FFI RAPPORT

RAPID AUTODYN-3D PENETRATION SIMULATIONS USING A VIRTUAL TARGET

OLSEN Åge Andreas Falnes, TELAND Jan Arild

FFI/RAPPORT-2002/00575

FFIBM/766/130

Approved
Kjeller 27. August 2002

Bjarne Haugstad
Director of Research

RAPID AUTODYN-3D PENETRATION SIMULATIONS USING A VIRTUAL TARGET

OLSEN Åge Andreas Falnes, TELAND Jan Arild

FFI/RAPPORT-2002/00575

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-2002/00575	2) SECURITY CLASSIFICATION UNCLASSIFIED	3) NUMBER OF PAGES 46
1a) PROJECT REFERENCE FFIBM/766/130	2a) DECLASSIFICATION/DOWNGRADING SCHEDULE -	

4) TITLE

RAPID AUTODYN-3D PENETRATION SIMULATIONS USING A VIRTUAL TARGET

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)	Autodyn	Autodyn
b)	Cavity expansion	Hulromsekspansjon
c)	User subroutines	Brukersubrutiner
d)	Virtual target	Virtuelt mål
e)		

THESAURUS REFERENCE:

8) ABSTRACT

By taking advantage of the possibilities for developing user subroutines in Autodyn-3D, we implement a method that uses a virtual target to decrease runtimes for 3D problems by several orders of magnitude. The new method uses results from analytical penetration theory based on cavity expansion to estimate the stress on the projectile. This eliminates the need to model the target explicitly. In this way we combine the Autodyn user interface and computational algorithms with analytical theory to create a very powerful tool.

9) DATE	AUTHORIZED BY This page only	POSITION
27. August 2002	Bjarne Haugstad	Director of Research

ISBN-82-464-0762-7

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE
(when data entered)

CONTENTS

		Page
1	INTRODUCTION	7
2	PROJECTILE DEFINITION	8
3	TARGET DEFINITION	9
3.1	Input from the Autodyn menu system	9
3.2	Matlab preprocessor	10
3.2.1	Target material model	10
3.2.2	Target geometry	11
3.2.3	Wrap-up condition	12
3.3	Target orientation	13
4	THE MODIFIED AUTODYN PROGRAM	13
4.1	An overview of the program	13
4.2	The various subtasks	15
4.2.1	Normal vector	15
4.2.2	Normal velocity	15
4.2.3	Distance to a free surface	15
4.2.4	Cell position relative to target	17
4.2.5	Pressure in a semi-infinite target	17
4.2.6	Decay function	17
4.2.7	Final pressure	18
5	COMPARISON WITH EXPERIMENTS AND OTHER METHODS	19
5.1	Comparison with semi-analytic expressions for normal impact	19
5.2	Oblique impact	20
5.3	Perforation	23
6	CONCLUSION AND FURTHER WORK	24
A	SOURCE CODE	25
A.1	Subroutine `exstr`	25
A.2	Subroutines `exedit`, `exsave` and `exload`	28
A.3	The subroutine `exval`	32
A.4	Sundry modules	34
A.5	Module maths_funcs2	35
A.5.1	real function surface_reduction	36
A.5.2	subroutine unit_normal	36
A.5.3	subroutine cube	37
A.5.4	subroutine perforation	38
A.5.5	subroutine cylinder	39

A.5.6 subroutine sphere 40
A.5.7 real function normal_component 42
A.5.8 real function surface_area 42

A.6 The pressure subroutine 42

B COMPILATION AND LINKING 43

B.1 Makefile commands 43

B.2 The Makefile 44

Distribution list 46

RAPID AUTODYN-3D PENETRATION SIMULATIONS USING A VIRTUAL TARGET

1 INTRODUCTION

3D hydrocode simulations of impact problems are extremely time consuming. Typical runtimes for such simulations in Autodyn range from a couple of days to several weeks on our current server (750 MHz processors) at FFI. The parallellisation feature in the most recent version of Autodyn seems to reduce the problem with long runtimes. However, for parameter sensitivity studies there is hardly any difference between running distinct simulations in parallell on different processors, or running distinct simulations sequentially in parallell on several processors. A tool that gives good results in a shorter time is therefore very much desirable.

Recently, Warren and Poormon (1) presented a scheme which combines hydrocode simulations with analytical expressions. In their approach, only the projectile was simulated using a finite element code, whereas the target response was found from analytical expressions and were implemented in the simulation as a pressure boundary condition on the projectile surface (see Figure 1.1). In addition to reducing the number of cells in the simulations, this approach also eliminates the need for a time consuming interaction logic between the projectile and the target.

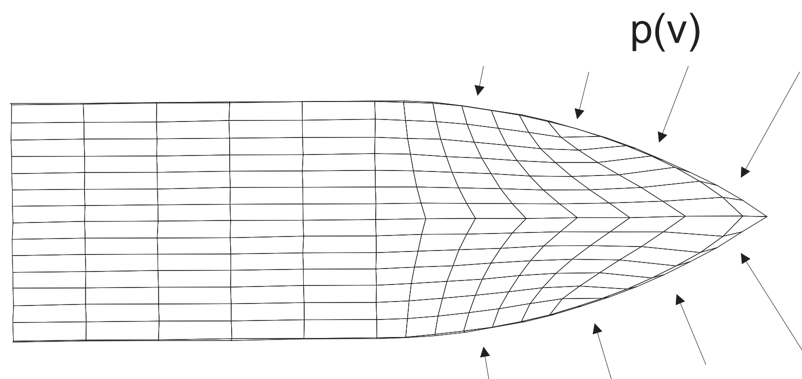

Figure 1.1 The projectile is modelled using a finite element mesh, whereas the target is modelled as a pressure boundary on the projectile surface elements.

Building on (1), we have implemented a similar scheme in the hydrocode Autodyn (2,3). User subroutines were programmed in Fortran 90 to define the pressure boundary condition, and

were linked with the Autodyn code through the standard Fortran 90 linker. In this way we retained the Autodyn user interface, which saved us a significant amount of work compared with creating our own hydrocode from scratch.

The new approach depends on the penetration model based on cavity expansion theory (4). This theory is valid for undeformed penetrators, which means that in principle the new approach is not applicable if the penetrator deforms significantly during the penetration process. However, it is believed to work well even for situastions when the projectile body bends, as long as there is no deformation of the nose.

This report explains how to use the extended Autodyn version as well as documenting the implemented user subroutines. It is assumed that the reader has some familiarity with both Fortran 90 and Autodyn. An elementary textbook may be useful as a reference for Fortran 90 illiterate readers.

2 PROJECTILE DEFINITION

After having started the extended Autodyn executable, the first step is to define the projectile mesh, which is most conveniently done using the Lagrangian processor. The projectile can be defined using two separate subgrids for the body and the nose, but it is quicker to use only one subgrid, and restrict the *ijk*-range during nose definition. This also makes it easier to later apply the pressure boundary condition to the surface cells, as one then gets away with only five boundary-commands instead of nine for two subgrids.

The most efficient way to define the projectile is thus the following:

- Define the *ijk*-range in the normal way.
- Enter the `Zoning—ijk-range` command and redefine the *ijk*-range so that only the upper *k*-range is used for zoning.
- Define the nose geometry within the redefined *ijk*-range.
- Redefine the *ijk*-range to include only the range that was not included in the nose definition.
- Enter the `Zoning—Generate—Block` command, and define the projectile body.

After having defined the projectile subgrid, it only remains to fill it with some kind of material. This is done in the usual way, except that the user is also prompted the following question: "*Use actual radius?*".

This refers to the projectile radius which is a factor in the analytical calculation. A more detailed explanation will be given later, but basically reponding "no" means the radius of the main body is used in the calculations, even at the nose, whereas "yes" implies that the smaller radius at the projectile nose is accounted for (i.e. a radius is calculated for each cell).

The radius is stored in the user variable `var01`, which before execution should be initialised from the Autodyn menu Global—Options—UserVar. It is *very important* that the user variable is assigned a non-zero value before the problem is run. If a *.000 file is loaded and ran without entering the fill session to initialise `var01`, the variable will be assigned the value 0. The calculations of distance involve division by this number, which will result in NaN (not-a-number). The most likely outcome is then that the calculation apparently proceeds normally, but without boundary effects being calculated. The result is thus a calculation for a semi-infinite target.

3 TARGET DEFINITION

As mentioned, the target is not modelled explicitly in the new approach, but accounted for through an analytical model. In this chapter we explain how to provide input for this analytical model.

3.1 Input from the Autodyn menu system

The target is modelled through a so-called user stress boundary condition, which is defined by selecting: Global – Boundary. As usual, the name of the boundary condition must first be specified. However, unlike in normal Autodyn this name is not arbitrary as it will define the target geometry. One of the reserved names shown in Table 3.1 must therefore be used.

Table 3.1 Boundary condition names and reserved user inputs.

Target geometry	Boundary condition name
Prism	CUBE
Cylindrical target	CYLINDER
Spherical target	SPHERE
Infinite slab	PERFORATE
Semi-infinite target	Arbitrary (except for the above names)

Autodyn then asks for the type of boundary condition, where *stress* and then *user* must be selected, after which Autodyn prompts for the values of five constants, named RBC(1) through RBC(5). The constants RBC(1) through RBC(3) define the stress on a cell as a function of velocity, while the two last constants are not used here. More details are given in Chapter 4.2.5.

After having defined the boundary conditions, the next step is to asssign them to the projectile nose surface cells. This is done with the menu choice Subgrid – Boundary. The user can for instance define one or more planes in the *ijk*-space. An example is shown in Figure 3.1 for an ogive subgrid, where the boundary condition ”PERFORATE” has been defined on the five planes $i=1$, $j=1$, $i=i_{max}$, $j=j_{max}$ and $k=k_{max}$, thereby covering the complete outer nose surface.

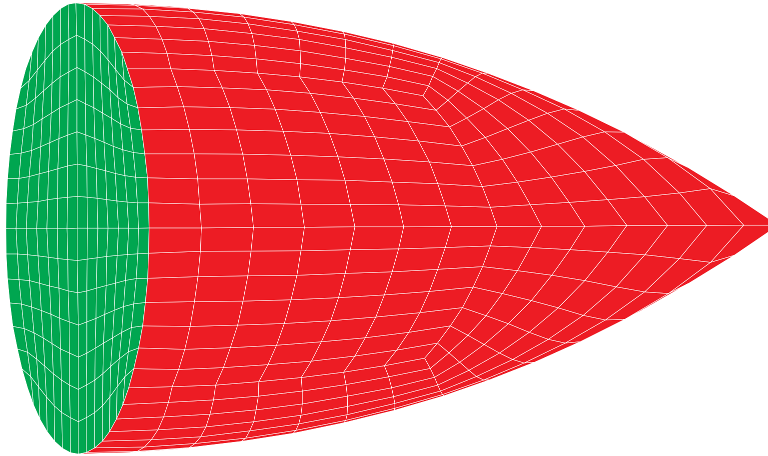

Figure 3.1 *The boundary condition "PERFORATE" has been defined for the planes* $i=1$, $j=1$, $i=i_{max}$, $j=j_{max}$ *and* $k=k_{max}$ *on the ogive subgrid.*

So far we have only defined the type of geometry for the target. It is also necessary to specify the values that define the target in the coordinate system, e.g. the radius of a cylindrical target and the position of the front face of the target. Such specifications are made using a Matlab preprocessor described in the next section.

