FFI RAPPORT

MODELLING OF GRANULAR MATERIALS IN THE AUTODYN HYDROCODE

OLSEN Åge Andreas Falnes, TELAND Jan Arild

FFI/RAPPORT-2003/02057

FFIBM/766/130

Approved
Kjeller 21. July 2003

Bjarne Haugstad
Director of Research

MODELLING OF GRANULAR MATERIALS IN THE AUTODYN HYDROCODE

OLSEN Åge Andreas Falnes, TELAND Jan Arild

FFI/RAPPORT-2003/02057

FORSVARETS FORSKNINGSINSTITUTT
Norwegian Defence Research Establishment
P O Box 25, NO-2027 Kjeller, Norway

FORSVARETS FORSKNINGSINSTITUTT (FFI)
Norwegian Defence Research Establishment

P O BOX 25
N0-2027 KJELLER, NORWAY

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

REPORT DOCUMENTATION PAGE

1) PUBL/REPORT NUMBER FFI/RAPPORT-2003/02057	2) SECURITY CLASSIFICATION UNCLASSIFIED	3) NUMBER OF PAGES 55
1a) PROJECT REFERENCE FFIBM/766/130	2a) DECLASSIFICATION/DOWNGRADING SCHEDULE -	

4) TITLE
MODELLING OF GRANULAR MATERIALS IN THE AUTODYN HYDROCODE

5) NAMES OF AUTHOR(S) IN FULL (surname first)
OLSEN Åge Andreas Falnes, TELAND Jan Arild

6) DISTRIBUTION STATEMENT
Approved for public release. Distribution unlimited. (Offentlig tilgjengelig)

7) INDEXING TERMS

	IN ENGLISH:		IN NORWEGIAN:
a)	Two-component material	a)	To-komponent materiale
b)	Autodyn	b)	Autodyn
c)	Concrete	c)	Betong
d)	User subroutine	d)	Brukersubrutine
e)		e)	

THESAURUS REFERENCE:

8) ABSTRACT

Two methods for creating a two-component material in Autodyn are described. In the first approach we fill two separate subgrids with different materials whereas the second method fills one subgrid with two different materials. Both methods rely on the use of special user subroutines to create an Autodyn version with extended functionality. Instructions for using the extended Autodyn program are provided and the implementation is discussed and documented.

9) DATE	AUTHORIZED BY This page only	POSITION
21. July 2003	Bjarne Haugstad	Director of Research

ISBN-82-464-0755-4

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

CONTENTS

		Page
1	INTRODUCTION	7
2	METHOD A: TWO SUBGRIDS	7
2.1	Initialisation	8
2.2	User input	9
3	INTERNAL WORKINGS OF THE USER SUBROUTINE	10
3.1	Determining the size of the individual aggregate rocks	10
3.2	Determining the positions of the individual aggregate rocks	12
3.3	Defining the subgrids based on the aggregate array	12
4	METHOD B: ONE SUBGRID	13
4.1	How to use the program	13
4.2	How the program works	14
5	COMPARISON OF AGGREGATE ROCKS FROM THE TWO APPROACHES	16
6	SUMMARY	17
A	FORMAT OF DATA FILES	17
A.1	Transformation from 3D array to a scalar	17
A.2	aggregatedata.dat	18
A.3	facelength.dat	19
A.4	seed.dat	19
B	THE AUTODYN MACRO FACILITY	19
C	FINDING AND LOCATING AGGREGATE ROCKS	20
D	THE SOURCE CODE OF THE EXZONE PROGRAM	25
D.1	The `exzone` subroutine	25
E	THE SOURCE CODE FOR THE MACRO GENERATING PROGRAM	33
E.1	The main program	33
E.2	The `fill_region_and_write` subroutine	35
E.3	The place_aggregate_macro subroutine	40
F	VARIOUS SHORT SUBROUTINES	40
F.1	The modules	40

F.1.1	integer function neighbour_sum	41
F.1.2	logical function joined_node	42
F.1.3	subroutine lower_corner	42
F.1.4	subroutine upper_corner	43
F.2	The subroutine `check_region`	43
F.3	The subroutine `fill_region`	44
F.4	Subroutine `write_macro_file`	45
G	THE EXVAL SUBROUTINE	45
H	THE MAKEFILE	48
I	THE 2D VERSION	50
I.1	The 2D filling subroutine	51
I.2	The aggregatedata.dat file	53
I.3	The 2D Makefile	53
	Distribution list	55

MODELLING OF GRANULAR MATERIALS IN THE AUTODYN HYDROCODE

1 INTRODUCTION

Concrete consists, broadly speaking, of a mixture between two materials, namely cement paste and aggregate. As a consequence, concrete is not homogenous as these two components have very different material properties. In penetration simulations with numerical codes such as Autodyn, concrete has until now been described as a single homogenous material with "average" properties to account for the two-component structure. In this report the possibility of modelling the different components separately is considered.

In principle, it is very easy to generate such a two-component material. It is just a matter of creating a subgrid and filling it with the appropriate materials. However, doing this manually would be a very tedious and time consuming task because of the large number of (single) cells that would have to be filled one by one. Fortunately, it is possible for the user to create his own subroutines in Autodyn, thereby creating an Autodyn version with added functionality. In this report, we show how this can be exploited to generate two-component materials.

Two different methods for creating a two-component material are discussed. In the first, the cement and the aggregate reside in two different subgrids, while the other approach genuinely fills one subgrid with two different materials. A similar idea has earlier been implemented [1], where the two-component structure was accounted for by a modification of the yield stress in certain cells. The present report goes one step further and creates genuine two-component materials.

We start by showing how to use the extended two-component Autodyn version and then go on to describe the internal workings of the new user subroutines. The focus of this report is on 3D, but a 2D version has also been programmed.

2 METHOD A: TWO SUBGRIDS

The method of using two subgrids is the approach which the user is most likely to apply. Besides being the fastest and easiest method, it also has the advantage of allowing greater flexibility in how the cement can interact with the aggregate. One can choose between joining the subgrids node to node or an interaction based coupling. In the latter case we may specify a frictional constant, allowing a fairly complex interaction to be modelled. This possibility is particularly interesting in concrete, as certain rock types provide a poor connection between cement and aggregate.

The drawback of using two subgrids in Autodyn is that aggregate rocks must be separated by at least two cells, which limits the number of aggregate rocks that can be placed in a subgrid. Therefore we will later look into an alternative approach where only one subgrid is used.

In this section we assume that an extended version of Autodyn containing the relevant user subroutines has been compiled. The new executable file, by default called `aggregate`, should then reside in the same directory (bin) as ordinary Autodyn. On running this program, an extended version of Autodyn is started which incorporates the ability to easily define two-component materials.

When the materials are placed in separate subgrids, filling the subgrids with the appropriate materials becomes very easy. The problem is instead to define the position of the actual elements of each subgrid, a process which is called “zoning” in Autodyn terminology. This is where the user subroutines come in handy. A subroutine that is particularly suited to our problem is called `Exzone`. It can be accessed by the user from the standard Autodyn menus: `Zoning—Predefs—User—Exzone`. However, in the original Autodyn version, `Exzone` was empty so nothing happened when it was selected. In the extended Autodyn version, we have programmed our own `Exzone` subroutine that helps us place nodes at positions not available from the standard menu system. In the current version, `Exzone` only generates rectangular (cubical) subgrids, though.

2.1 Initialisation

Before starting Autodyn (technically only before calling the `Exzone` subroutine), the user must create a small ASCII-file called *aggregatedata.dat* and put it in the same directory as the modified Autodyn executable (`aggregate`). This file provides information to the Autodyn `Exzone` subroutine about the size and distribution of the aggregate rocks.

The easiest way to created this file is by using the Matlab pre-processor *aggdef.m*. However, it is also possible to define the file directly using a text editor. The file format is described in Appendix A.2.

On running *aggdef.m* the user is prompted for the following input parameters:

- The volume fraction of aggregate rocks (0-100%). This number determines how much aggregate there is in the concrete.
- Minimum distance (measured in number of cells) between the various aggregate rocks. For technical reasons this number must be 2 or larger.
- Damage level for cement cells adjacent to an aggregate cell. This is a number between 0 (no damage) and 1 (completely damaged).
- A list of the various aggregate sizes with the corresponding percentage of aggregate rocks that are larger than the defined size. Up to five different sizes can be defined.

When everything has been defined, the pre-processor creates the file *aggregatedata.dat*.

The user is then prompted for the face length of each cell in the the *i*- ,*j*- and *k*-directions. By default the value is 10 mm in all directions. After pressing OK, a file called *facelength.dat* is created. This file can, of course, also be created with a text editor.

2.2 User input

After the *aggregatedata.dat* and *facelength.dat* files have been created, the extended Autodyn version may be executed. The procedure for creating a two-component material is now as follows:

1. Define two new subgrids having the names CEMENT and AGGREGATE with an appropriate *ijk*-range. It is important that exactly these names are used and that the same *ijk*-range is selected for both of them.
2. Select the AGGREGATE subgrid and use the `Exzone` command. This is found under `Zoning-Predefs-User` in the Autodyn menu system.
3. The user is now prompted with *"Read or generate?"*. This question is slightly misleading as the only possible answers are *"yes"* and *"no"*. Usually the answer is *"no"*, as Autodyn then generates the position of each aggregate rock stochastically (based on the data in *aggregatedata.dat*) and stores the result in a file called *aggpos.dat*. Only if the user wants to use the aggregate positions from a previously generated *aggpos.dat* file, the answer to the question is *"yes"*.
4. Fill the complete AGGREGATE subgrid with an appropriate material by using the `Fill-Block` command.
5. Select the CEMENT subgrid and use the `Exzone` command again.
6. Always answer *"yes"* when prompted with *"Read or generate?"*. This makes sure that the CEMENT subgrid completely surrounds the aggregate rocks defined under point 3.
7. Answer *"yes"* or *"no"* to the question of whether damage should be included in surface cells.
8. Fill the complete CEMENT subgrid with an appropriate material by using the `Fill-Block` command. A macro called *ovrlap.mac* has now been created and stored in the *bin*-directory. Copy it to the *data*-directory. (If an old *ovrlap.mac* already exists, it must be deleted or renamed before filling the subgrid.)
9. Enter the `Fill` menu of CEMENT, press F6 and run the macro *ovrlap.mac* to remove overlapping cells between the two subgrids. A full explanation is provided in Appendix B.
10. Join or define an interaction between the CEMENT and AGGREGATE subgrids.

A typical two-component material is displayed in Figure 2.1. For a more detailed explanation of the inner workings of the code (what is actually going on), see the next chapter.

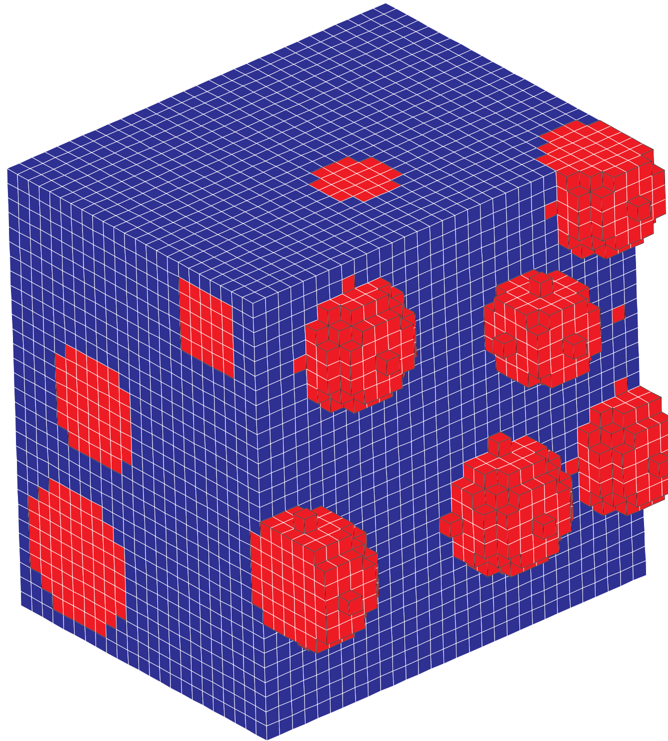

Figure 2.1 An example of a two-component material.

3 INTERNAL WORKINGS OF THE USER SUBROUTINE

In this chapter, we describe the internal workings of the `Exzone` subroutine and describe exactly what happens after it has been called from the Autodyn menu.

3.1 Determining the size of the individual aggregate rocks

In the following we assume that the user has answered "no" when prompted with "Read or generate?". This means that Autodyn will have to generate the position of each aggregate rock stochastically, and then store this information in a file called *aggpos.dat*.

The first thing Autodyn does is read the *aggregatedata.dat* file which contains all the information on the aggregate distribution. This information is then applied by the user subroutine `find_rocks2` to find the size of each individual rock. More precisely, the rock sizes are drawn randomly until the total volume of rocks within each interval is in compliance with the input defined in *aggregatedata.dat.* We are then left with an array containing the size of each individual rock in the aggregate.

