JNC TN8520 2002-001

Java Meshing Tool for Sphere Arrangements

(Manual)

April, 2002

JAPAN NUCLEAR CYCLE DEVELOPMENT INSTITUTE
TOKAI WORKS

〒319-1184　茨城県那珂郡東海村村松４番地４９
核燃料サイクル開発機構
技術展開部 技術協力課

JNC TN8520 2002-001

April, 2002

Java Meshing Tool for Sphere Arrangements

Manuel Alexandre Pouchon*

Abstract

A tool for meshing sphere arrangements was programmed in order to perform finite element calculations. Sphere arrangements are investigated in frame of the feasibility study of the sphere-pac nuclear fuel. One major concern of this study is the thermal conductivity of the arrangement. Further concerns are the mechanical behavior and sintering of the fuel. The thermal conductivity of the fuel was addressed with the computer code SPACON based on a unit cell approach and a radial heat flow experiment. However, a further approach using the finite element method is desirable, in order to better understanding the thermal flow through the package and to cross check with SPACON data and with experimental data. Also the mechanical behavior of the fuel could be addressed using the finite element technique.

* Plutonium Fuel Technology Group, Advanced Fuel Recycle Technology Division, Waste Management and Fuel Cycle Research Center

JNC TN8520 2002-001
2002 年 4 月

球状粒子充てん体のための JAVA によるメッシュ作成ツール（マニュアル）

マニュエル　アレクサンドレ　プーション*

要旨

有限要素法を用いた計算を行うための球状粒子充てん体にメッシュを作成するプログラムを開発した。球状粒子充てん体はスフェアパック燃料の適用性研究の中で検討されている。この研究の中で一つの大きな関心が充てん体の熱伝導度にある。また，機械的挙動及び焼結挙動についても関心がある。燃料の熱伝導度は燃料を代表する単位セルに着目した SPACON コードや中心加熱法を用いた実験により検討されている。しかしながら，充てん体中の熱流をより詳しく知り，SPACON の計算結果と実験結果の比較を行うために，有限要素法を用いた計算が望まれている。この有限要素法は燃料の機械的挙動を検討するためにも有効である。

*環境保全・研究開発センター　先進リサイクル研究開発部　プルトニウム燃料開発グループ

Table of Contents

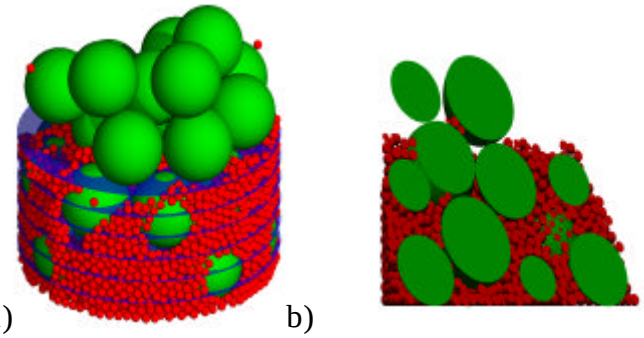
Table of Figures

1 Introduction

Beside the computer code SPACON [1] an additional computational approach to the thermal conduction through a sphere package is desirable. One further possible approach would be the finite element method.
A meshing tool was programmed in the object-oriented programming language JAVA [2]. The tool reads in a sphere arrangement formed by a particle flow code [3]. Only a representative cell is taken into account and the spheres are cut in order to fit into this cell. In a fist step only the meshing of the arrangement's solid phase is realized, which limits the finite element calculation to the thermal conduction through the solid or to mechanical considerations.

2 Theoretical background

2.1 Result from particle flow code



a) b)

Fig. 1: Particle Flow Code simulated sphere arrangement. a) shows the whole arrangement and b) a diagonal cut trough the arrangement.

Fig. 1 shows a sphere arrangement generated by the particle flow code PFC 3D. The filling of two size fractions, one with 0.1 mm and the other with 0.08 mm diameter into a cladding with a inner diameter of 1 cm was simulated.

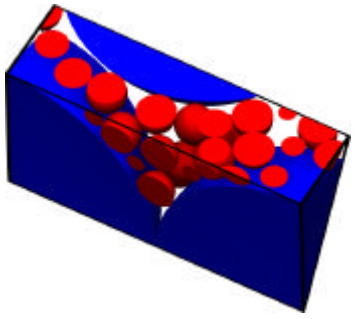


Fig 2: Representative cell being used for FEM approach

From this arrangement a representative cell was extracted. This cell can be used to consider the thermal conduction through the package. Symmetries were used to get the smallest possible representative wedge cell. However, along the extrusion of the base triangle, the cell represented in **Fig 2** could be further simplified into two half pieces. It was decided to remain this dimension for better statistics. The source code

for generating above graphics with the povray program [4] can be found in the appendix D on page 239.

2.2 Meshing Program

2.2.1 Language

Java was chosen as programming language, because of being object orientated and because of being free of charge, additionally it consists all possibilities of a modern programming language, including the automatic generation of documentation. The standard development kit of the Java platform 2 in its version 1.3.1 was used to compile the programs and to execute the java files [2].

2.2.2 File Structure

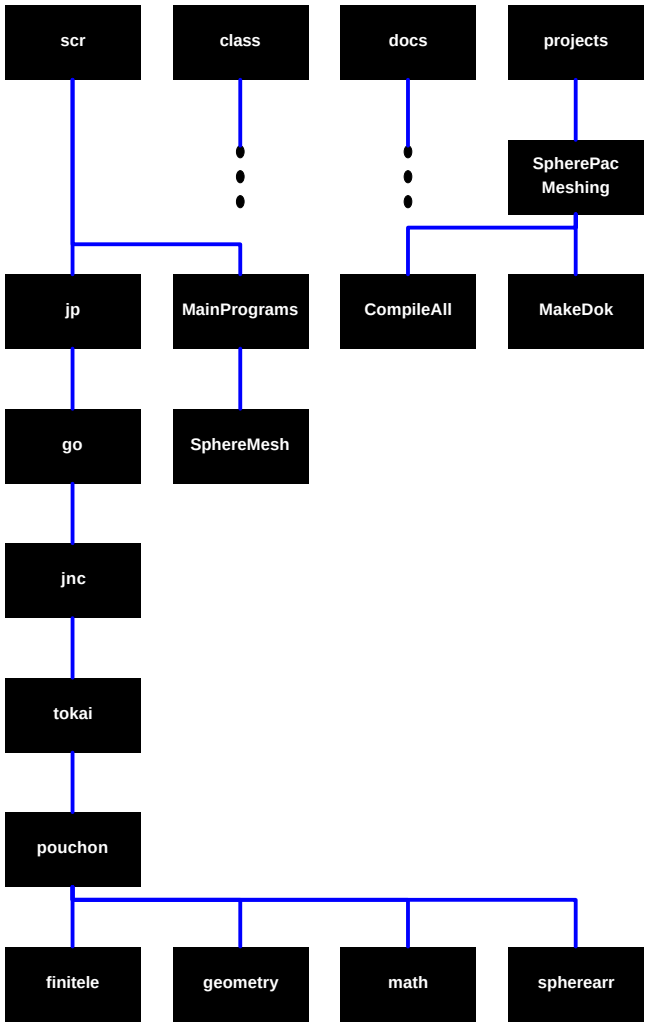


Fig 3: File structure used for the meshing tool.

The file structure of the java program is represented in Fig. **3**. The scr folder contains all the java source files, whereas the class folder contains the java compiled code which can be executed on the java virtual machine. The docs folder contains the automatically generated documentation.

The scr, the class and the docs folders contain the same file tree (except that the documentation for the MainPrograms was not performed). The tree is therefore only depicted for the scr folder. The projects folder contains the make files being necessary to compile the whole package and it contains also the make file for the documentation. These make files can be found in the appendix. The single java sources contained in the scr folder can also be found in the Appendix. Following java rules and guaranteeing uniqueness, the path to the program packages is inverse equivalent to the programmer's and institute's url, in this case, pouchon@tokai.jnc.go.jp. In the ../jp/go/jnc/tokai/pouchon folder one can find the different packages, in this case finitele, geometry, math and spherearr. The geometry package contains models and routines to handle the geometric concerns of the package. It has the basic coordinate, sphere, polygon… types with the necessary operations. The finitele package is mostly built on the geometry types, with additional identification entries. The math package contains some permutation routines, which is mainly used for permutation entities in geometrical shapes, in order to guarantee the right orientation. The spherearr package provides types to describe a entire sphere arrangement and to read in such an arrangement from a text file containing the PFC arrangement data. The generated documentation for all classes can be found in the section 4 on page 6 and the source code in the appendix C on page 110. It follows a more precise description of the packages.

2.3 Description of the packages

2.3.1 The geometry package

The geometry package handles geometric concerns. It implements the classes coordinate, SphereCoordinate, cell, line, plane, Polygon, Polyhedron, Sphere, Face, Matrix, MultiLine, MultiPolygon, MultiPolyhedron. The coordinate class is very basic and just defines an array with three entries of real numbers. These represent the coordinates in the Cartesian coordinate system. (It would be more natural to call it vector, however, the vector class is already contained in Java. In order to avoid conflicts, this class was called coordinate). In some cases the representation in spherical coordinates is preferable, therefore the class SphereCoordinate was created, it is built on two fields, one containing a real number, representing the radial distance, and the second being an array of two real number containing the angles. Instances of both class types can be transformed back and forward mainly by using the constructors of each type with an instance of the other class. For the coordinate class the common operations, like the vector addition, vector product are implemented. More complex operations, like the projection of a coordinate onto a surface are also implemented here.
The cell class was initially used to represent the representative cell, and to cut the meshed sphere into this cell. However, most of the coding is now performed in the polyhedron class, which represents the most general cell.
The line class just extends the coordinate class by a direction. It therefore represents infinitively extended lines in space or, if defining the extended coordinate as starting point, and the direction vector with its length as the line extension, the class also represents lines with defined start and endpoints. The MultiLine class defines a

collection of lines. It provides routines which try to connect the lines to a polygon. The endpoints of the multiple line are stored and the possible connections.
The polygon type is limited to defined topologies, the implemented topologies are triangle and quadrangle. If a quadrangle is given, an additional information is stored, which declares the possible division of the quadrangle into two triangles. This is necessary to know the surface points in the case when the four corners are not within one plane. The MultiPolygon defines similar to the MultiLine a collection of different polygons. This is originally programmed to have the possibility to reduce the polygon collection to a simple wedge or other voluminous shape. However, this is a very difficult topic and remains to be programmed.
The polyhedron class is a voluminous shape in space; here it is limited to a wedge or a box type shape. It replaces the cell class and provides routines to cut other polyhedrons within the own volume. MultyPolyhedron again represents a collection of Polyhedrons.
The Marix class mainly provides matrix operations.

2.3.2 The finitele package

The finitele package contains classes handling the nodes and elements representing FEM meshes. Many of the operations are based on the classes being defined in the geometry package, the node class just extends the coordinate class by an id number and a list of the elements which contain the node. It therefore inherits all the methods of the coordinate class. The NodeList class describes a collection of Nodes, this is useful for writing a collection of nodes to an output stream. The element class contains a collection of node ids and specifies the topology. However, geometrical operations are performed on a polyhedron class basis and affect directly the nodes being stated in the element, this might also involve the generation of more nodes and elements. An additional class called ElementCoordinate extends the element class in order to collect the node information with the coordinates. This class also provides methods to extract the polyhedron from the element. In the ElementList class a list of elements can be instanced. The entities class represents up to eight entities and provides methods to permute their order. This can be useful to change the orientation or geometrical shapes. In the ObjectsIds class a continuous identification with numbers is guaranteed for elements and nodes. Additionally ids can also be killed here. The ObjectCounter class serves as a object seeking tool. The id can be given, and the appropriate object is returned.

3 Main Program description and Example

3.1 Representative cell being extracted from the whole arrangement

Fig. 2 on page 1 already showed the representative cell being selected from the whole arrangement being depicted in Fig. 1. The characteristic of the selected cell can be found in appendix E.1 on page 242. There the source code of a C++ program performing the extraction of spheres interacting with the wedge or lying within it, is also given in appendix E.2. The identification numbers and the coordinates of the

selected spheres can be found in appendix E.3 (coarse fraction) and E.4 (fine fraction) on page 263ff.

3.2 Meshing of the arrangement

For the meshing of the arrangement, the coarse and fine spheres are read in by the main program of the Java tool, which can be found in the appendix C.1 on Page 110. The spheres being near the cell have already been selected by the program described in appendix E.2 on page 242. The program then performs a meshing procedure, where every sphere is meshed, in this first approach; only one radial segment is created, which is definitively not enough. But the routine performing the meshing for the single spheres is already prepared for more radial segments. The touching spheres are detected and the nearest finite elements are connected by shifting the nodes. This procedure is important to allow a connection for the thermal flow. However, the connection type is subject to changes and should be replaced with an element type appropriate for the used finite element software. The connection type should also take into consideration the necking of the spheres. A voluminous connection element is thinkable, which truly represents the necking, or a line element, which simulates it. In a next step, the generated elements are further eliminated, if they don't lie within the cell and if they don't interact with the cell surface. If an element lies partially within the cell, it is cut. The cutting procedure is rather complicated, and further improvements can be programmed. An example of the cell meshing can be seen in Fig. 4. It is visible, that only one radial segment was created. Most of the cutting procedures work fine, however, especially at the edges, further improvements can be achieved. The Java program generates a FEMAP-Neutral file. This was then imported into the FEM interface FEMAP [5], Fig. 4 was directly copied from this program. It is intended to use FINAS [6] as FEM software.

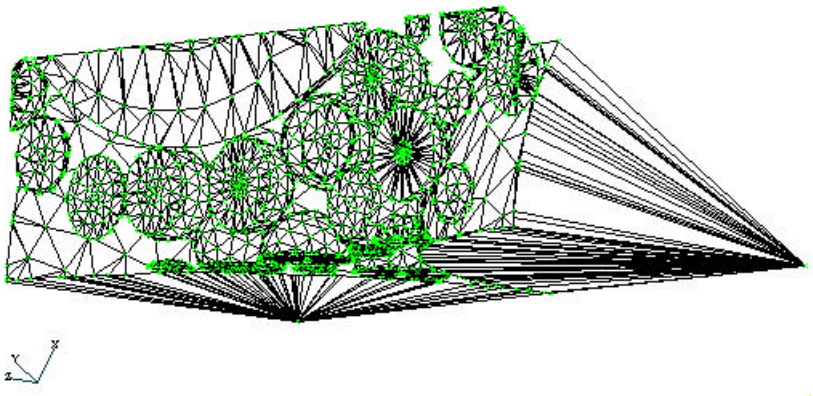


Fig 4: Meshing example of representative cell.

4 Generated documentation

4.1 Class hierarchy

A list of all classes in their hierarchy is represented here.

- class java.lang.Object
  - class jp.go.jnc.tokai.pouchon.spherearr.**CellSpheres**
  - class jp.go.jnc.tokai.pouchon.geometry.**Coordinate**
    - class jp.go.jnc.tokai.pouchon.geometry.**Cell**
    - class jp.go.jnc.tokai.pouchon.geometry.**Line**
    - class jp.go.jnc.tokai.pouchon.finitele.**Node**
    - class jp.go.jnc.tokai.pouchon.geometry.**Plane**
    - class jp.go.jnc.tokai.pouchon.geometry.**Polygon**
    - class jp.go.jnc.tokai.pouchon.geometry.**Polyhedron**
    - class jp.go.jnc.tokai.pouchon.geometry.**Sphere**
      - class jp.go.jnc.tokai.pouchon.spherearr.**CellSphereData**
        - class jp.go.jnc.tokai.pouchon.spherearr.**SphereMesh**
  - class jp.go.jnc.tokai.pouchon.finitele.**Element**
    - class jp.go.jnc.tokai.pouchon.finitele.**ElementCoordinate**
  - class jp.go.jnc.tokai.pouchon.finitele.**ElementList**
  - class jp.go.jnc.tokai.pouchon.finitele.**Entities**
  - class jp.go.jnc.tokai.pouchon.geometry.**Face**
  - class jp.go.jnc.tokai.pouchon.geometry.**Matrix**
  - class jp.go.jnc.tokai.pouchon.geometry.**MultiLine**
  - class jp.go.jnc.tokai.pouchon.geometry.**MultiPolygon**
  - class jp.go.jnc.tokai.pouchon.geometry.**MultiPolyhedron**
  - class jp.go.jnc.tokai.pouchon.finitele.**NodeList**
  - class jp.go.jnc.tokai.pouchon.finitele.**ObjectCounter**
  - class jp.go.jnc.tokai.pouchon.finitele.**ObjectIds**
  - class jp.go.jnc.tokai.pouchon.math.**Permutation**
  - class jp.go.jnc.tokai.pouchon.geometry.**SphereCoordinate**

4.2 The geometry package

Package handling basic geometric entities and operations

4.2.1 The coordinate class

jp.go.jnc.tokai.pouchon.geometry

Class Coordinate

java.lang.Object

|

+--**jp.go.jnc.tokai.pouchon.geometry.Coordinate**

Direct Known Subclasses:

Cell, Line, Node, Plane, Polygon, Polyhedron, Sphere

---

public class **Coordinate**

extends java.lang.Object

Representing vectors and coordinates in R^3. (Because of possible conflict with the Java-Vector type, representing an dynamic array, the term Coordinate is used here).

Version:

1.00 2001/01/10, 1.01 2000/02/19

Author:

Manuel Alexandre POUCHON - © JNC Tokai Works (JAPAN)

Field Summary	
static double	**precision** Specifies the tolerance in calculation results, if a distance of two points is smaller than this value, the points are identified as identical!
double[]	**x** *Coordinate* (or *vector*) in &real^3

Constructor Summary	
Coordinate() Produces coordinate/vector (0 0 0)	
Coordinate(Coordinate toCopy) Copys directly the coordinate/vector called "toCopy" to a new coordinate/vector	
Coordinate(Coordinate K1, Coordinate K2) Creates vector between K1 and K2	
Coordinate(double x1, double x2, double x3) Produces coordinate/vector (x1 x2 x3)	
Coordinate(double x1, double x2, double x3, double xe1, double xe2, double xe3) Produces vector between (x1 x2 x3) and (xe1 xe2 xe3)	
Coordinate(SphereCoordinate Trsf)	

Method Summary	
double	**abs**() Calculates the absolute value (the length) of a Vector
void	**add**(Coordinate ToAdd)
void	**add**(Coordinate ToAdd1, Coordinate ToAdd2)
void	**add**(Coordinate ToAdd1, Coordinate ToAdd2, double Scale)
void	**add**(Coordinate ToAdd, double Scale)

void	**assign**(Coordinate ToCopy)
void	**assign**(double x1Koor, double x2Koor, double x3Koor)
void	**assign**(SphereCoordinate Trsf)
void	**cross**(Coordinate vec1, Coordinate vec2) Cross or vector product of two vectors.
void	**cross**(Coordinate koo1, Coordinate koo2, Coordinate koo3) Cross or vector product of two vectors.
boolean	**cut**(Face F, Line L, boolean limitedLine) Cuts Face F with Line L and assigns the cutting-point (coordinate), returns true if cutting-point lies between Origin of the line and the endpoint of the line (origin plus direction)
boolean	**cut**(Plane P, Line L) Cuts Plane P with Line L and assigns the cutting-point (coordinate), returns true if cutting-point lies between Origin of the line and the endpoint of the line (origin plus direction)
boolean	**cut**(Polygon P, Line L, boolean limitedLine) Cutting of line with polygon!
double	**distance**(Coordinate second)
double	**dot**(Coordinate vec1)
static double	**dot**(Coordinate vec1, Coordinate vec2)
static double	**dot**(Coordinate koo1, Coordinate koo2, Coordinate koo3)
boolean	**equal**(Coordinate toComp) Deterines whether two coordinates are coincident (=equal here) or not.
static void	**main**(java.lang.String[] args)
void	**mult**(double multiplier) Multiplies with scalar "multiplier" and assigns.
void	**mult**(Matrix M) Multiplies vector with Matrix M and applies.
void	**normalize**() Applies the multiplication of vector vIn with the Matrix M.
void	**normalize**(double length) Normalizes a vector and multiplies with "length".
boolean	**project**(Face F, Coordinate K)
void	**project**(Line L, Coordinate K)
void	**project**(Plane P, Coordinate K)
boolean	**project**(Polygon projectOn, Coordinate toProject)

	Projects the point "toProject" on the polygon "porjectOn", the projection-coordinate is assigned to the object, a boolean value is returned to determine, whether projection point is on the surface (true) or outside (false).
void	**rotate**(double x0, double x1, double x2) Rotates around the x[0], x[1] and x[2] axis
void	**rotate**(double x0, double x1, double x2, double part) Rotates partially around the x[0], x[1] and x[2] axis with factor "part"
static double	**spat**(Coordinate vec1, Coordinate vec2, Coordinate vec3) Calculates the volume of the generally sheared box formed by the three vectors vec1, vec2 and vec3.
static double	**spat**(Coordinate origin, Coordinate end1, Coordinate end2, Coordinate end3) Calculates the volume of the generally sheared box formed by the origin and the endpoints end1, end2 and end3.
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

x

public double[] **x**

Coordinate (or *vector*) in &real^3

---

precision

public static final double **precision**

Specifies the tolerance in calculation results, if a distance of two points is smaller than this value, the points are identified as identical!

Constructor Detail

Coordinate

public **Coordinate**()

Produces coordinate/vector (0 0 0)

Coordinate

public **Coordinate**(double x1,
double x2,
double x3)

Produces coordinate/vector (x1 x2 x3)

Coordinate

public **Coordinate**(double x1,
double x2,
double x3,
double xe1,
double xe2,
double xe3)

Produces vector between (x1 x2 x3) and (xe1 xe2 xe3)

Coordinate

public **Coordinate**(Coordinate toCopy)

Copies directly the coordinate/vector called "toCopy" to a new coordinate/vector

---

Coordinate

public **Coordinate**(Coordinate K1,
Coordinate K2)

Creates vector between K1 and K2

Coordinate

public **Coordinate**(SphereCoordinate Trsf)

Method Detail

assign

public void **assign**(double x1Koor,
double x2Koor,
double x3Koor)

---

assign

public void **assign**(Coordinate ToCopy)

---

assign

public void **assign**(SphereCoordinate Trsf)

---

add

public void **add**(Coordinate ToAdd)

---

add

public void **add**(Coordinate ToAdd,
double Scale)

---

add

public void **add**(Coordinate ToAdd1,
Coordinate ToAdd2)

---

add

public void **add**(Coordinate ToAdd1,
Coordinate ToAdd2,
double Scale)

---

distance

public double **distance**(Coordinate second)

---

cross

public void **cross**(Coordinate vec1,
Coordinate vec2)

Cross or vector product of two vectors. Direct input of both vectors.

Parameters:

vec1 - fist (right) vector

vec2 - second (left) vector

---

cross

public void **cross**(Coordinate koo1,
Coordinate koo2,
Coordinate koo3)

Cross or vector product of two vectors. Endpoint-Coordinate input: not the vectors but the endpoints of the 2 vector arrangement are entered!