3.2 Matlab preprocessor

The Matlab preprocessor generates two input files called *material_data.dat* and *geometry.dat*, which must be present in the Autodyn bin-directory during execution. It is, of course, also possible to generate these files using a text editor.

It is very important to be aware that all input parameters must be given in the same units as used in the relevant Autodyn model. Thus, if lengths are measured in centimeters, all geometrical parameters must be given in centimeters etc.

3.2.1 Target material model

The first four target input parameters concern the target material model:

- Yield limit
- Shear modulus
- Young's modulus
- Density

These variables are only used in the calculation of boundary effects. If an infinite target is considered, the values of these variables are not used and it is sufficient to define RBC(1)-RBC(3) in the Autodyn user interface.

3.2.2 Target geometry

The next seven inputs deal with the target geometry:

- Radius
- Front coordinate
- Rear coordinate
- Top coordinate
- Bottom coordinate
- Left coordinate
- Right coordinate

The meaning of these input parameters depend on the target geometry, which was chosen by the name of the boundary condition.

The simplest case is the sphere: radius is obviously the radius of the sphere, and the centre of the sphere has the coordinates (left, top, front), i.e. x=left, y=top, and z=front.

The cylinder radius is obviously given by the radius constant. For a cylinder, the top and bottom is defined by front and rear: z_{min}=front and z_{max}=rear. The axis is always along the z-axis, with $x=y=0$. Figure 3.2 shows a sketch of a cylindrical target with the three required constants.

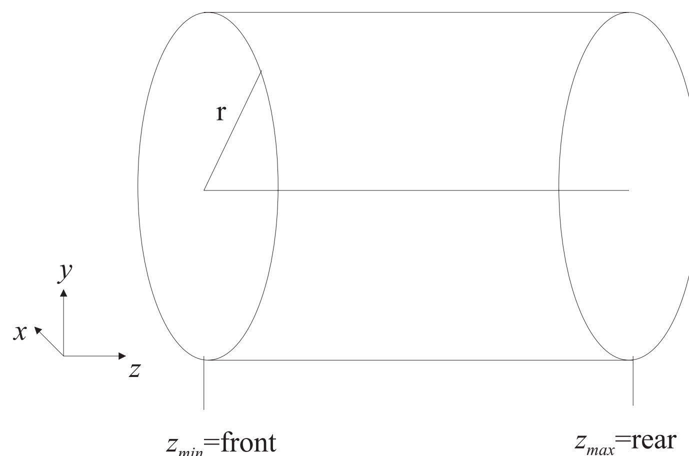

Figure 3.2 Definition of a cylindrical target.

Finally, a prism is defined from six surface planes. Note that the radius parameter is not used in this case and that the target does not have to be cubical (the name of the boundary condition might then be slightly confusing). In Figure 3.3 an example of a prism target is shown. The definition is straightforward as long as one remembers that the terms ”front”, ”rear”, ”left” and so on, refer to a viewpoint at a z-value smaller than z_{min}. The side defined by ”left” will therefore appear as the left side of the target, ”front” will appear as the front face, and so on.

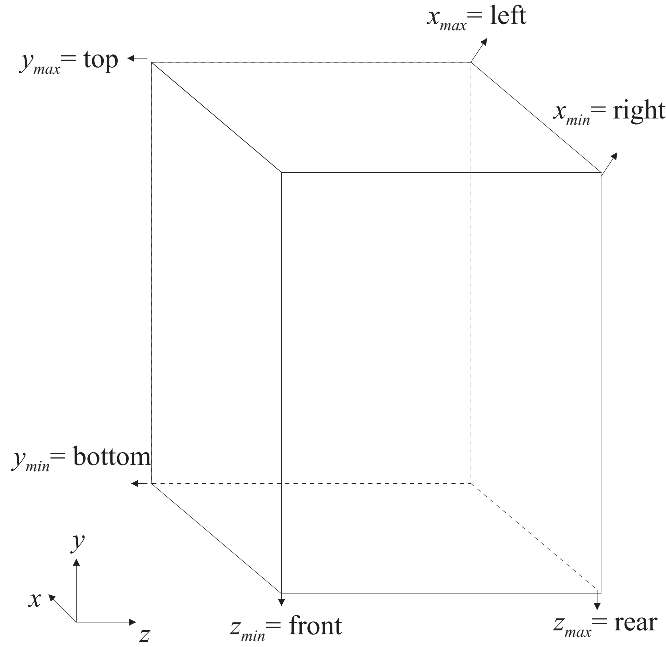

Figure 3.3 A prism with the 6 constants needed to define it.

3.2.3 Wrap-up condition

Finally, the last input lines in the preprocessor defines a wrap-up condition. It is possible to specify one of four wrap-up conditions, three of which are non-standard and are probably only useful in impact simulations. The conditions are as follows:

- *kin.energy*: wrap up when the kinetic energy is very small
- *momentum*: wrap up when the momentum is reversed
- *on perf.*: wrap up when the kinetic energy has ceased to change in time
- *cycle*: wrap-up when the simulation reaches a certain cycle

The *kin.energy* criterion is the following: If the initial kinetic energy is denoted K_i and the present kinetic energy is denoted K_p, then the calculation ends when

$$\frac{K_p}{K_i} < r$$

where r is the value specified by the user.

The *momentum* criterion is similar. Letting $\boldsymbol{p}_i$ and $\boldsymbol{p}_p$ denote initial and present momentum, the calculation wraps up at the condition

$$\boldsymbol{p}_i \cdot \boldsymbol{p}_p \leq 0 .$$

The criterion *on perf.* is also a kinetic energy criterion. It works like this: let K_p be the present kinetic energy again, and let K_{p-1} be the kinetic energy in the previous "cycle". Then the simulation wraps up when

$$\frac{K_p}{K_{p-1}} \geq r$$

Note that K_{p-1} is not literally the kinetic energy in the previous cycle, but the kinetic energy the last time `exedit` was called. The user sets the frequency at which this subroutine is called from the menu selection Global—Edit—User.

The *cycle* criterion is obviously the maximum cycle number in the calculation. If a standard Autodyn wrap-up criterion is desired, this can be achieved simply by never calling the `exedit` subroutine except in cycle 0.

3.3 Target orientation

Since the target is not modelled explicitly, the target orientation is always fixed relative to the Autodyn coordinate system with planes along the coordinate axes. In order to vary the impact, yaw and pitch angles, the projectile must be moved about instead of the target.

4 THE MODIFIED AUTODYN PROGRAM

After having defined the simulation set-up according to the instructions in Chapters 2 and 3, it is executed as a normal Autodyn simulation. In this chapter we explain how things actually work internally in the extended Autodyn code. We will not be concerned with the source code itself, but will rather examine the general idea, see how the calculations are divided into subtasks, and explain how each of them are carried out.

4.1 An overview of the program

In Autodyn, the user subroutine `exstr` allows the user to define a pressure boundary. This subroutine is called at every cycle in the computation for certain predefined cells, defined from the Autodyn menu under Global—Subgrid—Boundary.

The pressure boundary condition is defined on the surface and Autodyn automatically distributes it as a force on the four corner nodes of a cell face. The disturbance that results from this pressure is then advanced through the subgrid in the normal way.

Figure 4.1 shows a sketch of the modified Autodyn program structure. The subroutine `exedit` is called in the initialisation cycle (i.e. cycle 0), and a file containing initialisation data is opened and read. Some simple preliminary calculations are then performed to convert the input parameters into constants that are easier to work with internally.

The variables are stored in the modules `savevar`, `geometry`, `static_constants` and `constants_def`. The use of modules is the only way to make the variables available to

other subroutines. The final initialisations are executed from the `exstr` subroutine in cycle 1, setting the variable that determines the type of boundary condition used. The initialisation data are saved and loaded in the `exsave` and `exload` subroutines, respectively.

The essential part of the program is the calculation of the pressure boundary condition. The pressure will be a function of the cell velocity, the distance to any free surfaces in the target, and the diameter of the projectile. The task of finding this pressure is divided into seven subtasks, each of which must be performed on all surface cells in every cycle.

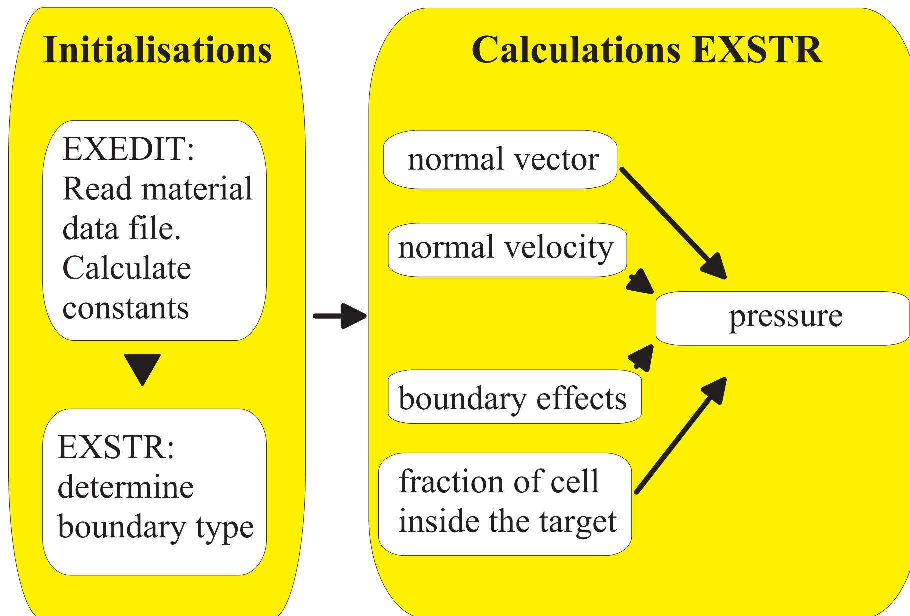

Figure 4.1 A coarse overview of the program.

The subtasks are as follows:

1. Calculate the outward pointing normal vector of the cell face
2. Calculate the node velocity in the direction of the normal vector for each of the four corner nodes, and find the average value
3. Find the distance to any free surfaces in the target
4. Find the fraction of the cell inside the target
5. Calculate the pressure on the projectile in an infinite target
6. Use the results from the above computations to calculate the decay function
7. Find the final pressure

Most of these calculations are carried out using other subroutines and functions placed in separate files or modules. In particular, the module `math_funcs2` contains several functions used to compute the normal vector, the decay function and so on.

In the remaining part of this chapter, each of the seven steps are explained in more detail. The focus is on the algorithms, the theoretical background, and the problems we need to consider in each step. The detailed source code can be found in Appendix A.

4.2 The various subtasks

4.2.1 Normal vector

The normal vector of an element is found from a straightforward cross product between the diagonals of the cell. However, care is required to make sure it points away from the projectile surface. Therefore we have to take care of the order of the vectors as well as their directions.

The subroutine `exstr` is called with the *ijk*-indices of two diagonally opposite nodes. These nodes define the surface that is subject to the pressure boundary. As an example, consider Figure 4.2 which shows a cell surface. When `exstr` is called, the indices of the nodes (i1,j1) and (i2,j2) are passed to the subroutine as formal parameters. Using this information we can access the positions of each node via the `xn`, `yn` and `zn`-arrays, and once the positions are known it is a simple task to find the diagonal vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Their direction is chosen with care to eventually give a normal vector pointing away from the projectile.