Figure 3.1 may be helpful in visualising an example. Let us imagine a sieve curve with four size intervals. The aggregate rocks are grouped in four intervals: rocks with diameter between s_1 and s_0, rocks with diameter between s_2 and s_1, and so on. The total volume fraction of aggregate, denoted p, is the sum of the fractions of each interval: $p=f_1+f_2+f_3+f_4$. In other words, a fraction f_1 of the subgrid is filled with aggregate with size between s_1 and s_0. Since we know the volume of the subgrid V, it is simple to find the total volume of aggregate with diameter inside the interval. We can then keep selecting rocks randomly inside this interval until the volume exceeds $f_1 \cdot V$, at which time we switch to the next interval (diameter between s_2 and s_1) and repeat the process.

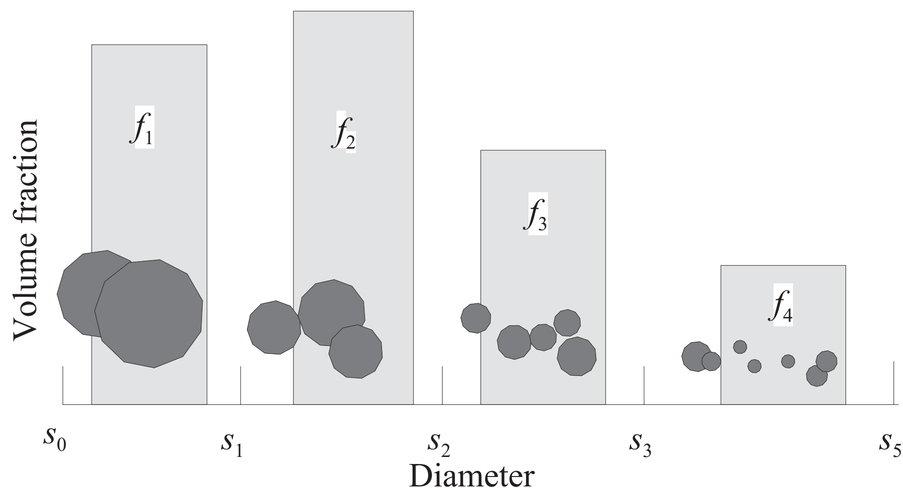

Figure 3.1 Within each interval, the sizes of each rock is drawn randomly.

However, on moving from real coordinates to *ijk*-indices, the rock sizes must be given in terms of integers rather than real numbers. Consequently, no rock can be smaller than one cell, which means that the sieve curve might need to be amended. The actual output sieve curve may therefore look slightly different from the input curve. The final result for the volume (number of cells) and diameter of each rock is stored in a file called *aggregate_output.dat*. The data in this file is used to create statistics on the output sieve curve and the output filling fraction, which is output to the screen. Typical output may look like this:

```
Finished creating aggregates!
Input aggregate ratio:  0.1
 # of aggregate rocks:   2
 Output ratio:           7.38125E-02
          Mesh size       Input fraction     Output fraction
           21.00                0.00                1.00
           18.00                1.00                1.00
```

The column "Mesh size" is the input sieve diameter. "Input fraction" is the input residual fraction for each mesh size, and the "Output fraction" is the corresponding output values. If the user is unsatisfied with the output curve, changes must be attempted in the input curve until a reasonable output result is obtained. Usually the discrepancies are fairly small, but the result gets worse for rocks that are small compared to the cell size. Further, since the output filling

fraction varies, the average density of the subgrid may differ from the measured values in the real concrete.

3.2 Determining the positions of the individual aggregate rocks

The program then moves on to the next step, which is performed by the subroutine `place_aggregate`. This subroutine selects a random position in the *ijk*-space for each individual rock, and then fills the internal variable `aggregate_array` (which has dimension exactly corresponding to the subgrid) with 1's in every element that should contain rock material. The task is non-trivial if we insist that aggregate rocks should not overlap. Indeed, with this constraint, we may not find room for every aggregate rocks regardless of how much we try. As a result, there are two important features of the subroutine:

- The order in which rocks are placed is random, that is, we draw a random rock from the array created in `find_rocks2`, then draw a random location for this rock. We repeat until we are done.
- If a position drawn is overlapping with previously placed rocks, the entire `aggregate_array` is searched in order to find an available position. Should none be found, the routine starts on a new rock.

3.3 Defining the subgrids based on the aggregate array

Assuming the `aggregate_array` has been defined, Autodyn creates corresponding subgrids when the `Exzone` subroutine is invoked by the user. For the AGGREGATE subgrid, cells are defined at all positions where the corresponding `aggregate_array` element has the value 1. Similarly, the cells of the CEMENT subgrid are defined at positions where the `aggregate_array` takes on a value of zero. The `Exzone` user subroutine is called with the present *ijk*-range as formal parameters.

Generating a cell is done by defining all eight corner nodes, after which the cell is recognised by Autodyn and included in all calculations unless explicitly declared as unused. However, such an automatic generation of cells gives rise to a problem. An example is illustrated in Figure 3.2. There we have two separate subgrids, and in one of them (the blue in the figure) a single cell protrudes from the rest of the subgrid. This single cell is meant to fit into a corresponding empty slot in the other subgrid (the red one). However, while defining the cells surrounding this slot, it is impossible to avoid defining all four corner nodes, and as a consequence, Autodyn inserts a cell where there should have been empty space. The result is overlapping cells when the two subgrids are joined.

The only feasible way to avoid this problem is by defining the overlapping cells as unused in one of the subgrids via a macro file. Therefore Autodyn (automatically) generates a macro file called *ovrlap.mac* to perform this. It is based on a simple algorithm explained in Appendix D and must be copied to the data directory before execution. It is important to keep in mind that

the macro file must be used after filling the subgrids with materials, since any subsequent fillings will override the action performed by the macro file.

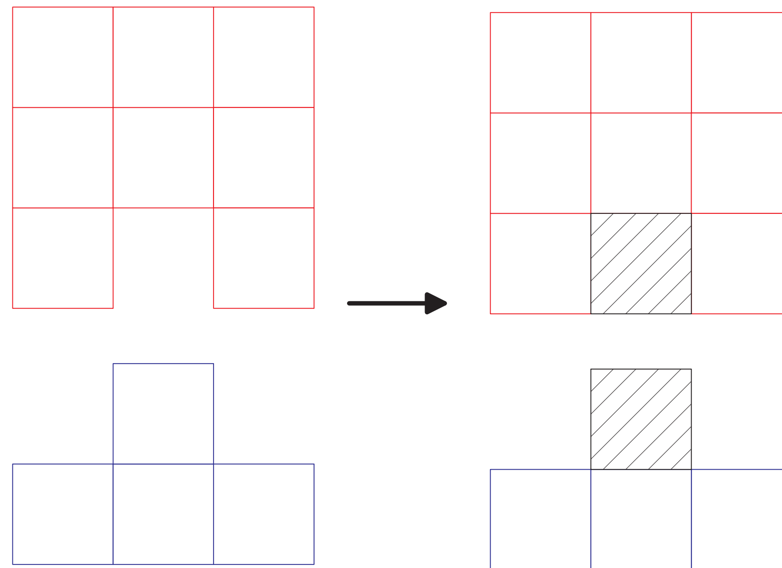

Figure 3.2 The problem of overlapping cells when defining two subgrids.

4 METHOD B: ONE SUBGRID

Instead of creating two different subgrids and filling them completely with separate materials, we here examine a different approach where a single subgrid is filled with two different materials. This method enables us to avoid the restriction of having at least two cells between each aggregate rock.

Instead of the difficult part being to generate the zoning for the two subgrids, we now instead have to find an effective way of filling a single subgrid with two different materials appropriately. This is done by automatically generating a macro file which is later executed in Autodyn by the user.

Note that for technical reasons explained later, the aggregate rocks will not be equally nice as in the two-subgrids method.

4.1 How to use the program

The program which generates this macro is called *generate_macro* and runs completely independent of Autodyn. However, before running the program, the file *aggregatedata.dat* has to be present in the same directory. This file has the same structure as in the two-subgrid case, but a few extra input parameters have to be supplied. The easiest way to generate this file is to use the Matlab pre-processor *aggdef2.m*. The user is then prompted for the following input parameters:

- Name of the macro file (exactly six characters).
- Name of the aggregate material for use in Autodyn
- Initial density of the aggregate material.
- The *ijk*-range of the subgrid region to be filled.

The other input parameters are identical to the two-subgrids case:

- The volume fraction of aggregate rocks (0-100%). This number determines how much aggregate there is in the cement.
- Minimum distance (measured in number of cells) between the various aggregate rocks.
- The various aggregate sizes and the corresponding percentage of aggregate rocks having size larger than the defined size. Up to five different sizes can be defined.

After the user has specified these input parameters, the *aggregatedata.dat* file appears in the directory. Now *generate_macro* can be executed, which results in a macro with the chosen filename being created. This macro must be put in the Autodyn *data* directory.

The regular version of Autodyn can now be started. The procedure is now to first define a subgrid and fill it completely with cement. When the macro is executed from the `Fill` menu, the aggregate rocks will be filled in as well.

Note that the file *aggregate_output.dat* must be removed every time the program is run. Although the program will operate even if this file exists, the output sieve curve data will be wrong because new rock sizes are only appended to the file.

4.2 How the program works

The macro file generator *generate_macro* is not linked with Autodyn in any way. A diagram showing the general structure of the program is shown in Figure 4.1.

Things are similar to the two-subgrids case when it comes to determining the size and position of each rock. However, there is a significant difference in the way that the subgrid is filled with aggregate rocks. In the two-subgrids case, the rocks were spherical (or rather a discrete version of a sphere). This could have been implemented here as well by filling each aggregate rock cell by cell, but this would have taken a fairly long time because of screen updates after each command. A different approach is thus needed, which minimises the number of operations yet retains the shape of the rocks at least roughly. The chosen method is illustrated in Figure 4.2.

The first stage is to define a central square. New cells are then added in stages as shown by the red cells in the figure. The advantage of defining a block of cells at a time is that each block only requires one interactive call to the block-command. For the particular case illustrated in the figure, we would need 73 calls if the rock was filled cell by cell, whereas now we only

need 1+4+4=9 calls. The extension to 3D is straightforward, in which case the starting object is a cube instead of a square.

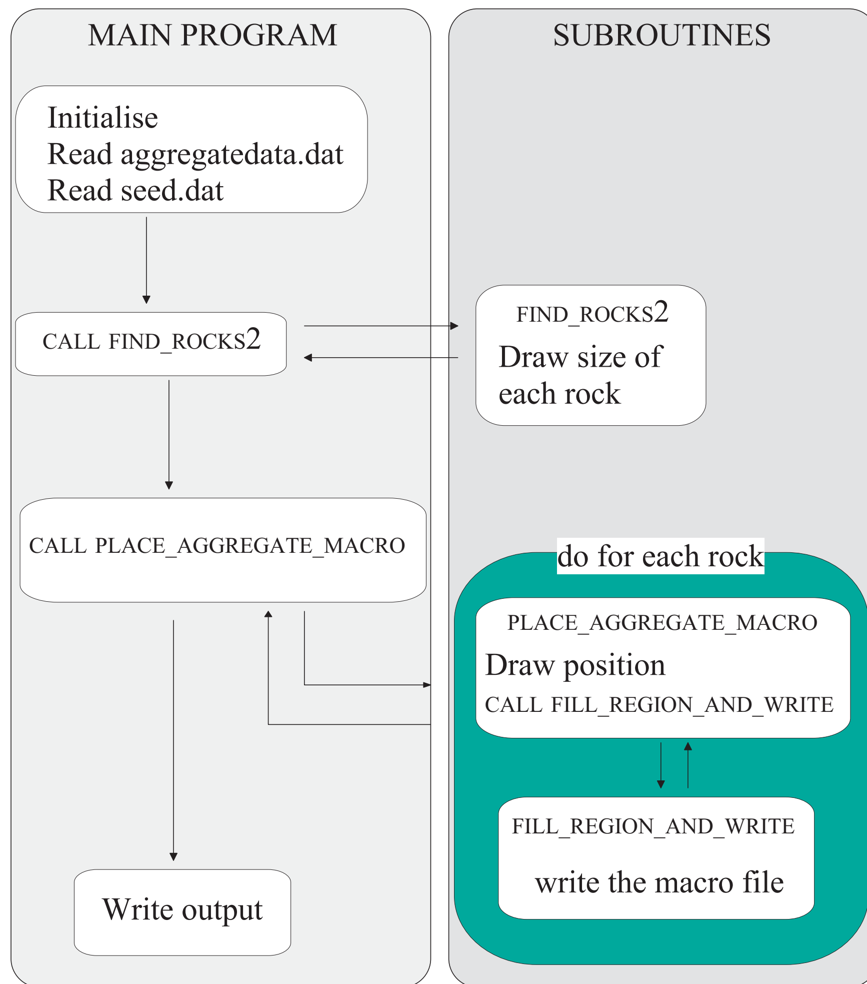

Figure 4.1 A sketch of the structure of the generate_macro program.