Parameters:

koo1 - starting point of both vectors

koo2 - endpoint of fist (right) vector

koo3 - endpoint of second (left) vector

---

dot

public double **dot**(Coordinate vec1)

---

dot

public static double **dot**(Coordinate vec1,
Coordinate vec2)

---

dot

public static double **dot**(Coordinate koo1,
Coordinate koo2,
Coordinate koo3)

---

spat

public static double **spat**(Coordinate vec1,
Coordinate vec2,
Coordinate vec3)

Calculates the volume of the generally sheared box formed by the three vectors vec1, vec2 and vec3.

---

spat

public static double **spat**(Coordinate origin,
Coordinate end1,
Coordinate end2,
Coordinate end3)

Calculates the volume of the generally sheared box formed by the origin and the endpoints end1, end2 and end3.

---

mult

public void **mult**(double multiplier)

Multiplies with scalar "multiplier" and assigns.

---

abs

public double **abs**()

Calculates the absolute value (the length) of a Vector

Returns:

The length of the vector as double

---

equal

public boolean **equal**(Coordinate toComp)

Determines whether two coordinates are coincident (=equal here) or not.

---

mult

public void **mult**(Matrix M)

Multiplies vector with Matrix M and applies.

---

rotate

public void **rotate**(double x0,
double x1,
double x2)

Rotates around the x[0], x[1] and x[2] axis

---

rotate

public void **rotate**(double x0,
double x1,
double x2,
double part)

Rotates partially around the x[0], x[1] and x[2] axis with factor "part"

---

normalize

public void **normalize**()

Applies the multiplication of vector vIn with the Matrix M. /* public void mult(Matrix M, Coordinate vIn) { Coordinate V = new Coordinate(vIn); V.mult(M); x[0]=V.x[0]; x[1]=V.x[1]; x[2]=V.x[2]; } /** Normalizes a vector (set to length 1).

---

normalize

public void **normalize**(double length)

Normalizes a vector and multiplies with "length".

---

cut

public boolean **cut**(Plane P,
Line L)

Cuts Plane P with Line L and assigns the cutting-point (coordinate), returns true if cutting-point lies beween Origin of the line and the endpoint of the line (origin plus direction)

---

cut

public boolean **cut**(Face F,
Line L,
boolean limitedLine)

Cuts Face F with Line L and assigns the cutting-point (coordinate), returns true if cutting-point lies between Origin of the line and the endpoint of the line (origin plus direction)

---

cut

public boolean **cut**(Polygon P,
Line L,
boolean limitedLine)

Cutting of line with polygon!

Returns:

true if unlimited line cuts the polygon. If limitedLine is set to true, routine only returns true, if the line defined by its origin and the length, cuts the polygon. If line is parallel to polygon, the result is false, even if the line lies on the polygon!

---

project

public boolean **project**(Polygon projectOn,
Coordinate toProject)

Projects the point "toProject" on the polygon "porjectOn", the projection-coordinate is assigned to the object, a boolean value is returned to determine, whether projection point is on the surface (true) or outside (false).

Parameters:

projectOn - Polygon which specifies the projection area

toProject - Coordinate of the point which is to be projected on the polygon

Returns:

Returns boolean which specifies whether the projection lies on the surface (true) of not (false)

---

project

public boolean **project**(Face F,
Coordinate K)

---

project

public void **project**(Plane P,
Coordinate K)

---

project

public void **project**(Line L,
Coordinate K)

WriteScr

public void **WriteScr**()

---

main

public static void **main**(java.lang.String[] args)

4.2.2 The SphereCoordinate class

jp.go.jnc.tokai.pouchon.geometry
Class SphereCoordinate

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.geometry.SphereCoordinate**

---

public class **SphereCoordinate**

extends java.lang.Object

Coordinate in spherical coordinates (in ^{3})

Author:

Manuel Alexandre Pouchon, © 2000-2001 JNC Japan

Field Summary	
`double[]`	**phi** Angles from origin to point in ^{3}, **ö₀** is the angle around the x_2 (or *z*) - axis, **ö₁** is the elevation angle relative to the x_0-x_1 (*or x-y*) - plane!
`double`	**r** Distance from origin to point in ^{3}

Constructor Summary	
SphereCoordinate()	
SphereCoordinate(Coordinate Trsf)	
SphereCoordinate(double PhiX3Inp, double PhiX12Inp, double Radius)	
SphereCoordinate(SphereCoordinate ToCopy)	

Method Summary	
void	**assign**(Coordinate Trsf)
void	**assign**(double PhiX3Inp, double PhiX12Inp, double Radius)
void	**assign**(SphereCoordinate ToCopy)
void	**rotate**(double phi0add, double phi1add)
void	**scale**(double factor)
void	**toTransform**(SphereCoordinate trFrom, SphereCoordinate trTo)
void	**transform**(double phi0add, double phi1add, double rscale)
void	**transform**(double phi0add, double phi1add, double rscale, double partial)
void	**transform**(SphereCoordinate trsf)
void	**transform**(SphereCoordinate trsf, double partial)
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

phi

public double[] **phi**

Angles from origin to point in R^3, **ö₀** is the angle around the x_2 (or *z*) - axis, **ö₁** is the elevation angle relative to the x_0-x_1 (*or x-y*) - plane!

---

r

public double **r**

Distance from origin to point in ^{3}

Constructor Detail

SphereCoordinate

public **SphereCoordinate**()

SphereCoordinate

public **SphereCoordinate**(double PhiX3Inp,
double PhiX12Inp,
double Radius)

SphereCoordinate

public **SphereCoordinate**(SphereCoordinate ToCopy)

SphereCoordinate

public **SphereCoordinate**(Coordinate Trsf)

Method Detail

assign

public void **assign**(double PhiX3Inp,
double PhiX12Inp,
double Radius)

---

assign

public void **assign**(SphereCoordinate ToCopy)

---

assign

public void **assign**(Coordinate Trsf)

---

scale

public void **scale**(double factor)

---

rotate

public void **rotate**(double phi0add,
double phi1add)

---

transform

public void **transform**(double phi0add,
double phi1add,
double rscale)

---

transform

public void **transform**(double phi0add,
double phi1add,
double rscale,
double partial)

---

transform

public void **transform**(SphereCoordinate trsf)

---

transform

public void **transform**(SphereCoordinate trsf,
double partial)

---

toTransform

public void **toTransform**(SphereCoordinate trFrom,
SphereCoordinate trTo)

WriteScr

public void **WriteScr**()

4.2.3 The cell class

jp.go.jnc.tokai.pouchon.geometry

Class Cell

java.lang.Object
|
+--jp.go.jnc.tokai.pouchon.geometry.Coordinate
|
+--**jp.go.jnc.tokai.pouchon.geometry.Cell**

---

public class **Cell**

extends Coordinate

Describes a cell in space, generally it is a sheared box or wedge, if type is true, then box (4 side prism), otherwise, if type is false, then it is a wedge (3 side prism). Origin is the base of the three sides, describing the cell. dir[0] describes the elevation of the prism, and dir[1,2] the two sides.

Field Summary	
Coordinate[]	**dir** Sides of the Cell
int	**type** Type of the cell, **true** 4-side prism (Brick), **false** 3-side prism (Wedge)

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Cell(java.lang.String FileName)	

Method Summary	
Face[]	**faces**() Returns teh faces of a cell, with the Faces being oriented towards the cell (cross product of side-vectors).
boolean	**within**(Coordinate isIn) verifies wether coordinate lies within the cell, returns boolean
boolean	**within**(Element isIn, Coordinate[] isInNodeKoor) verifies wether element interacts with the cell, returns boolean
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

dir

public Coordinate[] **dir**

Sides of the Cell

---

type

public int **type**

Type of the cell, **true** 4-side prism (Brick), **false** 3-side prism (Wedge)

Constructor Detail

Cell

public **Cell**(java.lang.String FileName)

Method Detail

within

public boolean **within**(Coordinate isIn)

verifies whether coordinate lies within the cell, returns boolean

---

within

public boolean **within**(Element isIn,
Coordinate[] isInNodeKoor)

verifies whether element interacts with the cell, returns boolean

---

faces

public Face[] **faces**()

Returns the faces of a cell, with the Faces being oriented towards the cell (cross product of side-vectors).

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

4.2.4 The line class

jp.go.jnc.tokai.pouchon.geometry

Class Line

java.lang.Object
|
+--jp.go.jnc.tokai.pouchon.geometry.Coordinate
|
+--**jp.go.jnc.tokai.pouchon.geometry.Line**

public class **Line**

extends Coordinate

Field Summary	
Coordinate	**dir**

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Line()	
Line(Coordinate o, Coordinate d)	
Line(Coordinate o, Coordinate d, boolean endPoints)	
Line(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3)	
Line(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3, boolean endPoints)	
Line(Line toCopy)	
Line(MultiLine toConvert) Construcor with a Multiple line as argument.	

Method Summary	
void	**assign**(Coordinate o, Coordinate d)
void	**assign**(Coordinate o, Coordinate d, boolean endPoints)
void	**assign**(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3)
void	**assign**(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3, boolean endPoints)
void	**assign**(Line toCopy)
void	**assign**(MultiLine toConvert) Simplifies multiple line to one simple line, works in the same way like the

	constructor with a multiple line as argument!
void	**cut**(Plane P1, Plane P2) Cuts two planes P1 and P2 and assigns the cutting line.
boolean	**cut**(Polygon pol, Plane pla) Cuts a polygons with a plane and assigns the cutting line.
boolean	**cut**(Polygon P1, Polygon P2) Cuts two polygons P1 and P2 and assigns the cutting line.
boolean	**cutOld**(Polygon P1, Polygon P2) **Deprecated.** *use cut instead, this old version contains a bug*
boolean	**equal**(Line toCompare)
double	**length**() Returns the lenght of the line.
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

dir

public Coordinate **dir**

Constructor Detail

Line

public **Line**()

Line

public **Line**(double ox1,
double ox2,
double ox3,
double dx1,
double dx2,
double dx3,
boolean endPoints)

Line

public **Line**(double ox1,
double ox2,
double ox3,
double dx1,
double dx2,
double dx3)

Line

public **Line**(Coordinate o,
Coordinate d,
boolean endPoints)

Line

public **Line**(Coordinate o,
Coordinate d)

Line

public **Line**(Line toCopy)

Line

public **Line**(MultiLine toConvert)

Constructor with a Multiple line as argument. The multiple line is simplified to one single line. If the line is one connected, non circular line, the first and last point span the new Line, if it is circular, the two points having the biggest distance span the new line, same for a multiple line composed of several separated lines. Of course this always signifies an information loss!

Parameters:

toConvert - Line which is going to be converted into the new single line.

Method Detail

assign

public void **assign**(double ox1,
double ox2,
double ox3,
double dx1,
double dx2,
double dx3,
boolean endPoints)

---

assign

public void **assign**(double ox1,

double ox2,
double ox3,
double dx1,
double dx2,
double dx3)

---

assign

public void **assign**(Coordinate o,
Coordinate d,
boolean endPoints)

---

assign

public void **assign**(Coordinate o,
Coordinate d)

---

assign

public void **assign**(Line toCopy)

---

assign

public void **assign**(MultiLine toConvert)

Simplifies multiple line to one simple line, works in the same way like the constructor with a multiple line as argument!

Parameters:

toConvert - Multiple line which is going to be assigned as a simplified line.

See Also:

Line(MultiLine)

---

length

public double **length**()

Returns the length of the line.

Returns:

length of the line.

---

cut

public void **cut**(Plane P1,
Plane P2)

Cuts two planes P1 and P2 and assigns the cutting line.

Parameters:

P1 - first plane to cut

P2 - second plane to cut

---

cut

public boolean **cut**(Polygon pol,
Plane pla)

Cuts a polygons with a plane and assigns the cutting line.

Parameters:

pol - polygon to cut

pla - plane to cut

---

cut

public boolean **cut**(Polygon P1,
Polygon P2)

Cuts two polygons P1 and P2 and assigns the cutting line.

Parameters:

P1 - first plane to cut

P2 - second plane to cut

---

equal

public boolean **equal**(Line toCompare)

---

cutOld

public boolean **cutOld**(Polygon P1,
Polygon P2)

Deprecated. *use cut instead, this old version contains a bug*

Cuts two planes P1 and P2 and assigns the cutting line.

Parameters:

P1 - first plane to cut

P2 - second plane to cut

Returns:

boolean if P1 and P2 intersect or not, returns also false, if Polygons are parallel

---

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

4.2.5 The plane class

jp.go.jnc.tokai.pouchon.geometry

Class Plane

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.geometry.Coordinate
      |
      +--**jp.go.jnc.tokai.pouchon.geometry.Plane**

---

public class **Plane**

extends Coordinate

Infinit plane: here represented by a *point* on the plane and the *normal* to the plane

---

Field Summary	
Coordinate	**normal**

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Plane() Generates plane with origin and normal being (0,0,0)	
Plane(Coordinate o, Coordinate n)	
Plane(Coordinate o, Coordinate n, boolean pointsOnPlane)	
Plane(Coordinate o, Coordinate d1, Coordinate d2) Generates a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).	
Plane(Coordinate o, Coordinate d1, Coordinate d2, boolean pointsOnPlane) Generates a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).	
Plane(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3)	
Plane(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3, boolean endPoints)	
Plane(double ox1, double ox2, double ox3, double d1x1, double d1x2, double d1x3, double d2x1, double d2x2, double d2x3) Generates a plane represented by an origin and two direction vectors:	

(ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).	
Plane(double ox1, double ox2, double ox3, double d1x1, double d1x2, double d1x3, double d2x1, double d2x2, double d2x3, boolean pointsOnPlane) Generates a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).	
Plane(Plane toCopy)	
Plane(Polygon toTrsf) Constructor of plane with Polygon as argument.	

Method Summary

void	**assign**(Coordinate o, Coordinate n)
void	**assign**(Coordinate o, Coordinate n, boolean pointsOnPlane)
void	**assign**(Coordinate o, Coordinate d1, Coordinate d2) Assigns a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).
void	**assign**(Coordinate o, Coordinate d1, Coordinate d2, boolean pointsOnPlane) Assigns a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).
void	**assign**(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3)
void	**assign**(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3, boolean endPoints)
void	**assign**(double ox1, double ox2, double ox3, double d1x1, double d1x2, double d1x3, double d2x1, double d2x2, double d2x3) Assigns a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
void	**assign**(double ox1, double ox2, double ox3, double d1x1, double d1x2, double d1x3, double d2x1, double d2x2, double d2x3, boolean pointsOnPlane) Assigns a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
void	**assign**(Plane toCopy)
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object

clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

normal

public Coordinate **normal**

Constructor Detail

Plane

public **Plane**()

Generates plane with origin and normal being (0,0,0)

Plane

public **Plane**(double ox1,
double ox2,
double ox3,
double d1x1,
double d1x2,
double d1x3,
double d2x1,
double d2x2,
double d2x3,
boolean pointsOnPlane)

Generates a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3). If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is produced.

Plane

public **Plane**(double ox1,
double ox2,
double ox3,
double d1x1,
double d1x2,
double d1x3,
double d2x1,
double d2x2,
double d2x3)

Generates a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).

Plane

public **Plane**(Coordinate o,
Coordinate d1,

Coordinate d2,
boolean pointsOnPlane)

Generates a plane represented by an origin and two direction vectors: o+u(d1)+v(d2). If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is produced.

Plane

public **Plane**(Coordinate o,
Coordinate d1,
Coordinate d2)

Generates a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).

Plane

public **Plane**(double ox1,
double ox2,
double ox3,
double nx1,
double nx2,
double nx3,
boolean endPoints)

Plane

public **Plane**(double ox1,
double ox2,
double ox3,
double nx1,
double nx2,
double nx3)

Plane

public **Plane**(Coordinate o,
Coordinate n,
boolean pointsOnPlane)

Plane

public **Plane**(Coordinate o,
Coordinate n)

Plane

public **Plane**(Plane toCopy)

Plane

public **Plane**(Polygon toTrsf)

Constructor of plane with Polygon as argument. In both cases, when the polygon is a triangle or a quadrangle, the plane is defined by the first three corners.

Parameters:

toTrsf - polygon being transformed into a plane

Method Detail

assign

public void **assign**(double ox1,
double ox2,
double ox3,
double d1x1,
double d1x2,
double d1x3,
double d2x1,
double d2x2,
double d2x3,
boolean pointsOnPlane)

Assigns a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3). If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is produced.

---

assign

public void **assign**(double ox1,
double ox2,
double ox3,
double d1x1,
double d1x2,
double d1x3,
double d2x1,
double d2x2,
double d2x3)

Assigns a plane represented by an origin and two direction vectors: (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).

---

assign

public void **assign**(Coordinate o,
Coordinate d1,
Coordinate d2,
boolean pointsOnPlane)

Assigns a plane represented by an origin and two direction vectors: o+u(d1)+v(d2). If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is produced.

---

assign

public void **assign**(Coordinate o,
Coordinate d1,
Coordinate d2)

Assigns a plane represented by an origin and two direction vectors: o+u(d1)+v(d2).

---

assign

public void **assign**(double ox1,
double ox2,
double ox3,
double nx1,
double nx2,
double nx3,
boolean endPoints)

---

assign

public void **assign**(double ox1,
double ox2,
double ox3,
double nx1,
double nx2,
double nx3)

---

assign

public void **assign**(Coordinate o,
Coordinate n,
boolean pointsOnPlane)

---

assign

public void **assign**(Coordinate o,
Coordinate n)

---

assign

public void **assign**(Plane toCopy)

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

4.2.6 The Polygon class

jp.go.jnc.tokai.pouchon.geometry

Class Polygon

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.geometry.Coordinate
      |
      +--**jp.go.jnc.tokai.pouchon.geometry.Polygon**

---

public class **Polygon**

extends Coordinate

Polygon in &real^3, it can be a triangle or a quadrangle, the quadrangle should be convex. If not so, the order procedure will modify correctly.

Field Summary	
Coordinate[]	**corner** Corners of the polygon relative to the Origin (number of corners depends from Polygon type defined in "type")
boolean	**rt_lf** Surface splitting parameter: For a triangle this parameter is irrelevant, for the quadrangle the subdivision into triangles can be performed in two ways, by calling the four corners c1, c2, c3, c4 the splitting line can be defined from c1 to c3 which is default and is called right to left (lower triangle is right, upper is left), in this case rt_lf is set to true, if splitting line is from c2 to c4 the rt_lf parameter is set to false.
int	**topology** Type of polygon, 4 types are defined as follows (compartible to FEMAP neutral file): **Triangle** - with 3 corners -> ***2*** - with 6 corners -> 3 , **Quadrangle** with 4 corners -> ***4*** - with 8 corners -> 5 , both, Triangle and Quadrangle define a surface, for the triangle in general two different surfaces, being trianles, can be defined.

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Polygon() Default constructor creates structure for triangle	
Polygon(boolean is_rt_lf) Constructs a Quadrangle, because only here setting the surface splitting orientation parameter rt_lf makes sense!	
Polygon(boolean toOrder, Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3) Constructs a Triangle.	
Polygon(boolean toOrder, Coordinate ori, Coordinate c1, Coordinate c2,	

Coordinate c3, Coordinate c4) Constructs a Quadrangle with the parameter "lt-rt" set to default value "true", this means that the surface is split into a lower-right and an upper-left triangle.	
Polygon(boolean toOrder, Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, boolean is_rt_lf) Constructs a Quadrangle.	
Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3) Constructs a Triangle.	
Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4) Constructs a Quadrangle with the parameter "lt-rt" set to default value "true", this means that the surface is split into a lower-right and an upper-left triangle.	
Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, boolean is_rt_lf) Constructs a Quadrangle.	
Polygon(int numCorn) Constructs Polygon with "numCorn" corners, the "rt_lf" parameter is set to "true"	
Polygon(Polygon toCopy)	
Polygon(Polygon toCopy, boolean toOrder) Copy constructor, assigns the given Polygon to a new created one.	

Method Summary	
boolean	**above**(Coordinate isAbove) Takes a coordinate "isAbove" and checks whether this is above the surface or not.
boolean	**above**(Polygon isAbove, boolean pos) Takes a Polygon "isAbove" and checks whether this is above the calling Polygon or not.
void	**assign**(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3) Assigns a Triangle.
void	**assign**(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4) Assigns a Quadrangle.
void	**assign**(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, boolean is_rt_lf) Assigns a Quadrangle.
void	**assign**(Polygon toCopy) Copy constructor, assigns the given Polygon to a new created one.
boolean	**convex**() Checks whether polygon surface is convex or concave, triangle surface is always convex, for the quadrangle surface the inner side is defined by the right hand rule (-> inner side in direction of cross product c1-c2 x c1-c4), the result also depends on the surface splitting orientation!
boolean	**cut**(Polygon cutting) Cuts a polygon with a second one.

boolean	**equal**(Polygon toCompare) Verifies whether two polygons are equal.
boolean	**order**(boolean out) Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.
boolean	**orderOld**(boolean out) Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.
boolean	**permute**(Permutation toApply) Permutates any Polygon-corner entities with the given permutation instance, toApply.
Coordinate[]	**sides**() Sides of the polygon, effectively the difference of the succeeding corners.
Polygon[]	**split**() Splits any polygons into triangles.
void	**WriteScr**() Writes a Polygon to the Screen,

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

corner

public Coordinate[] **corner**

Corners of the polygon relative to the Origin (number of corners depends from Polygon type defined in "type")

---

topology

public int **topology**

Type of polygon, 4 types are defined as follows (compartible to FEMAP neutral file): **Triangle** - with 3 corners -> ***2*** - with 6 corners -> 3 , **Quadrangle** with 4 corners -> ***4*** - with 8 corners -> 5 , both, Triangle and Quadrangle define a surface, for the triangle in general two different surfaces, being triangles, can be defined.

---

rt_lf

public boolean **rt_lf**

Surface splitting parameter: For a triangle this parameter is irrelevant, for the quadrangle the subdivision into triangles can be performed in two ways, by calling the four corners c1, c2, c3, c4 the splitting line can be defined from c1 to c3 which is default and is called right to left (lower triangle is right, upper is left), in this case rt_lf is set to true, if splitting line is from c2 to c4 the rt_lf parameter is set to false. The subdivision in necessary for the general case, where the four corners are not in one plane and when one wants to know, whether a point lies above or under the surface.

Constructor Detail

Polygon

public **Polygon**()

Default constructor creates structure for triangle

Polygon

public **Polygon**(int numCorn)

Consturcts Polygon with "numCorn" corners, the "rt_lf" parameter is set to "true"

Parameters:

numCorn - defines the number of corners, only 3 and 4 are accepted for the moment, everything else is set to 3

Polygon

public **Polygon**(boolean is_rt_lf)

Constructs a Quadrangle, because only here setting the surface splitting orientation parameter rt_lf makes sense!

Parameters:

is_rt_lf - sets the surface splitting orientation of the quadrangle, if set to true it is composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a upper-right triangle.