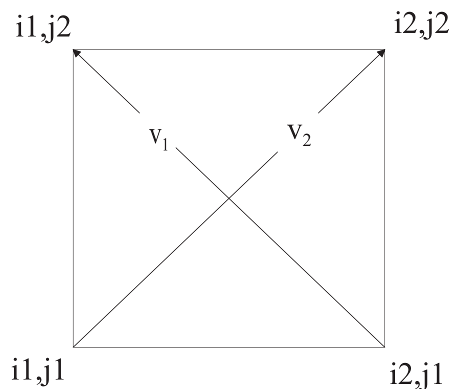

Figure 4.2 Cell face with either k=*1 or* k=k_{max}*, viewed from outside the subgrid. The vectors* $\mathbf{v}_1$ *and* $\mathbf{v}_2$ *are the diagonal vectors.*

A normal vector $\boldsymbol{n}$ in the direction coming out of the plane is then:

$$\boldsymbol{n} = \boldsymbol{v}_2 \times \boldsymbol{v}_1$$

4.2.2 Normal velocity

Once we have calculated the normal vector, it is straightforward to find the normal velocity. The inner product of the unit normal vector and the velocity vector for each node gives the component of the node velocity vector along the normal vector. We then compute a cell velocity by finding the mean velocity of the four corner nodes.

4.2.3 Distance to a free surface

The calculation of the distance between the projectile and the target boundary is actually less trivial than it may seem initially. In fact, even the definition of distance to a free boundary is not obvious, as shown in Figure 4.3.

If we are to take the cavity expansion formalism literally, we should use the distance along the projectile surface normal vector, which is the radius of the medium in which the "cavity" expands. This method has been implemented in the present program. Another possibility is to use the shortest distance to the surface. To implement this, only a trivial change in the source code is needed.

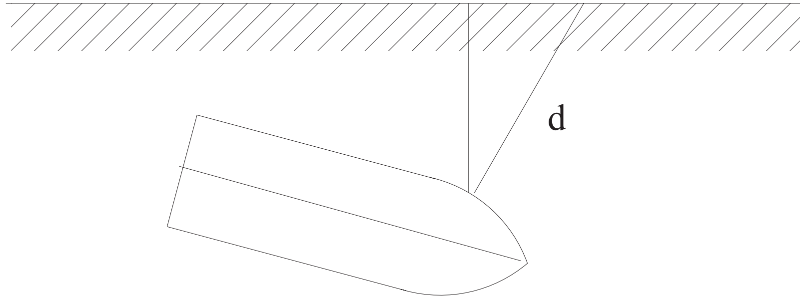

Figure 4.3 The distance to a free boundary may be defined in many ways. In this figure the two possibilites discussed in the text are shown.

A second issue is how to carry out the actual calculation. One possible method is to employ basic geometry to find a mathematical expression for the distance to a free boundary. This works well for simple geometries (objects consisting of one surface only, such as semi-infinite targets), and if the distance we want to find is the shortest possible. However, the calculations are more tedious for the implemented definition of d, whereas objects consisting of several surfaces (such as cubes) complicate the programming enormously.

Instead we have implemented a binary search algorithm. In this case we define a cutoff distance, beyond which it is known that boundary effects can safely be neglected. This provides us with an interval which represents the search range. The following algorithm is then applied:

1. Find the midpoint of the search range.
2. Check if this midpoint is inside or outside the target.
3. Redefine the search range: if the midpoint was inside the target, then use the upper half of the search range and in the opposite case use the lower half of the search range.

This procedure brings us closer and closer to the actual boundary, although it is never found exactly. If we loop n times, the resulting value is accurate to within $1/2^{n-1}$ of the cutoff distance. In the present version of the program we have used 100 projectile diameters as the cutoff value. As an example, assume the diameter is 7.5 cm and that we loop 11 times. The distance is then found to within

$$\frac{7.5 \times 100cm}{2^{10}} = 0.73 \ \ cm .$$

A higher accuracy can be obtained by increasing the number of iterations. Obviously, too many iterations will increase the runtime, but provided the number of iterations is kept at a reasonable level (i.e. between 10 and 20) the effect on the total runtime is small. In the current subroutine a default value of 13 iterations is used. It is trivial to change the source code to another number of iterations.

4.2.4 Cell position relative to target

When a projectile cell first enter the target, the pressure should be turned on gradually depending on how far the cell has penetrated. This can be achieved by multiplying the pressure with a factor depending on how much of the cell is inside the target. The simplest factor is just the number of nodes inside the target divided by four, which is the approach used in the current subroutine. This leads to a stepwise rise in the pressure, which should be a close enough approximation.

Another possibility would have been to actually calculate the relative cell face area inside the target. Although this gives a smooth transition between zero pressure and full pressure, it is a much more involved method, which have therefore not yet been implemented.

4.2.5 Pressure in a semi-infinite target

In a wide range of cases, the cavity expansion theory gives the cavity pressure as a function of expansion velocity as a quadratic velocity function:

$$p(v) = A + Bv + Cv^2 \tag{4.1}$$

The constants *A, B* and *C* in this expression depend on material behaviour, and are the constants input as RBC(1)-RBC(3). When applied to penetration problems, the expansion velocity is taken as the normal component of the velocity vector. In many cases the linear term is small and can be safely neglected.

4.2.6 Decay function

The decay function $\alpha(v,d)$ has a value between 0 and 1. It is calculated from cavity expansion theory as the ratio between the radial stress in a finite and infinite medium. The function depends on the distance d to the free boundaries, the projectile velocity, and possibly more parameters.

We must emphasise that the decay function is usually not an exact solution, but an approximation based on a simplified material model. Several alternative decay functions exist depending on the approximation used. However, in practise the difference between the various expressions is usually small. In our implementation we have used a dynamic expression derived in (1). The various geometrical parameters are defined in Figure 4.4.

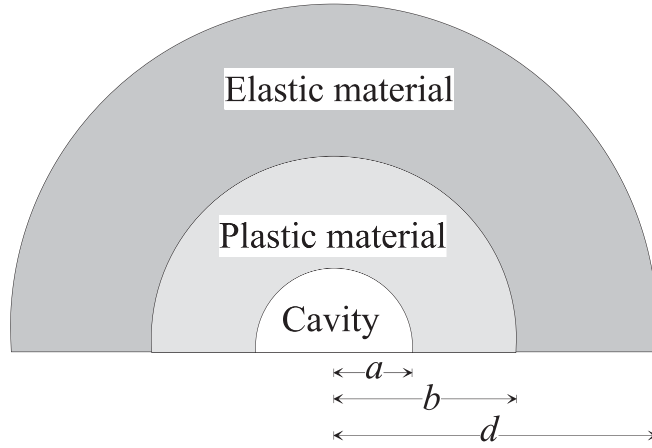

Figure 4.4 The physical problem in cavity expansion is to find the pressure at the cavity surface (r=a) as the cavity expands, while boundary conditions apply at the elastic-plastic boundary (r=b) and the elastic boundary (r=d).

For a plastic zone smaller than the target diameter, we have:

$$\alpha\left(d,a,v\right)=\frac{\frac{2Y}{3}\left(\ln\left(\left(\frac{b}{a}\right)^{3}\right)+1-\left(\frac{b}{a}\right)^{3}\left(\frac{a}{d}\right)^{3}\right)+\frac{\rho v^{2}}{2}\left(3+\left(\frac{a}{d}\right)^{4}-\frac{4a}{d}\right)}{\frac{2Y}{3}\left(1+\ln\left(\left(\frac{b}{a}\right)^{3}\right)\right)+\frac{3\rho v^{2}}{2}},\ d\geq b \tag{4.2}$$

whereas when the whole target is plastic, we use

$$\alpha\left(d,a,v\right)=\frac{2Y\ln\left(\frac{d}{a}\right)+\frac{\rho v^{2}}{2}\left(3+\left(\frac{a}{d}\right)^{4}-\frac{4a}{d}\right)}{\frac{2Y}{3}\left(1+\ln\left(\left(\frac{b}{a}\right)^{3}\right)\right)+\frac{3\rho v^{2}}{2}},\ d<b \tag{4.3}$$

The elastic-plastic boundary is given by:

$$\frac{b}{a}=\left(\frac{2G}{Y}\right)^{1/3} \tag{4.4}$$

4.2.7 Final pressure

The final pressure is found by multiplying the semi-infinite resistive pressure described in Chapter 4.2.5 with the decay function from Chapter 4.2.6:

$$p_{red}(v,d) = \alpha(v,d)p(v) \tag{4.5}$$

Further details about the calculation of boundary effects can be found in (6).

5 COMPARISON WITH EXPERIMENTS AND OTHER METHODS

In this chapter we apply our method to several different situations. In addition to demonstrating the capabilities of the approach, this enables us to check that the subroutine has been correctly implemented.

First we look at the simple case of penetration into a semi-infinite target with no boundary effects, and compare the results with the semi-analytical expressions from cavity expansion theory. Then we compare with results of Warren and Poormon (1) for the case of oblique impact and finally we look at some perforation experiments.

The projectile variables are explained in Figure 5.1.

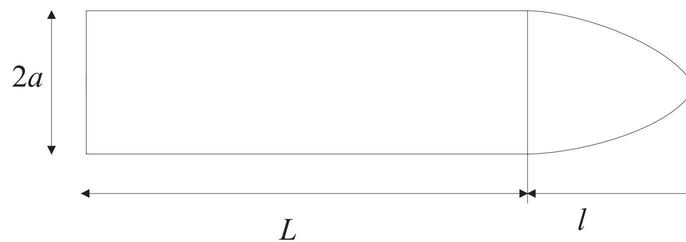

Figure 5.1 Definition of the projectile dimensions.

5.1 Comparison with semi-analytic expressions for normal impact

We start by making a direct comparison with semi-analytical theory, in which case we would expect to obtain the same result both from Autodyn and theory.

For this example we use projectile dimensions of L=50 mm, l=30 mm, mass=0.162 kg, and $2a$=20 mm. The impact velocity was 400 m/s and the standard Autodyn Johnson-Cook library 4340 steel model was employed.

The target material was arbitrarily chosen, with material constants roughly corresponding to a rather low-strength concrete of 48 MPa. The constants in the pressure function of Equation (4.1) are calculated from an empirical formula by Forrestal (6):

$$A = \sigma_c 49.5\left(\frac{\sigma_c}{10^6}\right)^{-0,43}$$

$$B = 0$$

$$C = \rho$$

The final input values are given in Table 5.1. (Keep in mind that Autodyn units are mm, ms and mg, so density is expressed in g/cm^3 and stress in kPa):

Table 5.1 Material parameters for the concrete target.

Yield stress Y	236 MPa
Youngs modulus E	62.5 GPa
Poisson ratio	0.25
Density	2.44 g/cm^3
RBC(1)	$4.5 \cdot 10^5$
RBC(2)	0.0
RBC(3)	2.44
RBC(4)	0.0
RBC(5)	0.0

Note that since RBC(4) and RBC(5) are not used by Autodyn, we could have entered whichever value we wanted for these parameters.

Mesh sensitivity was investigated by running two different simulations, one with a projectile consisting of 640 cells and one with 12000 cells making up the projectile.