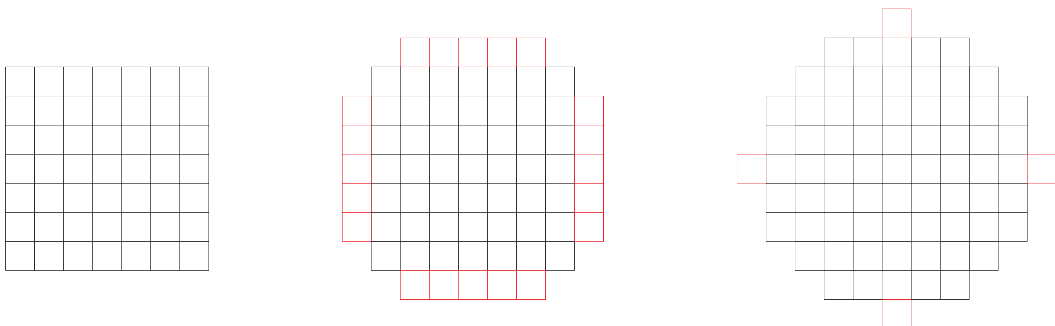

Figure 4.2 The algorithm for filling a rock in the one-subgrid case.

The corners of the square/cube are defined from the input radius r (one half the input diameter) relative to the center element. In 3D the distance is given by $r/\sqrt{3}$, or the nearest integer to this value. The padding is done until the diameter of the rock reaches the input value, or we

have padded single cells as in the final stage of Figure 4.2. Finally, the program also writes the map array to the file called *aggpos.dat*, just as in the two-subgrids case.

The disadvantages of this algorithm is that the rocks will not look exactly as spheres, and that the code is more tedious to write and read.

5 COMPARISON OF AGGREGATE ROCKS FROM THE TWO APPROACHES

In Figure 5.1 and 5.2 we show how the aggregate rocks may look after having been created using the two different methods. The rocks shown in the figures have all been generated with the same *aggregatedata.dat* file, and with the same ranges (1 to 41 in all three directions). The difference is quite apparent. Note that in concrete, rocks will usually be smaller compared with the cell size.

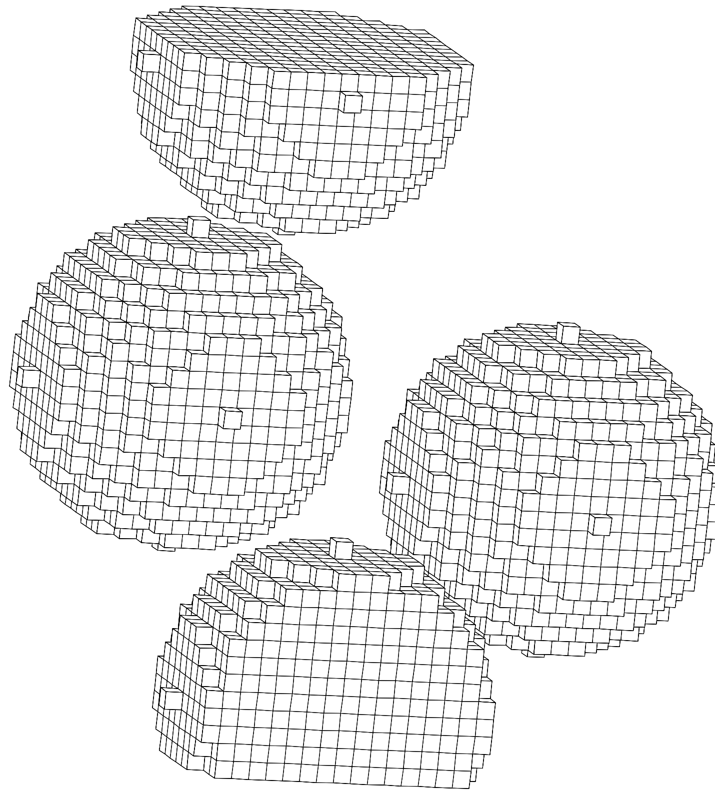

Figure 5.1 Aggregate rocks generated in the two-subgrids case.

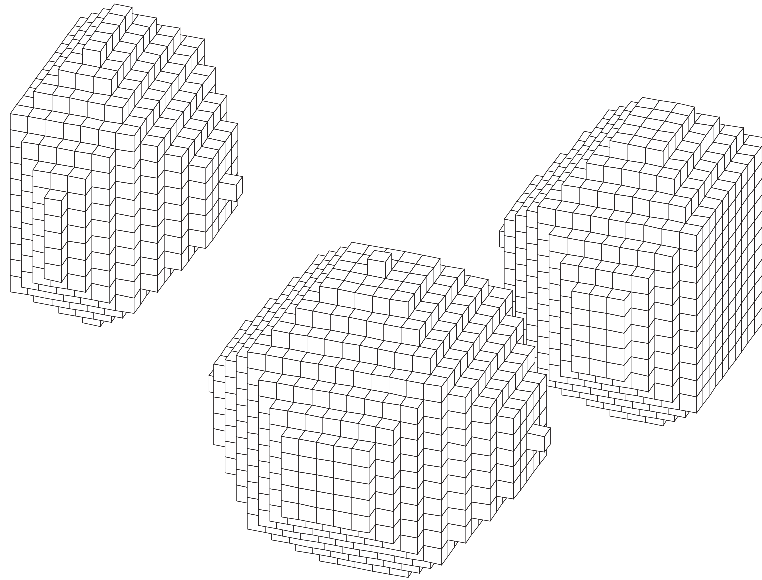

Figure 5.2 Aggregate rocks generated using the macro file generating program.

6 SUMMARY

Two methods for generating a two-component material have been described. This might prove very useful, particularly in penetration simulations against concrete or other heterogenous materials. Comparison of simulations on penetration into heterogenous and homogenous materials is beyond the scope of this report.

References

[1] Soleng H H, A Stochastic Two-Component Material Model: Documentation of an Implementation as Fortran 90 Subroutines in Autodyn, FFI/RAPPORT-2001/01089

A FORMAT OF DATA FILES

This chapter describes the format of the various data files that are used by Autodyn.

A.1 Transformation from 3D array to a scalar

The `aggregate_array` is written to a direct access, unformatted file. In such a file, we can access any value directly since we specify a record number in all read statements. Direct access requires a record length and a unique way of addressing each record in the file. Essentially this amounts to writing a 3D array on a 1D line, so we need a transformation rule between position in the file and element index. Figure A.1 shows the idea behind a 2D procedure that works.

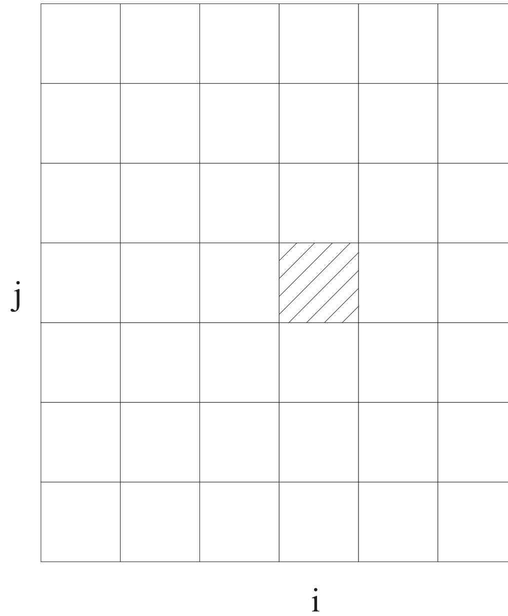

Figure A.1 *A sketch showing how we can count the cells in a unique way. The indicated cell has indices (i,j), but from the lower left corner the cell number is* $(j\text{-}1)i_{max}+i$.

Suppose the maximum number of cells in the *i*-direction is i_{max}. Also, suppose we are going to label each cell in the grid with a unique number. Let us focus on the cell (i,j) in the figure. There are $(j\text{-}1)i_{max}$ cells in the preceding rows, and i-1 preceding cells in the *j*th row: thus, the indicated cell is number *i* in this row. Hence the cell number is $(j\text{-}1)i_{max}$+i. An extension to 3D is achieved by the following formula, if we let the maximum ranges of the grid be $(i_{max},j_{max},k_{max})$ and the cell number be denoted n_{cell}:

$$n_{cell} = k + (j-1)k_{max} + (i-1)\,j_{max}k_{max}$$

It is easily seen that for the first cell (1,1,1) n_{cell}=1 and for the last cell $(i_{max},j_{max},k_{max})$ the cell number is $n_{cell}=i_{max}j_{max}k_{max}$.

A.2 aggregatedata.dat

The file *aggregatedata.dat* that is generated by the Matlab pre-processor has the following simple structure:

1. The name of the macro file, exactly 10 characters including the .mac extension (only relevant when using one single subgrid).
2. The name of the aggregate material in the Autodyn model, and the initial density of this material. The name must be written in capital letters, and must come first (only relevant when using one single subgrid).
3. The range of the subgrid region to be filled, in the following order: i_{min}, i_{max}, j_{min}, j_{max}, k_{min} and k_{max}. Each number must be an integer and they must be separated by a space. Then a text string with the name of the subgrid where damage is to be implemented (only relevant when using one subgrid), and finally a real number giving the damage level.

4. The volume fraction of rocks and an integer indicating the minimum distance between rocks.
5. The number of sieve curve points.
6. This and the following lines: the sieve curve points.

This file has been designed to have the same format independent of whether one or two subgrids are used. As a consequence, the two first lines and everything except the last number on the third line are not relevant (and not used) in this case. However, some (dummy) data must still be supplied to make sure Autodyn is not confused. This is done automatically by the Matlab pre-processor.

An example of an *aggregatedata.dat* file for a two-subgrid case is the following:

```
x
x x
1 1 1 1 1 1 z 0.30
0.15 2
3
10.0 0.0
8.0 0.5
6.0 1.0
```

A.3 facelength.dat

The format of the file *facelength.dat* is very simple as it just contains the three facelength values (*i,j* and *k* in that order), separated by a space. An example is:

```
10.0 5.0 8.0
```

which defines a length of 10 mm in the *i*-direction, 5 mm in the *j*-direction and 8 mm in the *k*-direction.

A.4 seed.dat

Finally, a file called *seed.dat* must be present in the bin directory. This file contains 17 integers between –30000 and 30000 and are used to initialize the random generator. Each time Autodyn is started, the random generator is initialized in the same way and therefore produces the same random numbers. To generate different numbers, a different seed file is needed. An example of a seed file may be generated by typing *make example* in the source file directory.

B THE AUTODYN MACRO FACILITY

Macros are stored sequences of keystrokes that can be executed via a single command in the Autodyn menu. They can be defined in the interactive menus at any time by pressing `F6-`

`Macro-Define`. All subsequent keystrokes are then saved in a file with the extension .mac, until F6 is pressed again. The macro file is located in the data directory.

When this macro is run `(F6-Macro-Run)`, Autodyn executes the same sequence of keystrokes automatically. However, the screen is still updated, and all dialogue boxes are displayed as usual.

It is a major disadvantage of the macros that the screen is updated after each command. As a consequence, it takes a long time to fill a subgrid one cell at a time using this approach. Further, this means that we can not create a long macro and let Autodyn run overnight in batch. Macros are also fuzzy about swapping of workspaces in the CDE environment, which may cause Autodyn to crash. (CDE=Common Desktop Environment, the interface in the HP-UX environment used at FFI).

The format of a macro file is quite simple, and a typical example may look like this:

```
$Block
%
2
7
16
21
27
32
PEBBLES
%
2.9
0.0
0.0
0.0
0.0
0.0
Spherical
f6
N
```

On running this macro, Autodyn enters the Block command, defines the *i*,*j*,*k*-indices of the region (*i*=2 to *i*=7, *j*=16 to *j*=21, and *k*=27 to *k*=32) and the material name (PEBBLES), and the density (2,9), velocities (0,0) and the Spherical option. Obviously all macro files will end with a F6 as this indicates that the macro has been defined. The “N” in the last line means that the macro is not rerun. Notice that this macro can only be run from the `Fill`-menu. Running from any other menus would cause an error, including the `Block`-menu.