Polygon

public **Polygon**(boolean toOrder,
Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3)

Constructs a Triangle. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back corner

toOrder - should polygon be controlled and reorganized when initialized

---

Polygon

public **Polygon**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3)

Constructs a Triangle. Define corners counterclockwise, as given as example in the parameter explanation. Control is performed.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back corner

Polygon

public **Polygon**(boolean toOrder,
Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4)

Constructs a Quadrangle with the parameter "lt-rt" set to default value "true", this means that the surface is split into a lower-right and an upper-left triangle. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

toOrder - should polygon be controlled and reorganized when initialized

---

Polygon

public **Polygon**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4)

Constructs a Quadrangle with the parameter "lt-rt" set to default value "true", this means that the surface is splittet into a lower-right and an upper-left triangle. Define corners counterclockwise, as given as example in the parameter explanation. Control is performed.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

Polygon

public **Polygon**(boolean toOrder,
Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4,
boolean is_rt_lf)

Constructs a Quadrangle. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

is_rt_lf - sets the surface splitting orientation of the quadrangle, if set to true it is composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a upper-right triangle.

toOrder - should polygon be controlled and reorganized when initialized

---

Polygon

public **Polygon**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4,
boolean is_rt_lf)

Constructs a Quadrangle. Define corners counterclockwise, as given as example in the parameter explanation. Control is perfomed.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

is_rt_lf - sets the surface splitting orientation of the quadrangle, if set to true it is composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a upper-right triangle.

Polygon

public **Polygon**(Polygon toCopy,
boolean toOrder)

Copy constructor, assigns the given Polygon to a new created one.

Parameters:

toCopy - Polygon to be assigned to the new created one

Polygon

public **Polygon**(Polygon toCopy)

Method Detail

assign

public void **assign**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3)

Assigns a Triangle. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back corner

---

assign

public void **assign**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4)

Assigns a Quadrangle. The parameter "lt-rt" is kept as defined before. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

---

assign

public void **assign**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4,
boolean is_rt_lf)

Assigns a Quadrangle. Define corners counterclockwise, as given as example in the parameter explanation.

Parameters:

ori - Origin and reference point of the triangle

c1 - e.g. front-left corner

c2 - e.g. front-right corner

c3 - e.g. back-right corner

c4 - e.g. back-left corner

is_rt_lf - sets the surface splitting orientation of the quadrangle, if set to true it is composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a upper-right triangle.

---

assign

public void **assign**(Polygon toCopy)

Copy constructor, assigns the given Polygon to a new created one.

Parameters:

toCopy - Polygon to be assigned to the new created one

---

order

public boolean **order**(boolean out)

Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.

Parameters:

out - Set to true if the routine should report corrections

Returns:

If the polygon was ok, or if successful corrections could be applied, the routine returns true, if the quadrangle is irreparable screwed, the routine returns false.

---

orderOld

public boolean **orderOld**(boolean out)

Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.

Parameters:

out - Set to true if the routine should report corrections

Returns:

If the polygon was ok, or if successful corrections could be applied, the routine returns true, if the quadrangle is irreparable screwed, the routine returns false.

---

permute

public boolean **permute**(Permutation toApply)

Permutates any Polygon-corner entities with the given permutation instance, toApply.

Parameters:

toApply - Permutation instance to be applied to the Polygon-corner entities.

Returns:

True if permutation was successful. False if the length of *toApply* does not correspond to the number of corners.

---

convex

public boolean **convex**()

Checks whether polygon surface is convex or concave, triangle surface is always convex, for the quadrangle surface the inner side is defined by the right hand rule (-> inner side in direction of cross product c1-c2 x c1-c4), the result also depends on the surface splitting orientation!

Returns:

Returns "true" if the surface is found to be convex, "false" otherwise, flat is defined as convex (of course)!

---

above

public boolean **above**(Coordinate isAbove)

Takes a coordinate "isAbove" and checks whether this is above the surface or not. The direction is thereby given by the right hand rule with the surface being oriented counterclockwise with the corners.

Parameters:

isAbove - Coordinates of the point being tested to be above surface

Returns:

Returns "true" if point "isAbove" is above the surface, "false" otherwise.

---

above

public boolean **above**(Polygon isAbove,
boolean pos)

Takes a Polygon "isAbove" and checks whether this is above the calling Polygon or not. The direction is thereby given by the right hand rule with the surface being oriented counterclockwise with the corners. When all corners are above, then the whole surface is above.

Parameters:

isAbove - Polygon being tested to be above surface

pos - If set to false, the orientation of the calling surface can be temporarily changed for this check, therefore the *isAbove* Polygon can be verified to be under the calling polygon!

Returns:

Returns "true" if Polygon "isAbove" is above the surface, "false" otherwise.

---

cut

public boolean **cut**(Polygon cutting)

Cuts a polygon with a second one. The procedure-object is modified in the following way: If the two polygons don't interact at all, no modification, if the Polygon given as parameter only partly cuts trough the calling polygon, the calling polygon is cut as if the parameter polygon was an infinite plane, this is also the case for a completely cutting second polygon! The routine can change the topology of the polygon, a triangle can become a quadrangle. If the resulting polygon would be a pentangle (e.g. when cutting a quadrangle with one corner lying outside), a simplification is performed, the corner lying outside is projected to the cutting plane.

Parameters:

cutting - Polygon which cuts the first polygon. The part of the polygon being above the plane is returned.

Returns:

True if the calling polygon was cut by the parameter-polygon.

---

sides

public Coordinate[] **sides**()

Sides of the polygon, effectively the difference of the succeeding corners.

Returns:

sides of the polygon as pure vectors without origin.

---

split

public Polygon[] **split**()

Splits any polygons into triangles.

Returns:

Triangle fragments of a polygon

---

equal

public boolean **equal**(Polygon toCompare)

Verifies whether two polygons are equal.

WriteScr

public void **WriteScr**()

Writes a Polygon to the Screen,

Overrides:

WriteScr in class Coordinate

4.2.7 The Polyhedron class

jp.go.jnc.tokai.pouchon.geometry

Class Polyhedron

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.geometry.Coordinate
      |
      +--**jp.go.jnc.tokai.pouchon.geometry.Polyhedron**

---

public class **Polyhedron**

extends Coordinate

Describes a cell in space, generally it is a sheared box or wedge, if type is true, then box (4 side prism), otherwise, if type is false, then it is a wedge (3 side prism). Origin is the base of the three sides, describing the cell. dir[0] describes the elevation of the prism, and dir[1,2] the two sides.

Version:

First created 2001.01.18 - version 0

Author:

Manuel Alexandre Pouchon - © JNC Tokai Works (JAPAN)

Field Summary	
Coordinate[	**corner**

]	Corners of the polyhedron relative to the Origin (number of corners depends from Polyhedron type defined in "type")
int	**topology** Type of polyhedron, 6 types are defined as follows (compartible to FEMAP neutral file): **Tetraeder** - with 4 corners -> ***6*** - with 10 corners -> 10 , **Wedge** - with 6 corners -> ***7*** - with 15 corners -> 11 , **Brick** with 8 corners -> ***8*** - with 20 corners -> 12 &hellip In the same layer (lower or upper), define the corners counterclockwise, start with the same corner in the upper layer, as you did in the lower layer (only for Wedge and Brick)

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary

Polyhedron() Empty constructor	
Polyhedron(Cell toTrsf) Constructor which takes an instance of the Cell-class to construct the polyhedron.	
Polyhedron(Coordinate ori, Coordinate[] corners)	
Polyhedron(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4) Constructor for type 6 (Tetraeder) Polyhedron with 4 corners	
Polyhedron(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, Coordinate c5, Coordinate c6) Constructor for type 8 (Wedge) Polyhedron with 6 corners	
Polyhedron(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, Coordinate c5, Coordinate c6, Coordinate c7, Coordinate c8) Constructor for type 8 (Brick) Polyhedron with 8 corners	
Polyhedron(Polyhedron toCopy) Copy constructor	

Method Summary

boolean	**cornerOutside**(Polygon cornAreOut) Verifies whether corners of polygon lie outside the polyhedron, returns boolean
boolean	**cornerOutside**(Polyhedron cornAreOut) Verifies whether corners of polyhedron lie outside the polyhedron, returns boolean
MultiPolyhedron	**cut**(Plane pla) Cutting a polyhedron with a plane.
MultiPolyhedron	**cut**(Polyhedron cell) Cutting a polyhedron with a cell.
MultiPolyhed	**cutOld**(Polyhedron cell)

ron	**Deprecated.**
boolean	**degenerated**() Is the polyhedron degenerated? Only implemented for tetraeder now!
boolean	**disjunct**(Polygon isDisj) Tests wether two polygon is disjunctive with calling polyhedron.
boolean	**disjunct**(Polyhedron isDisj) Tests wether two polyhedrons are disjunctive or not.
boolean	**disjunctOld**(Polygon isDisj) Tests whether two polygon is disjunctive with calling polyhedron.
boolean	**order**(boolean out) Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.
void	**orderOld**(boolean out) Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls the order of corners and also corrects if necessary.
boolean	**permute**(Permutation toApply) Permutates any Polyhedron-corner entities with the given permutation instance, toApply.
MultiPolygon	**polygons**() Returns the *polygonal faces* of a polyhedron, with the faces being oriented towards the cell (cross product of side-vectors).
int[][]	**primList**() Gives back a entity list of the primitive elements (tetraeder) of the defined polyhedrons
Line[]	**sides**() Sides of a Polyhedron
boolean	**within**(Coordinate isIn) Verifies wether coordinate lies within the polyhedron, returns boolean
boolean	**within**(Polygon isIn) Verifies wether polygon lies within the polyhedron, returns boolean
boolean	**within**(Polyhedron isIn) Verifies wether polyhedron lies within the polyhedron, returns boolean
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

corner

public Coordinate[] **corner**

Corners of the polyhedron relative to the Origin (number of corners depends from Polyhedron type defined in "type")

---

topology

public int **topology**

Type of polyhedorn, 6 types are defined as follows (compartible to FEMAP neutral file): **Tetraeder** - with 4 corners -> ***6*** - with 10 corners -> 10 , **Wedge** - with 6 corners -> ***7*** - with 15 corners -> 11 , **Brick** with 8 corners -> ***8*** - with 20 corners -> 12 &hellip In the same layer (lower or upper), define the corners counterclockwise, start with the same corner in the upper layer, as you did in the lower layer (only for Wedge and Brick)

Constructor Detail

Polyhedron

public **Polyhedron**()

Empty constructor

Polyhedron

public **Polyhedron**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4)

Constructor for type 6 (Tetraeder) Polyhedron with 4 corners

Parameters:

ori - Origin of the Tetraeder

c1 - lower-left-front - corner of Tetraeder

c2 - lower-right-front - corner of Teraeder

c3 - lower-back - corner of Tetraeder

c4 - upper - corner of Tetraeder

Polyhedron

public **Polyhedron**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4,
Coordinate c5,
Coordinate c6)

Constructor for type 8 (Wedge) Polyhedron with 6 corners

Parameters:

ori - Origin of the Wedge

c1 - lower-left-front - corner of Wedge

c2 - lower-right-front - corner of Wedge

c3 - lower-back - corner of Wedge

c4 - upper-left-front - corner of Wedge

c5 - upper-right-front - corner of Wedge

c6 - upper-back - corner of Wedge

Polyhedron

public **Polyhedron**(Coordinate ori,
Coordinate c1,
Coordinate c2,
Coordinate c3,
Coordinate c4,
Coordinate c5,
Coordinate c6,
Coordinate c7,
Coordinate c8)

Constructor for type 8 (Brick) Polyhedron with 8 corners

Parameters:

ori - Origin of the Brick

c1 - lower-left-front - corner of Brick

c2 - lower-right-front - corner of Brick

c3 - lower-right-back - corner of Brick

c4 - lower-left-back - corner of Brick

c5 - upper-left-front - corner of Brick

c6 - upper-right-front - corner of Brick

c7 - upper-right-back - corner of Brick

c8 - upper-left-back - corner of Brick

Polyhedron

public **Polyhedron**(Coordinate ori,
Coordinate[] corners)

Polyhedron

public **Polyhedron**(Cell toTrsf)

Constructor which takes an instance of the Cell-class to construct the polyhedron. This is a special simple case of a Wedge, or Brick, with straight surfaces and opposite sides being parallel (of course in case of Wedge the sides are not parallel, only the bottom and ceiling)

Parameters:

toTrsf - Cell to be transformed into the polyhedron.

Polyhedron

public **Polyhedron**(Polyhedron toCopy)

Copy constructor

Parameters:

toCopy - Polyhedron to be copied into the new polyhedron

Method Detail

order

public boolean **order**(boolean out)

Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the order.

Parameters:

out - Set to true if the routine should report corrections

Returns:

If the polygon was ok, or if sucessful corrections could be applied, the routine returns true, if the quadrangle is irreparable screwed, the routine returns false.

---

orderOld

public void **orderOld**(boolean out)

Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls the order of corners and also corrects if necessary. Error messages for degenerated polyhedrons will also be printed.

Parameters:

out - Set to *true* if correction information should be printed to screen, *false* otherwise.

---

permute

public boolean **permute**(Permutation toApply)

Permutates any Polyhedron-corner entities with the given permutation instance, toApply.

Parameters:

toApply - Permutation instance to be applied to the Polyhedron-corner entities.

Returns:

True if permutation was successful. False if the length of *toApply* does not correspond to the number of corners.

---

within

public boolean **within**(Coordinate isIn)

Verifies wether coordinate lies within the polyhedron, returns boolean

Parameters:

isIn - Coordinate to be verified

Returns:

true when **isIn** is contained in the polyhedron, false otherwise

---

within

public boolean **within**(Polygon isIn)

Verifies whether polygon lies within the polyhedron, returns boolean

Parameters:

isIn - Polygon to be verified

Returns:

true when **isIn** is contained in the polyhedron, false otherwise

---

within

public boolean **within**(Polyhedron isIn)

Verifies whether polyhedron lies within the polyhedron, returns boolean

Parameters:

isIn - Polyhedron to be verified

Returns:

true when **isIn** is contained in the polyhedron, false otherwise

---

cornerOutside

public boolean **cornerOutside**(Polyhedron cornAreOut)

Verifies wether corners of polyhedron lie outside the polyhedron, returns boolean

Parameters:

cornAreOut - Polyhedron whose corners are to be verified

Returns:

true when all corners of **cornsAreOut** are outside the polyhedron, false otherwise

---

cornerOutside

public boolean **cornerOutside**(Polygon cornAreOut)

Verifies whether corners of polygon lie outside the polyhedron, returns boolean

Parameters:

cornAreOut - Polygon whose corners are to be verified

Returns:

true when all corners of **cornsAreOut** are outside the polyhedron, false otherwise

---

disjunct

public boolean **disjunct**(Polyhedron isDisj)

Tests whether two polyhedrons are disjunctive or not.

Parameters:

isDisj - second polyhedron to be tested against the first one (the object-instance).

Returns:

true if the two polyhedrons are disjunctive, false otherwise.

---

disjunct

public boolean **disjunct**(Polygon isDisj)

Tests whether two polygon is disjunctive with calling polyhedron.

Parameters:

isDisj - polygon to be tested against the polyhedron (the object-instance).

Returns:

true if the polygon is disjunct, false otherwise.

---

disjunctOld

public boolean **disjunctOld**(Polygon isDisj)

Tests whether two polygon is disjunctive with calling polyhedron.

Parameters:

isDisj - polygon to be tested against the polyhedron (the object-instance).

Returns:

true if the polygon is disjunctive, false otherwise.

---

polygons

public MultiPolygon **polygons**()

Returns the *polygonal faces* of a polyhedron, with the faces being oriented towards the cell (cross product of side-vectors).

Returns:

multiple polygon containing all faces of the polyhedron.

---

degenerated

public boolean **degenerated**()

Is the polyhedron degenerated? Only implemented for tetraeder now!

Returns:

true if tetraeder is degenerated

---

cut

public MultiPolyhedron **cut**(Plane pla)

Cutting a polyhdron with a plane.

Parameters:

pla - plane cutting the polyhedon

Returns:

list of polyhedrons representing the cut volume

---

cutOld

public MultiPolyhedron **cutOld**(Polyhedron cell)

Deprecated.

Cutting a polyhdron with a cell.

Parameters:

cell - cell cutting the polyhedon

Returns:

list of polyhedrons representing the cutten volume

---

cut

public MultiPolyhedron **cut**(Polyhedron cell)

Cutting a polyhdron with a cell.

Parameters:

cell - cell cutting the polyhedon

Returns:

list of polyhedrons representing the cut volume

---

primList

public int[][] **primList**()

Gives back a entity list of the primitive elements (tetraeder) of the defined polyhedrons

Returns:

entity list in the form [elements][tetraeder entities], each row contains a different element, each element is described by the number of the original corner-points of the polyhedron.

---

sides

public Line[] **sides**()

Sides of a Polyhedron

Returns:

Sides of the polyhedron as Line-Type in a list

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

4.2.8 The Sphere class

jp.go.jnc.tokai.pouchon.geometry

Class Sphere

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.geometry.Coordinate
       |
       +--**jp.go.jnc.tokai.pouchon.geometry.Sphere**

Direct Known Subclasses:

CellSphereData

---

public class **Sphere**

extends Coordinate

Defines sphere in R3.

Field Summary	
double	**r** Radius of the sphere

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Sphere() Constructor with no argument.	
Sphere(Coordinate loc, double radius) Constructor with the coordinate and the radius as argument.	
Sphere(double x0, double x1, double x2, double radius) Constructor with the three coordinates and the radius as argument.	
Sphere(Sphere toCopy) Copy-constructor with a sphere as argument	

Method Summary	
void	**assign**() Assigns degenerated sphere (Radius = 0.0) in the origin
void	**assign**(Coordinate loc, double radius) Assign operator with the coordinate and the radius as argument.
void	**assign**(double x0, double x1, double x2, double radius) Assigns sphere with center-coordinates and radius.
void	**assign**(Sphere toCopy) Copy-operator with a sphere as argument
void	**WritelnScr**()
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

r

public double **r**

Radius of the sphere

Constructor Detail

Sphere

public **Sphere**(double x0,
double x1,
double x2,
double radius)

Constructor with the three coordinates and the radius as argument.

Parameters:

xi - three coordinates of the sphere-center

r - radius of the sphere

Sphere

public **Sphere**()

Constructor with no argument. Constructs a degenerated sphere (radius = 0.0) in the origin.

Sphere

public **Sphere**(Coordinate loc,
double radius)

Constructor with the coordinate and the radius as argument.

Parameters:

loc - coordinate of the sphere-center location

r - radius of the sphere

Sphere

public **Sphere**(Sphere toCopy)

Copy-constructor with a sphere as argument

Parameters:

toCopy - Sphere which is subject to be copied

Method Detail

assign

public void **assign**(double x0,
double x1,
double x2,
double radius)

Assigns sphere with center-coordinates and radius.

Parameters:

xi - three coordinates of the sphere-center

r - radius of the sphere

---

assign

public void **assign**()

Assigns degenerated sphere (Radius = 0.0) in the origin

---

assign

public void **assign**(Coordinate loc,
double radius)

Assign operator with the coordinate and the radius as argument.

Parameters:

loc - coordinate of the sphere-center location

r - radius of the sphere

---

assign

public void **assign**(Sphere toCopy)

Copy-operator with a sphere as argument

Parameters:

toCopy - Sphere which is subject to be copied

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

WritelnScr

public void **WritelnScr**()

4.2.9 The Face Class

jp.go.jnc.tokai.pouchon.geometry

Class Face

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.geometry.Face**

---

public class **Face**

extends java.lang.Object

Field Summary	
Coordinate[]	**dir**
Coordinate	**orig**

Constructor Summary	
Face()	
Face(Coordinate origin, Coordinate dir0, Coordinate dir1, int faceType)	
Face(Face F)	

Method Summary	
void	**assign**(Coordinate origin, Coordinate dir0, Coordinate dir1, int faceType)
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

orig

public Coordinate **orig**

---

dir

public Coordinate[] **dir**

Constructor Detail

Face

public **Face**()

Face

public **Face**(Coordinate origin,
Coordinate dir0,
Coordinate dir1,
int faceType)

Face

public **Face**(Face F)

Method Detail

assign

public void **assign**(Coordinate origin,
Coordinate dir0,
Coordinate dir1,
int faceType)

WriteScr

public void **WriteScr**()

4.2.10 The Matrix class

jp.go.jnc.tokai.pouchon.geometry

Class Matrix

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.geometry.Matrix**

---

public class **Matrix**

extends java.lang.Object

Representing a Matrix; mainly for vector transformations

Version:

1.00 2001/01/10

Author:

Manuel Alexandre POUCHON - International Fellow of JNC Tokai Works (JAPAN)

Field Summary

double[][]	**x**

Constructor Summary

Matrix()	

Matrix(Coordinate v1, Coordinate v2, Coordinate v3)	
Matrix(double x11, double x12, double x13, double x21, double x22, double x23, double x31, double x32, double x33)	
Matrix(Matrix M)	

Method Summary

void	**assign**(Coordinate v1, Coordinate v2, Coordinate v3)
void	**assign**(double x11, double x12, double x13, double x21, double x22, double x23, double x31, double x32, double x33)
void	**assign**(Matrix M)
void	**inv**()
void	**inv**(Matrix mIn)
void	**mult**(Matrix M, boolean first)
void	**mult**(Matrix M0, Matrix M2)
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

x

public double[][] **x**

Constructor Detail

Matrix

public **Matrix**(double x11,
double x12,
double x13,
double x21,
double x22,
double x23,

double x31,
double x32,
double x33)

Matrix

public **Matrix**()

Matrix

public **Matrix**(Coordinate v1,
Coordinate v2,
Coordinate v3)

Matrix

public **Matrix**(Matrix M)

Method Detail

assign

public void **assign**(double x11,
double x12,
double x13,
double x21,
double x22,
double x23,
double x31,
double x32,
double x33)

---

assign

public void **assign**(Coordinate v1,
Coordinate v2,
Coordinate v3)

---

assign

public void **assign**(Matrix M)

---

mult

public void **mult**(Matrix M,
boolean first)

---

mult

public void **mult**(Matrix M0,
Matrix M2)

---

inv

public void **inv**()

---

inv

public void **inv**(Matrix mIn)

WriteScr

public void **WriteScr**()

4.2.11 The MultiLine class

jp.go.jnc.tokai.pouchon.geometry

Class MultiLine

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.geometry.MultiLine**

---

public class **MultiLine**

extends java.lang.Object

Class which represents multiple lines, these can be a random collection of single lines, possible connections at the endpoints are detected and declared.

Field Summary	
`boolean`	**circular** Is true if the multiple line connects to one single closed path.
`int[][]`	**connect** Gives a list for each point, which describes possible connections to other points.
`int`	**numPaths** Is true if the multiple line connects to one single path.
`int[]`	**path** Possible path along the lines.
Coordinate[ ]	**points** Collection of single points describing the multiple line.