The Autodyn results are presented in Table 5.2 along with the result of an analytical calculation. The small disagreement can be explained by the phase when a cell face is partly embedded in the target. Notice that meshing appears to have no effect on the results, which is a good sign.

Table 5.2 Penetration depths for three calculations.

Type of calculation	Penetration depth
Analytical	100.1 mm
Simulation, 640 cells	100.8 mm
Simulation, 12000 cells	100.8 mm

5.2 Oblique impact

A more challenging problem is the oblique impact of a projectile on a target. This would normally require a full 3D-simulation, which is very timeconsuming, and was part of the original motivation for implementing the cavity expansion algorithm in Autodyn.
One specific situation has been studied carefully, both numerically and experimentally by Warren and Poormon (1), using their own implementation of the cavity expansion algorithm in a different code.

A complete description of the simulations and experiments can be found in the original article, but the basic details are included here for completeness. The dimensions of the projectile were L=59.3 mm, l=11.8 mm, and $2a$ = 7.11 mm and it was modelled using a steel model that differed slightly from the standard Autodyn Johnson-Cook model, one of the differences being a higher value for the bulk and Young moduli and yield strength. Furthermore, a different strain hardening model was applied:

$$Y = Y_0 \left(1 + \frac{\varepsilon_p}{\varepsilon_{p0}}\right)^{1/n}$$

where Y is the yield stress, Y_0 is the initial yield stress (or input yield stress), ε_p is the effective plastic strain, and ε_{p0} and n are input constants. This has been implemented in Autodyn using the subroutine `exyld`. Table 5.3 shows the input data used in the simulations.

Table 5.3 Input material data for the steel model.

Material model: Von mises plasticity with strain dependence in subroutine `exyld`	
Density	8025 kg/m^3
Bulk modulus	206 GPa
Shear modulus	76 GPa
Yield stress Y_0	1.481 GPa
n	25.0
ε_{p0}	$7.189 \cdot 10^{-3}$

The projectile body did bend during the penetration process, which means that it was not rigid and in principle CET theory was not valid. However, since the nose remained undeformed at all times, this was not expected to be a serious problem.

The target was a 6061-T6511 aluminium cylinder. Using the material model in (1), we have $RBC(1) = 5.0394Y$, $RBC(2) = 0.983\sqrt{\rho Y}$ and $RBC(3) = 0.9402\rho$. Table 5.4 shows the input values used.

Table 5.4 Material parameters for the aluminium target

Yield stress Y	276 MPa
Youngs modulus E	69 GPa
Poisson ratio	0.33
Density	2.71 gc/m^3
RBC(1)	$1.3909 \cdot 10^6$
RBC(2)	$8.5014 \cdot 10^2$
RBC(3)	2.5479
RBC(4)	0,0
RBC(5)	0,0

Four simulations were selected for comparison. Unfortunately, Warren and Poormon had used a target geometry consisting of a “skewed” cylinder, which was impossible to model with our current implementation of the user subroutine. Instead we used a cylinder target with a radius of 254 mm and a length of 217 mm. Since this is not exactly the same as Warren and

Poormon, we would not expect to exactly reproduce their results, although they ought to be quite similar. The various results are presented in Table 5.5, with the name of each simulation corresponding to the experiment number in Warren and Poormon's (WP) article. The *Y* and *Z* values are the coordinates of the nose tip after the projectile has come to rest.

Table 5.5 Comparison between Autodyn, WP and experiments for four selected simulations.

Simulation	Impact velocity	Impact angle	Y (AD)	Z (AD)	Y (WP)	Z (WP)	Y(exp)	Z (exp)
1-0453	1156	30	91,5	-118,8	88,7	138.1	113.2	149.5
1-0461	759	15	32,7	-94,2	30,4	95.5	34.4	85.0
1-0466	802	45	Ricochet (Y= 54)		104,3	8,9	Ricochet (Y= 38)	
1-0468	1184	45	Ricochet (Y= 169)		190,0	5,4	203.3	7.25

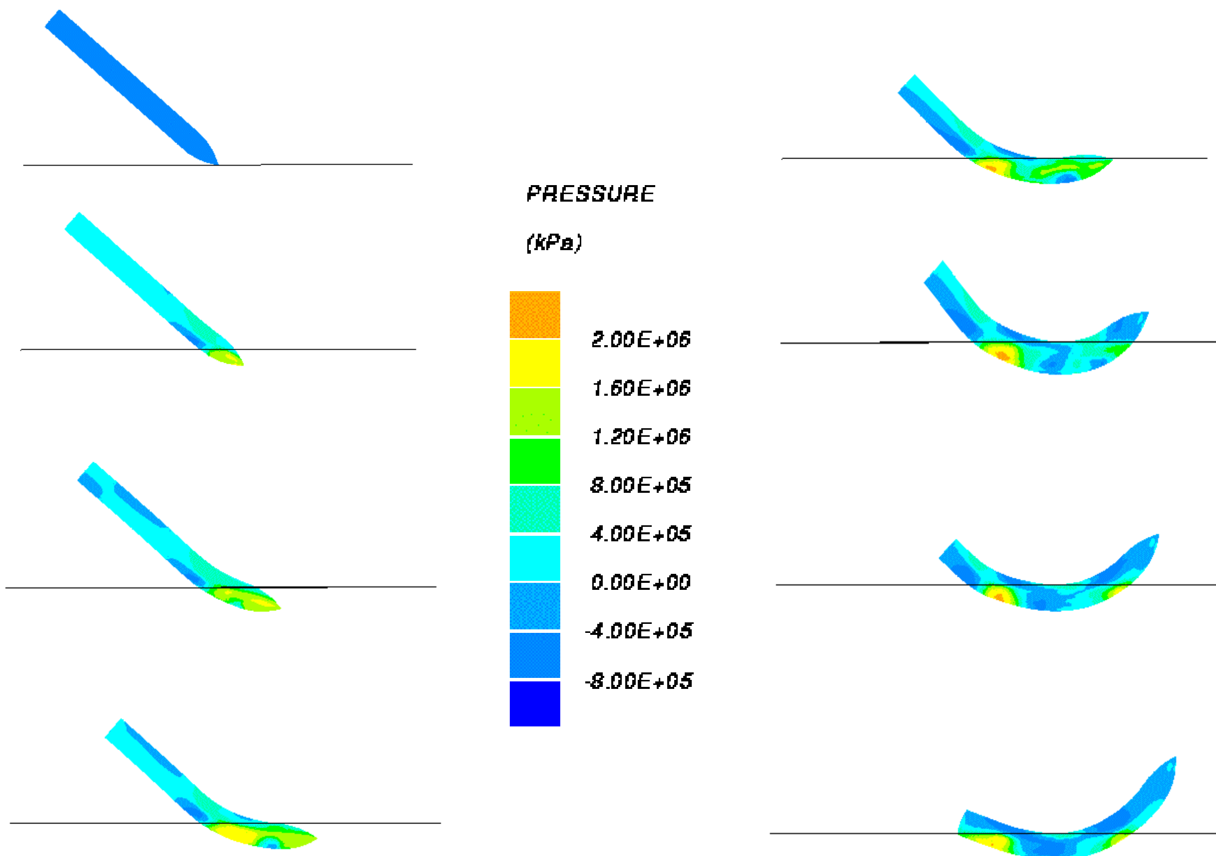

Figure 5.2: Pressure distribution in the projectile for impact at 45 degrees and velocity of 802 m/s (Simulation 1-0466).

Each simulation is completed within half an hour, which is at least 99% faster than a full In Figure 5.2 we show a contour plot of the pressure distribution in the projectile at various stages during the penetration process for Simulation 1-0466.

For the simulations 1-0453 and 1-0461, the correspondence between our results, WP and experiments is reasonably close. This is to be expected since these simulations have the smallest impact angle and boundary effects should therefore be less important. For simulation 1-0466 and 1-0468 with a larger impact angle, the results appear slightly different, though. However, this is not a reason for concern since these simulations are quite sensitive to boundary effects, which are different since we are unable to model exactly the same situation as Warren and Poormon.

5.3 Perforation

Finally, we compare with perforation experiments by Hanchak *et al* (8). They launched steel projectiles normally at concrete slabs for two very different concrete qualities. We have tried to simulate the experiments for the weakest concrete, having a compression strength of 48 MPa.

The projectile had dimensions L=101,6 mm, l=42,1 mm and $2a$=25,4 mm, and the steel model was standard Johnson-Cook 4340 steel from the Autodyn material library.

The target thickness was 17.8 cm. Inputs to the target material model are shown in Table 5.6. The yield stress for the concrete should be pressure-dependent, but in order to use a simple Mises material model, we selected a constant “average” value for this parameter.

Simulations were run for two different ways of calculating the distance to free boundaries. We used both the standard CUBE stress boundary condition, which calculates the distance along the projectile surface normal vector, and the PERFORATE stress boundary condition which finds the shortest distance to free boundaries in a semi-infinite slab.

Table 5.6 Target data for the perforation simulations.

Yield stress Y	273 MPa
Poisson ratio	0.25
Young’s modulus E	$6.25 \cdot 10^4$ MPa
Density ρ	2440 kg/m^3
RBC(1)	$4.6 \cdot 10^5$
RBC(2)	0
RBC(3)	2.44
RBC(4)	0
RBC(5)	0

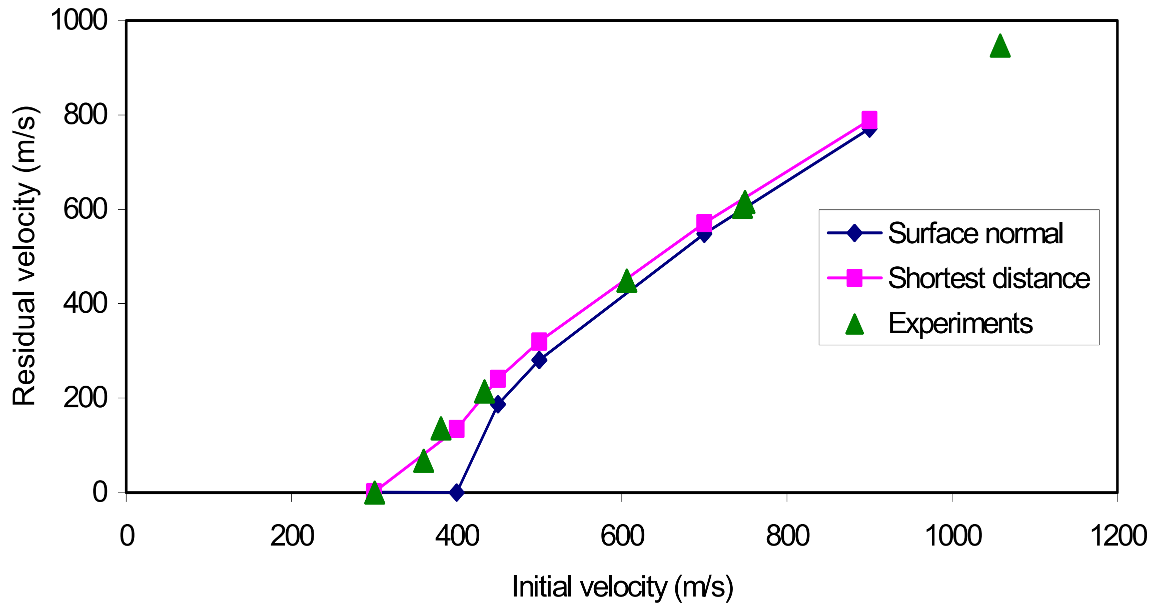

Figure 5.3 Comparison of perforation experiments by Hanchak et.al. (8) with results from the new Autodyn algorithm.