C FINDING AND LOCATING AGGREGATE ROCKS

Firstly, the size of each rock is drawn. The sizes are then returned in an array. The following subroutine handles this:

```
subroutine find_rocks2(rckvol, distribution)
```

```
  use map_array_functions, only : rocks=>sizes_of_rocks
  use inputs, only : no_of_sieves=>sieves, sieve_data=>sieve_array
```

The modules contain certain variables that are tedious to pass to the subroutine as formal parameters. The modules are documented in Appendix F. It suffices to say here that they contain the arrays that store information on the sieve curve and the array `aggregate_array`. In the module inputs, `sieve_array` is a 2D array. In the first column it contains data on sieve mesh size. In the second it contains the present proportion of residue (in other words, the relative mass left in the sieve with the given mesh size). The arrays are stored in modules, so provided the arrays have been filled correctly in the main program there should be no problems.

```
real, dimension(1:no_of_sieves), intent(out) :: distribution
```

`distribution` is an array that contains the same information as `sieve_data`, but for the output values: the actual size distribution in the rocks array.

```
integer :: i, rocknr
real, intent(in) :: rckvol
```

The number `rckvol` is the total rock volume in the subgrid. When enough rocks have been drawn that their total volume adds up to this value, the drawing is completed and we return to the user subroutine again.

```
real :: currentvol, rocksize, this_sieve

distribution=0
rocknr=1
```

It is necessary to draw sizes systematically. We loop through all sieve sizes, and draw rocks within the boundaries set by the sieves until the rock volume is consistent with the input fraction inside each size interval.

```
do i=1,no_of_sieves-1
```

Loop through each sieve:

```
this_sieve=rckvol*(sieve_data(2,i+1)-sieve_data(2,i))
```

Find the total volume of rocks that have a size within the interval set by the sieve curve and multiply the total rock volume with the fraction of rocks inside this sieve interval.

```
currentvol=0.0
```

Initiates the variable that keeps track of how large the total volume is.

```
do while (currentvol<this_sieve)
```

We have now entered the loop that actually draws the rock sizes. `random_number` returns a value between 0 and 1. Keep drawing until the volume is large enough.

```
call random_number(rocksize)
rocksize=(rocksize*(sieve_data(1,i)-sieve_data(1,i+1))+&
sieve_data(1,i+1))/2
currentvol=currentvol+4*3.1416*(rocksize**3)/3
```

We have now calculated the current volume of rock.

```
rocks(rocknr)=rocksize
rocknr=rocknr+1
end do
distribution(i)=rocknr
```

`rocknr` holds the number of rocks within the present interval. Distribution stores this information.

```
end do
return
end subroutine find_rocks2
```

We now have an array called `rocks` that contains the size of each rock. The trouble now is that this array contains real numbers: we can only fill rocks with a discrete volume. In the present implementation it is even worse: the rocks are spherical but can only have a discrete radius. This restriction drastically limits the actual size distribution we eventually see.
Still, let us move on to the main subroutine, the one that actually fills the `aggregate_array`, the array that is a map of the subgrid with information in which cells are filled with aggregate. Inside this subroutine the `aggregate_array` is called `aggregate_positions`. Its dimensions are determined outside this subroutine, and so the minimum and maximum index in each of the three spatial directions is passed as formal parameters.

```
subroutine place_aggregate(no_of_rocks)

  use map_array_functions
  use inputs, only : safety

  implicit none

  integer, intent(in) :: no_of_rocks
  real,dimension(:),allocatable :: help
  integer :: rock,this_rock,sizeint,p1,p2,p3
  logical :: free
  integer, dimension(1:3) :: low_corner, high_corner, center
```

`no_of_rocks` is the number of rocks that were drawn in the `find_rocks2` subroutine documented above. `safety` is a number that determines the minimum distance between neighbouring aggregate rocks. `help` is just an array used to store the values of the

sizes_of_rocks array locally. This is because the array will be manipulated in this subroutine, and I would prefer the original array to stay unchanged for other parts of the program.

```
rock=no_of_rocks
  allocate(help(1:rock))
  help=sizes_of_rocks(1:rock)
  do
     call random_uniformint(1,rock,this_rock)
```

Return a random integer in the range (1, rock).

```
sizeint=nint(size_array(this_rock))+safety
```

Round off the radius of each rock, and add the safety margin.

```
!draw positions
     call random_uniformint(imn,imx,p1)
     call random_uniformint(jmn,jmx,p2)
     call random_uniformint(k_min,kmx,p3)
     center(1)=p1
     center(2)=p2
     center(3)=p3
```

We have now drawn a random position for a rock. We want to check if the position is such that the rock overlaps with other rocks. In order to do this, we define a subregion of the array to search in, and pass this on to the check_region subroutine. We also want to keep looking for a position for the rock until we have searched the entire subgrid.

```
!we have to find somewhere to place the rock. If the random location
     !is occupied, we want to keep looking until we either find a location
     !or we come back to the starting point, in which case there is no free
     !location available.
     free=.false.
     do
        !define the cubic region in which the rock will be embedded.
        !first define the lower i, j and k values.
        call lower_corner(center,sizeint,low_corner)
        call upper_corner(center,sizeint,high_corner)
        !now we're ready to check if the region contains any rocks already
        call check_region(low_corner, center, high_corner, sizeint, free)
```

The variable free is true if the region was empty, and false otherwise. We then either exit this do-loop, or we move on to the next array element, repeat the check, and so on until we have looped through the entire array, or we have actually found a position for the rock.

```
if (free) then
           exit
        else
           center(1)=center(1)+1
           if (center(1)==imx) then
              center(1)=imn
```

```
            center(2)=center(2)+1
            if(center(2)==jmx) then
               center(2)=jmn
               center(3)=center(3)+1
               if (center(3)==kmx) then
                  center(3)=k_min
               end if
            end if
         end if
         if (center(1)==p1 .and. center(2)==p2 .and. &
              center(3)==p3) then
            ! If it is impossible to find a position for the rock.
            write(6,*) 'Warning: could not find room for rock ', this_rock
            exit
         end if
      end if
   end do
if (free) then!if a location has been found, then fill with a rock
```

The variable `free` is initialised as false before the previous serach loop is executed. If the loop has been finished without any free spot available for the rock, `free` is still false. Otherwise it will be true, and hence `free` is suited to determine whether we should fill the `aggregate_positions` array with 1's for this rock.

```
if (free) then!if a location has been found, then fill with a rock
        sizeint=sizeint-safety!this is the actual size of the rock
        call fill_region(low_corner, center, high_corner, sizeint)
```

The subroutine `fill_region` replaces all 0 elements with 1's inside the rocks. Filling a rock in the map array is done simply by looping through the cubic region around the center element of the rock, and every element closer than the desired radius is given the value 1. Expressing this as an equation, suppose that the center element index is given by (i_c, j_c, k_c) and the radius of the spherical rock is R. Then every element (i,j,k) satisfying the following condition is filled:

$$(I - I_c)^2 + (J - J_c)^2 + (K - K_c)^2 \leq R^2$$

Only the elements inside a cube with sides $2R$ are checked, thus avoiding a loop through the entire array.

We now want to avoid drawing the same rock again, and this is where the array `help` is useful. We rearrange the array of rock sizes so that the last element of the array is set to 0 while all elements after the present rock is moved one index forward. By using the array `help` we achieve this without altering the original `sizes_of_rocks` array.

```
!now we need to redefine some variables to make sure we don't draw
        !the same rock again
        if (rock<no_of_rocks) then !we are not at the last element in the
           !array of rock sizes
           help(this_rock:no_of_rocks-1)=help(this_rock+1:no_of_rocks)
           help(no_of_rocks)=0.0
           !now we have moved the rock already placed to the last position
in
```

```
           !the array. When the rock variable is reduced by one, we are
certain
           !that this rock will never be drawn again
        end if
        rock=rock-1
        if (rock<1) then
           write(6,*) 'Finished creating aggregates!'
           exit
        end if
     else
```

We stop placing rocks if we encounter one there is no room for.

```
exit
     end if
  end do
  deallocate(help)

  return
end subroutine place_aggregate
```

We now have an array filled with 0's and 1's, where 0's indicate a cement cell in a subgrid and 1's indicate an aggregate cell.

The subroutines `fill_region` and `check_region` are documented in Appendix E.

D THE SOURCE CODE OF THE EXZONE PROGRAM

D.1 The `exzone` subroutine

We now proceed to the details.

First, some declarations:

```
SUBROUTINE EXZONE (SUBGRD,I1,I2,J1,J2,K1,K2)

USE mdgrid
USE kindef

use map_array_functions
use inputs

IMPLICIT NONE

INTEGER (INT4) ::   IJK,I,      J,      K
INTEGER (INT4) ::    I1,     I2,     J1,     J2,     K1,     K2
CHARACTER (LEN=10) SUBGRD
integer :: jj, map_sum
integer :: openstat,readstat
real :: stepi,stepj,stepk,rockvol
real, allocatable, dimension(:,:) :: output_distr
real, allocatable, dimension(:,:) :: output_sieve_array
```

```
integer :: rocknumber
character :: yes_or_no


imn=i1
jmn=j1
k_min=k1
imx=i2
jmx=j2
kmx=k2
```

Ask whether to generate an `aggregate_array`. If the user wants to generate the array, we first need to read the input file:

```
call getyon(yes_or_no,'$Read or generate (read=yes, generate=no)$')
if (yes_or_no=='N') then
   allocate(aggregate_array(imn:imx,jmn:jmx,k_min:kmx))
   aggregate_array=0
   open(11,file='aggregatedata.dat',form='formatted',&
        iostat=openstat,status='old')
   if (openstat==0) then
      readstat=0
      read(11,*,iostat=jj)!the strange reading makes it possible to read
      readstat=readstat+jj!the same aggregatedata.dat file in both the
      read(11,*,iostat=jj)!this and the EXVAL user subroutine.
      readstat=readstat+jj
      read(11,*,iostat=jj) sieves, sieves, sieves, sieves, sieves, sieves, &
                   subgrid_name, damage_level
      readstat=readstat+jj
      read(11,*,iostat=jj) rockratio, safety
      readstat=readstat+jj
      read(11,'(I4)',iostat=jj) sieves
      readstat=readstat+jj
      allocate(sieve_array(1:2,1:sieves))
      read(11,*,iostat=jj) sieve_array
      readstat=readstat+jj
      close(11)
      if (readstat==0) then
         call read_seed_data()
         allocate(distr(1:sieves))
         rockvol =rockratio*real((i2-i1)*(j2-j1)*(k2-k1))
         rocknumber=int(rockvol)
         allocate(sizes_of_rocks(1:rocknumber))
         call find_rocks2(rockvol, distr)
         rocknumber=int(distr(sieves-1))
         if (rocknumber<=1) then
            call messag('$Rocknumber defaults to 1$')
            rocknumber=1
         end if
```

And now we have allocated all the memory we need, and we are ready to start filling the `aggregate_array`:

```
         call place_aggregate(rocknumber)
         call messag('$Aggregate generated. See screen for details.$')
```

Write the output to a file. This is data on how the actual sieve curve looks, and how large fraction of the target have been filled with aggregate.

The output data is stored in the file aggregate_output.dat by the subroutine `fill_region` during the filling phase. The file contains one column of numbers indicating the volume of each rock (in numbers of cells), and one column containing the diameter. The volume is used to calculate the mass fraction, and the diameter is used to calculate sieve interval relevant for each rock.

First we find how many rocks that have been generated. We need this number in order to allocate memory for the array `output_distr`.

```
    open(43,file='aggregate_output.dat',form='formatted')
         i=1
         do
            read(43,*,iostat=readstat)
            if (readstat/=0) exit
            i=i+1
         end do
         i=i-1
         close(43)
         allocate(output_distr(1:2,1:i))
```

We now read the rock data. For each rock we find the relevant sieve interval, and add the rock volume to the total volume within that sieve interval. Finally we calculate fractions by dividing with the total number of cells in the target.

```
open(43,file='aggregate_output.dat',form='formatted')
         j=0
         allocate(output_sieve_array(1:2,1:sieves))
         output_sieve_array=sieve_array
         output_sieve_array(2,1:sieves)=0.0
         do k=1,i
            read(43,*,iostat=readstat) output_distr(1,k),&
                 output_distr(2,k)
            j=j+output_distr(1,k)
            do rocknumber=1,sieves
               if (sieve_array(1,rocknumber)<&
                    output_distr(2,k)) then
                  output_sieve_array(2,rocknumber)=&
                       output_sieve_array(2,rocknumber)+&
                       real(output_distr(1,k))
               end if
               if (rocknumber==sieves .and. sieve_array(1,rocknumber)==&
                    output_distr(2,k)) then
                  output_sieve_array(2,rocknumber)=&
                       output_sieve_array(2,rocknumber)+&
                       real(output_distr(1,k))
               end if
            end do
         end do
         close(43)
         do k=1,sieves
            output_sieve_array(2,k)=&
                 output_sieve_array(2,k)/real(sum(output_distr(1,1:i)))
         end do
         write(6,*) 'Input aggregate ratio: ',rockratio
         write(6,*) '# of aggregate rocks:  ',i
```

```
      write(6,*) 'Output ratio:          ',real(j)/&
           (real((i2-i1)*(j2-j1)*(k2-k1)))
      write(6,'(3a20)') 'Mesh size','Input fraction','Output fraction'
      do j=1,sieves
         write(6,'(3(12x,f5.2,3x))') sieve_array(1,j),&
              sieve_array(2,j),&
              output_sieve_array(2,j)
      end do
      deallocate(sieve_array)
      deallocate(sizes_of_rocks)
      deallocate(distr)
   else
      write(6,*) 'Readstat error ', readstat
   end if
else
   write(6,*) 'Openstat error ', openstat
end if
close(11)
```

We must store the `aggregate_array` in a file so that it is possible to make another subgrid later based on the same `aggregate_array`. For this purpose we use a direct access, unformatted file.

```
inquire(iolength=jj) openstat
open(31,file='aggpos.dat',form='unformatted',access='direct',recl=jj,&
     iostat=openstat)
if (openstat/=0) then
   call messag('$Could not open file aggpos.dat.$')
else
   do i=1,i2-i1+1
      do j=1,j2-j1+1
         do k=1,k2-k1+1
            write(31,rec=k+(j-1)*(k2-k1+1)+(i-1)*(j2-j1+1)*(k2-k1+1))&
                 aggregate_array(i+i1-1,j+j1-1,k+k1-1)
         end do
      end do
   end do
   close(31)
end if
```

We also need to generate a macro file which is to be used to declare overlapping cells unused. We declare cells unused if all corner nodes have been defined in the CEMENT subgrid, that is, each of the eight corner nodes are also a corner of at least one element containing a 0 in the `aggregate_array`. The way we find unwanted cells is shown schematically in Figure 6.1 for the 2D case. The extension to 3D is straightforward, the only difference being that each node in 3D is common to eight cells.