Constructor Summary	
MultiLine()	
MultiLine(Line[] in) Generates a multiple line from a list of single lines, automatic endpoint-connection detection.	

Method Summary	
void	**assign**(MultiLine toCopy)
boolean	**cut**(Polygon pol, Plane pla) Cuts a polygons with a plane and assigns the cutting multiple-line.
boolean	**cut**(Polygon P1, Polygon P2)
void	**WriteScr**() Writes a multiple line to the screen.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

points

public Coordinate[] **points**

Collection of single points describing the multiple line.

---

connect

public int[][] **connect**

Gives a list for each point, which describes possible connections to other points.

---

path

public int[] **path**

Possible path along the lines. All lines should be included. Single connected multiple lines start and end with "-1", if there are several line-fragments, this is then indicated with additional "-1" entities.

---

numPaths

public int **numPaths**

Is true if the multiple line connects to one single path.

---

circular

public boolean **circular**

Is true if the multiple line connects to one single closed path.

Constructor Detail

MultiLine

public **MultiLine**()

MultiLine

public **MultiLine**(Line[] in)

Generates a multiple line from a list of single lines, automatic endpoint-connection detection.

Method Detail

assign

public void **assign**(MultiLine toCopy)

---

cut

public boolean **cut**(Polygon pol,
Plane pla)

Cuts a polygons with a plane and assigns the cutting multiple-line.

Parameters:

pol - polygon to cut

pla - plane to cut

---

cut

public boolean **cut**(Polygon P1,
Polygon P2)

WriteScr

public void **WriteScr**()

Writes a multiple line to the screen. It declares all points, all connections between them and the detected path.

4.2.12 The MultyPolygon class

jp.go.jnc.tokai.pouchon.geometry

Class MultiPolygon

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.geometry.MultiPolygon**

---

public class **MultiPolygon**

extends java.lang.Object

Field Summary	
boolean	**closed**
int[][]	**connect** Connection between the polygons, only perfect connections (coincidence of a side) are listed.
boolean[]	**fullyConnected**
Polygon[]	**polygons** List of independent polygons

Constructor Summary	
MultiPolygon()	
MultiPolygon(Polygon[] in)	

Method Summary	
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

polygons

public Polygon[] **polygons**

List of independent polygons

---

connect

public int[][] **connect**

Connection between the polygons, only perfect connections (coincidence of a side) are listed.

---

fullyConnected

public boolean[] **fullyConnected**

---

closed

public boolean **closed**

Constructor Detail

MultiPolygon

public **MultiPolygon**()

MultiPolygon

public **MultiPolygon**(Polygon[] in)

Method Detail

WriteScr

public void **WriteScr**()

4.2.13 The MultiPolyhedron class

jp.go.jnc.tokai.pouchon.geometry

Class MultiPolyhedron

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.geometry.MultiPolyhedron**

---

public class **MultiPolyhedron**

extends java.lang.Object

Class representing a collection of polyhedrons, originally created to reduce wedge and brick polyhedrons to tetraeder.

Version:

First created 2001.03.09 - Version 0

Author:

Manuel Alexandre Pouchon - © JNC Tokai Works (JAPAN)

Field Summary

Polyhedron[]	**polyhedrons** Collection of polyhedrons.

Constructor Summary

MultiPolyhedron()	
MultiPolyhedron(MultiPolyhedron mph, Polyhedron ph)	
MultiPolyhedron(Polyhedron ph1)	
MultiPolyhedron(Polyhedron[] ph)	
MultiPolyhedron(Polyhedron ph1, Polyhedron ph2)	

MultiPolyhedron(Polyhedron ph1, Polyhedron ph2, Polyhedron ph3)	
MultiPolyhedron(Polyhedron ph1, Polyhedron ph2, Polyhedron ph3, Polyhedron ph4)	
MultiPolyhedron(Polyhedron ph1, Polyhedron ph2, Polyhedron ph3, Polyhedron ph4, Polyhedron ph5)	
MultiPolyhedron(Polyhedron ph1, Polyhedron ph2, Polyhedron ph3, Polyhedron ph4, Polyhedron ph5, Polyhedron ph6)	

Method Summary

void	**graphicFile**(java.io.File fileName, double zoom) Writes the multiple polyhedorn to a file with the format understood by the (external) Java-Applet ThreeD.java
void	**reduceTopo**(Polyhedron toReduce) Gives back a list of polyhedrons (tetraeder) from the reduction of a single polyhedron of any defined type.
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

polyhedrons

public Polyhedron[] **polyhedrons**

Collection of polyhedrons.

Constructor Detail

MultiPolyhedron

public **MultiPolyhedron**()

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1,
Polyhedron ph2)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1,
Polyhedron ph2,
Polyhedron ph3)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1,
Polyhedron ph2,
Polyhedron ph3,
Polyhedron ph4)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1,
Polyhedron ph2,
Polyhedron ph3,
Polyhedron ph4,
Polyhedron ph5)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron ph1,
Polyhedron ph2,
Polyhedron ph3,
Polyhedron ph4,
Polyhedron ph5,
Polyhedron ph6)

MultiPolyhedron

public **MultiPolyhedron**(Polyhedron[] ph)

MultiPolyhedron

public **MultiPolyhedron**(MultiPolyhedron mph,
Polyhedron ph)

Method Detail

reduceTopo

public void **reduceTopo**(Polyhedron toReduce)

Gives back a list of polyhedrons (tetraeder) from the reduction of a single polyhedron of any defined type.

Parameters:

toReduce - Polyhedron from which the tetraeder list is derived.

See Also:

Polyhedron.primList()

WriteScr

public void **WriteScr**()

---

graphicFile

public void **graphicFile**(java.io.File fileName,
double zoom)

Writes the multiple polyhedron to a file with the format understood by the (external) Java-Applet ThreeD.java

Parameters:

fileName - file name which should be used for the object to be stored.

double - a zooming factor to all coordinates

4.3 The math package

4.3.1 The permutation class

jp.go.jnc.tokai.pouchon.math

Class Permutation

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.math.Permutation**

---

public class **Permutation**

extends java.lang.Object

Class describing permutations, eg for initialisation of geometrical bodies, where an input order of points may want to be changed because of possible problems.

Field Summary	
`int[]`	**order**

Constructor Summary	
Permutation(int num) Basic initialisation with a trivial order of entities.	

Permutation(Polygon num) Basic initialisation with a trivial order of entities.	
Permutation(Polyhedron num) Basic initialisation with a trivial order of entities.	

Method Summary

boolean	**pointOrder**(Polygon toOrder, boolean out) Takes a polygon and checks wether it is valid or not.
boolean	**pointOrder**(Polyhedron toOrder, boolean out) Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls the order of corners and also corrects if necessary.
boolean	**transpose**(int t1, int t2) Tranposes two entities in a permutation-vector.
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

order

public int[] **order**

Constructor Detail

Permutation

public **Permutation**(int num)

Basic initialisation with a trivial order of entities. [0,1,2,3,4, ... num]

Parameters:

num - Number of entities to initialise.

Permutation

public **Permutation**(Polygon num)

Basic initialisation with a trivial order of entities. [0,1,2,3,4, ... Num] where Num is derived from the given Polygon num.

Parameters:

num - Polygon with which corner-number the permutation should be initialised.

Permutation

public **Permutation**(Polyhedron num)

Basic initialisation with a trivial order of entities. [0,1,2,3,4, ... Num] where Num is derived from the given Polyhedorn num.

Parameters:

num - Polyhedron with which corner-number the permutation should be initialised.

Method Detail

transpose

public boolean **transpose**(int t1,
int t2)

Tranposes two entities in a permutation-vector.

Parameters:

t1 - first element to be transposed

t2 - second element to be transposed

Returns:

True if the two elements existed, false if one (or both) of the elements were ouside the range.

---

pointOrder

public boolean **pointOrder**(Polygon toOrder,
boolean out)

Takes a polygon and checks wether it is valid or not. If possible it makes permutations to validate. The instance is then modified to represent this correction.

Parameters:

toOrder - Input polygon to be checked for validity

out - Set to true if permutation informations should be printed

Returns:

True if Polygon is valid, if some irrecoverable errors apeared, the routine retruns false!

---

pointOrder

public boolean **pointOrder**(Polyhedron toOrder,
boolean out)

Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls the order of corners and also corrects if necessary. Error messages for degenerated polyhedrons will also be printed.

Parameters:

out - Set to *true* if correction information should be printend to screen, *false* otherwise.

WriteScr

public void **WriteScr**()

4.4 The finitele package

4.4.1 The element class

jp.go.jnc.tokai.pouchon.finitele

Class Element

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.finitele.Element**

Direct Known Subclasses:

ElementCoordinate

---

public class **Element**

extends java.lang.Object

Field Summary	
`long`	**id** id of the element
`long[]`	**nodeId**
`int`	**topology** Topology of the element, same topology-notation like for the geometry-class Polyhedron!

Constructor Summary	
Element()	
Element(Element toCopy) Copy constructor.	
Element(long idIn, Node node0, Node node1, Node node2, Node node3) Constructor for **Tetraeder** with 4 nodes.	
Element(long idIn, Node node0, Node node1, Node node2, Node node3, Node node4, Node node5) Constructor for **Wedge** with 6 nodes.	
Element(long idIn, Node node0, Node node1, Node node2, Node node3, Node node4, Node node5, Node node6, Node node7) Constructor for **Brick** with 8 nodes.	

Method Summary	
void	**assign**(Element toCopy) Assigns a copy to the existing element.
void	**assign**(long idIn, Node node0, Node node1, Node node2, Node node3) Assign operator for **Tetraeder** with 4 nodes.
void	**assign**(long idIn, Node node0, Node node1, Node node2, Node node3, Node node4, Node node5) Assign operator for **Wedge** with 6 nodes.
void	**assign**(long idIn, Node node0, Node node1, Node node2, Node node3, Node node4, Node node5, Node node6, Node node7) Assign operator for **Brick** with 8 nodes.
Coordinate[]	**coordinates**(Node[] listOfNodes) Procedure to find out the coordinates of an element.
long[]	**ids**() Gives back the ids contained in an Element, the ids are normally contained in a list of 20 long-numbers.
void	**modify**(ElementCoordinate modification) modifies the object instance to the modification
Node[]	**modify**(ElementCoordinate modification, ObjectCounter ids) modifies the object instance to the modification
Node[]	**nodes**(Node[] listOfNodes) Procedure to find out the nodes of an element.
void	**putId**(long newId, int numNewId) Puts a new Id in the element.
void	**WriteScr**() Writes an element to the screen.

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

id

public long **id**

id of the element

---

topology

public int **topology**

Topology of the element, same topology-notation like for the geometry-class Polyhedron!

See Also:

Polyhedron

nodeId

public long[] **nodeId**

Constructor Detail

Element

public **Element**()

Element

public **Element**(long idIn,
Node node0,
Node node1,
Node node2,
Node node3)

Constructor for **Tetraeder** with 4 nodes.

Parameters:

idIn - identity of this tetraeder

$node_{i=0...3}$ - nodes needed for initialization of tetraeder.

Element

public **Element**(long idIn,
Node node0,
Node node1,
Node node2,
Node node3,
Node node4,
Node node5)

Constructor for **Wedge** with 6 nodes.

Parameters:

idIn - identity of this wedge

$node_{i=0...5}$ - nodes needed for initialization of wedge.

Element

public **Element**(long idIn,
Node node0,
Node node1,
Node node2,
Node node3,
Node node4,
Node node5,
Node node6,
Node node7)

Constructor for **Brick** with 8 nodes.

Parameters:

idIn - identity of this Brick

$node_{i=0...7}$ - nodes needed for initialization of brick.

Element

public **Element**(Element toCopy)

Copy constructor.

Parameters:

toCopy - Element being copied to new Element.

Method Detail

assign

public void **assign**(long idIn,
Node node0,
Node node1,
Node node2,
Node node3)

Assign operator for **Tetraeder** with 4 nodes.

Parameters:

idIn - identity of this tetraeder

$node_{i=0...3}$ - nodes needed for initialization of tetraeder.

---

assign

public void **assign**(long idIn,
Node node0,
Node node1,
Node node2,
Node node3,
Node node4,
Node node5)

Assign operator for **Wedge** with 6 nodes.

Parameters:

idIn - identity of this wedge

$node_{i=0...5}$ - nodes needed for initialization of wedge.

---

assign

public void **assign**(long idIn,
Node node0,
Node node1,

Node node2,
Node node3,
Node node4,
Node node5,
Node node6,
Node node7)

Assign operator for **Brick** with 8 nodes.

Parameters:

idIn - identity of this Brick

$node_{i=0...7}$ - nodes needed for initialization of brick.

---

assign

public void **assign**(Element toCopy)

Assigns a copy to the existing element.

Parameters:

toCopy - Element being copied to element.

---

coordinates

public Coordinate[] **coordinates**(Node[] listOfNodes)

Procedure to find out the coordinates of an element. The procedure seeks all the nodes given in a list if they are contained in the element and returns the corresponding list of coordinates. The order of the returned coordinates corresponds to the id order.

Parameters:

listOfNodes - List of nodes where the element should be contained.

Returns:

list of Coordinates of the element-nodes.

---

nodes

public Node[] **nodes**(Node[] listOfNodes)

Procedure to find out the nodes of an element. The procedure seeks all the nodes given in a list if they are contained in the element and returns the corresponding list of nodes. The order of the returned nodes corresponds to the id order.

Parameters:

listOfNodes - List of nodes where the element should be contained.

Returns:

list of Nodes contained in Element.

---

modify

public Node[] **modify**(ElementCoordinate modification,
ObjectCounter ids)

modifies the object instance to the modification

Parameters:

modification - the modification to be applied, negative node index means the creation of a new node!

ids - Object counter

Returns:

nodes with new ids, the old ids are set to 0!

---

modify

public void **modify**(ElementCoordinate modification)

modifies the object instance to the modification

Parameters:

modification - the modification to be applied, negative node index means the cration of a new node!

---

ids

public long[] **ids**()

Gives back the ids contained in an Element, the ids are normally contained in a list of 20 long-numbers. But only 4, 6 or 8 of these are actually used. The information is returned depending on the topology.

Returns:

List of used ids

---

putId

public void **putId**(long newId,
int numNewId)

Puts a new Id in the element. Because each element can contain 20 ids, this procedure is needed to put the new Id at the right place.

Parameters:

newId - The new id to be set in the existing element

numNewId - The number of the new id.

WriteScr

public void **WriteScr**()

Writes an element to the screen.

4.4.2 The ElementList class

jp.go.jnc.tokai.pouchon.finitele

Class ElementList

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.finitele.ElementList**

---

public class **ElementList**

extends java.lang.Object

Field Summary	
java.util.Vector	**elements**

Constructor Summary	
ElementList()	

Method Summary	
void	**GetElements**(SphereMesh newElements)
void	**WriteFile**(java.io.PrintWriter os)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

elements

public java.util.Vector **elements**

Constructor Detail

ElementList

public **ElementList**()

Method Detail

GetElements

public void **GetElements**(SphereMesh newElements)

WriteFile

public void **WriteFile**(java.io.PrintWriter os)
throws java.io.IOException

4.4.3 The node class

jp.go.jnc.tokai.pouchon.finitele

Class Node

java.lang.Object
|
+--jp.go.jnc.tokai.pouchon.geometry.Coordinate
|
+--**jp.go.jnc.tokai.pouchon.finitele.Node**

---

public class **Node**

extends Coordinate

Field Summary	
long	**id**
java.util.Vector	**inEle**

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
Node()	
Node(long idIn, Coordinate loc)	
Node(long idIn, double x0, double x1, double x2)	
Node(Node toCopy)	

Method Summary	
void	**assign**(long idIn, Coordinate loc)

void	**assign**(long idIn, double x0, double x1, double x2)
void	**assign**(Node nToAss)
void	**WriteScr**()

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

id

public long **id**

---

inEle

public java.util.Vector **inEle**

Constructor Detail

Node

public **Node**(Node toCopy)

Node

public **Node**(long idIn,
Coordinate loc)

Node

public **Node**()

Node

public **Node**(long idIn,
double x0,

double x1,
double x2)

Method Detail

assign

public void **assign**(long idIn,
Coordinate loc)

---

assign

public void **assign**(long idIn,
double x0,
double x1,
double x2)

---

assign

public void **assign**(Node nToAss)

WriteScr

public void **WriteScr**()

Overrides:

WriteScr in class Coordinate

4.4.4 The NodeList class

jp.go.jnc.tokai.pouchon.finitele

Class NodeList

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.finitele.NodeList**

---

public class **NodeList**

extends java.lang.Object

Field Summary

java.util.Vector	**nodes**

Constructor Summary

NodeList()	

Method Summary	
Void	**GetNodes**(SphereMesh newNodes)
Void	**WriteFile**(java.io.PrintWriter os)

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

nodes

public java.util.Vector **nodes**

Constructor Detail

NodeList

public **NodeList**()

Method Detail

GetNodes

public void **GetNodes**(SphereMesh newNodes)

WriteFile

public void **WriteFile**(java.io.PrintWriter os)
throws java.io.IOException

4.4.5 The Entities class

jp.go.jnc.tokai.pouchon.finitele

Class Entities

java.lang.Object
|
+--**jp.go.jnc.tokai.pouchon.finitele.Entities**

---

public class **Entities**

extends java.lang.Object

General Class for representing a number of entities, for example the node-entities of a element, or an element collection!

Field Summary	
`long[]`	**id** List of entities.

Constructor Summary	
Entities`(long[] e)` Constructor for n entities	
Entities`(long e0, long e1)` Constructor for 2 entities	
Entities`(long e0, long e1, long e2)` Constructor for 3 entities	
Entities`(long e0, long e1, long e2, long e3)` Constructor for 4 entities	
Entities`(long e0, long e1, long e2, long e3, long e4)` Constructor for 5 entities	
Entities`(long e0, long e1, long e2, long e3, long e4, long e5)` Constructor for 6 entities	
Entities`(long e0, long e1, long e2, long e3, long e4, long e5, long e6)` Constructor for 7 entities	
Entities`(long e0, long e1, long e2, long e3, long e4, long e5, long e6, long e7)` Constructor for 8 entities	

Method Summary	
`boolean`	**permute**(Permutation `toApply`) Permutates any entity-list with the given permutation instance, toApply.

Methods inherited from class java.lang.Object
`clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

Field Detail

id

public long[] **id**

List of entities.

Constructor Detail

Entities

public **Entities**(long e0,
long e1)

Constructor for 2 entities

Parameters:

$e_{i=0...1}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2)

Constructor for 3 entities

Parameters:

$e_{i=0...2}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2,
long e3)

Constructor for 4 entities

Parameters:

$e_{i=0...3}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2,
long e3,
long e4)

Constructor for 5 entities

Parameters:

$e_{i=0...4}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2,
long e3,
long e4,

long e5)

Constructor for 6 entities

Parameters:

$e_{i=0...5}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2,
long e3,
long e4,
long e5,
long e6)

Constructor for 7 entities

Parameters:

$e_{i=0...6}$ - list of entities

Entities

public **Entities**(long e0,
long e1,
long e2,
long e3,
long e4,
long e5,
long e6,
long e7)

Constructor for 8 entities

Parameters:

$e_{i=0...7}$ - list of entities

Entities

public **Entities**(long[] e)

Constructor for n entities

Parameters:

e[] - list of entities

Method Detail

permute

public boolean **permute**(Permutation toApply)

Permutates any entity-list with the given permutation instance, toApply.

Parameters:

toApply - Permutation instance to be applied to the entity-list.

Returns:

True if permutation was successful. False if the length of *toApply* does not correspond to the number of entities.

4.4.6 The ElementCoordinate class

jp.go.jnc.tokai.pouchon.finitele

Class ElementCoordinate

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.finitele.Element
      |
      +--**jp.go.jnc.tokai.pouchon.finitele.ElementCoordinate**

---

public class **ElementCoordinate**

extends Element

Additional Class to **Element**. The class **Element** only contains the Node-Number information, **id**. This class completes this information with the corresponding coordinates.

Field Summary	
Coordinate[]	**coords**

Fields inherited from class jp.go.jnc.tokai.pouchon.finitele.Element
id, nodeId, topology

Constructor Summary	
ElementCoordinate(Element toComplete, Node[] nodeList) Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes, contained in the Element, are specified.	
ElementCoordinate(Element toComplete, NodeList toLookIn) Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes, contained in the Element, are specified.	
ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4) Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.	
ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4, Node node5, Node node6) Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.	

ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4, Node node5, Node node6, Node node7, Node node8) Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.	

Method Summary	
ElementCoordinate[]	**adjust**(Polyhedron cutting) Routine to cut an Element with a polyhedron.
Node[]	**nodes**()
Polyhedron	**polyhedron**() Takes an element with the coordinate-information and froms the geometrical object Polyhedron.

Methods inherited from class jp.go.jnc.tokai.pouchon.finitele.Element
assign, assign, assign, assign, coordinates, ids, modify, modify, nodes, putId, WriteScr

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

coords

public Coordinate[] **coords**

Constructor Detail

ElementCoordinate

public **ElementCoordinate**(Element toComplete,
NodeList toLookIn)

Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes, contained in the Element, are specified.

Parameters:

toComplete - the Element information which is going to be completed with the corresponding coordinates.

toLookIn - the list of nodes, where the information (here, the coordinates) about the single nodes is searched.

ElementCoordinate

public **ElementCoordinate**(Element toComplete,
Node[] nodeList)

Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes, contained in the Element, are specified.

Parameters:

toComplete - the Element information which is going to be completed with the corresponding coordinates.

nodeList - the list of nodes, where the information (here, the coordinates) about the single nodes is searched.

ElementCoordinate

public **ElementCoordinate**(long id,
Node node1,
Node node2,
Node node3,
Node node4)

Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.

Parameters:

node - the list of nodes, where the information (here, the coordinates) about the single nodes is searched.

ElementCoordinate

public **ElementCoordinate**(long id,
Node node1,
Node node2,
Node node3,
Node node4,
Node node5,
Node node6)

Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.

Parameters:

node - the list of nodes, where the information (here, the coordinates) about the single nodes is searched.

ElementCoordinate

public **ElementCoordinate**(long id,
Node node1,
Node node2,
Node node3,
Node node4,
Node node5,
Node node6,
Node node7,
Node node8)

Constructor of ElementCoordinate, it takes an a NodeList representing the element in the correct order.

Parameters:

node - the list of nodes, where the information (here, the coordinates) about the single nodes is searched.

Method Detail

polyhedron

public Polyhedron **polyhedron**()

Takes an element with the coordinate-information and forms the geometrical object Polyhedron.

Returns:

polyhedron from the infomation on the element

---

nodes

public Node[] **nodes**()

---

adjust

public ElementCoordinate[] **adjust**(Polyhedron cutting)

Routine to cut an Element with a polyhedron. It uses a cutting procedure of the Polyhedron-class. New indices are marked with a negative index, for recognition! Starting from 1. These are not the definitive indices, but have to be initialized later.