The results are plotted in Figure 5.3 along with the experimental results. The plot suggests that there is little difference between the two methods for large impact velocities. Since the target material model is a very crude approximation of the experimental concrete, these simulations can not determine whether the CUBE or PERFORATE boundary condition is the best, but still it is nice to see that both seem to give reasonable results.

6 CONCLUSION AND FURTHER WORK

We have implemented a combined analytical and numerical method for 3D penetration simulations of rigid projectiles. The approach takes advantage of the possibilities for developing user subroutines in Autodyn. After these new subroutines have been linked to Autodyn, a new executable file with extended capabilities is created.

The new method gives runtimes of less than 1% of a corresponding full 3D simulation. This makes it possible to perform sensitivity studies on very complex problems, something which was out of reach with normal Autodyn-3D.

Suggestions for further work include:

- Investigate the importance of projectile meshing when boundary effects are important and, if possible, optimise it.
- Validate the use of the decay function on concrete targets. If necessary and possible, implement improvements.
- Implement more complicated target geometries, e.g. boulder targets in the form of spheres stacked on top of each other.

References

(1) Warren T L, Poormon K L, Penetration of 6061-T6511 aluminum targets by ogive-nosed VAR 4340 steel projectiles at oblique angles: experiments and simulations, Int. J. Imp. Engng. Vol 25, pp. 993-1022, 2001
(2) Autodyn Release Notes v4.2, Century Dynamics, 2001
(3) Autodyn Theory Manual, Century Dynamics
(4) Teland J A, A review of penetration mechanics, FFI/RAPPORT-99/01264
(5) Littlefield D L, Anderson Jr C E, Partom Y, Bless S J, The penetration of steel targets finite in radial extent, Int J. Impact Engng. Vol 19 No 1, pp. 49 - 62, 1997
(6) Teland J A, Sjøl H, Boundary Effects in Penetration into Concrete*,* FFI/RAPPORT-2000/05414
(7) Hopkins H G, Dynamic expansion of spherical cavities in metals, in I. Sneddon and R. Hill (eds): Progress in solid mechanics, vol 1, New York: North Holland, 1960, pp. 85-164
(8) Hanchak S J, Forrestal M J, Perforation of concrete slabs with 48 MPa (7 ksi) and 140 MPa (20 ksi) unconfined compressive strength, Int. J. Imp. Engng., Vol 12, No. 1, pp. 1-7, 1992

A SOURCE CODE

All user subroutines in the present versions of Autodyn are written in Fortran 90. Readers who are unfamiliar with this programming language will find a textbook useful. In addition, the Autodyn User Subroutine Tutorial may shed some light on parts of the code. Mostly, however, it should be rather straightforward to follow the program lines, albeit a bit tedious at times.

In the source code there are a few program lines that have been commented. This happens in the subroutine `surface_reduction`, for example, in the calculation of the decay function. Both the static and the dynamic expressions have been programmed, and choosing between the two is done at compile time by commenting the lines for the undesired expression.

Furthermore, in the subroutine `exyld` a three-line modification to the yield criterion has been defined. This modification was introduced to comply with the material model of Warren and Poormon, which was slightly different from the standard Autodyn material library model. A recompilation of the program is needed to use this special yield model.

A description of the various parts of the code follows in the next chapters.

A.1 Subroutine `exstr`

```
SUBROUTINE EXSTR (NAMSTR1,RBC,I1,J1,K1,I2,J2,K2,PRES)

  USE kindef
  USE bnddef
  USE ijknow
  USE cycvar
  USE wrapup
  USE mdgrid
```

The two following modules are documented below.

```
  USE maths_funcs
  use savevar
```

We start out by declaring some variables.

```
  IMPLICIT NONE

  INTEGER (INT4) ::     I1,     I2,     J1,     J2,     K1,     K2
  REAL (REAL8)   ::   PRES
  REAL (REAL8), DIMENSION(5) :: RBC
  CHARACTER (LEN=10) :: NAMSTR1
  real, dimension(1:3) :: normal_vector, diag1, diag2
  real, dimension(1:12) :: corner_positions
  real :: normal_velocity, smearing, area, normal_distance
  integer :: n1
  integer, dimension(1:4) :: ijk_array
```

Initialisations:

```
if (ncycle==1) then
   if (namstr1=='CUBE') then
      boundary_type_int=1
   else if (namstr1=='CYLINDER') then
      boundary_type_int=2
   else if (namstr1=='SPHERE') then
      boundary_type_int=3
   else
      write(6,*) 'No known boundary type specified. Target is infinite.'
      boundary_type_int=0
   end if
end if
end if
```

Now we have set the integer variable `boundary_type_int` to a value which corresponds to the type of boundary chosen by the user.

Next, we find the ijk-index of each of the four corner nodes:

```
  if (i1==i2) then
     ijk_array(1)=ijkset(i1,j1,k1)
     ijk_array(2)=ijkset(i1,j1,k2)
     ijk_array(3)=ijkset(i1,j2,k2)
     ijk_array(4)=ijkset(i1,j2,k1)
  else if (j1==j2) then
     ijk_array(1)=ijkset(i2,j1,k1)
     ijk_array(2)=ijkset(i2,j1,k2)
     ijk_array(3)=ijkset(i1,j1,k2)
     ijk_array(4)=ijkset(i1,j1,k1)
  else if (k1==k2) then
     ijk_array(1)=ijkset(i1,j1,k1)
     ijk_array(2)=ijkset(i1,j2,k1)
     ijk_array(3)=ijkset(i2,j2,k1)
     ijk_array(4)=ijkset(i2,j1,k1)
  end if
```

And now we are ready to find the positions of the corner nodes:

```
do n1=0,3
   corner_positions(n1*3+1)=xn(ijk_array(n1+1))
   corner_positions(n1*3+2)=yn(ijk_array(n1+1))
   corner_positions(n1*3+3)=zn(ijk_array(n1+1))
end do
```

When the positions are known, we can calulate an area close to the actual area of the cell face. The subroutine `surface_area` which accomplishes this is explained below. Usually this area is not needed, so the relevant part of the code is commented at present.

```
area=surface_area(&
     corner_positions(4:6)-corner_positions(1:3), &
     corner_positions(10:12)-corner_positions(1:3),&
     corner_positions(4:6)-corner_positions(7:9),&
     corner_positions(10:12)-corner_positions(7:9))
```

Now we find the diagonals of the cell face, defined as the vectors between two opposing corner nodes. Later, these are used in the calculation of a normal vector. Hence, to ensure it always points outwards, the direction of the diagonals must be chosen with care.

```
diag1=corner_positions(7:9)-corner_positions(1:3)
if ((i1==i2 .and. i1==1) .or. (j1==j2 .and. j1==1) .or. &
     (k1==k2 .and. k1==1)) then
   diag2=corner_positions(4:6)-corner_positions(10:12)
else
   diag2=corner_positions(10:12)-corner_positions(1:3)
end if
```

We calculate the outward pointing unit normal, and find the average velocity of the four corner nodes.

```
call unit_normal(diag2,diag1,normal_vector)
normal_velocity=0.0
diag2=0.0
do n1=1,4
   diag2(1)=uxn(ijk_array(n1))+diag2(1)
   diag2(2)=uyn(ijk_array(n1))+diag2(2)
   diag2(3)=uzn(ijk_array(n1))+diag2(3)
end do
diag2=diag2/4
```

The function `normal_component` is simply calculating the inner product of two vectors.

```
normal_velocity=normal_component(diag2,normal_vector)
```

Now the main part of the subroutine starts, using `boundary_type_int` to choose between the different possible boundary conditions: one free surface, two free surfaces, with or without the linear term in the pressure function, and a corner. The variable `smearing` is used to reduce the pressure calculated from the subroutine `pressure_function`.

```
select case (boundary_type_int)
    case(0)
          smearing=real(ncycle)/rbc(3)
          if (smearing>=1.0) smearing=1.0
```

```
      case(1)
            call cube(normal_vector,corner_positions,&
                  var01(ijk_now),smearing,normal_distance)
            if (normal_distance<100.0 .and. normal_distance>0.0) then
                  normal_distance=2.0*normal_distance+1.0!Conversion to
!radii, and translation to the zero point of the decay function
                  smearing=smearing*surface_reduction(normal_distance,&
           normal_velocity, rbc(3))
      end if
      case(2)
            call cylinder(normal_vector,corner_positions,&
                  var01(ijk_now),smearing,normal_distance)
            if (normal_distance<100.0 .and. normal_distance>0.0) then
                  normal_distance=2.0*normal_distance+1.0!Conversion to
!radii
                  smearing=smearing*surface_reduction(normal_distance,&
              normal_velocity, rbc(3))
            end if
      case(3)
            call sphere(normal_vector,corner_positions,&
                  var01(ijk_now),smearing,normal_distance)
            if (normal_distance<100.0 .and. normal_distance>0.0) then
                  normal_distance=2.0*normal_distance+1.0!Conversion to
!radii
                  smearing=smearing*surface_reduction(normal_distance,&
                    normal_velocity, rbc(3))
            end if
      case(4)
            call perforation(normal_vector,corner_positions,&
                  var01(ijk_now),smearing,normal_distance)
            if (normal_distance<100.0 .and. normal_distance>0.0) then
                  normal_distance=2.0*normal_distance+1.0!Conversion to
!radii
                  smearing=smearing*surface_reduction(normal_distance,&
                    normal_velocity, rbc(3))
            end if


  end select
  call pressure_function(rbc,normal_velocity,pres)
  pres=smearing*pres

  RETURN

END SUBROUTINE EXSTR
```

A.2 Subroutines `exedit`, `exsave` and `exload`

The subroutine `exedit` reads the material data file in cycle 0, as well as doing some calculations of static variables. Results are stored to avoid doing the same computation in every cycle, for every relevant cell. In addition, a wrap-up criterion is programmed, and it is possible to write values of the decay function for five different velocities to a file called `alpha_values.dat`. (This was implemented for debugging and the user will probably never need to do this).

```
SUBROUTINE EXEDIT
```

```
USE kindef
USE wrapup
USE subdef
USE mdgrid
USE cycvar

use savevar
use constants_def
use static_constants
use maths_funcs

IMPLICIT NONE

integer, dimension(1:8) :: values
character(len=8) :: date
character(len=10) :: time1
character(len=5) :: zone
integer :: previous,ivel
character :: create_plot
real, dimension(1:5) :: velocities

if (ncycle==0) then
```

First, read the material data file:

```
   open(15,file='material_data.dat',form='formatted')
   read(15,*)
   read(15,*) yld
   read(15,*)
   read(15,*) shrmd
   read(15,*)
   read(15,*) youngmd
   read(15,*)
   read(15,*) exp_n, exp_m
   read(15,*)
   read(15,*) epsdot_user, eps_user
   read(15,*)
   read(15,*) wrap_up_condition
   read(15,*)
   read(15,*) create_plot
   read(15,*)
   read(15,*) velocities
   close(15)
   read(15,*) top
   read(15,*)
   read(15,*) bottom
   read(15,*)
   read(15,*) left
   read(15,*)
   read(15,*) right
   close(15)
```