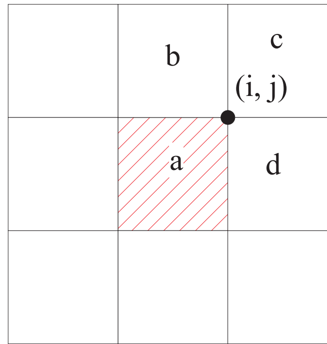

Figure D.1 *Let the center cell, shown here with index (i,j), be a cell with element value 1. The node (i,j) is common to the four cells a, b, c and d. If at least one of the four cells have element value 0, then the node (i,j) will be defined in both subgrids. Furthermore, if all four corner nodes to the cell (i,j) are defined in both subgrids, then a cell overlap will occur.*

The code below is a realisation of that algorithm. The logical function `joined_node` returns `.true.` if one or more of the eight cells that share the input node has the value 0. If this is the case, the node will be defined in both subgrids. Thus we test all eight corner nodes to find how many are defined in both subgrids: if all eight nodes are, then the cell must be declared unused in one of the subgrids.

```
 open(31,file='ovrlap.mac',form='formatted',iostat=openstat,&
        status='new')
   if (openstat/=0) then
      call messag('$Trouble with overlap-files.$')
   else
      readstat=0
do i=i1+1,i2
         do j=j1+1,j2
            do k=k1+1,k2
               if (aggregate_array(i,j,k)==1 .and. &
                     neighbour_sum(i,j,k)>1) then
                  if ((joined_node(i,j,k)) .and. &
                     (joined_node(i-1,j,k)) &
                     .and. (joined_node(i-1,j-1,k)) .and. &
                     (joined_node(i,j-1,k)) &
                     .and. (joined_node(i,j,k-1)) .and. &
                     (joined_node(i-1,j,k-1)) &
                     .and. (joined_node(i-1,j-1,k-1)) .and. &
                     (joined_node(i,j-1,k-1)))&
                     then
                     write(31,'(A7)') '$Unused'
                     write(31,'(A1)') '%'
                     write(31,*) i-1
                     write(31,*) i
                     write(31,*) j-1
                     write(31,*) j
                     write(31,*) k-1
                     write(31,*) k
                  end if
               end if
            end do
         end do
      end do write(31,'(A2)') 'f6'
```

```
      write(31,'(A1)') 'N'
   end if
   close(31)
else
```

The reader is perhaps a little desoriented at this point. The else-branch we are about to enter, is the alternative to generating the `aggregate_array`. In other words, all the previous code is executed only if the user replyed NO in the prompt. In case the answer was yes, we assume that the unformatted file containing the `aggregate_array` exists, and that the macro file has been generated already.

So, first we need to read the data in the *aggpos.dat* file.

```
      do i=1,i2-i1+1
         do j=1,j2-j1+1
            do k=1,k2-k1+1
               read(31,rec=k+(j-1)*(k2-k1+1)+(i-1)*(j2-j1+1)*(k2-k1+1),&
                    iostat=readstat)&
                    aggregate_array(i+i1-1,j+j1-1,k+k1-1)
            end do
         end do
      end do
      close(31)
   end if
   if (readstat/=0) then
      call messag('$File reading error. See screen for more info.$')
      write(6,*) 'The file aggpos.dat could not be read. '
      write(6,*) 'Make sure the file exists, if not, choose ',&
           'NO when prompted READ OR GENERATE. '
      write(6,*) 'A further mistake can be a mismatch between the ',&
           'i,j,k-range'&
           ,' in this subgrid and the one used when the file was '&
           ,'generated.'
      aggregate_array=0
   end if
end if
```

If the user has specified a file called facelength.dat, containing the lengths of the cell faces in all three directions, then use these values when the subgrid is generated. In fact, it does not really matter what these cell face lengths are, since it is possible to scale any subgrid from the Autodyn menus.

```
open(12,file='facelength.dat', form='formatted', status='old',&
     iostat=openstat)
read(12,*, iostat=readstat) stepi, stepi, stepk
close(12)
if(readstat/=0 .or. openstat/=0) then
   call messag('$Cell size defaults to 10 mm$')
   stepi=10.0
   stepj=10.0
   stepk=10.0
end if
```

We are now ready to embark on the main part of the program. We have an `aggregate_array`, either generated or read from a file, and we know the cell faces.

```
if (subgrd=='CEMENT') then
```

This is if the present subgrid is the cement subgrid. Loop through all prospective cells. If the `aggregate_array` contains a 0 at the element corresponding to the present cell, then generate all eight nodes necessary to define this cell. Otherwise, do nothing.

The user is prompted whether to include damage in the cells close to the aggregate rocks. The damage is stored in the user variable `var01`, and this variable must be defined in the menus before generating the subgrid.

```
call getyon(yes_or_no,'$Include damage in surface cells$')
   if (yes_or_no=='Y') then
      write(6,*) 'Damage level is ',damage_level
   end if
   do i=i1+1,i2
      do j=j1+1,j2
         do k=k1+1,k2
            if (aggregate_array(i,j,k)==0) then
               ijk=ijkset(i,j,k)!Corner node #1
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i-1,j,k)!Corner node #2
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i-1,j-1,k)!Corner node #3
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i,j-1,k)!Corner node #4
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i,j,k-1)!Corner node #5
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i-1,j,k-1)!Corner node #6
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i-1,j-1,k-1)!Corner node #7
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i,j-1,k-1)!Corner node #8
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-2)
               if (yes_or_no=='Y') then
```

The function `neighbour_sum` is found in the `map_array_functions` module.

```
                  map_sum=neighbour_sum(i,j,k)
                  if (map_sum/=0) then
                     var01(ijkset(i,j,k))=damage_level
                  else
```

```
                    var01(ijkset(i,j,k))=0.0
                 end if
              end if
           end if
        end do
     end do
  end do
end if
```

Now do exactly the same if the subgrid is called aggregate, only this time generate cells when `aggregate_array` contains a 1 in the relevant element.

```
if (subgrd=='AGGREGATE') then
   do i=i1+1,i2
      do j=j1+1,j2
         do k=k1+1,k2
            if (aggregate_array(i,j,k)==1) then
               ijk=ijkset(i,j,k)!Corner node #1
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i-1,j,k)!Corner node #2
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i-1,j-1,k)!Corner node #3
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i,j-1,k)!Corner node #4
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-1)
               ijk=ijkset(i,j,k-1)!Corner node #5
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i-1,j,k-1)!Corner node #6
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-1)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i-1,j-1,k-1)!Corner node #7
               xn(ijk)=stepi*(i-2)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-2)
               ijk=ijkset(i,j-1,k-1)!Corner node #8
               xn(ijk)=stepi*(i-1)
               yn(ijk)=stepj*(j-2)
               zn(ijk)=stepk*(k-2)
            end if
         end do
      end do
   end do
end if
deallocate(aggregate_array)

RETURN

END SUBROUTINE EXZONE
```

E THE SOURCE CODE FOR THE MACRO GENERATING PROGRAM

E.1 The main program

The main program consists of three main parts: initialisations, calls to subroutines that actually do the job, and then a summary part that writes output regarding the filling fraction and size distribution of the rocks stored in the macro file.

```
program macrofill

  use inputs
  use map_array_functions

  integer :: jj
  integer :: i,j,k,openstat,readstat
  real :: rockvol, density_cement
  integer, allocatable,dimension(:,:) :: output_distr
  real, allocatable, dimension(:,:) :: output_sieve_array
  integer :: rocknumber


  open(11,file='aggregatedata.dat',form='formatted',&
       iostat=openstat,status='old')
  if (openstat==0) then
     read(11,*,iostat=jj) macrofile
     readstat=readstat+jj
     read(11,*,iostat=jj) aggregate_name, density_aggregate
     readstat=readstat+jj
     read(11,*,iostat=jj) imn,imx,jmn,jmx,k_min,kmx
     readstat=readstat+jj
     read(11,*,iostat=jj) rockratio, safety
     readstat=readstat+jj
     read(11,'(I4)',iostat=jj) sieves
     readstat=readstat+jj
     allocate(sieve_array(1:2,1:sieves))
     read(11,*,iostat=jj) sieve_array
     readstat=readstat+jj
     close(11)
     allocate(aggregate_array(imn:imx,jmn:jmx,k_min:kmx))
     aggregate_array=0
     if (readstat==0) then
        call read_seed_data()
        allocate(distr(1:sieves))
        rockvol =rockratio*real((imx-imn)*(jmx-jmn)*(kmx-k_min))
        rocknumber=int(rockvol)
        allocate(sizes_of_rocks(1:rocknumber))
```

Now we have completed all initialisations. We first call the `find_rocks2` subroutine, which is identical to the one used in the `exzone` program, and thereafter the `place_aggregate_macro` subroutine, which is a modified version of the subroutine `place_aggregate`.

```
        call find_rocks2(rockvol, distr)
        rocknumber=int(distr(sieves-1))
        if (rocknumber<=1) then
```

```
          write(6,*) '$Rocknumber defaults to 1$'
          rocknumber=1
       end if
       call place_aggregate_macro(rocknumber-1)
       deallocate(sizes_of_rocks)
       deallocate(distr)
    end if
 end if
```

Now write the array of rock positions to the file aggpos.dat, exactly as in the `exzone` subroutine. Also finish the macro file by adding lines to it that will tell Autodyn to plot a material plot when the fill session is completed.

```
open(31,file='aggpos.dat',form='unformatted',access='direct',&
       iostat=openstat,recl=4)
  do i=1,imx-imn+1
     do j=1,jmx-jmn+1
        do k=1,kmx-k_min+1
           write(31,rec=k+(j-1)*(kmx-k_min+1)+(i-1)*(jmx-jmn+1)*(kmx-
k_min+1))&
                 aggregate_array(i+imn-1,j+jmn-1,k+k_min-1)
        end do
     end do
  end do
  close(31)
  open(31,file=macrofile,form='formatted',position='append'&
       ,iostat=openstat)
  if (openstat/=0) then
     write(6,*) 'Could not open the macrofile ',macrofile,'. '
  else
     write(31,'(a5)') '$View'
     write(31,'(a10)') '$Materials'
     write(31,'(a9)') '$Location'
     write(31,'(a2)') 'f6'
     write(31,'(a1)') 'N'
  end if
  deallocate(aggregate_array)
  close(31)
```

Lastly, write the output data to the screen. The code is exactly as in the `exzone` subroutine.

```
open(43,file='aggregate_output.dat',form='formatted')
  i=1
  do
     read(43,*,iostat=readstat)
     if (readstat/=0) exit
     i=i+1
  end do
  i=i-1
  close(43)
  allocate(output_distr(1:2,1:i))
  open(43,file='aggregate_output.dat',form='formatted')
  j=0
  allocate(output_sieve_array(1:2,1:sieves))
  output_sieve_array=sieve_array
  output_sieve_array(2,1:sieves)=0.0
  do k=1,i
     read(43,*,iostat=readstat) output_distr(1,k),output_distr(2,k)
     j=j+output_distr(1,k)
     do rocknumber=1,sieves
```

```
        if (sieve_array(1,rocknumber)<&
             output_distr(2,k)) then
           output_sieve_array(2,rocknumber)=&
                output_sieve_array(2,rocknumber)+real(output_distr(1,k))
        end if
        if (rocknumber==sieves .and. sieve_array(1,rocknumber)==&
             output_distr(2,k)) then
           output_sieve_array(2,rocknumber)=&
                output_sieve_array(2,rocknumber)+real(output_distr(1,k))
        end if
     end do
  end do
  close(43)
  do k=1,sieves
     output_sieve_array(2,k)=&
          output_sieve_array(2,k)/real(sum(output_distr(1,1:i)))
  end do
  write(6,*) 'Input aggregate ratio: ',rockratio
  write(6,*) '# of aggregate rocks:  ',i
  write(6,*) 'Output ratio:          ',real(j)/&
       (real((imx-imn)*(jmx-jmn)*(kmx-k_min)))
  write(6,'(3a20)') 'Mesh size','Input fraction','Output fraction'
  do j=1,sieves
     write(6,'(3(12x,f5.2,3x))') sieve_array(1,j),sieve_array(2,j),&
          output_sieve_array(2,j)
  end do
  deallocate(output_distr)
  deallocate(output_sieve_array)
  deallocate(sieve_array)
end program macrofill
```

E.2 The `fill_region_and_write` subroutine

There are probably better ways to realise the macro file needed for this, but the following code is a suggestion. Unfortunately, it is fairly long and perhaps a little confusing, possibly because we want to achieve a couple things simultaneously:

1. find the *i*-, *j*- and *k*-ranges for each new layer
2. make sure that the program does not loop outside the array boundaries.
3. update the `aggregate_array` for every cell that is filled with aggregate material
4. write the macro file

As usual, some initial code is needed, this time to read the name of the macro file, the aggregate material and the density of the aggregate material from the file *aggragedata.dat*. The initial lines are as follows, first the declarations:

```
subroutine fill_region_and_write(lower, center, upper, mindist)

  use map_array_functions

 implicit none

 real, intent(in) :: mindist
 integer, dimension(1:3), intent(in) :: lower, center, upper
! lower(x) give minimum boundary of the region.
! upper(x) give maximum boundary of the region.
```

```
! center(x) give center of the region.
 integer, dimension(1:3,1:3) :: ranges
 integer :: i,j,k, maxi, rangek1, rangej1,cornerpos,plane_no
 integer :: plane,rangek2, rangej2,size_this_rock
 logical :: skip
```

The necessary material data we need in order to write a macro file are stored in variables accessible via the `inputs` module.