Parameters:

cutting - cell which defines the cutting limits.

Returns:

list of elements with the coordinates, representing the cut-volume. This might contain new elements and nodes. These must be written to the general element- and node-list!

See Also:

Polyhedron.cut(Polyhedron)

4.4.7 The ObjectIds class

jp.go.jnc.tokai.pouchon.finitele

Class ObjectIds

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.finitele.ObjectIds**

---

public class **ObjectIds**

extends java.lang.Object

Element and Node ids, the next vacant id is specified as double value, a vector specifies the vacant ids, if elements or nodes have been erased.

Field Summary	
int	**nextVacEle** The next vacant Element in the list
int	**nextVacNode** The next vacant Node in the list
java.util.Vector	**vacEle** Vector of vacant element entities, which have been released due to erasing of an element.
java.util.Vector	**vacNode** Vector of vacant node entities, which have been released due to erasing of an elements.

Constructor Summary	
ObjectIds()	

Method Summary	
int[]	**actIds**() Gives back the actual ids (last initialized ids) of the node (as first integer) and the element (as second integer) collection.
boolean	**killEle**(int toKill) When an element is deleted, its id is given free to be used again, therefore it is added to the vacEle list, or, if it is the last element in the list, the counter nextVacEle is just lowered.
boolean	**killNode**(int toKill) When an node is deleted, its id is given free to be used again, therefore it is added to the vacNode list, or, if it is the last element in the list, the counter nextVacNode is just lowered.
int	**nextEle**() Gives back the next possible id and increases the **nextVacEle** entity or, if possible, deletes this entity from the vacant list **vacEle**.
int	**nextEle**(int num) Gives back the next new id (does not take any elements from the vacant list **vacEle**!) and increases the **nextVacEle** entity by num, useful for the creation of a new Object where a large, connected amount of elements is created.
int	**nextNode**() Gives back the next possible id and increases the **nextVacNode** entity or, if possible, deletes this entity from the vacant list **vacEle**.
int	**nextNode**(int num) Gives back the next new id (does not take any elements from the vacant list **vacNode**!) and increases the **nextVacNode** entity by num, useful for the creation of a new Object where a large, connected amount of nodes is created.
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

nextVacEle

public int **nextVacEle**

The next vacant Element in the list

---

nextVacNode

public int **nextVacNode**

The next vacant Node in the list

---

vacEle

public java.util.Vector **vacEle**

Vector of vacant element entities, which have been released due to erasing of an element.

---

vacNode

public java.util.Vector **vacNode**

Vector of vacant node entities, which have been released due to erasing of an elements.

Constructor Detail

ObjectIds

public **ObjectIds**()

Method Detail

nextEle

public int **nextEle**()

Gives back the next possible id and increases the **nextVacEle** entity or, if possible, deletes this entity from the vacant list **vacEle**.

---

nextEle

public int **nextEle**(int num)

Gives back the next new id (does not take any elements from the vacant list **vacEle**!) and increases the **nextVacEle** entity by num, useful for the creation of a new Object where a large, connected amount of elements is created.

---

killEle

public boolean **killEle**(int toKill)

When an element is deleted, its id is given free to be used again, therefore it is added to the vacEle list, or, if it is the last element in the list, the counter nextVacEle is just lowered.

Parameters:

toKill - id of the element to be killed, this id is between 0 and 9999, it is only the element part of the complete id and does not contain the object-number.

Returns:

gives back true, if the element could be erased, false, if the element id given was bigger than **nextVacEle** or was contained in the list of already killed elements.

---

nextNode

public int **nextNode**()

Gives back the next possible id and icreases the **nextVacNode** entity or, if possible, deletes this entity from the vacant list **vacEle**.

---

nextNode

public int **nextNode**(int num)

Gives back the next new id (does not take any elements from the vacant list **vacNode**!) and increases the **nextVacNode** entity by num, useful for the creation of a new Object where a large, connected amount of nodes is created.

---

killNode

public boolean **killNode**(int toKill)

When an node is deleted, its id is given free to be used again, therefore it is added to the vacNode list, or, if it is the last element in the list, the counter nextVacNode is just lowered.

Parameters:

toKill - id of the element to be killed, this id is between 0 and 9999, it is only the element part of the complete id and does not contain the object-number.

Returns:

gives back true, if the element could be erased, false, if the element id given was bigger than **nextVacNode** or was contained in the list of already killed nodes

---

actIds

public int[] **actIds**()

Gives back the actual ids (last initialized ids) of the node (as first integer) and the element (as second integer) collection.

Returns:

Two integer, first: id of last node, second: id of last element.

WriteScr

public void **WriteScr**()

4.4.8 The ObjectCounter class

jp.go.jnc.tokai.pouchon.finitele

Class ObjectCounter

java.lang.Object
  |
  +--**jp.go.jnc.tokai.pouchon.finitele.ObjectCounter**

---

public class **ObjectCounter**

extends java.lang.Object

Field Summary	
java.util.Vector	**nodesAndElements**
java.util.Vector	**objectId**

Constructor Summary	
ObjectCounter()	

Method Summary	
int	**getEntityNr**(long num) Takes a whole id and determines the element or node number and returns it.
int	**getObjPlace**(long num) Takes a whole id and determines the Object number and returns it.
void	**killEle**(int toKill) Kills an element
void	**killElementInObj**(int eleNr, int objNr)

	Kills an element specified by the total id-number.
void	**killNode**(int toKill) Kills Node in the actual Object.
void	**killNodeInObj**(int nodeNr, int objNr) Kills a node specified by the total id-number.
long	**newElement**() Adds a new element to the last Object
long	**newElement**(int num) Adds num elements to the last Object
long	**newElementInObj**(int num) Creates a new element in the specified element, given by the number.
long	**newNode**() Adds a nodes to the last Object
long	**newNode**(int num) Adds num nodes to the last Object
long	**newNodeInObj**(int num) Creates a new node in the specified object, given by the number.
void	**newObject**() Just initializes a new Object, assigns a new Object-id and opens a placeholder for the node and element ids.
void	**WriteScr**()

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

nodesAndElements

public java.util.Vector **nodesAndElements**

---

objectId

public java.util.Vector **objectId**

Constructor Detail

ObjectCounter

public **ObjectCounter**()

Method Detail

newObject

public void **newObject**()

Just initializes a new Object, assigns a new Object-id and opens a placeholder for the node and element ids.

---

getObjPlace

public int **getObjPlace**(long num)

Takes a whole id and determines the Object number and returns it.

Parameters:

num - whole id of any object entity.

Returns:

the object number.

---

getEntityNr

public int **getEntityNr**(long num)

Takes a whole id and determines the element or node number and returns it.

Parameters:

num - whole id of any object entity.

Returns:

the element of node number.

---

newElement

public long **newElement**()

Adds a new element to the last Object

Returns:

new complete Id of the Element.

---

newElement

public long **newElement**(int num)

Adds num elements to the last Object

Parameters:

num - number of elements to be added!

Returns:

new complete Id of the fist Element.

---

newElementInObj

public long **newElementInObj**(int num)

Creates a new element in the specified element, given by the number.

Parameters:

num - object number in List (to be found with getObjPlace(long num) - function)

Returns:

complete id of the created element

See Also:

getObjPlace(long)

---

killEle

public void **killEle**(int toKill)

Killst an element

---

killElementInObj

public void **killElementInObj**(int eleNr,
int objNr)

Kills an element specified by the total id-number.

Parameters:

eleNr - Number of the Element

objNr - Number of the Object

---

newNode

public long **newNode**()

Adds a nodes to the last Object

Returns:

new complete Id of the Element.

---

newNode

public long **newNode**(int num)

Adds num nodes to the last Object

Parameters:

num - number of nodes to be added!

Returns:

new complete Id of the fist Element.

---

newNodeInObj

public long **newNodeInObj**(int num)

Creates a new node in the specified object, given by the number.

Parameters:

num - object number in List (to be found with getObjPlace(long num) - function)

Returns:

complete id of the created node

See Also:

getObjPlace(long)

---

killNode

public void **killNode**(int toKill)

Kills Node in the actual Object.

Parameters:

toKill - number of the node to be killed.

---

killNodeInObj

public void **killNodeInObj**(int nodeNr,
int objNr)

Kills a node specified by the total id-number.

Parameters:

nodeNr - Number of the Node

objNr - Number of the Object

WriteScr

public void **WriteScr**()

4.5 The spherearr package

4.5.1 The CellSpheres class

jp.go.jnc.tokai.pouchon.spherearr

Class CellSpheres

java.lang.Object
|

+--**jp.go.jnc.tokai.pouchon.spherearr.CellSpheres**

---

public class **CellSpheres**

extends java.lang.Object

Class describing a collection of spheres in the cell. It mainly provides tools to import sphere data from a file.

Field Summary	
`int`	**Size** Number of spheres in the collection.
CellSphereData[]	**Spheres** List of spheres, the basic type is CellSphereData, which contains many information about the sphere, like number of neighbors, ...

Constructor Summary	
CellSpheres(java.lang.String fineName, java.lang.String coarseName, int nFine, int nCoarse) Constructor, which reads in two text files, one for the fine, and one for the coarse fraction.	

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

Spheres

public CellSphereData[] **Spheres**

List of spheres, the basic type is CellSphereData, which contains much information about the sphere, like number of neighbors,....

Size

public int **Size**

Number of spheres in the collection.

Constructor Detail

CellSpheres

public **CellSpheres**(java.lang.String fineName,
java.lang.String coarseName,
int nFine,
int nCoarse)

Constructor, which reads in two text files, one for the fine, and one for the coarse fraction. The amount in each fraction is specified here. This function calls a constructor of CellSphereData with a Buffered Reader as parameter, which extracts each sphere with its specifications from the file-lines

Parameters:

fineName - name of the text-file containing the fine fraction.

coarseName - name of the text-file containing the coarse fraction.

nFine - number of fine spheres in the collection

nCoarse - number of coarse spheres in the collection

See Also:

CellSphereData.CellSphereData(java.io.BufferedReader, boolean)

4.5.2 The CellSphereData class

jp.go.jnc.tokai.pouchon.spherearr

Class CellSphereData

java.lang.Object
  |
  +--jp.go.jnc.tokai.pouchon.geometry.Coordinate
      |
      +--jp.go.jnc.tokai.pouchon.geometry.Sphere
          |
          +--**jp.go.jnc.tokai.pouchon.spherearr.CellSphereData**

Direct Known Subclasses:

SphereMesh

---

public class **CellSphereData**

extends Sphere

Complete information about the sphere. Additional information is used for the meshing routine and later connection between the spheres.

Field Summary	
SphereCoordinate[]	**direction** The direction of the next neighbors, this is given relative to the midpoint in spherical coordinates.
long	**id** id of the sphere, this is not an entity for the elements but just the original appellation of each sphere.
double[]	**neigDist** Distance of each neighbor.
long[]	**neigId** id's of the next 12 neighbors.
long	**NONeig** Number of neighbor for each sphere, when the distance is smaller

	than a critical value, the spheres are considered to be touching.

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Sphere
r

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary

CellSphereData() Constructor of an empty sphere, empty means here: in the origin (0,0,0) with a negative radius (of course impossible), radius -1 is therefore a sign for not being initialized.	
CellSphereData(java.io.BufferedReader is, boolean coarse) Constructor with a buffered reader as argument, this contains the ASCII-text Stream whose lines contain the sphere data, one line is taken at each call.	
CellSphereData(CellSphereData in) Copy constructor.	
CellSphereData(long idIn, Coordinate OriginIn, double rIn, long NONeigIn, long[] neigIdIn, double[] neigDistIn) Constructor with the basic values.	

Method Summary

void	**WriteScr**() Writes the complete information of one sphere to the screen.

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Sphere
assign, assign, assign, assign, WritelnScr

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

id

public long **id**

id of the sphere, this is not an entity for the elements but just the original appellation of each sphere.

NONeig

public long **NONeig**

Number of neighbor for each sphere, when the distance is smaller than a critical value, the spheres are considered to be touching.

---

neigId

public long[] **neigId**

id's of the next 12 neighbors. They don't have to touch, just the first NONeig's are touching.

---

neigDist

public double[] **neigDist**

Distance of each neighbor.

---

direction

public SphereCoordinate[] **direction**

The direction of the next neighbors, this is given relative to the midpoint in spherical coordinates.

Constructor Detail

CellSphereData

public **CellSphereData**()

Constructor of an empty sphere, empty means here: in the origin (0,0,0) with a negative radius (of course impossible), radius -1 is therefore a sign for not being initialized.

CellSphereData

public **CellSphereData**(long idIn,
Coordinate OriginIn,
double rIn,
long NONeigIn,
long[] neigIdIn,
double[] neigDistIn)

Constructor with the basic values.

Parameters:

idIn - id of the new sphere.

OriginIn - input of the sphere mid-point (origin)

rIn - input of radius

NONeigIn - number of neighbors (distance smaller than a critical value)

$neigIdIn_{i=0...12}$ - input of the neighbor id's

$neigDistIn_{i=0...12}$ - input of the neighbor distances

CellSphereData

public **CellSphereData**(CellSphereData in)

Copy constructor.

CellSphereData

public **CellSphereData**(java.io.BufferedReader is,
boolean coarse)
throws java.io.IOException

Constructor with a buffered reader as argument, this contains the ASCII-text Stream whose lines contain the sphere data, one line is taken at each call. The boolean variable tells if the fine or coarse fraction is read in.

Parameters:

is - ASCII-text stream which contains all sphere data (of all spheres).

coarse - true if the coarse fraction is read in, false otherwise.

Method Detail

WriteScr

public void **WriteScr**()

Writes the complete information of one sphere to the screen.

Overrides:

WriteScr in class Sphere

4.5.3 The SphereMesh class

jp.go.jnc.tokai.pouchon.spherearr

Class SphereMesh

java.lang.Object
|
+--jp.go.jnc.tokai.pouchon.geometry.Coordinate
|
+--jp.go.jnc.tokai.pouchon.geometry.Sphere
|
+--jp.go.jnc.tokai.pouchon.spherearr.CellSphereData
|
+--**jp.go.jnc.tokai.pouchon.spherearr.SphereMesh**

---

public class **SphereMesh**

extends CellSphereData

Class to generate, represent and modify meshes for spheres.

Field Summary	
Element[]	**elements** Elements in this sphere mesh.
java.util.Vector	**newElements**
java.util.Vector	**newNodes**
Node[]	**nodes** Node coordinates in Cartesian system.
SphereCoordinate[]	**nodesSpherical** Node coordinates in spherical system, this information is kept for later easy transformations, this information is more natural for the system than the Cartesian description, but is not compatible with all the transformation and notation tools.

Fields inherited from class jp.go.jnc.tokai.pouchon.spherearr.CellSphereData
direction, id, neigDist, neigId, NONeig

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Sphere
r

Fields inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
precision, x

Constructor Summary	
SphereMesh(CellSphereData sphIn, long startId, int stufe) Basic constructor of a sphere mesh.	
SphereMesh(CellSphereData shpIn, ObjectCounter ids, int stufe) Procedure witch takes ObjectCounter instead of direct integer, otherwise see procedure with integer argument for new ids.	
SphereMesh(SphereMesh toCopy) Simple copy constructor.	

Method Summary	
void	**adjust**(Polyhedron adjustingCell, ObjectCounter ids) Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.
void	**adjust1**(Polyhedron adjustingCell, ObjectCounter ids) Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.
void	**adjustOld**(Cell adjCell)

void	**connectPoint**(Coordinate toConn, double region) Transforms sphere-mesh, so that the closest node is shifted to the Coordinate "toConn", the rest of the nodes are transformed spherically in the same direction but with decreasing distance (decreasing transformation factor).
void	**cut**(Cell cuttingCell)
void	**cut**(Polyhedron cuttingCell)
static long[][][]	**newNodesInd**(ElementCoordinate[] in)
void	**vectorToMatrix**()
void	**WriteScr**() Writes the complete information of one sphere to the screen.

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Sphere
assign, assign, assign, assign, WritelnScr

Methods inherited from class jp.go.jnc.tokai.pouchon.geometry.Coordinate
abs, add, add, add, add, assign, assign, assign, cross, cross, cut, cut, cut, distance, dot, dot, dot, equal, main, mult, mult, normalize, normalize, project, project, project, project, rotate, rotate, spat, spat

Methods inherited from class java.lang.Object
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait

Field Detail

nodesSpherical

public SphereCoordinate[] **nodesSpherical**

Node coordinates in spherical system, this information is kept for later easy transformations, this information is more natural for the system than the Cartesian description, but is not compatible with all the transformation and notation tools.

---

nodes

public Node[] **nodes**

Node coordinates in cartesian system.

---

elements

public Element[] **elements**

Elements in this sphere mesh.

---

newNodes

public java.util.Vector **newNodes**

---

newElements

public java.util.Vector **newElements**

Constructor Detail

SphereMesh

public **SphereMesh**(SphereMesh toCopy)

Simple copy constructor.

Parameters:

toCopy - mesh of a shere to be copied.

SphereMesh

public **SphereMesh**(CellSphereData shpIn,
ObjectCounter ids,
int stufe)

Procedure which takes ObjectCounter instead of direct integer, otherwise see procedure with integer argument for new ids. The startId of the nodes and elements is the same here.

Parameters:

sphIn - sphere to be meshed

ids - Object counter for determination of the ids.

degreeSf - subdivision degree of the sphere-surface.

See Also:

SphereMesh(CellSphereData,long,int)

SphereMesh

public **SphereMesh**(CellSphereData sphIn,
long startId,
int stufe)

Basic constructor of a sphere mesh. It takes the sphere information as input and generates the nodes and elements to represent it.

Parameters:

sphIn - sphere to be meshed

startId - id of the first entry in the new mesh.

degreeSf - subdivision degree of the sphere-surface.

degreeRa - radial subdivisions of the sphere.

Method Detail

connectPoint

public void **connectPoint**(Coordinate toConn,
double region)

Transforms sphere-mesh, so that the closest node is shifted to the Coordinate "toConn", the rest of the nodes are transformed spherically in the same direction but with decreasing distance (decreasing transformation factor). The most far point to be transformed is given with the parameter region, where 1 would just contain the whole sphere, if the whole mesh should be transformed identically, region should be set to infinite, or just a large number (of course).

Parameters:

toConn - point to be connected with the sphere-mesh

region - region of the sphere mesh to be adjusted, 0 only one mesh node is displaced, 1 the whole sphere-mesh is transformed, the most far point with a factor of 0, the whole sphere-mesh is transformed with the same factor.

---

cut

public void **cut**(Polyhedron cuttingCell)

---

cut

public void **cut**(Cell cuttingCell)

---

adjust1

public void **adjust1**(Polyhedron adjustingCell,
ObjectCounter ids)

Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.

---

adjust

public void **adjust**(Polyhedron adjustingCell,
ObjectCounter ids)

Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.

---

adjustOld

public void **adjustOld**(Cell adjCell)

---

vectorToMatrix

public void **vectorToMatrix**()

---

newNodesInd

public static long[][][] **newNodesInd**(ElementCoordinate[] in)

Parameters:

from - list of elements (with Coordinate information), which necessities new nodes;

Returns:

list of new nodes which should be initialized

WriteScr

public void **WriteScr**()

Description copied from class: CellSphereData

Writes the complete information of one sphere to the screen.

Overrides:

WriteScr in class CellSphereData

5 Conclusion

A meshing tool for sphere arrangements was programmed successfully. The software creates elements suitable for thermal conductivity and mechanical considerations. The result is outputted in the FEMAP neutral format and can be easily be processed. The program is split up into four packages with many classes, which are relatively easy to understand and maintain. Most classes including methods are commented within the source codes, which also lead to an automatically generated documentation. This also provides a good possibility to maintain and further develop the tool. Some tasks remain to be refined or programmed. Especially the cut routines, which limit the mesh to the representative cell, can be improved. Also the connection types should be adjusted to the FEM software being used. The basic structure of the tool is designed and the corpus of the software is programmed and represents a good basis for future implementations.

Appendix

A References

[1] F. Botta, Ch. Hellwig, “SPACON – A Theoretical Model for Calculating the Heat Transport Properties in Sphere-Pac Fuel Pins”, Nucl. Sci. Eng., Vol. 135, p.165 (2000)
[2] Java programming language: http://java.sun.com
[3] PFC 3D program from Itasca: www.itasca.de/pages/software/pfc3d/
[4] povray program for ray tracing: www.povray.org
[5] Femap, a FEM modeling editor and interface: http://www.eds.com/products/plm/femap/
[6] Finas, FEM software: http://www.crc.co.jp/science/Finas/finas.html

B Make Files

B.1 For program package compilation

The make file for generating the class files is:

```
# 2001.02.28, Manuel Alexandre Pouchon, JNC Japan
# Makefile for compiling the full project.

# JAVA_DEV_ROOT, JAVA_HOME must be set as an enviromental variable (DOS -> autoexec.bat)
# If UNIX System, also set IS_UNIX to true, because of different delimiter (: Unix, ; Dos)

ifdef IS_UNIX
SEP = :
else
SEP = ;
endif

# Input your JRE-Home Directory
JRE           = $(JAVA_HOME)/jre/lib/rt.jar

# Set the documentation, class and source directory plus the class target: DEV_CLASS_DIRECTORY
DOC_DIRECTORY      = $(JAVA_DEV_ROOT)/docs
CLASSPATH          = .$(SEP)$(JRE)$(SEP)$(JAVA_DEV_ROOT)/classes
SOURCEPATH         = .$(SEP)$(JRE)$(SEP)$(JAVA_DEV_ROOT)/scr
DEV_CLASS_DIRECTORY = $(JAVA_DEV_ROOT)/classes
DEV_SCR_DIRECTORY   = $(JAVA_DEV_ROOT)/scr

# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# Insert program to be evaluated !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
#!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

TESTPROG = jp.go.jnc.tokai.pouchon.spherearr.SphereArrTest

# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# Insert Packages to be used !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

include $(JAVA_DEV_ROOT)/projects/SpherePacMeshing/packages
# (A) list of packages called PACKAGES

# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
# !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

TESTPROG_P = $(subst .,/,$(TESTPROG))
TESTPROG_PATH_SCR = $(addprefix $(DEV_SCR_DIRECTORY)/, $(TESTPROG_P).java)

PACKAGE_D = $(subst .,/,$(PACKAGES))
PACKAGE_DIR = $(addprefix $(DEV_SCR_DIRECTORY)/, $(PACKAGE_D))
# (B) include the class list for each package
PACKAGE_CLASS_LIST = $(addsuffix /packageProgs,$(PACKAGE_DIR))
include $(PACKAGE_CLASS_LIST)

ALL_PACKAGE_PROGS_DIR = $(subst .,/,$(LocClasses))
ALL_PACHAGE_PROGS = $(addprefix $(DEV_SCR_DIRECTORY)/, $(ALL_PACKAGE_PROGS_DIR))
```

```
ALL_CLASSES = $(patsubst %, %.java, $(ALL_PACKAGE_PROGS)) $(TESTPROG_PATH_SCR)

# set compiler options
JAVAPRE    = javac
JAVA       = java
JAVAPREOPT = -g -d $(DEV_CLASS_DIRECTORY) -classpath $(CLASSPATH) -sourcepath
$(SOURCEPATH) -verbose
JAVAOPT    = -classpath $(CLASSPATH)
JAVADOCOPT = -d $(DEV_DOC_DIRECTORY) -classpath $(CLASSPATH) -sourcepath
$(SOURCEPATH) -use -version -author

GeometryTest: AllClassesIn.o
        $(JAVA) $(JAVAOPT) $(TESTPROG)

AllClassesIn.o: $(ALL_CLASSES)
        $(JAVAPRE) $(JAVAPREOPT) $(ALL_CLASSES)
```

make imports here (A) a list of packages, the tree where this list is saved can be seen in the make file, the list itself is stored in a file called “packages”, the content is:

```
PACKAGES = jp.go.jnc.tokai.pouchon.geometry\
        jp.go.jnc.tokai.pouchon.math\
        jp.go.jnc.tokai.pouchon.finitele\
        jp.go.jnc.tokai.pouchon.spherearr
```

Each package has its own folder, this folder contains a list of the classes being contained in the package, this list is read in (B) and is used to generate the information about all packages which need to be compiled. The file containing the classes of each package is always called “packageProgs”, an example for the geometry package is:

```
PackageGeom = jp.go.jnc.tokai.pouchon.geometry

LocClassGeom = \
 Coordinate\
 SphereCoordinate\
 Line\
 Plane\
 Face\
 Cell\
 Matrix\
 Polygon\
 MultiLine\
 MultiPolygon\
 Sphere\
 Polyhedron\
 MultiPolyhedron

LocClasses += $(patsubst %, $(PackageGeom).%, $(LocClassGeom) )
```

First the package url must be given, this is prefixed to each class name, in order to make calling each compilation with the whole path. The procedure of prefixing the url is performed in the last line. The class list is assigned to the variable “LocClasses” which in turn is again called in the make file.