Calculate some constants to be used later, and place the values in variables stored in modules so that these values are accessible from all subroutines:

```
   exp_n=1/exp_n
   plastic_boundary=(2*youngmd)/(3*yld)
   plastic_boundary=plastic_boundary**0.333333333
   red_yld=2*yld/3
   log_constant=red_yld+red_yld*log((2*youngmd)/(3*yld))
```

```
   previous=0
   poisson=youngmd/(2*shrmd)-1
   static2=(1+poisson)/(4*(1-poisson))
   static3=1+log(2*shrmd/yld)
   static4=4*shrmd/(static2*yld)
   static1=2*shrmd*(1+poisson)/(yld*(3-poisson))
   static1=static1**0.333333333
```

If the user has requested it, create a file containing value pairs of the decay function, one value being a distance from a free boundary and the other the calculated decay function value. Do this for five velocities.

```
   if (create_plot=='y') then
      open(83,file='alpha_values.dat',form='formatted')
      do ivel=1,5
         write(83,*) 'Velocity=',velocities(ivel)
         do previous=20,120
            write(83,'(f20.10,3x,f20.10)') real(previous)/20.0, &
              surface_reduction(real(previous)/20.0, &
              velocities(ivel), 2.8)
        end do
        do previous=61,261
            write(83,'(f20.10,3x,f20.10)') real(previous)/10.0, &
               surface_reduction(real(previous)/10.0, &
               velocities(ivel), 2.8)
        end do
        do previous=150,250
            write(83,'(f20.10,3x,f20.10)') real(previous)/5.0, &
               surface_reduction(real(previous)/5.0, &
               velocities(ivel), 2.8)
        end do
     end do
     close(83)
     end if
   else
```

Wrap the calculation up either when the projectile stops, or turns around. The stop condition is calculated by calculating the ratio between the starting kinetic energy and the present kinetic energy. If the ratio is less than a user specified small value then the calculation is wrapped up. The turning condition is that the inner product between the starting momentum and the present momentum is negative.

```
     if (wrap_up_condition>0.0) then
            if (subke(1)/subkeb(1)<wrap_up_condition) then
            nswrap=99
     end if
   else if (wrap_up_condition<0.0) then
      if (subxmb(1)*subxm(1)+subymb(1)*subym(1)+subzmb(1)*subzm(1)<0.0)&
            then
      nswrap=99
      end if
   end if
end if


RETURN
```

```
END SUBROUTINE EXEDIT
```

The subroutines `exsave` and `exload` are included to handle the saving of the non-standard variables. With them, we may stop the execution of the program and still be able to remember the initialisations (which are only called if the cycle number is 0 and 1, depending on the variable). The variables in question are defined in the module `savevar`.

```
SUBROUTINE EXSAVE (NTYPE)

USE kindef
USE fildef

use savevar
use geometry

IMPLICIT NONE

INTEGER (INT4) ::  NTYPE
integer :: writestat

if (ntype==1) then
   write(nut1,iostat=writestat) shrmd, yld, detection_distance, &
         displacement,boundary_type_int, exp_n, exp_m, epsdot_user, &
         eps_user, youngmd, front, rear, top, bottom, right, left,  &
         radius, wrap_up_condition
else if (ntype==2) then
   write(nut2,'(4(es13.6),i4,13(es13.6))',iostat=writestat) shrmd, yld, &
         detection_distance, displacement,boundary_type_int, exp_n, &
         exp_m, epsdot_user, eps_user, youngmd , radius, top, bottom, &
         right, left, front, rear, wrap_up_condition
end if
if (writestat/=0) write(6,*) 'Error writing custom data to file.'

RETURN
END SUBROUTINE EXSAVE


SUBROUTINE EXLOAD (NTYPE)

USE kindef
USE fildef

use savevar
use constants_def
use static_constants
use geometry

IMPLICIT NONE

INTEGER (INT4) ::  NTYPE
integer :: readstat

if (ntype==1) then
   read(nut1,iostat=readstat) shrmd, yld, detection_distance, &
         displacement,boundary_type_int, exp_n, exp_m, epsdot_user, &
         eps_user, youngmd, front, rear, top, bottom, right, left,  &
         radius, wrap_up_condition
else if (ntype==2) then
   read(nut2,'(4(es13.6),i4,13(es13.6))',iostat=readstat) shrmd, yld, &
```

```
        detection_distance, displacement,boundary_type_int, exp_n, &
        exp_m, epsdot_user, eps_user, youngmd , radius, top, bottom, &
        right, left, front, rear, wrap_up_condition
end if
if (readstat/=0) write(6,*) 'Error reading custom data from file.'
plastic_boundary=(2*youngmd)/(3*yld)
plastic_boundary=plastic_boundary**0.333333333
red_yld=2*yld/3
log_constant=red_yld+red_yld*3*log(plastic_boundary)
poisson=youngmd/(2*shrmd)-1
static2=(1+poisson)/(4*(1-poisson))
static3=1/(1-log(2*shrmd/yld))
static4=16*shrmd*(1-poisson)/(yld*(1+poisson))
static1=2*shrmd*(1+poisson)/(yld*(3-poisson))
static1=static1**0.333333333

RETURN
END SUBROUTINE EXLOAD
```

A.3 The subroutine `exval`

The sole purpose of this subroutine is to calculate the diameter of the projectile for each surface cell, and store the value in the user variable `var01`. The way this calculation is done is shown in Figure A.1.

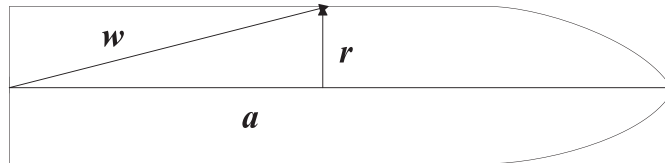

Figure A.1: Sketch of a projectile with the vectors needed to find the radius at any point on the surface.

We first need to identify the axis vector, in other words the vector between the tip point and the endpoint. The vector is shown as ***a*** in the figure. When this vector has been identified, we can find the vector between the endpoint and any point on the surface. This vector is shown as ***w*** in the figure. We want to find ***r***, the normal vector from the axis to this point on the surface. Simple geometry gives ***r*** expressed in terms of ***a*** and ***w*** as

$$\boldsymbol{r} = \boldsymbol{w} - \frac{\boldsymbol{a} \cdot \boldsymbol{w}}{\|\boldsymbol{a}\|^2}\boldsymbol{a}$$

The radius is then taken as the length of this vector. We use this formula to calculate the radius to the four corner nodes of each surface cell, and then taking the average distance as the value stored in the `var01` variable. The program assumes elsewhere that the value stored is the diameter rather than the radius, so we finally multiply by 2.

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

USE kindef
USE bnddef
```

```
USE subdef
USE mdgrid

IMPLICIT NONE

INTEGER (INT4) ::    IJK,I,J,K,   MATI,     NP
INTEGER (INT4) ::     NS
REAL (REAL8)   ::   RHOI,   RREF,   SIEI,    URI,    UXI,    UYI,   UZI
real, dimension(1:3) :: axis_vec, radius_vec, pos_vec
real :: length_axis, angle
character :: yon
integer :: axis_i, axis_j, n1
integer, dimension(1:4) :: corner_ijk
```

After the declarations we start out by prompting the user whether to calculate the actual radius for every surface cell, or just calculate the maximum radius. We also find the axis vector as the vector between nodes `((imax+1)/2,(jmax+1)/2,2)` and `((imax+1)/2,(jmax+1)/2,kmax)`. Since these tasks only need to be performed once, we do them only in the first cell, that is, for *ijk*-index (2,2,2).

```
if (i==2 .and. j==2 .and. k==2) then
   call getyon(yon, '$Use actual radius$')
   axis_i=(imax+1)/2
   axis_j=(jmax+1)/2
   axis_vec(1)=xn(ijkset(axis_i,axis_j,2))-xn(ijkset(axis_i,axis_j,kmax))
   axis_vec(2)=yn(ijkset(axis_i,axis_j,2))-yn(ijkset(axis_i,axis_j,kmax))
   axis_vec(3)=zn(ijkset(axis_i,axis_j,2))-zn(ijkset(axis_i,axis_j,kmax))
   length_axis=axis_vec(1)**2+axis_vec(2)**2+axis_vec(3)**2
   length_axis=sqrt(length_axis)
end if
```

The following `if`-branching is needed to correctly identify the corner nodes in each surface cell face. The `exval` subroutine is only called for cells. Since Autodyn cells are identified by their upper corner cell (see the Autodyn theory manual (3)), the surface cells for the lower values of *i* and *j* have indices (2,2,*k*) or similarly. If we just use the *i,j,k*-indices that are formal parameters in the subroutine, we do not use the surface nodes to calculate the radius, and the value is thus wrong. We need the following branches to use the correct nodes in the calculation of the radius.

```
if (yon=='Y') then
   if (i==2) then
      corner_ijk(1)=ijkset(1,j,k)
      corner_ijk(2)=ijkset(1,j-1,k)
      corner_ijk(3)=ijkset(1,j,k-1)
      corner_ijk(4)=ijkset(1,j-1,k-1)
   else if (i==imax) then
      corner_ijk(1)=ijkset(i,j,k)
      corner_ijk(2)=ijkset(i,j-1,k)
      corner_ijk(3)=ijkset(i,j,k-1)
      corner_ijk(4)=ijkset(i,j-1,k-1)
   else if (j==2) then
      corner_ijk(1)=ijkset(i,1,k)
      corner_ijk(2)=ijkset(i-1,1,k)
      corner_ijk(3)=ijkset(i,1,k-1)
      corner_ijk(4)=ijkset(i-1,1,k-1)
   else if (j==jmax) then
```

```
      corner_ijk(1)=ijkset(i,j,k)
      corner_ijk(2)=ijkset(i-1,j,k)
      corner_ijk(3)=ijkset(i,j,k-1)
      corner_ijk(4)=ijkset(i-1,j,k-1)
   else if (k==kmax) then
      corner_ijk(1)=ijkset(i,j,k)
      corner_ijk(2)=ijkset(i-1,j,k)
      corner_ijk(3)=ijkset(i,j-1,k)
      corner_ijk(4)=ijkset(i-1,j-1,k)
   else
      corner_ijk(1)=ijkset(i,j,k)
      corner_ijk(2)=ijkset(i,j,k)
      corner_ijk(3)=ijkset(i,j,k)
      corner_ijk(4)=ijkset(i,j,k)
   end if
```

The *ijk*-indices of the corner nodes have now been identified correctly. We then proceed to calculate the ***w*** vector and the radius for each of the four nodes. The user variable is assigned the average value, and finally converted to a diameter by multiplication with 2.