The distance in *i*-, *j*- or *k*-directions of the cube boundaries from the cube center. I now define the region in which the cubical center is defined.

```
size_this_rock=0
 maxi=nint(2*mindist)
 ranges(1,1:3)=lower
 ranges(2,1:3)=upper
 ranges(3,1:3)=center
!ranges will contain the ijk-indices that describe the inner cube of each
rock.
 if (mod(maxi,2)==0) then
    maxi=nint(2.0*mindist/sqrt(3.0))
    if (mod(maxi,2)/=0) then
       maxi=maxi+1
    end if
 else
    maxi=nint(2.0*mindist/sqrt(3.0))
    if (mod(maxi,2)==0) then
       maxi=maxi+1
    end if
 end if
```

The if-loop above defines the length of the sides (the variable `maxi`) of the central cube. If the number of cells in the diameter is even, then the number of cells in `maxi` is rounded up to the nearest even integer. Otherwise, if the number of cells in the diameter is odd, the sides of the central cube is also odd. This is to ensure that there is an even number of cells remaining to be defined outside the central cube in the diameter. This is because planes are defined pairwise: if there is one plane padded on the cube on one side, then the same happen also on the opposite side (see Figure 4.2).

```
call lower_corner(ranges(3,1:3), int(real(maxi)/2.0), ranges(1,1:3))
 if (mod(maxi,2)==0) then
    call upper_corner(ranges(3,1:3), int(real(maxi)/2.0)-1, ranges(2,1:3))
 else
    call upper_corner(ranges(3,1:3), int(real(maxi)/2.0), ranges(2,1:3))
 end if
```

The subroutines `lower_corner` and `upper_corner` define the ijk-index of those corners, taking into account the possibility that we are outside the array boundaries. The if-branching controls the two different cases of even and odd side lengths. An odd number of cells means that there is a well defined center cell, and to find the corners we can simply add the distance `maxi`/2 to the *ijk*-index for the upper corner, and the subtract the same value to find the lower corner. For an even number of cells on each side we must add `maxi`/2-1.

The cubical central part is now defined in the map array with the following code, the first if-statement controls that the cubic region is well defined:

```
!First fill the rock in the aggregate_array
 if (ranges(2,1)-ranges(1,1)<=0 .or. ranges(2,2)-ranges(1,2)<=0 .or. &
      ranges(2,3)-ranges(1,3)<=0) then
!Do nothing since the ranges are invalid
 else
    do i=ranges(1,1)+1,ranges(2,1)
       do j=ranges(1,2)+1,ranges(2,2)
          do k=ranges(1,3)+1,ranges(2,3)
             aggregate_array(i,j,k)=1
             size_this_rock=size_this_rock+1
          end do
       end do
    end do
    !now start creating the rock, first the center cube
```

and then the macro file lines are written which defines the cube:

```
call write_macro_file(ranges(1,1),ranges(1,2),ranges(1,3),ranges(2,1),&
         ranges(2,2),ranges(2,3))
    plane_no=1
```

We are now ready to start padding the surfaces outside the cube, as explained previously. Firstly, we need a way to define the surfaces that are going to be padded. We do this by looking for a position for one of the corners, and creating a square of thickness one. The variable `plane_no` indicates the layer number, or stage number to use the terminology adopted in Figure 4.2. The middle figure has `plane_no=1`, while the right figure has `plane_no=2`. The variable `cornerpos` indicates how many elements away from the cubical corner the square begins. In the figure `cornerpos=1` in the middle and `cornerpos=3` in right figure.

```
    do
       cornerpos=1
       do
          if ((real(maxi)/2+real(plane_no))**2+&
               (real(maxi)/2.0-real(cornerpos))**2+&
               (real(maxi)/2.0-real(cornerpos))**2<=(mindist)**2) then
             exit
          else
             cornerpos=cornerpos+1
          end if
          if (cornerpos>nint(mindist)) exit
       end do
       if (cornerpos>nint(mindist)) exit
```

The value of cornerpos is found by requiring the distance between the cube center and the corner element be less than the sphere radius.
The next part of the code performs the padding now that we know the relative position of the square corner. The `select case` construct is not necessary here, but I thought it was

slightly easier to see what is going on with it, and it also makes it easier to skip any surfaces that extend beyond the subgrid boundaries.

Basically, I place a square on each of the six sides of the cube. Notice that the variables `rangej1`, `rangej2`, `rangek1` and `rangek2` have nothing to do with *i*, *j* or *k* indices, but changes meaning according to which of the six sides we are on. For example, if we pad a plane which is constant in the *k*-direction, then the range definition refers to the *i*- and *j*-direction.

```
do i=1,6
          skip=.false.
          select case(i)
          case (1)
             plane=ranges(1,1)-plane_no !Let imin be one less than in the
cube
             if (plane<imn) skip=.true.
             rangej1=ranges(1,2)+cornerpos
             rangek1=ranges(1,3)+cornerpos
             rangej2=ranges(2,2)-cornerpos
             rangek2=ranges(2,3)-cornerpos
             if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
             if (skip) then
             else
                do j=rangej1+1,rangej2
                   do k=rangek1+1,rangek2
                      aggregate_array(plane+1,j,k)=1
                      size_this_rock=size_this_rock+1
                   end do
                end do
                call
write_macro_file(plane,rangej1,rangek1,plane+1,rangej2,&
                     rangek2)
             end if
          case (2)
             plane=ranges(2,1)+plane_no
!Let imax be one greater than in the cube
             if (plane>imx) skip=.true.
             rangej1=ranges(1,2)+cornerpos
             rangek1=ranges(1,3)+cornerpos
             rangej2=ranges(2,2)-cornerpos
             rangek2=ranges(2,3)-cornerpos
             if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
             if (skip) then
             else
                do j=rangej1+1,rangej2
                   do k=rangek1+1,rangek2
                      aggregate_array(plane,j,k)=1
                      size_this_rock=size_this_rock+1
                   end do
                end do
                call write_macro_file(plane&
                     -1,rangej1,rangek1,plane,rangej2,&
                     rangek2)
             end if
          case (3)
             plane=ranges(1,2)-plane_no
!Let jmin be one less than in the cube
             if (plane<jmn) skip=.true.
             rangej1=ranges(1,1)+cornerpos
             rangek1=ranges(1,3)+cornerpos
             rangej2=ranges(2,1)-cornerpos
```

```
               rangek2=ranges(2,3)-cornerpos
               if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
               if (skip) then
               else
                  do j=rangej1+1,rangej2
                     do k=rangek1+1,rangek2
                        aggregate_array(j,plane+1,k)=1
                        size_this_rock=size_this_rock+1
                     end do
                  end do
                  call write_macro_file(rangej1,plane,rangek1,&
                       rangej2,plane+1,&
                       rangek2)
               end if
            case (4)
               plane=ranges(2,2)+plane_no
!Let jmax be one greater than in the cube
               if (plane>jmx) skip=.true.
               rangej1=ranges(1,1)+cornerpos
               rangek1=ranges(1,3)+cornerpos
               rangej2=ranges(2,1)-cornerpos
               rangek2=ranges(2,3)-cornerpos
               if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
               if (skip) then
               else
                  do j=rangej1+1,rangej2
                     do k=rangek1+1,rangek2
                        aggregate_array(j,plane,k)=1
                        size_this_rock=size_this_rock+1
                     end do
                  end do
                  call write_macro_file(rangej1,plane-&
                       1,rangek1,rangej2,plane,&
                       rangek2)
               end if
            case (5)
               plane=ranges(1,3)-plane_no
!Let kmin be one less than in the cube
               if (plane<k_min) skip=.true.
               rangej1=ranges(1,1)+cornerpos
               rangek1=ranges(1,2)+cornerpos
               rangej2=ranges(2,1)-cornerpos
               rangek2=ranges(2,2)-cornerpos
               if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
               if (skip) then
               else
                  do j=rangej1+1,rangej2
                     do k=rangek1+1,rangek2
                        aggregate_array(j,k,plane+1)=1
                        size_this_rock=size_this_rock+1
                     end do
                  end do
                  call write_macro_file(rangej1,rangek1,&
                       plane,rangej2,rangek2,&
                       plane+1)
               end if
            case (6)
               plane=ranges(2,3)+plane_no
!Let kmax be one greater than in the cube
               if (plane>kmx) skip=.true.
               rangej1=ranges(1,1)+cornerpos
```

```
          rangek1=ranges(1,2)+cornerpos
          rangej2=ranges(2,1)-cornerpos
          rangek2=ranges(2,2)-cornerpos
          if (rangej2-rangej1<=0 .or. rangek2-rangek1<=0) skip=.true.
          if (skip) then
          else
             do j=rangej1+1,rangej2
                do k=rangek1+1,rangek2
                   aggregate_array(j,k,plane)=1
                   size_this_rock=size_this_rock+1
                end do
             end do
             call write_macro_file(rangej1,rangek1,plane-1,rangej2,&
                  rangek2,plane)
          end if
       end select
    end do !End the select case loop.
```

We have now padded a plane on each side of the cube. We increase `plane_no` by one, and repeat the whole process until the `maxi+plane_no` is equal to or greater than the diameter of the rock.

```
plane_no=plane_no+1
       if ((2*plane_no+maxi)>nint(2.0*mindist)) exit
    end do
    open(13,file='aggregate_output.dat',form='formatted',position='append')
    write(13,*) size_this_rock,maxi+2*plane_no
    close(13)
 end if
 return
end subroutine fill_region_and_write
```

The file *aggregate_output.dat* contains the sizes of each of the rocks filled in this subroutine, and can later be accessed by other parts of the program (see the last lines in the main program code).

E.3 The place_aggregate_macro subroutine

This subroutine is very similar to the `place_aggregate` subroutine. The only difference is that rather than calling `the fill_region` subroutine, it calls the `fill_region_and_write` subroutine. That subroutine was documented in the previous section.

F VARIOUS SHORT SUBROUTINES

F.1 The modules

The module `map_array_functions` contains some short subroutines and a function, and also some declarations that are used in several different subroutines. Among the variables declared here is the `aggregate_array`.

```
module map_array_functions

  implicit none

  real, allocatable, dimension(:,:,:) :: aggregate_array
  real, allocatable, dimension(:) :: sizes_of_rocks
  integer :: imn, jmn, k_min, imx, jmx, kmx

  contains
  integer function neighbour_sum

  logical function joined_node

  subroutine lower_corner

  subroutine upper_corner

end module map_array_functions
```

The following module contains input variables. They are used in several different subroutines, and to avoid passing values around as formal parameters I have included them in this module. That makes subroutine calls much easier.

```
module inputs

  implicit none

  character(len=10) :: macrofile, aggregate_name, subgrid_name
  real :: density_aggregate, damage_level, rockratio
  real, allocatable, dimension(:,:) :: sieve_array
  real, allocatable, dimension(:) :: distr
  integer :: safety, sieves

end module inputs
```

F.1.1 integer function neighbour_sum

The function adds the 27 elements in a 3x3x3 subset of the aggregate array.

```
integer function neighbour_sum(posi,posj,posk)
!The function returns the sum of 27 elements in an integer array; those
!elements that form a 3x3x3 subset of the array centered on
(posi,posj,posk).
    implicit none
    integer, intent(in) :: posi, posj, posk
    integer :: ix, in, jx, jn, kx, kn

    ix=posi+1
    in=posi-1
    jx=posj+1
    jn=posj-1
    kx=posk+1
    kn=posk-1
    if (posi==imx) ix=posi
    if (posj==jmx) jx=posj
```

```
    if (posk==kmx) kx=posk
    if (posi==imn) in=posi
    if (posj==jmn) jn=posj
    if (posk==k_min) kn=posk
    neighbour_sum=sum(aggregate_array(in:ix,jn:jx,kn:kx))

    return
  end function neighbour_sum
```