B.2 For generation of documentation

The make file for generating the documentation is in principle very similar to the last one, except for calling “javadoc” as compiler.

```
# 2001.02.27, Manuel Alexandre Pouchon, JNC Japan
# Makefile for generating the full documentation of the project.

# JAVA_DEV_ROOT must be set as an enviromental variable (DOS -> autoexec.bat)
# If UNIX System, also set IS_UNIX to true, because of different delimiter (: Unix, ; Dos)

ifdef IS_UNIX
SEP = :
else
SEP = ;
endif

# Input your JRE-Home Directory
JRE           = $(JAVA_HOME)/jre/lib/rt.jar

# Set the documentation, class and source directory
DOC_DIRECTORY      = $(JAVA_DEV_ROOT)/docs  # target directory
CLASSPATH          = .$(SEP)$(JRE)$(SEP)$(JAVA_DEV_ROOT)/classes
SOURCEPATH         = .$(SEP)$(JRE)$(SEP)$(JAVA_DEV_ROOT)/scr
DEV_SCR_DIRECTORY  = $(JAVA_DEV_ROOT)/scr

# The Text-file called packages contains the name of all packages wich are assigned to
# the variable PACKAGES
# (eg: jp.go.jnc.tokai.pouchon.geometry jp.go.jnc.tokai.pouchon.math .....)
# It is stored in the project Folder
# Each package-Folder also containes a Text-file called packageProgs, which lists all
# Classes which should be included in the Documentation, please look at one of these
# packageProgs-Files for reference, the Variable being defined here is LocClasses
# which is assigned to ALL_PACKAGE_PROGS here!
include $(JAVA_DEV_ROOT)/projects/SpherePacMeshing/packages    #list of packages called
PACKAGES
PACKAGE_D   = $(subst .,/,$(PACKAGES))
PACKAGE_DIR = $(addprefix $(DEV_SCR_DIRECTORY)/, $(PACKAGE_D))
PACKAGE_CLASS_LIST = $(addsuffix /packageProgs,$(PACKAGE_DIR))
include $(PACKAGE_CLASS_LIST)

ALL_PACKAGE_PROGS = $(LocClasses)


# define your options
JAVADOCOPT = -d $(DOC_DIRECTORY) -classpath $(CLASSPATH) -sourcepath $(SOURCEPATH) -
use -version -author

DOC:
        javadoc $(JAVADOCOPT) $(ALL_PACKAGE_PROGS)
```

C Java source code

C.1 The main class

```
package MainPrograms.SphereMesh;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;
import jp.go.jnc.tokai.pouchon.spherearr.*;
import jp.go.jnc.tokai.pouchon.finitele.*;

public class SphereMeshing
{

   // ****************** Procedure TxtFileCopyToFile ****************************************

    static void TxtFileCopyToFile(String fileName, PrintWriter os) throws IOException
           {
               try
                   {
                       BufferedReader in = new BufferedReader(new FileReader(fileName));
                       String S;
                       while ((S = in.readLine()) != null && (S.trim().length()) > 0)
                           {
                               os.println(S);
                           }
                       in.close();
                   }
               catch(IOException e)
                   {
                       System.out.print(" **** Error when reading  ****: "+fileName+"  "+ e);
                       System.exit(1);
                   }
           }

    // ***************** Procedure CountFileLines ***************************************************

    static int CountFileLines(String fileName)
           {
               int N=0;
               try
                   {
                       BufferedReader in = new BufferedReader(new FileReader(fileName));
                       String S;
                       while ((S = in.readLine()) != null && (S.trim().length()) > 0)
                           {
                               N++;
                           }
                       in.close();
                       System.out.println("Anzahl: "+N);
                   }
               catch(IOException e)
                   {
                       System.out.print(" **** Error when reading ****: "+fileName+"   "+  e);
                       System.exit(1);
                   }
               return N;
           }

    // ********************************** Procedure WriteFile  ************************

    static void WriteFile(String fileName, NodeList nodeToWr, ElementList eleToWr)
    {
        try
            {
               PrintWriter out = new PrintWriter(new FileWriter(fileName));
               TxtFileCopyToFile("PreFix.txt", out);
               // Print of NODE section (403) ****************************
               out.println("   -1");out.println("   403");
               nodeToWr.WriteFile(out);
               out.println("   -1");
               // Print of ELEMENT section (404) ************************
               out.println("   -1");out.println("   404");
```

```
            eleToWr.WriteFile(out);
            out.println("   -1");
            TxtFileCopyToFile("PostFix.txt", out);
            out.close();
        }
    catch(IOException e)
        {
            System.out.print(" - Fehler: (beim Schreiben von " + fileName + ")--> " + e);
            System.exit(1);
        }
}




public static void main(String[] args)
{
    System.out.print("\n\n\n\n\n\n _____________________________________________________ \n\n");

    double toleranz = 5.0E-6;
    String FileInName = "InDerZelle.txt";
    String CoarseFileInName = "DatenCoarse.txt";
    String FileOutName = "FemapNormal.neu";
    int nFine = CountFileLines(FileInName);
    //int nFine = 3;
    int nCoarse = CountFileLines(CoarseFileInName);
    CellSpheres A = new CellSpheres(FileInName,CoarseFileInName,nFine,nCoarse);

    NodeList liste = new NodeList();
    ElementList ellist = new ElementList();
    ObjectCounter ids = new ObjectCounter();

    System.out.print("\n\n _____________________THE CELL IS_____________________ \n\n");
    Cell ZZ = new Cell("Cell.txt");
    Polyhedron Z = new Polyhedron(ZZ);
    Z.WriteScr();
    System.out.print("\n\n");



    SphereMesh kugelAll[] = new SphereMesh[nFine+nCoarse];
    ids.newObject();

    // read-in fine spheres
    //nFine = 1;
    for(int i=0; i<nFine; i++)
        {
            ids.newObject();
            SphereMesh kugelG = new SphereMesh(A.Spheres[i],ids,5);
                        //SphereMesh kugelG = new SphereMesh(A.Spheres[i],10000*(i+1),5);
            if(i==0){kugelG.WriteScr();}
            kugelAll[i] = new SphereMesh(kugelG);
        }

    kugelAll[0].WriteScr();

    for(int i=nFine; i<(nFine+nCoarse); i++)
        {
            System.out.println(" counter "+i);
            //Coordinate toConn = new Coordinate(0,0,0); // test
            ids.newObject();
            SphereMesh kugelG = new SphereMesh(A.Spheres[i],ids,17);
            kugelAll[i] = new SphereMesh(kugelG);
            //kugelG.cut(Z);
            //if(i>0/*==(nFine)*/){kugelG.adjust(Z,ids);}
            //else{kugelG.adjustOld(ZZ);}
            //liste.GetNodes(kugelG);
            //ellist.GetElements(kugelG);
        }

    // connection of the fine spheres to each other

    for(int i=0; i<(nFine); i++)
```

```
            {
               //int id = kugelAll[i].sphereData.id;
               int jj = kugelAll[i].neigId.length;
               for(int j=0;j<jj;j++)
                   {
                      for(int k=0; k<(nFine+nCoarse); k++)
                          {
                             //System.out.println(" "+kugelAll[i].neigId[j]);
                             if(kugelAll[i].neigId[j]==kugelAll[k].id &&
                                Math.abs(kugelAll[i].neigDist[j])<toleranz)
                                 {
                                    //System.out.println(" *"+kugelAll[i].neigId[j]);
                                    long id1=0,id2=0;
                                    Coordinate midPt = new Coordinate(0,0,0);
                                    midPt.add(kugelAll[i],
                                           (kugelAll[k].r/(kugelAll[i].r+kugelAll[k].r)));
                                    midPt.add(kugelAll[k],
                                           (kugelAll[i].r/(kugelAll[i].r+kugelAll[k].r)));
                                    id1=kugelAll[i].connectPoint(midPt,.1,true);
                                    if(k>26){id2=kugelAll[k].connectPoint(midPt,.01,true);}
                                    System.out.println(" Connected: Sphere "+
                                                           i+" with Sphere "+k+" node Id "+
                                                  id1+"  "+id2);
                                 }
                          }
                   }
            }

        ids.WriteScr();

        for(int i=0; i<(nFine+nCoarse); i++)
            {
               //kugelAll[i].cut(Z);
               //kugelAll[i].adjust(Z,ids);
               liste.GetNodes(kugelAll[i]);
               ellist.GetElements(kugelAll[i]);
            }

        System.out.println(" coarse Sphere reading and evaluation");

        /*
         Readin Coarse Spheres
        for(int i=nFine; i<(nFine+nCoarse); i++)
            {
               System.out.println(" counter "+i);
               //Coordinate toConn = new Coordinate(0,0,0); // test
               ids.newObject();
               SphereMesh kugelG = new SphereMesh(A.Spheres[i],ids,15);
               //kugelG.cut(Z);
               if(i>0){kugelG.adjust(Z,ids);}
               else{kugelG.adjustOld(ZZ);}
               liste.GetNodes(kugelG);
               ellist.GetElements(kugelG);
            }
*/

        WriteFile(FileOutName, liste, ellist);



        Coordinate rotrot = new Coordinate(1,0,0);
        rotrot.rotate(Math.PI/2,0,0);
        rotrot.WriteScr();

        Coordinate fo = new Coordinate(1,0,0),
            fd1 = new Coordinate(0,1,0),
            fd2 = new Coordinate(0,0,1),
            lo = new Coordinate(.5,0,.5),
            ld = new Coordinate(.6,0,0);

        Line testl = new Line(lo,ld);
        Face test = new Face(fo,fd1,fd2,1);

        Coordinate cutlf = new Coordinate();
        boolean iscut = cutlf.project(test,lo);
        cutlf.WriteScr();
```

```
        System.out.println(iscut);

        ids.WriteScr();

    }
}
```

C.2 The geometry package

C.2.1 The coordinate class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;
import java.lang.*;
import java.text.*;



// ********************************************************************************
// *********************************   Class Coordinate ***************************
// ********************************************************************************

/**
   Representing vectors and coordinates in &#8476<sup>3</sup>. (Because of possible conflict with the
Java-Vector type, representing an dynamic array, the term Coordinate is used here).

   @version 1.00 2001/01/10, 1.01 2000/02/19
   @author Manuel Alexandre POUCHON - &copy JNC Tokai Works (JAPAN)
*/
public class Coordinate
{
    /**
       <em>Coordinate</em> (or <em><blink>vector</blink></em>) in &real<sup>3</sup>
    */
    public double x[];

    /**
Specifies the tollerance in calculation results, if a distance of two points is smaller than this
value, the points are identified as identic!
     */
    public static final double precision = 1.e-20;

    /**
       Produces coordinate/vector (0 0 0)
    */
    public Coordinate()
    {
        x = new double[3];
        x[0] = 0; x[1] = 0; x[2] = 0;
    }

    /**
       Produces coordinate/vector (x1 x2 x3)
    */
    public Coordinate(double x1, double x2, double x3)
    {
        x = new double[3];
        x[0] = x1;
        x[1] = x2;
        x[2] = x3;
    }

    /**
       Produces vector beween (x1 x2 x3) and (xe1 xe2 xe3)
    */
    public Coordinate(double x1, double x2, double x3, double xe1, double xe2, double xe3)
    {
        x = new double[3];
        x[0] = xe1-x1;
```

```
        x[1] = xe2-x2;
        x[2] = xe3-x3;
    }

    /**
       Copys directly the coordinate/vector called "toCopy" to a new coordinate/vector
     */
    public Coordinate(Coordinate toCopy)
    {
        x = new double[3];
        for(int i=0; i<3; i++){x[i] = toCopy.x[i];}
    }

    /**
       Creates vector between K1 and K2
     */
    public Coordinate(Coordinate K1, Coordinate K2)
    {
        x = new double[3];
        for(int i=0; i<3; i++){x[i] = K2.x[i]-K1.x[i];}
    }

    public Coordinate(SphereCoordinate Trsf)
    {
        x = new double[3];
        x[0] = Trsf.r*(Math.cos(Trsf.phi[0])*Math.cos(Trsf.phi[1]));
        x[1] = Trsf.r*(Math.sin(Trsf.phi[0])*Math.cos(Trsf.phi[1]));
        x[2] = Trsf.r*Math.sin(Trsf.phi[1]);
    }

    public void assign(double x1Koor, double x2Koor, double x3Koor)
    {
        x[0] = x1Koor;
        x[1] = x2Koor;
        x[2] = x3Koor;
    }

    public void assign(Coordinate ToCopy)
    {
        for(int i=0; i<3; i++){x[i] = ToCopy.x[i];}
    }

    public void assign(SphereCoordinate Trsf)
    {
        x[0] = Trsf.r*(Math.cos(Trsf.phi[0])*Math.cos(Trsf.phi[1]));
        x[1] = Trsf.r*(Math.sin(Trsf.phi[0])*Math.cos(Trsf.phi[1]));
        x[2] = Trsf.r*Math.sin(Trsf.phi[1]);
    }

    public void add(Coordinate ToAdd)
    {
        for(int i=0; i<3; i++){x[i] += ToAdd.x[i];}
    }

    public void add(Coordinate ToAdd, double Scale)
    {
        for(int i=0; i<3; i++){x[i] += (Scale*ToAdd.x[i]);}
    }

    public void add(Coordinate ToAdd1, Coordinate ToAdd2)
    {
        for(int i=0; i<3; i++){x[i] = ToAdd1.x[i]+ToAdd2.x[i];}
    }

    public void add(Coordinate ToAdd1, Coordinate ToAdd2, double Scale)
    {
        for(int i=0; i<3; i++){x[i] = (ToAdd1.x[i]+Scale*ToAdd2.x[i]);}
    }

    public double distance(Coordinate second)
    {
        double a=0;
        for(int i=0; i<3; i++){a += Math.pow((x[i]-second.x[i]),2);};
        return Math.sqrt(a);
    }

    /**
```

```
       Cross or vector product of two vectors.
       Direct input of both vectors.
       @param vec1 fist (right) vector
       @param vec2 second (left) vector
    */
    public void cross(Coordinate vec1, Coordinate vec2)
    {
        x[0] = (vec1.x[1] * vec2.x[2]) - (vec1.x[2] * vec2.x[1]);
        x[1] = (vec1.x[2] * vec2.x[0]) - (vec1.x[0] * vec2.x[2]);
        x[2] = (vec1.x[0] * vec2.x[1]) - (vec1.x[1] * vec2.x[0]);
    }

    /**
       Cross or vector product of two vectors.
       Endpoint-Coordinate input: not the vectors but the enpoints of the 2 vector arrangement
        are entered!
       @param koo1 starting point of both vectors
       @param koo2 endpoint of fist (right) vector
       @param koo3 endpoint of second (left) vector
     */
    public void cross(Coordinate koo1, Coordinate koo2, Coordinate koo3)
// Input of common origin with endpoints
    {
        Coordinate vec1 = new Coordinate();
        Coordinate vec2 = new Coordinate();
        vec1.add(koo2,koo1,-1.0);
        vec2.add(koo3,koo1,-1.0);

        x[0] = (vec1.x[1] * vec2.x[2]) - (vec1.x[2] * vec2.x[1]);
        x[1] = (vec1.x[2] * vec2.x[0]) - (vec1.x[0] * vec2.x[2]);
        x[2] = (vec1.x[0] * vec2.x[1]) - (vec1.x[1] * vec2.x[0]);
    }


    public double dot(Coordinate vec1)
    {
        return ((vec1.x[0]*x[0]) + (vec1.x[1]*x[1]) + (vec1.x[2]*x[2]));
    }

    public static double dot(Coordinate vec1, Coordinate vec2)
    {
        return ((vec1.x[0]*vec2.x[0]) + (vec1.x[1]*vec2.x[1]) + (vec1.x[2]*vec2.x[2]));
    }

    public static double dot(Coordinate koo1, Coordinate koo2, Coordinate koo3)
        // Input of common origin with endpoints
    {
        Coordinate vec1 = new Coordinate();
        Coordinate vec2 = new Coordinate();
        vec1.add(koo2,koo1,-1.0);
        vec2.add(koo3,koo1,-1.0);

        return ((vec1.x[0]*vec2.x[0]) + (vec1.x[1]*vec2.x[1]) + (vec1.x[2]*vec2.x[2]));
    }

    /**
       Calculates the volume of the generally sheared box formed by the three vectors vec1,
vec2 and vec3.
     */
    public static double spat(Coordinate vec1, Coordinate vec2, Coordinate vec3)
    {
        Coordinate vecCross12 = new Coordinate();
        vecCross12.cross(vec1,vec2);
        return dot(vecCross12,vec3);
    }

    /**
       Calculates the volume of the generally sheared box formed by the origin and the
endpoints end1, end2 and end3.
     */
    public static double spat(Coordinate origin, Coordinate end1, Coordinate end2, Coordinate end3)
    {
        Coordinate vec1 = new Coordinate();
        Coordinate vec2 = new Coordinate();
        Coordinate vec3 = new Coordinate();
        vec1.add(end1,origin,-1.0);
        vec2.add(end2,origin,-1.0);
```

```
    vec3.add(end3,origin,-1.0);
    return spat(vec1,vec2,vec3);
}

/**
   Multiplies with scalar "multiplier" and assigns.
 */
public void mult(double multiplier)
{
    x[0] *= multiplier;
    x[1] *= multiplier;
    x[2] *= multiplier;
}

/**
   Calculates the absolute value (the length) of a Vector
   @return The length of the vector as double
*/
public double abs()
{
    return Math.sqrt((x[0]*x[0])+(x[1]*x[1])+(x[2]*x[2]));
}

/**
   Deterines whether two coordinates are coincident (=equal here) or not.
*/
public boolean equal(Coordinate toComp)
{
    double dist = this.distance(toComp);
    if(dist<precision)
        {return true;}
    else
        {return false;}
}


/**
   Multiplies vector with Matrix M and applies.
 */
public void mult(Matrix M)
{
    double b[] = new double[3];

    for (int i=0; i<3; i++)
        {
           for (int j=0; j<3; j++)
               {
                  b[i] += M.x[i][j] * x[j];
               };
        };

    x[0]=b[0]; x[1]=b[1]; x[2]=b[2];
}

/**
   Rotates around the x[0], x[1] and x[2] axis
*/
public void rotate(double x0, double x1, double x2)
{
    Matrix rot[] = new Matrix[3];
    Matrix totRot = new Matrix();

    for(int i=0;i<3;i++){rot[i] = new Matrix();}

    double rotAng[] = new double[3]; rotAng[0]=x0;rotAng[1]=x1;rotAng[2]=x2;

    for(int mat=0; mat<3; mat++)
        {
           for(int i=0;i<3;i++)
               {
                  for(int j=0;j<3;j++)
                      {
                         if(i==j)
                             {
                                if(i==mat){rot[mat].x[i][j]=1.0;}
                                else{rot[mat].x[i][j]=Math.cos(rotAng[mat]);}
                             }
```

```
                            else
                                {
                                    if(i==mat||j==mat){rot[mat].x[i][j]=0.0;}
                                    else{
                                        int vorz; if(i<mat){vorz=mat+i+1;}else{vorz=mat+i;}
                                        rot[mat].x[i][j]=(Math.sin(rotAng[mat]))
                                                                    *(Math.pow(-1,vorz));}
                                }
                        }
                }
        }
    totRot.mult(rot[0],rot[1]);
    totRot.mult(totRot,rot[2]);

    mult(totRot);

}


/**
   Rotates partially around the x[0], x[1] and x[2] axis with factor "part"
*/
public void rotate(double x0, double x1, double x2, double part)
{
    Matrix rot[] = new Matrix[3];
    Matrix totRot = new Matrix();

    for(int i=0;i<3;i++){rot[i] = new Matrix();}

    double rotAng[] = new double[3]; rotAng[0]=x0;rotAng[1]=x1;rotAng[2]=x2;

    for(int mat=0; mat<3; mat++)
        {
            for(int i=0;i<3;i++)
                {
                    for(int j=0;j<3;j++)
                        {
                            if(i==j)
                                {
                                    if(i==mat){rot[mat].x[i][j]=1.0;}
                                    else{rot[mat].x[i][j]=(1-part)+part*Math.cos(rotAng[mat]);}
                                }
                            else
                                {
                                    if(i==mat||j==mat){rot[mat].x[i][j]=0.0;}
                                    else{
                                        int vorz; if(i<mat){vorz=mat+i+1;}else{vorz=mat+i;}
                                        rot[mat].x[i][j]=(part*Math.sin(rotAng[mat]))
                                                            *(Math.pow(-1,vorz));}
                                }
                        }
                }
        }
    totRot.mult(rot[0],rot[1]);
    totRot.mult(totRot,rot[2]);

    mult(totRot);

}




/**
   Applies the multiplication of vector vIn with the Matrix M.
/*
public void mult(Matrix M, Coordinate vIn)
{
    Coordinate V = new Coordinate(vIn);
    V.mult(M);
    x[0]=V.x[0]; x[1]=V.x[1]; x[2]=V.x[2];
}