```
   var01(ijk)=0.0
   do n1=1,4
      pos_vec(1)=xn(ijkset(axis_i,axis_j,2))-xn(corner_ijk(n1))
      pos_vec(2)=yn(ijkset(axis_i,axis_j,2))-yn(corner_ijk(n1))
      pos_vec(3)=zn(ijkset(axis_i,axis_j,2))-zn(corner_ijk(n1))
      angle=pos_vec(1)*axis_vec(1)+pos_vec(2)*axis_vec(2)+&
           pos_vec(3)*axis_vec(3)
      angle=angle/(length_axis**2)
      radius_vec=pos_vec-angle*axis_vec
      var01(ijk)=var01(ijk)+2.0*sqrt(radius_vec(1)**2+&
           radius_vec(2)**2+radius_vec(3)**2)
   end do
   var01(ijk)=var01(ijk)/4
else
   pos_vec(1)=xn(ijkset(axis_i,axis_j,2))-xn(ijkset(imax,jmax,2))
   pos_vec(2)=yn(ijkset(axis_i,axis_j,2))-yn(ijkset(imax,jmax,2))
   pos_vec(3)=zn(ijkset(axis_i,axis_j,2))-zn(ijkset(imax,jmax,2))
   angle=pos_vec(1)*axis_vec(1)+pos_vec(2)*axis_vec(2)+&
        pos_vec(3)*axis_vec(3)
   angle=angle/(length_axis**2)
   radius_vec=pos_vec-angle*axis_vec
   var01(ijk)=2.0*sqrt(radius_vec(1)**2+radius_vec(2)**2+&
        radius_vec(3)**2)
end if

RETURN
END SUBROUTINE EXVAL
```

A.4 Sundry modules

These modules contain various variables, as well as a function, needed by several different subroutines. The constants contained in these modules could be passed to subroutines as arguments, but the reason for placing them in modules is the much simpler programming this entails.

```
module savevar
```

```
  real :: shrmd, yld, detection_distance, displacement, youngmd, side_face
  integer :: boundary_type_int, body_number
  real :: exp_n, exp_m, epsdot_user, eps_user, front_face, upper_face
  real :: wrap_up_condition, poisson
end module savevar

module constants_def
  real :: red_yld, log_constant, plastic_boundary
end module constants_def

module static_constants
  real :: static1, static2, static3, static4
contains
  real function plastic_b(var)
    real :: var
    plastic_b=static2*(sqrt(1+static4/(var**3))-1)
  end function plastic_b
end module static_constants

module geometry
   real :: radius, front, rear, top, bottom, left, right
end module geometry
```

A.5 Module maths_funcs2

This module contains a number of different subroutines and functions used in the program. Firstly, the following is an overview of the module with the names of all functions and subroutines defined, along with the call variables in each case.

```
module maths_funcs2
contains

      real function surface_reduction(d,velocity,dens)
      ...
      end function surface_reduction

      subroutine unit_normal(vec1,vec2,result_vector)
      ...
      end subroutine unit_normal

      subroutine cube(norm_vec,corner_pos,diam,fraction,distance)
      ...
      end subroutine cube

      subroutine cube(norm_vec,corner_pos,diam,present_time, snd_spd,&
      fraction,distance)
      ...
      end subroutine cube

      subroutine cylinder(norm_vec,corner_pos,diam,fraction,distance)
      ...
      end subroutine cylinder

      subroutine sphere(norm_vec,corner_pos,diam,fraction,distance)
      ...
      end subroutine sphere

      real function normal_component(vec1,unit_vec)
      ...
```

```
      end function normal_component

      real function surface_area(span1,span2,span3,span4)
      ...
      end function surface_area

end module maths_funcs
```

In more detail, the contents of each of these routines are shown below.

A.5.1 real function surface_reduction

```
real function surface_reduction(d,velocity, dens)
!d is in units of projectile radii

! STATIC EXPRESSION FOR THE REDUCTION OF PRESSURE AS FUNCTION OF DISTANCE
! TO FREE BOUNDARIES
    use static_constants
    real :: d, velocity, dens


    if (d>=static1) then
       surface_reduction=(1-
plastic_b(d)+log(plastic_b(d))+log(d**3))/static3
    else
       surface_reduction=3*log(d)/static3
    end if

! DYNAMIC SPHERICAL CAVITY EXPANSION
!!$    use constants_def
!!$    real :: d, velocity, dens, dens2
!!$
!!$    dens2=dens/2
!!$    if (plastic_boundary<=d) then
!!$       d=1/d
!!$       surface_reduction=(log_constant-&
!!$            red_yld*(plastic_boundary**3)*(d**3)+&
!!$            (dens2*velocity**2)*(3+d**4-4*d))/&
!!$            (log_constant+3*dens2*(velocity**2))
!!$    else
!!$       surface_reduction=(3*red_yld*log(d)+&
!!$            dens2*velocity*velocity*(3+1/(d**4)-4/d))/&
!!$            (log_constant+3*dens2*velocity*velocity)
!!$    end if
    if (surface_reduction>1.0) surface_reduction=1.0
    if (surface_reduction<0.0) surface_reduction=0.0

    return
  end function surface_reduction
```

A.5.2 subroutine unit_normal

```
    implicit none
    real, dimension(1:3), intent(in) :: vec1,vec2
    real, dimension(1:3), intent(inout) :: result_vector
    real :: length_vec

    result_vector(1)=vec1(2)*vec2(3)-vec1(3)*vec2(2)
    result_vector(2)=vec1(3)*vec2(1)-vec1(1)*vec2(3)
    result_vector(3)=vec1(1)*vec2(2)-vec1(2)*vec2(1)
```

```
    length_vec=sqrt(result_vector(1)**2+result_vector(2)**2+&
         result_vector(3)**2)
    result_vector=result_vector/length_vec

    return
  end subroutine unit_normal
```

A.5.3 subroutine cube

This subroutine calculates the smearing factor and the distance from free boundaries in the case of a prism-shaped target. The smearing factor is used to reduce the pressure on a cell face when that cell is only partially embedded in the target. The factor is calculated in the simplest possible way: when only one node is inside the target, the factor is 0.25, when two nodes are inside it is 0.5, and so on.

```
subroutine cube(norm_vec, nodes, diam, fraction, distance)
    use geometry
!The variables front, left, right, top etc refers to the target as seen
along
!the positive z-axis. These variables are defined in the module.
    implicit none

    real, dimension(1:3), intent(in) :: norm_vec
    real, dimension(1:12), intent(in) :: nodes
    real, intent(in) :: diam
    real, intent(out) :: fraction, distance
    integer :: no_nodes, i
    real, dimension(1:3) :: distance_vec
    real :: distance, nodex, nodey, nodez, minimum, maximum

    !Check how many nodes are within the target, compute fraction
    no_nodes=0
    accuracy=0.0
    distance_vec=0.0
    do i=0,3!The target is _always_ along coordinate axes
       nodex=nodes(i*3+1)
       nodey=nodes(i*3+2)
       nodez=nodes(i*3+3)
       if (nodex<left .and. nodex>right .and. nodey<top .and. &
            nodey>bottom .and. nodez>front .and. nodez<rear) then
          no_nodes=no_nodes+1
       end if
       distance_vec(1)=distance_vec(1)+nodex
       distance_vec(2)=distance_vec(2)+nodey
       distance_vec(3)=distance_vec(3)+nodez
    end do!distance_vec is now the vector sum of all four corner node
positions
!relative to the absolute coordinate system in the calculation
    distance_vec=distance_vec/4
    nodex=distance_vec(1)
    nodey=distance_vec(2)
    nodez=distance_vec(3)!calculation of geometric mean of cell face.
    if (no_nodes>0) then
!Binaersoek-algoritmen
       maximum=100.0*diam
       minimum=0.0
       distance_vec=maximum*norm_vec
       if (nodex+distance_vec(1)<left .and. nodex+distance_vec(1)>right&
```

```
             .and. nodey+distance_vec(2)<top .and. &
             nodey+distance_vec(2)>bottom .and. &
             nodez+distance_vec(3)>front .and. &
             nodez+distance_vec(3)<rear) then
        distance=101.0*diam
     else
        do i=1,13
           distance=(minimum+maximum)/2
           distance_vec=distance*norm_vec
           if (nodex+distance_vec(1)<left .and.
nodex+distance_vec(1)>right&
             .and. nodey+distance_vec(2)<top .and. &
             nodey+distance_vec(2)>bottom .and. &
             nodez+distance_vec(3)>front .and. &
             nodez+distance_vec(3)<rear) then
          minimum=distance
           else
          maximum=distance
           end if
        end do
        distance=(minimum+maximum)/(2*diam)
     end if
     fraction=real(no_nodes)/4.0
  else if (no_nodes==0) then
     distance=-1.0
     fraction=0.0
  end if

  return
end subroutine cube
```

A.5.4 subroutine perforation

This subroutine only needs two geometry inputs: the front and the rear faces of the target. It calculates the shortest distance to the rear face, but only after enough time has passed that elastic waves can travel across the target thickness and back to the projectile. It also calculates the shortest distance to the front face for oblique impacts. The target is infinite in the other two directions.

```
subroutine perforation(norm_vec, nodes, present_time, snd_spd, &
       diam, fraction, distance)
    use geometry
!The variables front, left, right, top etc refers to the target as seen
!along the positive z-axis. These variables are defined in the module.
    implicit none

    real, dimension(1:3), intent(in) :: norm_vec
    real, dimension(1:12), intent(in) :: nodes
    real, intent(in) :: diam, present_time, snd_spd
    real, intent(out) :: fraction, distance
    integer :: no_nodes, i
    real, dimension(1:3) :: distance_vec
    real :: distance, nodex, nodey, nodez, minimum, maximum

    !Check how many nodes are within the target, compute fraction
    no_nodes=0
    distance_vec=0.0
    do i=0,3!The target is _always_ along coordinate axes
       nodex=nodes(i*3+1)
```

```
       nodey=nodes(i*3+2)
       nodez=nodes(i*3+3)
       if (nodex<left .and. nodex>right .and. nodey<top .and. &
            nodey>bottom .and. nodez>front .and. nodez<rear) then
          no_nodes=no_nodes+1
       end if
       distance_vec(1)=distance_vec(1)+nodex
       distance_vec(2)=distance_vec(2)+nodey
       distance_vec(3)=distance_vec(3)+nodez
    end do
!distance_vec is now the vector sum of all four corner node positions
!relative to the absolute coordinate system in the calculation
    distance_vec=distance_vec/4
    nodex=distance_vec(1)
    nodey=distance_vec(2)
    nodez=distance_vec(3)!calculation of geometric mean of cell face.
    if (no_nodes>0) then
       if (norm_vec(3)>=0.0) then
          if (present_time>(rear-front+nodez)/snd_spd) then
             distance=(rear-nodez)/diam
          else
             distance=101.0
          end if
          fraction=real(no_nodes)/4.0
       else
          distance=(nodez-front)/diam
          fraction=real(no_nodes)/4.0
          if (distance>100.0) distance=101.0
       end if
    else if (no_nodes==0) then
       distance=-1.0
       fraction=0.0
    end if

    return
  end subroutine perforation
```

A.5.5 subroutine cylinder

In case of a cylindrical target, the following subroutine is needed:

```
subroutine cylinder(norm_vec, nodes, diam, fraction, distance)
    use geometry
!The radius is the radius of the cylindrical target. The axis is along
!the positive z-axis, through the origin (x=0 y=0). These variables are
!defined in the module.
    implicit none

    real, dimension(1:3), intent(in) :: norm_vec
    real, dimension(1:12), intent(in) :: nodes
    real, intent(in) :: diam
    real, intent(out) :: fraction, distance
    integer :: no_nodes, i
    real, dimension(1:3) :: distance_vec
    real :: distance, nodex, nodey, nodez, minimum, maximum, act_rad