F.1.2 logical function joined_node

The function returns `.true.` if one or more of the eight neighbouring cells that share the node (*posi,posj,posk*) contains a 0.

```
logical function joined_node(posi,posj,posk)

    implicit none

    integer, intent(in) :: posi, posj, posk

    integer :: ix, jx, kx

    ix=posi+1
    jx=posj+1
    kx=posk+1
    if (posi==imx) ix=posi
    if (posj==jmx) jx=posj
    if (posk==kmx) kx=posk
    if (sum(aggregate_array(posi:ix,posj:jx,posk:kx))<8) then
       joined_node=.true.
    else
       joined_node=.false.
    end if

    return
  end function joined_node
```

F.1.3 subroutine lower_corner

This subroutine finds the lower corner in a cubical (in *ijk*-space) subset of the aggregate array. It checks if any of the three indices run beyond the array extent.

```
subroutine lower_corner(center_indices, rad, corner_indices)
!When looping occurs outside the borders of an array, any program will
crash.
!This routine checks that the lower corner in a subset is inside the array
!boundaries.
    implicit none
    integer, intent(in) :: rad
    integer, dimension(1:3), intent(in) :: center_indices
    integer, dimension(1:3), intent(out) :: corner_indices
```

```
    if (center_indices(1)-rad<imn) then
       corner_indices(1)=imn
    else
       corner_indices(1)=center_indices(1)-rad
    end if
    if (center_indices(2)-rad<jmn) then
       corner_indices(2)=jmn
    else
       corner_indices(2)=center_indices(2)-rad
    end if
    if (center_indices(3)-rad<k_min) then
       corner_indices(3)=k_min
    else
       corner_indices(3)=center_indices(3)-rad
    end if

    return
  end subroutine lower_corner
```

F.1.4 subroutine upper_corner

This subroutine performs the same action as the previous one, but in order to find the upper corner of the same cubical subset.

```
subroutine upper_corner(center_indices,rad, corner_indices)
    !This is identical to the above, only for the maximum limits.
    implicit none
    integer, intent(in) :: rad
    integer, dimension(1:3), intent(in) :: center_indices
    integer, dimension(1:3), intent(out) :: corner_indices

    if (center_indices(1)+rad>imx) then
       corner_indices(1)=imx
    else
       corner_indices(1)=rad+center_indices(1)
    end if
    if (center_indices(2)+rad>jmx) then
       corner_indices(2)=jmx
    else
       corner_indices(2)=rad+center_indices(2)
    end if
    if (center_indices(3)+rad>kmx) then
       corner_indices(3)=kmx
    else
       corner_indices(3)=rad+center_indices(3)
    end if

    return
  end subroutine upper_corner
```

F.2 The subroutine `check_region`

The subroutine finds if the region with diagonally opposite corners `lower` and `upper` is free of any rock elements (i.e. elements with value 1).

```
subroutine check_region(lower, center, upper, mindist,free)

  use map_array_functions, only: nn_array=>aggregate_array

  implicit none
  integer, intent(in) ::  mindist
  integer, dimension(1:3), intent(in) :: lower, center, upper
  ! low_corner(x) give minimum boundary of the region.
  ! upper(x) give maximum boundary of the region.
  ! center(x) give center of the region.
  logical, intent(out) :: free
  integer :: i,j,k

  free=.true.
  do i=lower(1),upper(1)
     do j=lower(2),upper(2)
        do k=lower(3),upper(3)
           if ((i-center(1))**2+(j-center(2))**2+&
                (k-center(3))**2<=mindist**2 .and. &
                nn_array(i,j,k)==1) then
              free=.false.
           end if
        end do
     end do
  end do
  return
end subroutine check_region
```

F.3 The subroutine `fill_region`

The subroutine fills a region defined with lower and upper corner given by `lower` and `upper`, respectively, with a spherical rock. Its diameter is `mindist`.

```
subroutine fill_region(lower, center, upper, mindist)

  use map_array_functions, only : nn_array=>aggregate_array

 implicit none
 integer, intent(in) :: mindist
 integer, dimension(1:3), intent(in) :: lower, center, upper
! lower(x) give minimum boundary of the region.
! upper(x) give maximum boundary of the region.
! center(x) give center of the region.
 integer :: volume, i, j, k

 volume=0
 do i=lower(1),upper(1)
    do j=lower(2),upper(2)
       do k=lower(3),upper(3)
          if ((i-center(1))**2+(j-center(2))**2+&
               (k-center(3))**2<=mindist**2) then
             nn_array(i,j,k)=1
             volume=volume+1
          end if
       end do
    end do
 end do
```

```
 open(13,file='aggregate_output.dat',form='formatted',position='append')
 write(13,*) volume,2*mindist+1
 close(13)
 return
end subroutine fill_region
```

F.4 Subroutine `write_macro_file`

The subroutine write macro file lines for a fill command in the block (`in,jn,kn`) to (`ix,jx,kx`). The variable `name` is the material name, `macrofile` is the name of the macro file, and `dens` is the density that the material should be filled with.

```
subroutine write_macro_file(in,jn,kn,ix,jx,kx)

  use inputs, only : name=>aggregate_name, dens=>density_aggregate,
macrofile

  implicit none

  integer, intent(in) :: in,jn,kn,ix,jx,kx
  integer :: i

  open(91,file=macrofile,form='formatted',position='append')
  write(91,'(A6)') '$Block'
  write(91,'(A1)') '%'
  write(91,*) in
  write(91,*) ix
  write(91,*) jn
  write(91,*) jx
  write(91,*) kn
  write(91,*) kx
  write(91,'(A10)') name
  write(91,'(A1)') '%'
  write(91,*) dens
  do i=1,5
     write(91,*) 0.0
  end do
  write(91,'(A9)') 'Spherical'
  close(91)
  return
end subroutine write_macro_file
```

G THE EXVAL SUBROUTINE

I mentioned in the introduction that in 3D I have not yet been able to fill a subgrid with two materials from the exval subroutine. Even though it does not work satisfactorily, I have written a suggestion for a program that would work in 2D. If someone finds a way to reset the material from the exval subroutine, then only a few lines need be modified in this program to make it work.

```
SUBROUTINE EXVAL (NS,I,J,K,IJK,MATI,NP,RHOI,RREF,SIEI,UXI,UYI,UZI,URI)
```

```
  USE kindef
  USE bnddef

  use subdef
  use mdgrid
  use map_array_functions
  use inputs

  IMPLICIT NONE

  INTEGER (INT4) ::    IJK,      I,      J,      K,   MATI,     NP
  INTEGER (INT4) ::     NS
  REAL (REAL8)   ::   RHOI,   RREF,   SIEI,    URI,    UXI,    UYI,   UZI

  character :: yes_or_no, read_old
  integer(int4) :: filestat, subnumber, matnumber, actual_fill
  integer(int4) :: rocknumber,  ii, jj, kk, aggpos_element
  real(real8) :: rockratio_out
  real :: rockvol
  logical :: no_aggpos_file


  var01(ijk)=var01(ijk)
  if (i==2 .and. j==2 .and. k==2) then
     imn=1
     jmn=1
     k_min=1
     imx=imax
     jmx=jmax
     kmx=kmax
     yes_or_no='N'
!         call getyon(yes_or_no, &
!              '$Do you wish to fill this subgrid with two materials$')
```

To activate the program, uncomment the two lines above.

```
     if (yes_or_no=='Y') then
open(11,file='aggregatedata.dat',form='formatted',&
             iostat=filestat)
        if (filestat==0) then
           read(11,*,iostat=jj)
           read(11,*,iostat=jj) aggregate_name, density_aggregate
           filestat=filestat+jj
           matnumber=0
           do ii=1,nummat
              if (nammat(ii)==aggregate_name) then
                 matnumber=ii
                 exit
              end if
           end do
           read(11,*,iostat=jj) ii,ii,ii,ii,ii,ii,subgrid_name
           filestat=filestat+jj
           read(11,*,iostat=jj) rockratio, safety
           filestat=filestat+jj
           read(11,'(I4)',iostat=jj) sieves
           filestat=filestat+jj
           allocate(sieve_array(1:2,1:sieves))
           read(11,*,iostat=jj) sieve_array
```

```
filestat=filestat+jj
close(11)
write(6,*) ' '
write(6,*) 'Aggregate material: ',aggregate_name
write(6,*) 'Target subgrid: ', &
     namsub(ns)
write(6,*) rockratio
write(6,*) sieves
write(6,*) sieve_array
call getyon(read_old,'$Use existing aggregate array file$')
if (read_old=='Y') then
   no_aggpos_file=.false.
   inquire(iolength=jj) jj
   open(34,file='aggpos.dat',form='unformatted',&
        access='direct',recl=jj,&
        status='old',iostat=actual_fill)
   if (actual_fill/=0) then
      call messag('$Could not open file aggpos.dat.$')
      no_aggpos_file=.true.
   end if
else
   if (filestat==0 .and. matnumber/=0) then
      call read_seed_data()
      allocate(distr(1:sieves))
      rockvol =rockratio*real(imax*jmax&
           *kmax)
      rocknumber=int(rockvol)
      allocate(sizes_of_rocks(1:rocknumber))
      allocate(aggregate_array(1:imax,&
           1:jmax,1:kmax),stat=filestat)
      if (filestat/=0) write(6,*) 'Allocation error ',filestat
      call find_rocks2(rockvol, distr)
      rocknumber=distr(sieves-1)
      call place_aggregate(rocknumber)
      write(6,*) 'Aggregate database completed'
      inquire(iolength=jj) jj
      open(31,file='aggpos.dat',form='unformatted',&
           access='direct',recl=jj,&
           iostat=actual_fill)
      if (actual_fill/=0) then
         call messag('$Could not open file aggpos.dat.$')
      else
         do ii=1,imax
            do jj=1,jmax
               do kk=1,kmax
                  write(31,rec=kk+(jj-1)*(kmax)+&
                       (ii-1)*(jmax)*(kmax))&
                       aggregate_array(ii,jj,kk)
               end do
            end do
         end do
         close(31)
      end if
      deallocate(aggregate_array)
      deallocate(sizes_of_rocks)
      deallocate(distr)
      deallocate(sieve_array)
      inquire(iolength=jj) jj
      open(34,file='aggpos.dat',form='unformatted',&
           access='direct',recl=jj,&
           status='old')
```

```
            end if
         end if
      end if
      actual_fill=0
   else
      call messag('$This subgrid will only be filled with one '//&
           'material.$')
   end if
end if
if (i==imax .and. j==jmax .and. k==kmax) then
   if (yes_or_no=='Y') then
      do ii=1,imax
         do jj=1,jmax
            do kk=1,kmax
               if (no_aggpos_file) then
                  aggpos_element=0
               else
                  read(34,rec=kk+(jj-1)*(kmax)+(ii-1)*(jmax)*(kmax),&
                       iostat=filestat)&
                       aggpos_element
               end if
               if (aggpos_element==1) then
                  npkmn(ijkset(ii,jj,kk))=matnumber
               end if
            end do
         end do
      end do
   end if
   close(34)
   rockratio_out=REAL(actual_fill)/REAL(numi(ns)*&
        numj(ns)*numk(ns))
   open(22,file='fillingdata.txt',form&
        ='formatted',position='append')
   write(22,*) '-----------------------------------------'
   write(22,*) 'Subgrid name:                       ', &
        namsub(ns)
   write(22,*) '# cells filled with ', aggregate_name, ': ',&
        actual_fill
   write(22,*) 'Desired fraction:                   ', rockratio
   write(22,*) 'Actual fraction:                    ', &
        rockratio_out
   write(22,*) '-----------------------------------------'
   close(22)
end if !The last cell

RETURN

END SUBROUTINE EXVAL
```

H THE MAKEFILE

A makefile has been written to make it easier to compile the program. As it stands, it must be used in a directory called Aggregate, with the path ~/autodyn/3dv42/usrsub/Aggregate (in fact, the name of the directory is not important in this case, but level in the directory hierarchy is). The syntax for using the make utility is

make <*keyword*>

The Makefile accepts the following keywords:

- Nothing or auto_aggregate: compiles the .f90 files aggregate.f90, random.f90 and 2mat.f90, and links them with the Autodyn object file and libraries/modules.
- adslav3: compiles the same files, and links them to make the slave processor for 3D parallell processing. I can not see how this is necessary to do at present, but if it would be possible to achieve a two-material filling from the exedit subroutine as in 2D, then it might be necessary to make a slave process.
- generate_macro: compiles and links the files used to create the generate_macro program.
- example: generates examples of the seed.dat-file and the aggregatedata.dat-file.
- cleandir: removes all .mod- and .o-files in the directory, as well as the executable files.
- clean: the same as clean, but it also removes generated files from the ~/autodyn/3dv42/bin/ directory.
- cleanzip: the same as clean, but in addition it gzips the .f90 files.

The Makefile can be rewritten, of course, to allow for different directory structures and different filenames and output program names. To aid in such rewriting, the most important definitions in the Makefile are written in the top lines.