/**
   Normalizes a vector (set to length 1).
```

```
     */
    public void normalize()
    {
        double z[] = new double[3];
        for(int i=0; i<3;      i++)
        {z[i]=x[i]/(Math.sqrt(Math.pow(x[0],2)+Math.pow(x[1],2)+Math.pow(x[2],2)));}
        for(int i=0; i<3; i++){x[i]=z[i];};
    }

    /**
       Normalizes a vector and multiplies with "length".
     */
    public void normalize(double length)
    {
        double z[] = new double[3];
        for(int i=0; i<3; i++)
        {z[i]=length*x[i]/(Math.sqrt(Math.pow(x[0],2)+Math.pow(x[1],2)+Math.pow(x[2],2)));}
        for(int i=0; i<3; i++){x[i]=z[i];}
    }

    // ************************************ Cut ********************************

    /**
       Cuts Plane P with Line L and assigns the cutting-point (coordinate), returns true if cutting-
point lies beween Origin of the line and the endpoint of the line (origin plus direction)
    */
    public boolean cut(Plane P, Line L)
    {
        Coordinate pl1 = new Coordinate(), pl2 = new Coordinate();
        double t;
        double gross=0.0;
        int grossNum=-1, i2, i3;

        // finding biggest coordinate of plane-normal (index grossNum)
        // <- avoid cross product with zero vectors
        for (int i=0; i<3; i++) {if (Math.abs(P.normal.x[i]) > Math.abs(gross))
                                    {gross = P.normal.x[i]; grossNum = i;};}
        i2 = (grossNum+1)%3; // determining second index
        i3 = (grossNum+2)%3; // determining third index

        pl1.x[grossNum] = - P.normal.x[i2];
        // formation of normal to plane-normal -> vector pl1 along plane
        pl1.x[i2] = P.normal.x[grossNum];
        pl1.x[i3] = 0;

        pl2.cross(P.normal,pl1);
        // fromation of second vector pl2 along plane -> P.origin & pl1 & pl1: new definition of P

        t = (L.x[2]*pl1.x[1]*pl2.x[0] - P.x[2]*pl1.x[1]*pl2.x[0] - L.x[1]*pl1.x[2]*pl2.x[0]
             +  P.x[1]*pl1.x[2]*pl2.x[0] - L.x[2]*pl1.x[0]*pl2.x[1] + P.x[2]*pl1.x[0]*pl2.x[1]
             + L.x[0]*pl1.x[2]*pl2.x[1] - P.x[0]*pl1.x[2]*pl2.x[1] + L.x[1]*pl1.x[0]*pl2.x[2]
             - P.x[1]*pl1.x[0]*pl2.x[2] - L.x[0]*pl1.x[1]*pl2.x[2] + P.x[0]*pl1.x[1]*pl2.x[2])/
            (-L.dir.x[2]*pl1.x[1]*pl2.x[0] + L.dir.x[1]*pl1.x[2]*pl2.x[0] +
               L.dir.x[2]*pl1.x[0]*pl2.x[1]
             - L.dir.x[0]*pl1.x[2]*pl2.x[1] - L.dir.x[1]*pl1.x[0]*pl2.x[2] +
             L.dir.x[0]*pl1.x[1]*pl2.x[2]);
        // t: cutting parameter along L

        x[0] = L.x[0] + t*L.dir.x[0];
        x[1] = L.x[1] + t*L.dir.x[1];
        x[2] = L.x[2] + t*L.dir.x[2];

        if(t<0 || t>1){return false;}else{return true;}
    }

    /**
       Cuts Face F with Line L and assigns the cutting-point (coordinate), returns true if cutting-
point lies beween Origin of the line and the endpoint of the line (origin plus direction)
    */
    public boolean cut(Face F, Line L, boolean limitedLine)
    {
        Plane P = new Plane(F.orig,F.dir[0],F.dir[1]); // corresponding infinite plane
        Coordinate cuttingPt = new Coordinate(), inUnitKoo = new Coordinate(),
        orthoDir = new Coordinate();
        boolean btwPts;
        boolean inFace;
```

```
        btwPts = cuttingPt.cut(P,L);
        assign(cuttingPt);

        orthoDir.cross(F.dir[0], F.dir[1]);

        Matrix cellM = new Matrix(F.dir[0], F.dir[1], orthoDir), toUnit = new Matrix();
        toUnit.inv(cellM);

        inUnitKoo.assign(cuttingPt);
        inUnitKoo.add(F.orig,-1.0);
        inUnitKoo.mult(toUnit);

        if(F.type==1)
            {
               if(inUnitKoo.x[0]<0 || inUnitKoo.x[0]>1 ||
                  inUnitKoo.x[1]<0 || inUnitKoo.x[1]>1)
                   {inFace = false;}
               else
                   {inFace = true;}
            }
        else
            {
               if(inUnitKoo.x[0]<0 || inUnitKoo.x[1]<0 ||
                  (inUnitKoo.x[0]+inUnitKoo.x[1])>1)
                   {inFace = false;}
               else
                   {inFace = true;}
            }
        if(limitedLine)
            { if(btwPts && inFace){return true;}else{return false;} }
        else
            { if(inFace){return true;}else{return false;} }

    }

    /**
       Cutting of line with polygon!
@return true if unlimited line cuts the polygon. If limitedLine is set to true, routine only returns
true, if the line defined by its origin and the length, cuts the polygon.
If line is parallel to polygon, the result is false, even if the line lies on the polygon!
     */
    public boolean cut(Polygon P, Line L, boolean limitedLine)
    {
        boolean doesCut = false;

        int N = P.corner.length;

        if(N==3)
            {
               double t_hit = -1;                       // lenght of cutting after line origin

               Coordinate side[] = P.sides(); side[2].mult(-1.0); // side of the polygon

               Coordinate normal = new Coordinate();
               normal.cross(side[0],side[2]);            // normal of the polygon

               Coordinate diff = new Coordinate(L,P); // difference fo line and polygon difference
               diff.add(P.corner[0]);

               double nen = Coordinate.dot(normal,L.dir);   // cutting parameters
               double zaehl = Coordinate.dot(normal,diff);

               if(nen==0.0)                             // parallel line to polygon
                   {
                      if(zaehl==0.0){t_hit=0.0;doesCut = false;}
                      else{t_hit=0.0;doesCut=false;}
                      this.assign(L);
                   }
               else                                     // general line, non parallel
                   {
                      doesCut = true;

                      t_hit = zaehl/nen;
                      Coordinate cutPt = new Coordinate(L);
                      cutPt.add(L.dir,t_hit);
                      this.assign(cutPt);
```

```
                    Coordinate in0 = new Coordinate(P.corner[0],cutPt); in0.add(P,-1.0);
                    Coordinate side0 = new Coordinate(P.corner[0],P.corner[1]);
                    Coordinate nor0 = new Coordinate(); nor0.cross(side0,in0);

                    for(int i=1;i<3;i++)
                    // are all normal-vectors in the same direction, if not -> false
                        {
                            Coordinate in12 = new Coordinate(P.corner[i],cutPt); in12.add(P,-1.0);
                            Coordinate side12 = new Coordinate(P.corner[i],P.corner[(i+1)%3]);
                            Coordinate nor12 = new Coordinate(); nor12.cross(side12,in12);
                            double same = Coordinate.dot(nor12,nor0);

                            if(same<=0){doesCut=false;/*System.out.println(" opposite");*/}
                            else{/*System.out.println(" same");*/}
                        }

                    /*Matrix trsf = new Matrix(side[0],side[2],normal);
                    trsf.inv();
                    cutPt.mult(trsf);
                    if(cutPt.x[0]>=0 && cutPt.x[1]>=0 &&(cutPt.x[0]+cutPt.x[1] <= 1 ))
                    {doesCut = true;}
                    else{doesCut = false;}*/

                    if((doesCut && limitedLine) && (t_hit<0.0 || t_hit>1.0)){doesCut = false;}
                }
            }
        else
            {
                //System.out.println("     Splitting");

                doesCut = false;

                Polygon frags[] = P.split();
                Coordinate defCoor = new Coordinate();

                int M = frags.length;

                for(int i=0;i<M;i++)
                    {
                        boolean fragCuts;
                        fragCuts = this.cut(frags[i],L,limitedLine);
                        if(fragCuts){doesCut=true; defCoor.assign(this);
                        /*System.out.println("   and cutting")*/;}
                    }
                this.assign(defCoor);
            }

        return doesCut;

    }

    // ***************************************** Project ***********************************

    /**
       Projects the point "toProject" on the polygon "porjectOn", the projection-coordinate is
assigned to the object, a boolean value is returned to determine, whether projection point is on the
surface (true) or outside (false).
       @author Manuel Alexandre Pouchon, JNC Japan
       @version 1.0 created 2001.01.22
       @param projectOn Polygon which specifies the projection area
       @param toProject Coordinate of the point which is to be projected on the polygon
       @return Returns boolean which specifies whether the projection lies on the surface (true) of
        not (false)
     */
    public boolean project(Polygon projectOn, Coordinate toProject)
    {
        boolean result = false;
        Coordinate P = new Coordinate(), toProjRel = new Coordinate(toProject),
        rel = new Coordinate(0,0,0);
        if(projectOn.topology == 3)     // (quadrangle)
            {
                Coordinate M, R, L, NR, NL;
                // defining the two subtriangles, right: R-M, left: M-L, and the Normals
                Coordinate PR, PL;
                // Projection on right and left triangle and the final projection
                Matrix MR, MRI, ML, MLI;
                // transformaions matrix of each triangle and its normal to unit
```

```
                if(projectOn.rt_lf)
                    {
                        rel.add(projectOn,+1); rel.add(projectOn.corner[0],+1);
                        // isAbove relative to c1
                        M = new Coordinate(projectOn.corner[0],projectOn.corner[2]);
                        R = new Coordinate(projectOn.corner[0],projectOn.corner[1]);
                        L = new Coordinate(projectOn.corner[0],projectOn.corner[3]);
                    }
                else
                    {
                        rel.add(projectOn,+1); rel.add(projectOn.corner[1],+1);
                        // isAbove relative to c2
                        M = new Coordinate(projectOn.corner[1],projectOn.corner[3]);
                        R = new Coordinate(projectOn.corner[1],projectOn.corner[2]);
                        L = new Coordinate(projectOn.corner[1],projectOn.corner[0]);
                    }
                toProjRel.add(rel,-1);
                NR = new Coordinate(); NL = new Coordinate();
                NR.cross(R,M); NR.normalize();
                NL.cross(M,L); NL.normalize();
                MR = new Matrix(R,M,NR); MRI = new Matrix(MR); MRI.inv();
                ML = new Matrix(M,L,NL); MLI = new Matrix(ML); MLI.inv();
                PR = new Coordinate(toProjRel); PR.mult(MRI);
                // Projected point in unit system of right triangle
                PL = new Coordinate(toProjRel); PL.mult(MLI);
                // Projected point in unit system of left trianagle
                if(PR.x[0]>=0 &&
                   PL.x[1]<0) // is pojection right from middle-line PR.x[1] ?
                    {
                        P.assign(PR.x[0],PR.x[1],0); P.mult(MR); P.add(rel);
                        if((PR.x[0]+PR.x[1])<=1 && PR.x[1]>=0){result = true;}
                        else{result = false;}
                    }
                else if(PL.x[1]>=0 &&
                        PR.x[0]<0)  // is projection left from middle-line PL.x[0] ?
                    {
                        P.assign(PL.x[0],PL.x[1],0); P.mult(ML); P.add(rel);
                        if((PL.x[1]+PL.x[0])<=1 && PL.x[0]>=0){result = true;}
                        else{result = false;}
                    }
                else if((PL.x[1]>=0 && PL.x[0]>=0 && (PL.x[0]+PL.x[1])<=1 &&
                        // proj on both triangles possible
                         PR.x[0]>=0 && PR.x[1]>=0 && (PR.x[0]+PR.x[1])<=1)  ||
                        (PL.x[1]>=0 && (PL.x[0]<0 || (PL.x[0]+PL.x[1])>1)  &&
                        // proj on neither of  both triangl poss
                         PR.x[0]>=0 && (PR.x[1]<0 || (PR.x[0]+PR.x[1])>1)))
                    {                           // -> projection on middle-line
                        Coordinate NML = new Coordinate();
                        NML.cross(M,NL); NML.normalize();
                        Matrix MML = new Matrix(M,NML,NL); Matrix MMLI = new Matrix(MML);
                        MMLI.inv();
                        Coordinate PML = new Coordinate(toProjRel); PML.mult(MMLI);
                        P.assign(PML.x[0],0,0); P.mult(MML); P.add(rel);
                        if((PL.x[0]+PL.x[1])<=1 && (PR.x[0]+PR.x[1])<=1)
                        {result = true;}else{result = false;}
                    }
                else if((PL.x[1]>=0 && PL.x[0]>=0 && (PL.x[0]+PL.x[1])<=1 &&
                // proj on L- triangle is possible
                         PR.x[0]>=0 && (PR.x[1]<0 || (PR.x[0]+PR.x[1])>1)))
                    {
                        P.assign(PL.x[0],PL.x[1],0); P.mult(ML); P.add(rel);
                        result = true;
                    }
                else if((PR.x[0]>=0 && PR.x[1]>=0 && (PR.x[0]+PR.x[1])<=1 &&
                // proj on R- triangle is possible
                         PL.x[1]>=0 && (PL.x[0]<0 || (PL.x[0]+PL.x[1])>1)))
                    {
                        P.assign(PR.x[0],PR.x[1],0); P.mult(MR); P.add(rel);
                        result = true;
                    }
                else if(PL.x[1]<0 &&
// projection on neither of both triangles possible
                        PR.x[0]<0)                      // -> projection on middle-line
                    {
                        Coordinate NML = new Coordinate();
                        NML.cross(M,NL); NML.normalize();
                        Matrix MML = new Matrix(M,NML,NL); Matrix MMLI = new Matrix(MML); MMLI.inv();
```

```
                        Coordinate PML = new Coordinate(toProjRel); PML.mult(MMLI);
                        P.assign(PML.x[0],0,0); P.mult(MML); P.add(rel);
                        if(PML.x[0]>=0 && PML.x[0]<=1){result = true;}else{result = false;}
                    }
                else
                    {
                        System.out.println(" *** Error in project for Polygon:
                        unknown case for quadrangle");
                    }
            }
        else // topology 2 is assumed (triangle)
            {
                Coordinate R, L, NO;
                // defining the two subtriangles, right: R-M, left: M-L, and the Normals
                Coordinate PJ; // Projection on right and left triangle and the final projection
                Matrix MT, MTI; // transformaions matrix of each triangle and its normal to unit
                rel.add(projectOn,+1); rel.add(projectOn.corner[0],+1);// isAbove relative to c1
                R = new Coordinate(projectOn.corner[0],projectOn.corner[1]);
                L = new Coordinate(projectOn.corner[0],projectOn.corner[2]);
                toProjRel.add(rel,-1);
                NO = new Coordinate();
                NO.cross(R,L); NO.normalize();
                MT = new Matrix(R,L,NO); MTI = new Matrix(MT); MTI.inv();
                PJ = new Coordinate(toProjRel); PJ.mult(MTI);
                // Projected point in unit system of triangle
                P.assign(PJ.x[0],PJ.x[1],0); P.mult(MT); P.add(rel);
                if(PJ.x[0]>=0 && PJ.x[1]>=0 && (PJ.x[0] + PJ.x[1])<=1){result = true;}
else{result = false;}
            }
        this.assign(P);
        return result;
    }


    public boolean project(Face F, Coordinate K)
    {
        Plane P = new Plane(F.orig, F.dir[0], F.dir[1]);
        Line L = new Line(K, P.normal);

        Coordinate cuttingPt = new Coordinate();

        boolean isCutting = cuttingPt.cut(F,L,false);
        assign(cuttingPt);

        return isCutting;
    }

    public void project(Plane P, Coordinate K)
    {
        Line L = new Line(K, P.normal);

        Coordinate cuttingPt = new Coordinate();

        boolean noNeed = cuttingPt.cut(P,L);
        assign(cuttingPt);
    }

    public void project(Line L, Coordinate K)
    {
        Coordinate origL2K = new Coordinate(K); origL2K.add(L,-1.0);
        double onLine = Coordinate.dot(L.dir,origL2K);
        Coordinate res = new Coordinate(L); res.add(L.dir,onLine);
        assign(res);
    }




    // ******************************** Write ***************************************

    public void WriteScr()
    {
        //NumberFormat nf = NumberFormat.getNumberInstance();
        //nf.setMaximumFractionDigits(4);
        //nf.setMinimumFractionDigits(2);
        String x0 = ScNotation(x[0],false);// nf.format(x[0]);
```

```
        String x1 = ScNotation(x[1],false);// nf.format(x[1]);
        String x2 = ScNotation(x[2],false);// nf.format(x[2]);
        System.out.print("|"+x0+" "+ x1+" "+x2+"|");

    }

    /**
       transforms an real (double) number into a string represeinting the scientifical notation!
       @param number number to be transfromed into the fromated string
       @param except shall there be an exception for number which can easily be represented in normal
        notation?
       @return string representing the sceintifical notation of the number
     */
    static String ScNotation(double number, boolean except)
    {
        long digits;
        String bas;
        double numAbs = Math.abs(number);
        if(numAbs<precision)
            {
                bas = "0";
                digits = 0;
            }
        else
            {
                digits = (long)(Math.floor(Math.log(numAbs)/Math.log(10)));
                double basis = number/(Math.pow(10,digits));

                NumberFormat nf = NumberFormat.getNumberInstance();
                nf.setMaximumFractionDigits(2);
                nf.setMinimumFractionDigits(0);
                bas = nf.format(basis);
            }

        return(bas+"E"+digits);

    }

    public static void main(String[] args)
    {

    }

}
```

C.2.2 The cell class

```
/*
 * Copyright 1996-2000 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.finitele.*;

// ********************************************************************************************
// *********************************   Class Cell  ********************************************
// ********************************************************************************************

/**
Describes a cell in space, generally it is a spheared cubus or wedge, if type is true, then cubus (4
side prism), otherwise, if type is false, then it is a wedge (3 side prism). Origin is the base of the
three sides, describing the cell. dir[0] describes the elevation of the prism, and dir[1,2] the two
sides.
 */
public class Cell extends Coordinate
{

    /**
       Origin of the Cell, reference point for the Cell sides
     */
    //    public Coordinate orig;
```

```
    /**
       Sides of the Cell
    */
    public Coordinate dir[];
    /**
       Type of the cell, <b>true</b> &#8658; 4-side prism (Brick),
        <b>false</b> &#8658; 3-side prism (Wedge)
    */
    public int type;

    public Cell(String FileName)
    {
        super();
        dir = new Coordinate[3]; for(int i=0; i<3; i++){dir[i] = new Coordinate();}
        try
            {
                BufferedReader in = new BufferedReader(new FileReader(FileName));
                String s = in.readLine();
                StringTokenizer t = new StringTokenizer(s, ",");
                for(int i=0; i<3; i++){this.x[i] = Double.parseDouble(t.nextToken().trim());}
                for(int j=0; j<3; j++){for(int i=0; i<3; i++){dir[j].x[i] =
                Double.parseDouble(t.nextToken().trim());}}
                type = Integer.parseInt(t.nextToken().trim());
                in.close();
            }
        catch(IOException e)
            {
                System.out.print(" **** Fehler **** (Datei nicht gefunden) "+ e);
                System.exit(1);
            }
    }

    // ******************************** In Cell ***********************************************

    /**
       verifies wether coordinate lies within the cell, returns boolean
     */
    public boolean within(Coordinate isIn)
    {
        Matrix cellM = new Matrix(dir[0], dir[1], dir[2]), toUnit = new Matrix();
        toUnit.inv(cellM);

        Coordinate cooToOrig = new Coordinate(isIn), trsf = new Coordinate();
        cooToOrig.add(this,-1.0);
        cooToOrig.mult(toUnit);

        if(type==1)
            {
                if(cooToOrig.x[0]<0 || cooToOrig.x[0]>1 ||
                   cooToOrig.x[1]<0 || cooToOrig.x[1]>1 ||
                   cooToOrig.x[2]<0 || cooToOrig.x[2]>1)
                    {return false;}
                else {return true;}
            }
        else
            {
                if(cooToOrig.x[0]<0 || cooToOrig.x[0]>1 ||
                   cooToOrig.x[1]<0 || cooToOrig.x[2]<0 ||
                   (cooToOrig.x[1]+cooToOrig.x[2])>1)
                    {return false;}
                else {return true;}
            }
    }

    /**
       verifies wether element interacts with the cell, returns boolean
     */
     public boolean within(Element isIn, Coordinate isInNodeKoor[])
    {
        int isout=1;
        int nodesNum = isIn.nodeId.length;
        if(nodesNum != isInNodeKoor.length){System.out.println( "*** Fehler ****"); return true;}
        else // Test if element and cell are not disjunct
            {
                // Test if one (or more) node(s) of element is (are) withing cell
                boolean nodeIsIn = false; //initial setting
                for(int i=0; i<nodesNum; i++)
```

```
                {
                    if(this.within(isInNodeKoor[i]))
                        {nodeIsIn = true; break;}
                    else{}
                }
            if(nodeIsIn)
                {return true;} // no additional test necessary
            else // test, whether one of the element-sides interacts with the cell
                {
                    Face cellFaces[] =  this.faces();
                    int numOfFaces = cellFaces.length;
                    boolean isCutting = false; // initial setting
                    for(int i=0; i<nodesNum-1; i++)
                        {
                            for(int j=i+1; j<nodesNum; j++)
                                {
                                    Line connLine = new Line(isInNodeKoor[i], isInNodeKoor[j],
                                                             true);
                                    Coordinate cutPt = new Coordinate();
                                    for(int k=0; k<numOfFaces; k++)
                                        {
                                            isCutting = cutPt.cut(cellFaces[k],connLine,true);
                                            if(isCutting){break;}
                                            else{}
                                        }
                                }
                        }
                    if(isCutting){return true;}
                    else{return false;}
                }
        }
}


/**
   Returns the faces of a cell, with the Faces being oriented towards the cell
     (cross product of side-vectors).
*/
public Face[] faces()
{
    // Orientation of the cell
    Coordinate d[]= new Coordinate[2];
    if((Coordinate.spat(dir[1],dir[2],dir[0]))>=0.0)
        {d[0] = new Coordinate(dir[1]); d[1] = new Coordinate(dir[2]);}
    else
        {d[0] = new Coordinate(dir[2]); d[1] = new Coordinate(dir[1]);}

    Coordinate U1 = new Coordinate(this); U1.add(d[0]);
    Coordinate U2 = new Coordinate(this); U2.add(d[1]);
    Coordinate O = new Coordinate(this); O.add(dir[0]);
    int planeNum;if(type==1){planeNum=6;}else{planeNum=5;}
    Face P[] = new Face[planeNum];