    !Check how many nodes are within the target, compute fraction
    no_nodes=0
    distance_vec=0.0
```

```
    do i=0,3!The target is _always_ along coordinate axes
       nodex=nodes(i*3+1)
       nodey=nodes(i*3+2)
       nodez=nodes(i*3+3)
       act_rad=nodex**2+nodey**2
       if (act_rad<radius**2 .and.  nodez>front .and. nodez<rear) then
          no_nodes=no_nodes+1
       end if
       distance_vec(1)=distance_vec(1)+nodex
       distance_vec(2)=distance_vec(2)+nodey
       distance_vec(3)=distance_vec(3)+nodez
    end do!distance_vec is now the vector sum of all four corner node
positions
!relative to the absolute coordinate system in the calculation
    distance_vec=distance_vec/4
    nodex=distance_vec(1)
    nodey=distance_vec(2)
    nodez=distance_vec(3)!calculation of geometric mean of cell face.
    if (no_nodes>0) then
maximum=100.0*diam
       minimum=0.0
       distance_vec=maximum*norm_vec
       act_rad=(nodex+distance_vec(1))**2+(nodey+distance_vec(2))**2
       if (act_rad<radius**2 .and.  &
            nodez+distance_vec(3)>front .and. &
            nodez+distance_vec(3)<rear) then
          distance=101.0*diam
       else
          do i=1,13
             distance=(minimum+maximum)/2
             distance_vec=distance*norm_vec
             act_rad=(nodex+distance_vec(1))**2+(nodey+distance_vec(2))**2
             if (act_rad<radius**2 .and.  &
                  nodez+distance_vec(3)>front .and. &
                  nodez+distance_vec(3)<rear) then
            minimum=distance
             else
            maximum=distance
             end if
          end do
          distance=(maximum+minimum)/(2*diam)
       end if
       fraction=real(no_nodes)/4.0
    else if (no_nodes==0) then
       distance=-1.0
       fraction=0.0
    end if

    return
  end subroutine cylinder
```

A.5.6 subroutine sphere

Finally, the same calculations are performed for a spherical target using the subroutine `sphere`:

```
subroutine sphere(norm_vec, nodes, diam, fraction, distance)
    use geometry
```

```
!The radius is the radius of the spherical target. The variable is defined
in
!the module. The centre of the sphere is at (x=left, y=top, z=front).
    implicit none

    real, dimension(1:3), intent(in) :: norm_vec
    real, dimension(1:12), intent(in) :: nodes
    real, intent(in) :: diam
    real, intent(out) :: fraction, distance
    integer :: no_nodes, i
    real, dimension(1:3) :: distance_vec
    real :: distance, nodex, nodey, nodez, minimum, maximum, act_rad

    !Check how many nodes are within the target, compute fraction
    no_nodes=0
    distance_vec=0.0
    do i=0,3!The target is _always_ along coordinate axes
       nodex=nodes(i*3+1)
       nodey=nodes(i*3+2)
       nodez=nodes(i*3+3)
       act_rad=(nodex-left)**2+(nodey-top)**2+(nodez-front)**2
       if (act_rad<=radius**2) then
          no_nodes=no_nodes+1
       end if
       distance_vec(1)=distance_vec(1)+nodex
       distance_vec(2)=distance_vec(2)+nodey
       distance_vec(3)=distance_vec(3)+nodez
    end do!distance_vec is now the vector sum of all four corner node
positions
!relative to the absolute coordinate system in the calculation
    distance_vec=distance_vec/4
    nodex=distance_vec(1)
    nodey=distance_vec(2)
    nodez=distance_vec(3)!calculation of geometric mean of cell face.
    if (no_nodes>0) then
maximum=100.0*diam
       minimum=0.0
       distance_vec=maximum*norm_vec
       act_rad=(nodex+distance_vec(1))**2+(nodey+distance_vec(2))**2+&
            (nodez+distance_vec(3))**2
       if (act_rad<radius**2) then
          distance=101.0*diam
       else
          do i=1,13
             distance=(minimum+maximum)/2
             distance_vec=distance*norm_vec
             act_rad=(nodex+distance_vec(1))**2+(nodey+distance_vec(2))**2+&
                  (nodez+distance_vec(3))**2
             if (act_rad<radius**2) then
            minimum=distance
             else
            maximum=distance
             end if
          end do
          distance=(maximum+minimum)/(2*diam)
       end if
       fraction=real(no_nodes)/4.0
    else if (no_nodes==0) then
       distance=-1.0
       fraction=0.0
    end if
```

```
    return
  end subroutine sphere
```

A.5.7 real function normal_component

```
    implicit none
    real, dimension(1:3),intent(in) :: vec1,unit_vec

    normal_component=vec1(1)*unit_vec(1)+vec1(2)*unit_vec(2)+&
         vec1(3)*unit_vec(3)
    return
  end function  normal_component
```

A.5.8 real function surface_area

```
    implicit none
    real, dimension(1:3),intent(in) :: span1,span2,span3,span4
    real :: area

    area=sqrt((span1(2)*span2(3)-span1(3)*span2(2))**2+&
          (span1(3)*span2(1)-span1(1)*span2(3))**2+&
          (span1(1)*span2(2)-span1(2)*span2(1))**2)+&
         sqrt((span3(2)*span4(3)-span3(3)*span4(2))**2+&
          (span3(3)*span4(1)-span3(1)*span4(3))**2+&
          (span3(1)*span4(2)-span3(2)*span4(1))**2)
    surface_area=area/2
    return
  end function surface_area
```

A.6 The pressure subroutine

Although it is not necessary, the actual pressure calculation has been placed in a separate subroutine in a separate file. The idea behind this is that it will be easier to find and alter the pressure function this way. Whether this is sensible from efficiency considerations has not been tested, but any loss in efficiency should not be important compared with all the other calculations that are performed in each cycle.

A different issue regards the layout of the subroutine. As it stands, the constants in the quadratic formula are determined from user input, i.e. the array called `rbc` in the `exstr` subroutine. This array only contains five constants, and we need more to completely define the target material. As the program now stands, the remaining constants (yield limit, elastic moduli and geometry) are all input to the program via files. It would perhaps be simpler to use the program if all the target material data were input via the same file, including also the constants in the pressure function. Such an improvement to the code can be written by the diligent reader.

```
subroutine pressure_function(user_inputs,velocity,distance,pressure)
  implicit none
  real, dimension(1:5), intent(in) :: user_inputs
  real ,intent(in) :: velocity, distance
!The variable "velocity" is the normal component of velocity
  real, intent(out) :: pressure
```

```
  if (velocity<=0) then
     pressure=0.0
  else

pressure=user_inputs(1)+user_inputs(2)*velocity+user_inputs(3)*velocity**2
  end if

  return
end subroutine pressure_function
```

B COMPILATION AND LINKING

The compilation and linking can be done with the Makefile discussed here. To use it, one needs to copy the source code files and the Makefile to a directory `usrsub/forcing_function`. The standard Autodyn directory hierarchy should also be used, that is, the `usrsub` directory should reside in the same folder as `bin` and `data`. With alterations to the Makefile, it is of course possible to change this folder structure.

The first line in the Makefile defines the path to the Autodyn directory, a path referred to as `PATH1`. This path must be changed to the appropriate name. For instance, if you have the autodyn files in a directory called `autodyn` in your home directory, 3D version 4.2 in a directory called `3dv42`, with `bin`, `data` and `usrsub` in this directory, then `PATH1` is `/user/<username>/autodyn/3dv42`.

In addition to compiling and linking, the makefile can be used to generate examples of input files.

B.1 Makefile commands

The following commands are possible:

make:

> The command compiles the necessary .f90-files and links them with the standard Autodyn program. The name of the output file is defined in the Makefile.

make $(SLAVE):

> The command is as above, but generates the slave process for 3D parallell runs. The $(SLAVE) symbol is defined in the Makefile, and is the name of the slave process. The input name must be identical to the name in the Makefile.

make example:

> The command generates a program called input_examples, and runs the program. The program generates examples of the material_data.dat-file and the geometry.dat-file.

make clean:

The command removes all files except the *.f90-files. It also removes all executables generated with the Makefile.

make zipclean:

This command is the same as the clean command, except that in addition it uses gzip to compress the *.f90-files.

B.2 The Makefile

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

PATH1=      /user/aao/autodyn/3dv42
FILES1= $(PATH1)/usrsub/admain3.o $(PATH1)/usrsub/autodyn3.a
ADSLAVES= $(PATH1)/usrsub/adslav3.o $(PATH1)/usrsub/autodyn3.a
FILES2= variables.f90 math_funcs.f90 pressure_function.f90 $(F90FILE).f90
OBJFILES= ${FILES2:.f90=.o}
#MODFILES= ${FILES3:.f90=.mod}
GKSDIR= $(gksdir)
PVM_DIR= $(PVM_ROOT)/lib/HPPA/
PROGRAM= forcing_dynamic_binary
DATADIR= testruns
SLAVE= adslav3
FLAGS=      -L$(GKSDIR)  -lgksflb -lgksw5300 \
      -lgksw1900 -lgkswiss \
      -lgksgksm -lgksmsc -L$(PVM_ROOT)/libfpvm/HPPA -lX11 -lm
FLAGS2= +save +noshared +O2 +DA2.0 -I $(PATH1)/usrsub \
      -I $(PATH1)/usrsub/forcing_function
FLAGS3= -Wl,-a,shared -lnsl -ldld -I $(PATH1)/usrsub \
      -I $(PATH1)/usrsub/forcing_function
DEBUG= +gprof
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/.

input_example : example.f90
      f90 -o input_example example.f90

example : input_example
      input_example
      mv geometry.dat material_data.dat $(PATH1)/bin/.

clean :
      rm -f *~ *.o *.mod *.dat
      rm -f $(PROGRAM) $(PATH1)/bin/$(PROGRAM)
      rm -f $(SLAVE) $(PATH1)/bin/$(SLAVE)
```

```
zipclean : clean
      gzip *.f90
      gzip input_examples
```

DISTRIBUTION LIST

FFIBM **Dato:** 27 august 2002

RAPPORTTYPE (KRYSS AV)	RAPPORT NR.	REFERANSE	RAPPORTENS DATO
X RAPP ☐ NOTAT ☐ RR	2002/00575	FFIBM/766/130	27 august 2002
RAPPORTENS BESKYTTELSESGRAD		ANTALL EKS UTSTEDT	ANTALL SIDER
Unclassified		28	46
RAPPORTENS TITTEL		FORFATTER(E)	
RAPID AUTODYN-3D PENETRATION SIMULATIONS USING A VIRTUAL TARGET		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		Henrik Sjøl, FFIBM
1		Ove Dullum, FFIBM
1		John F Moxnes, FFIBM
5		Restopplag til Bibliotket
		Elektronisk fordeling:
		FFI-veven
		Lars Kvifte (LKv), FFIBM
		Bjarne Haugstad (BjH), FFIBM
		Svein Rollvik (SRo), FFIS

FFI-K1 Retningslinjer for fordeling og forsendelse er gitt i Oraklet, Bind I, Bestemmelser om publikasjoner for Forsvarets forskningsinstitutt, pkt 2 og 5. Benytt ny side om nødvendig.