Here are the lines in the Makefile:

```
.SUFFIXES: .f90 .o .a

PATH1=      /user/aao/autodyn/3dv42
PROGRAM= auto_aggregate
SLAVE= adslav3
F90FILE= aggregate
#Declaring filenames etc for the macrofile generating program
MACROPROG= generate_macro
MACROSRC= random.f90 2mat.f90 macrofill.f90
MACROOBJ= ${MACROSRC:.f90=.o}
DATADIR= $(PATH1)/data/

FILES1= $(PATH1)/usrsub/admain3.o $(PATH1)/usrsub/autodyn3.a
ADSLAVES= $(PATH1)/usrsub/adslav3.o $(PATH1)/usrsub/autodyn3.a
FILES2= random.f90 2mat.f90 $(F90FILE).f90
OBJFILES= ${FILES2:.f90=.o}
GKSDIR= $(gksdir)
PVM_DIR= $(PVM_ROOT)/lib/HPPA/
FLAGS= -L$(GKSDIR)  -lgksflb -lgksw5300 -lgksw1900 -lgksswiss \
       -lgksgksm -lgksmsc -L$(PVM_ROOT)/libfpvm/HPPA -lX11 -lm
FLAGS2= +save +noshared +O2 +DA2.0 -I $(PATH1)/usrsub
FLAGS3= -Wl,-ashared -lnsl -ldld -I $(PATH1)/usrsub
PVM_LIB1= $(PVM_ROOT)/libfpvm/HPPA/libfpvm3.a
PVM_LIB2= $(PVM_DIR)libpvm3.a $(PVM_DIR)libgpvm3.a

.f90.o : $(FILES2)
      f90 -c $< $(FLAGS2)

$(PROGRAM) : $(OBJFILES)
      f90  -o $(PROGRAM) $(FLAGS3) $(OBJFILES) $(FILES1) $(FLAGS) \
```

```
      $(PVM_LIB1) $(PVM_LIB2)
      cp $(PROGRAM) $(PATH1)/bin/.

$(SLAVE) : $(OBJFILES)
      f90  -o $(SLAVE) $(FLAGS3) $(OBJFILES) $(ADSLAVES) $(FLAGS) \
      $(PVM_LIB1) $(PVM_LIB2)
      cp $(SLAVE) $(PATH1)/bin/.

$(MACROOBJ) : $(MACROSRC)
      f90 -c $< $(FLAGS2)

$(MACROPROG) : $(MACROOBJ)
      f90 -o $(MACROPROG) $(MACROOBJ) $(FLAGS3)
      cp $(MACROPROG) $(PATH1)/bin/.
      echo rm -f aggregate_output.dat > macrogen.sh
      echo $(MACROPROG) >> macrogen.sh
      echo mv $(MACROFILE) $(PATH1)/data/$(DATADIR)/. >>\
      macrogen.sh
      chmod +x macrogen.sh
      mv macrogen.sh $(PATH1)/bin/.

input_example_aggregate : example_gen.f90
      f90 -o input_example_aggregate example_gen.f90

example : input_example_aggregate
      input_example_aggregate
      mv seed.dat aggregatedata.dat $(PATH1)/bin/.

cleandir :
      rm -f $(PROGRAM) $(SLAVE) *.mod *.o input_example_aggregate
      rm -f *~ $(MACROPROG)

clean : cleandir
      rm -f $(PATH1)/bin/$(SLAVE) $(PATH1)/bin/$(PROGRAM) \
      $(PATH1)/bin/seed.dat $(PATH1)/bin/aggregatedata.dat
      rm -f $(PATH1)/bin/macrogen.sh $(PATH1)/bin/$(MACROPROG)

cleanzip : clean
      gzip $(FILES2)
```

I THE 2D VERSION

In 2D, it is possible to use the `exval` user subroutine to set the material number in each cell through the `nkpmn`-array. This has the advantage of allowing us to access this array from other user subroutines. In particular, if we reset the material number in the `exedit` subroutine then we are able to run a large number of simulations with different realisations of the aggregate, and thereby obtain better statistics on the effect of the aggregate. This is unfortunately not possible to achieve in Autodyn-3D. A workaround suggested by Century Dynamics is to use a macro file to fill the subgrid interactively, as has been described in Chapter 4.

Generally, the differences are small between the 3D version and the 2D version. From a technical point of view there is a difference in the programming structure. The 2D version is programmed without the use of modules, and consequently all variables used in subroutines

must be passed as formal parameters. In addition, we can fill any subgrid in 2D from the `exval` subroutine in the present implementation. It is even possible to fill from `exedit` by accessing grid variables from the `putnpks` subroutine (see the Autodyn User Subroutine Tutorial for details).

Despite the small differences, the basic idea remains the same in that a map array is created that contains information on which cells are filled with an aggregate material. We leave most of the code uncommented.

I.1 The 2D filling subroutine

The subroutine resembles the `place_aggregate` subroutine used in 3D.

Initialisations and declarations:

```
subroutine place_rocks_2D(rcknr, imx, jmx, separation, sizearr, rock_or_not)

!Without due care, some of the loops in this subroutine will go outside the
!range of the array. What has been done, is to let the array have a frame of
!elements arounf the main part, which are not used for anything. The looping
!never extends beyond the range in this way. However, it means we have to be
!careful about what values are sent with this routine from the outside:
notice
!that imx and jmx are used to define the array dimensions as well as
defining
!the range from which the positions of rocks are drawn. These problems do
not
!occur in 3D because there I am specifically checking the ranges before
!all loops.
  implicit none
  real, dimension(1:*), intent(in) :: sizearr
  integer, intent(in) :: rcknr, imx, jmx, separation
  integer, dimension(0:imx+1,0:jmx+1), intent(out) :: rock_or_not
  integer :: pos1, pos2, telle1, telle2, telle3, sizeint, cell_no
  integer :: unplaced_rocks
  real :: rock_vol
  logical :: occupied_space

  unplaced_rocks=0
  rock_or_not=0
```

Loop through every rock:

```
  do telle1=1,rcknr
```

The first step is to find the radius of the current rock in number of cells. Thereafter, draw a random position in the subgrid.

```
     rock_vol=sizearr(telle1)
!The sizes in the array are radii.
     sizeint=nint(rock_vol)
     call random_uniformint(1+sizeint+separation,imx-sizeint-
separation,pos1)
```

```
      call random_uniformint(1+sizeint+separation,jmx-sizeint-separalife
insution,pos2)
```

Now check if the drawn position is free, that is, make sure there are no overlap with existing rocks. If there is overlap, then loop through the entire subgrid looking for an available spot.

```
      do cell_no=1,imx*jmx
         occupied_space=.false.
         do telle2=pos1-sizeint-separation,pos1+sizeint+separation
            do telle3=pos2-sizeint-separation,pos2+sizeint+separation
               if ((telle2-pos1)**2+(telle3-
pos2)**2<=(sizeint+separation)**2&
                    .and. rock_or_not(telle2,telle3)==1) then
                  occupied_space=.true.
               end if
            end do
         end do
         if (occupied_space) then
            pos1=pos1+1
            if (pos1==imx-sizeint) then
               pos1=1+sizeint
               pos2=pos2+1
               if (pos2==jmx-sizeint) then
                  pos2=1+sizeint
               end if
            end if
            if (cell_no==imx*jmx) then
```

If we ever come this far, there will not be any room available for this rock. If so, write a message to the file unplaced.log, and a warning message to the screen.

```
               unplaced_rocks=unplaced_rocks+1
               if (unplaced_rocks==1) then
                  open(21,file='unplaced.log',form='formatted')
                  write(21,'(a18,i4)') 'No place for rock ', &
                       telle1
                  close(21)
                  write(6,*) 'Warning: some rocks can not be placed. '
                  write(6,*) 'See the file "unplaced.log" for details.'
               else
                  open(21,file='unplaced.log',form='formatted',&
                       position='append')
                  write(21,'(a18,i4)') 'No place for rock ', &
                       telle1
                  close(21)
               end if
            end if
         else
```

On the other hand, if the original position was available, or a different position has been found, then fill the aggregate array called `rock_or_not` with this rock in the relevant elements.

```
            if (sizeint>0) then
               do telle2=pos1-sizeint,pos1+sizeint
                  do telle3=pos2-sizeint,pos2+sizeint
                     if ((telle2-pos1)**2+(telle3-pos2)**2<=(sizeint)**2)
then
                        rock_or_not(telle2,telle3)=1
                     end if
```

```
                end do
             end do
          end if
          exit
       end if
    end do
 end do
 open(21,file='unplaced.log',form='formatted',position='append')
 write(21,'(a9,i4,a20)') 'In total ', unplaced_rocks, &
      'could not be placed.'
 close(21)
 return
end subroutine place_rocks_2D
```

I.2 The aggregatedata.dat file

The first lines in the input file looks very different though. The following file is a typical example:

```
PEBBLES
TARGET
1
0.2
4
8.0 0.0
6.0 0.053
4.0 0.56
2.0 1.0
```

The information in each line is as follows:

1. Aggregate material name
2. Name of the subgrid to be filled with two materials.
3. Wrap-up cycle number for `exedit`.
4. Filling fraction
5. This line and the following lines are the sieve curve data, exactly as in 3D.

The program also requires a seed.dat file which looks exactly as in 3D.

I.3 The 2D Makefile

The Makefile in 2D has some keywords, just as the 3D Makefile:

- Nothing or auto_aggregate: compiles and links the necessary files to generate the modified Autodyn executable.
- example: creates example versions of the input files
- cleandir: removes the object files and executables in the current directory.

- clean: the same as above, but also removes executables from the ~/autodyn/2dv42/bin/ directory.

The Makefile looks like this:

```
.SUFFIXES: .f90 .o .a

PROGRAM= auto_aggregate
#Name of output program
F90FILE= aggregate2D.f90
#Name of .f90 sourcecode
PATH1=      /user/aao/autodyn/2dv42
MODPATH= $(PATH1)/usrsub
#Path to Autodyn modules.
BINDIR= $(PATH1)/bin
#Path to Autodyn bin directory

FILES1= $(MODPATH)/admain2.o $(MODPATH)/autodyn2.a
FILES2= random.f90 subroutines.f90 $(F90FILE)
FILES3=     ${FILES2:.f90=.o}

GKSDIR= /user/aao/autodyn/gks
FLAGS= -L$(GKSDIR) -lgksflb -lgksw5300 \
      -lgksw1900 -lgkswiss -lgksgksm -lgksmsc \
      -L$(PVM_ROOT)/libfpvm/HPPA -lX11 -lm
FLAGS2= +save +noshared +O2 +DA2.0 -I $(MODPATH)/
FLAGS3= -Wl,-ashared -lnsl -ldld -I $(MODPATH)/

.f90.o : $(FILES2)
      f90 -c $< $(FLAGS2)

$(PROGRAM) : $(FILES3)
      f90  -o $(PROGRAM) $(FLAGS3) $(FILES3) $(FILES1) $(FLAGS) $(FLAGS2)
      mv $(PROGRAM) $(PATH1)/bin/.

example_inputs : example2D.f90
      f90 -o example_inputs $?

example : example_inputs
      example_inputs
      mv seed.dat $(PATH1)/bin/.
      mv aggregatedata.dat $(PATH1)/bin/.

cleandir :
      rm -f *.o *~ $(PROGRAM)

clean : cleandir
      rm -f $(PATH1)/bin/$(PROGRAM)
      rm -f $(PATH1)/bin/seed.dat
      rm -f $(PATH1)/bin/aggregatedata.dat
```

DISTRIBUTION LIST

FFIBM **Dato:** 21. juli 2003

RAPPORTTYPE (KRYSS AV)	RAPPORT NR.	REFERANSE	RAPPORTENS DATO
X RAPP ☐ NOTAT ☐ RR	2003/02057	FFIBM/766/130	21. juli 2003
RAPPORTENS BESKYTTELSESGRAD		ANTALL TRYKTE UTSTEDT	ANTALL SIDER
Unclassified		21	55
RAPPORTENS TITTEL		FORFATTER(E)	
MODELLING OF GRANULAR MATERIALS IN THE AUTODYN HYDROCODE		OLSEN Åge Andreas Falnes, TELAND Jan Arild	
FORDELING GODKJENT AV FORSKNINGSSJEF		FORDELING GODKJENT AV AVDELINGSSJEF:	
Bjarne Haugstad		Jan Ivar Botnan	

EKSTERN FORDELING

ANTALL	EKS NR	TIL
1		Eirik Svinsås Paulus Plass 5 0554 Oslo
1		Åge Andreas Falnes Olsen Fysisk institutt Postboks 1048 – Blindern 0316 Oslo
1		Otto Munthe Anker Zemer Engineering Grindbakken 1 0764 Oslo
1		Jim Sheridan DSTL Missiles and Countermeasures Dept. Room G056, Building A2 Ively Road Farnborough, Hants., Gu14 0LX England
1		Jaap Weerheijm TNO Lange Kleiweg 137 P. O. Box 45 2280 AA Rijswijk Nederland

INTERN FORDELING

ANTALL	EKS NR	TIL
9		FFI-Bibl
1		Adm direktør/stabssjef
1		FFIE
1		FFISYS
1		FFIBM
2		Jan Arild Teland, FFIBM
1		John F Moxnes, FFIBM
		Elektronisk fordeling: FFI-veven
		Lars Kvifte (LKv), FFIBM Bjarne Haugstad (BjH), FFIBM Svein Rollvik (SRo), FFIS Ove Dullum (OSD), FFIBM Henrik Sjøl, (HSj) ,FFIBM

Benytt ny side om nødvendig.