    P[0] = new Face(this,dir[0],d[0],1); // Seite1
    P[1] = new Face(this,d[1],dir[0],1); // Seite2
    if(type==1) // Brick - case
        {
            P[2]=new Face(U1,dir[0],d[1],1); // Seite3
            P[3]=new Face(U2,d[0],dir[0],1); // Seite4
            P[4]=new Face(this,d[0],d[1],1); // unterer Deckel
            P[5]=new Face(O,d[1],d[0],1); // oberer Deckel
        }
    else // Wedge - case
        {
            Coordinate U1U2 = new Coordinate(U2); U1U2.add(U1,-1.0);
            P[2]=new Face(U1,dir[0],U1U2,1); // schraege Hinterseite
            P[3]=new Face(this,d[0],d[1],0); // unterer Deckel
            P[4]=new Face(O,d[1],d[0],0); // oberer Deckel
        }
    return P;
}



public void WriteScr()
{
    super.WriteScr(); for(int i=0; i<3; i++){dir[i].WriteScr();}
```

```
        System.out.println(type);
    }

}
```

C.2.3 The SpereCoordinate class

```
/*
 * Copyright 2000-2002 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

// ********************************************************************************************
// *********************************   Class SphereCoordinate  ********************************
// ********************************************************************************************
/**
Coordinate in spherical coordinates (in &#8476;<sup>3</sup>)
@author Manuel Alexandre Pouchon, &copy 2000-2001 JNC Japan
 */
public class SphereCoordinate
{
    /**
Angles from origin to point in &#8476;<sup>3</sup>, <b>&#966;<sub>0</sub></b> is the angle around the
x<sub>2</sub> (or <em>z</em>) - axis, <b>&#966;<sub>1</sub></b> is the elevation angle relative to the
x<sub>0</sub>-x<sub>1</sub> (<em>or x-y</em>) - plane!
     */
    public double phi[]; // phi[0] angle around x2 axis; phi[1] angle to x0x1 plane
    /**
        Distance from origin to point in &#8476;<sup>3</sup>
     */
    public double r;

    public SphereCoordinate()
    {
        phi = new double[2];
        phi[0] = 0; phi[1] = 0;
        r = 1;
    }

    public SphereCoordinate(double PhiX3Inp, double PhiX12Inp, double Radius)
    {
        phi = new double[2];
        phi[0] = PhiX3Inp;
        phi[1] = PhiX12Inp;
        r = Radius;
    }

    public SphereCoordinate(SphereCoordinate ToCopy)
    {
        phi = new double[2];
        phi[0] = ToCopy.phi[0];
        phi[1] = ToCopy.phi[1];
        r = ToCopy.r;
    }

    public SphereCoordinate(Coordinate Trsf)
    {
        phi = new double[2];

        double rx0x1 = Math.sqrt(Math.pow(Trsf.x[0],2.0)+Math.pow(Trsf.x[1],2.0));
        // radius in x0x1-Plane
        r = Math.sqrt(Math.pow(Trsf.x[0],2.0)+Math.pow(Trsf.x[1],2.0)+Math.pow(Trsf.x[2],2.0));
        // radius

        if (r==0)
            {
                phi[0]=0; phi[1]=1;
            }
        else
            {
                if (rx0x1==0)
                    {
```

```
                        phi[0] = 0;
                        phi[1] = Math.asin(1);
                    }
                else
                    {
                        if(Trsf.x[1]<0)
                            {
                                phi[0] = -Math.acos(Trsf.x[0]/rx0x1);
                            }
                        else
                            {
                                phi[0] = Math.acos(Trsf.x[0]/rx0x1);
                            }
                        phi[1] = Math.asin(Trsf.x[2]/r);
                    }
            }
    }


    public void assign(double PhiX3Inp, double PhiX12Inp, double Radius)
    {
        phi[0] = PhiX3Inp;
        phi[1] = PhiX12Inp;
        r = Radius;
    }

    public void assign(SphereCoordinate ToCopy)
    {
        phi[0] = ToCopy.phi[0];
        phi[1] = ToCopy.phi[1];
        r = ToCopy.r;
    }


    public void assign(Coordinate Trsf)
    {
        phi = new double[2];

double rx0x1 = Math.sqrt(Math.pow(Trsf.x[0],2.0)+Math.pow(Trsf.x[1],2.0));
// radius in x0x1-Plane
r = Math.sqrt(Math.pow(Trsf.x[0],2.0)+Math.pow(Trsf.x[1],2.0)+Math.pow(Trsf.x[2],2.0));
// radius

        if (r==0)
            {
                phi[0]=0; phi[1]=1;
            }
        else
            {
                if (rx0x1==0)
                    {
                        phi[0] = 0;
                        phi[1] = Math.asin(1);
                    }
                else
                    {
                        if(Trsf.x[1]<0)
                            {
                                phi[0] = -Math.acos(Trsf.x[0]/rx0x1);
                            }
                        else
                            {
                                phi[0] = Math.acos(Trsf.x[0]/rx0x1);
                            }
                        phi[1] = Math.asin(Trsf.x[2]/r);
                    }
            }
    }

    public void scale(double factor)
    {
        double phi0 = phi[0], phi1 = phi[1], rr = r;
        if(factor<0.0)
            {
                phi[1] = -phi1;
                phi[0] = (phi0+(2*Math.acos(.0)))-
                    (((int)((phi0+(2*Math.acos(.0)))/(4*Math.acos(.0))))*(4*Math.acos(.0)));
```

```
                r = rr*Math.abs(factor);
            }
        else
            {
                r = rr*factor;
            }
    }

    public void rotate(double phi0add, double phi1add)
    {
        double phi0 = phi[0], phi1 = phi[1], phi0Tot = phi0+phi0add, phi1Tot = phi1+phi1add;
        if(Math.abs(phi1Tot)>(Math.acos(.0)))
            {
                phi[1] = -(((2*Math.acos(.0))*(Math.abs(phi1Tot)/phi1Tot))-phi1Tot);
                phi[0] = (phi0Tot+(2*Math.acos(.0)))-
                    (((int)((phi0Tot+(2*Math.acos(.0)))/(4*Math.acos(.0))))*(4*Math.acos(.0)));
            }
        else
            {
                phi[1] = phi1Tot;
                phi[0] = (phi0Tot)-(((int)((phi0Tot)/(4*Math.acos(.0))))*(4*Math.acos(.0)));
            }
    }

    public void transform(double phi0add, double phi1add, double rscale)
    {
        scale(rscale);
        rotate(phi0add, phi1add);
    }

    public void transform(double phi0add, double phi1add, double rscale, double partial)
    {
        scale(rscale*partial);
        rotate((phi0add*partial), (phi1add*partial));
    }


    public void transform(SphereCoordinate trsf)
    {
        scale(trsf.r);
        rotate(trsf.phi[0], trsf.phi[1]);
    }

    public void transform(SphereCoordinate trsf, double partial)
    {
        scale(trsf.r*partial);
        rotate((trsf.phi[0]*partial), (trsf.phi[1]*partial));
    }

    public void toTransform(SphereCoordinate trFrom, SphereCoordinate trTo)
    {
        SphereCoordinate a = new SphereCoordinate(trFrom);
        SphereCoordinate b = new SphereCoordinate(trTo);
        r = b.r/a.r; // radius scaling-factor
        b.rotate(-a.phi[0],-a.phi[1]);
        if(b.phi[0]>(2*Math.acos(0)))
            {
                b.phi[0]=b.phi[0]-(4*Math.acos(0));
            }
        phi[0] = b.phi[0];
        phi[1] = b.phi[1];
    }



    public void WriteScr()
    {
        System.out.println( " ( r: " + r + " / phi1 : " + phi[0] + " / phi2: " + phi[1] + " ) " );
    }
}
```

C.2.4 The line class

```
/*
 * Copyright 2000-2002 JNC Japan, Inc. All Rights Reserved.
 */
```

```
package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;


// ********************************************************************************************
// *********************************   Class Line  ********************************************
// ********************************************************************************************

public class Line extends Coordinate
{
    //    public Coordinate orig;
    public Coordinate dir;

    //*******************  Constructors ***********************

    public Line()
    {
        super();
        dir = new Coordinate();
    }

    public Line(double ox1, double ox2, double ox3, double dx1,
double dx2, double dx3, boolean endPoints)
    {
        super();
        if(endPoints)  // input is origin and endpoint (endPoints = true)
            {

                Coordinate oVec = new Coordinate(ox1,ox2,ox3);
                Coordinate eVec = new Coordinate(dx1,dx2,dx3);
                 Coordinate dVec = new Coordinate();
                dVec.add(eVec,oVec,-1.0);
                super.assign(oVec);
                dir = new Coordinate(dVec);
            }
        else   // input is origin and direction
            {
                super.assign(ox1,ox2,ox3);
                dir = new Coordinate(dx1,dx2,dx3);
            }
    }

    public Line(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3)
    {
        super(ox1,ox2,ox3);
        dir = new Coordinate(dx1,dx2,dx3);
    }

    public Line(Coordinate o, Coordinate d, boolean endPoints)
// endPoints: input of endpoints -> true (see above)
    {
        super();
        Line L = new Line(o.x[0],o.x[1],o.x[2],d.x[0],d.x[1],d.x[2], endPoints);
        super.assign(L);
        dir = new Coordinate(L.dir);
    }

    public Line(Coordinate o, Coordinate d)
    {
        super(o);
        dir = new Coordinate(d);
    }

    public Line(Line toCopy)
    {
        super(toCopy);
        dir = new Coordinate(toCopy.dir);
    }

    /**
       Constructor with a Multiple line as argument. The multiple line is simplified to one single
line. If the line is one connected, non circular line, the first and last point span the new Line, if
it is circular, the two points having the biggest distance span the new line, same for a multiple line
composed of several separated lines. Of course this always signifies an information loss!
```

```
       @param toConvert Line which is going to be converted into the new single line.
     */
    public Line(MultiLine toConvert)
    {
        super();
        int start = -1;
        int end = -1;
        int pathLength = toConvert.path.length; //System.out.println(" "+pathLength);
        if(pathLength>=4)
            {
                if( (toConvert.numPaths == 1) && !toConvert.circular )
                    {
                        start = toConvert.path[1]; end = toConvert.path[pathLength-2];
                    }
                else
                    {
                        double maxDist = 0.0;
                        int numPoints = toConvert.points.length;
                        for(int i=0; i<numPoints-1; i++){for(int j=i+1; j<numPoints; j++){
                            double dist = toConvert.points[i].distance(toConvert.points[j]);
                            if(dist > maxDist){maxDist = dist; start = i; end = j;}
                            }}
                    }
                //System.out.println(" "+start+" "+end);
                super.assign(toConvert.points[start]);
                this.dir = new Coordinate(toConvert.points[end]);
                this.dir.add(toConvert.points[start],-1);
            }
        else
            {
System.out.println(" ***** ERROR: MultiLine - Element is false,
not enough entities in the path!");
                new Line(0.0,0.0,0.0,0.0,0.0,0.0);
            }
    }

    //*******************  assign ******************************

    public void assign(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3,
                       boolean endPoints)
    {
        if(endPoints)  // input is origin and endpoint (endPoints = true)
            {
                Coordinate oVec = new Coordinate(ox1,ox2,ox3);
                Coordinate eVec = new Coordinate(dx1,dx2,dx3);
                 Coordinate dVec = new Coordinate();
                dVec.add(eVec,oVec,-1.0);
                super.assign(oVec);
                dir.assign(dVec);
            }
        else   // input is origin and direction
            {
                super.assign(ox1,ox2,ox3);
                dir.assign(dx1,dx2,dx3);
            }
    }

    public void assign(double ox1, double ox2, double ox3, double dx1, double dx2, double dx3)
    {
        super.assign(ox1,ox2,ox3);
        dir.assign(dx1,dx2,dx3);
    }

    public void assign(Coordinate o, Coordinate d, boolean endPoints)
   // endPoints: input of endpoints -> true (see above)
    {
        Line L = new Line(o.x[0],o.x[1],o.x[2],d.x[0],d.x[1],d.x[2], endPoints);
        super.assign(L);
        dir.assign(L.dir);
    }

    public void assign(Coordinate o, Coordinate d)
    {
        super.assign(o);
        dir.assign(d);
    }
```

```
public void assign(Line toCopy)
{
    super.assign(toCopy);
    dir.assign(toCopy.dir);
}

/**
    Simplifies multiple line to one simple line,
    works in the same way like the constructor with a multiple line as argument!
   @param toConvert Multiple line which is going to be assigned as a simplified line.
   @see jp.go.jnc.tokai.pouchon.geometry.Line#Line(MultiLine)
 */
public void assign(MultiLine toConvert)
{
    Line toAssign = new Line(toConvert);
    this.assign(toAssign);
}

//****************************************** length ************************************

/**
   Returns the lenght of the line.
   @return length of the line.
 */
    public double length()
{
    return dir.abs();
}



//********************************* CUT ****************************************

/**
   Cuts two planes P1 and P2 and assigns the cutting line.
   @param P1 first plane to cut
   @param P2 second plane to cut
 */
public void cut(Plane P1, Plane P2)
{
    Coordinate diff = new Coordinate(), b = new Coordinate(), midpt = new Coordinate();

    // different origin of both planes P1 and P2, otherwise trivial case
    if (P2.x[0] != P1.x[0] || P2.x[1] != P1.x[1] || P2.x[2] != P1.x[2])
        {
            Line c = new Line();
            diff.add(P2,P1,-1.0); // difference vector of two plane origines
            // which plane is more parallel to diff?
            //-> line along this plane to other plane for finding new common orig
            if ( Math.abs(diff.dot(P2.normal)) < Math.abs(diff.dot(P1.normal)))
                {
                    b.cross(P2.normal,diff);
                    c.assign(P2);
                    c.dir.cross(P2.normal,b);
                    midpt.cut(P1,c);
                }
            else
                {
                    b.cross(P1.normal,diff);
                    c.assign(P1);
                    c.dir.cross(P1.normal,b);
                    midpt.cut(P2,c);
                };
        }
    else
        {
            midpt.assign(this);
        };
    this.assign(midpt);
    b.cross(P2.normal,P1.normal);
    dir.assign(b);
}


/**
   Cuts a polygons with a plane and assigns the cutting line.
   @param pol polygon to cut
```

```
   @param pla plane to cut
 */
public boolean cut(Polygon pol, Plane pla)
{
    boolean result = false;
    boolean isCut = false;

nonparallel:
    if(pol.topology!=4) // Polygons being a triangle
        {
            Coordinate relCorns[] = new Coordinate[3];
            // triangle-corners relativ to plane-origin
            for(int i=0;i<3;i++)
    {relCorns[i]=new Coordinate(pol);relCorns[i].add(pol.corner[i]);relCorns[i].add(pla,-1.0);}

            /*Coordinate sides[] = new Coordinate[3];     // side-vectors of the triangle
            for(int i=0;i<3;i++)
            {sides[i]=new Coordinate(pol.corner[(i+1)%3]);sides[i].add(pol.corner[i],-1.0);}*/

            double dist[] = new double[3];    // distance of triangle-corners to plane
            for(int i=0;i<3;i++){dist[i]=Coordinate.dot(pla.normal,relCorns[i]);}

            int order[] = new int[3];
            // distances in an incresing order (signum is important -> not abs distance!)
            for(int i=0;i<3;i++)  // finding smallest
                {if(dist[i]<=dist[(i+1)%3] && dist[i]<=dist[(i+2)%3]){order[0]=i;}}
            if(dist[(order[0]+1)%3]<=dist[(order[0]+2)%3])         // second and third order
                {order[1]=(order[0]+1)%3;order[2]=(order[0]+2)%3;}
            else{order[2]=(order[0]+1)%3;order[1]=(order[0]+2)%3;}

            if((dist[order[0]]*dist[order[2]])<0.0)
            // most far points are on oposit sides of the plane -> is cutting
                {
                    isCut = true;

                    {                   //connection between farest points
                    Coordinate side =
                    new Coordinate(pol.corner[order[2]]);side.add(pol.corner[order[0]],-1.0);
                    double inters = dist[order[0]]/(dist[order[0]]-dist[order[2]]);
                    this.assign(pol); this.add(pol.corner[order[0]]); this.add(side,inters);
    //  <--- intersection of line between farest points with plane assigned to line origin.
                    }

                    if(dist[order[1]]==0.0)
                    // middle point lies on the plane -> already cutting point
                        {this.dir.assign(pol); this.dir.add(pol.corner[order[1]]);
                        this.dir.add(this,-1.0);} // <---- dir vector of line assigned

                    else{if(dist[order[1]]<0.0)
                    // middle point lies under the plane -> intersecion between upper two
                        {
                            Coordinate side = new Coordinate(pol.corner[order[2]]);
                            side.add(pol.corner[order[1]],-1.0);
                            double inters = dist[order[1]]/(dist[order[1]]-dist[order[2]]);
                            this.dir.assign(pol); this.dir.add(pol.corner[order[1]]);
                            this.dir.add(side,inters);
                            this.dir.add(this,-1.0);
                        }
                    else
                    // middle point lies above the plane -> 2'nd secion between lower two points
                        {
                            Coordinate side = new Coordinate(pol.corner[order[1]]);
                            side.add(pol.corner[order[0]],-1.0);
                            double inters = dist[order[0]]/(dist[order[0]]-dist[order[1]]);
                            this.dir.assign(pol); this.dir.add(pol.corner[order[0]]);
                            this.dir.add(side,inters);
                            this.dir.add(this,-1.0);
                        }}
                }
            else
                {
                    isCut = false;
                    this.assign(0.0,0.0,0.0);
                }
        }
    else
        {
```

```
            MultiLine tempLine = new MultiLine();
            isCut = tempLine.cut(pol,pla);    // generally a multiple line is returned
            this.assign(tempLine);            // simplification with to a simple line
        }
    return isCut;
}



// ************************************************
// ************************************************

/**
   Cuts two polygons P1 and P2 and assigns the cutting line.
   @param P1 first plane to cut
   @param P2 second plane to cut
 */
public boolean cut(Polygon P1, Polygon P2)
{
    boolean result = false;

nonparallel:
    if(P1.topology!=4 && P2.topology!=4) // both Polygons being triangles
        {
            // three sides of first trianle, defined counterclockwise
            Coordinate P1S[] = new Coordinate[3];
            P1S[0] = new Coordinate(P1.corner[0], P1.corner[1]);
            P1S[1] = new Coordinate(P1.corner[1], P1.corner[2]);
            P1S[2] = new Coordinate(P1.corner[2], P1.corner[0]);

            // three sides of second trianle, defined counterclockwise
            Coordinate P2S[] = new Coordinate[3];
            P2S[0] = new Coordinate(P2.corner[0], P2.corner[1]);
            P2S[1] = new Coordinate(P2.corner[1], P2.corner[2]);
            P2S[2] = new Coordinate(P2.corner[2], P2.corner[0]);

            // normal-vectors of both triangles
            Coordinate P1Normal = new Coordinate(); P1Normal.cross(P1S[0],P1S[1]);
            P1Normal.mult(1.0);
            Coordinate P2Normal = new Coordinate(); P2Normal.cross(P2S[0],P2S[1]);
            P2Normal.mult(1.0);

            if(P1Normal.abs()==0 || P2Normal.abs()==0)
                {
                    System.out.println(" ******  ERROR, at leaste one of the triangles is
                                        degenerated to a line !!!");
                    break nonparallel;
                }

            // Cutting-line of two planes
            Line cuttingPlanes = new Line();

            // intersections of first triangle with infinit plane of second triangle
            // triangle-line intersections given by (P2Normal dot DO) / (P2Normal dot P1S[x])

            double iDiv[] = new double[3];
            // (P2Normal dot P1S[x]) determination because of possible div by 0
            for(int i=0;i<3;i++){iDiv[i] = P2Normal.dot(P1S[i]);}
            // --> special cases like: planes being parallel, .....

            int indSmall = 0, ind1 = 0, ind2 = 0;

            // search smallest iDiv[x], x <- indSmall, others: ind1 & ind2
            for(int i1=0; i1<3; i1++)
            //  ---->  iDiv[indSmall] <= iDiv[ind1] <= iDiv[ind2]
                {
                    int i2 = (i1+1)%3, i3 = (i1+2)%3;
                    if((Math.abs(iDiv[i1])<=Math.abs(iDiv[i2]))
                            && (Math.abs(iDiv[i1])<=Math.abs(iDiv[i3])))
                        {
                            indSmall = i1;
                            if(Math.abs(iDiv[i2])<Math.abs(iDiv[i3]))
                                    {ind1=i2; ind2=i3;}else{ind1=i3; ind2=i2;}
                        }
                }

            // System.out.println(" "+indSmall+" "+ind1+" "+ind2+"  "+iDiv[indSmall]+"
```

```
// "+iDiv[ind1]+" "+iDiv[ind2]);

if(iDiv[ind1] == 0 || iDiv[ind2] == 0)
// then iDiv[indSmall] is also 0 of course and planes are parallel
    {
        System.out.println(" ******  Special case:
                              planes defined by triangles are parallel in 3D !!!");
        break nonparallel;
    }
else if(iDiv[indSmall] == 0)
 // this line is parallel to second plane -> can directly be taken as cuttin dir
    {
        Coordinate p2corn0_p1cornInd2 = new Coordinate(P1, P2);
        p2corn0_p1cornInd2.add(P2.corner[0],1.0);
        p2corn0_p1cornInd2.add(P1.corner[ind2],-1.0);
        // for numerical stability: div by biggest number
        double cuttingPar = P2Normal.dot(p2corn0_p1cornInd2)/iDiv[ind2];

        cuttingPlanes.assign(P1); cuttingPlanes.add(P1.corner[ind2]);
        // corner position
        cuttingPlanes.add(P1S[ind2],cuttingPar);

        cuttingPlanes.dir.assign(P1S[indSmall]);
    }
else   //  general case
    {
        Coordinate p2corn0_p1cornInd1 = new Coordinate(P1, P2);
        p2corn0_p1cornInd1.add(P2.corner[0],1.0);
        Coordinate p2corn0_p1cornInd2 = new Coordinate(p2corn0_p1cornInd1);

        p2corn0_p1cornInd1.add(P1.corner[ind1],-1.0);
        p2corn0_p1cornInd2.add(P1.corner[ind2],-1.0);

        double cuttingPar1 = P2Normal.dot(p2corn0_p1cornInd1)/iDiv[ind1];
        // System.out.println(cuttingPar1);
        double cuttingPar2 = P2Normal.dot(p2corn0_p1cornInd2)/iDiv[ind2];
        // System.out.println(cuttingPar2);

        Coordinate end = new Coordinate(P1); end.add(P1.corner[ind1]);
        // corner position
        end.add(P1S[ind1],cuttingPar1);

        cuttingPlanes.assign(P1); cuttingPlanes.add(P1.corner[ind2]);
        // corner position
        cuttingPlanes.add(P1S[ind2],cuttingPar2);

        cuttingPlanes.dir.assign(end); cuttingPlanes.dir.add(cuttingPlanes,-1);
    }

// defining normals of the trianle sides, normals are oriented towards inner side
// and the origin of each side, relative to the line origin
Coordinate PSN[][] = new Coordinate[2][3];
Coordinate PSO[][] = new Coordinate[2][3];    // two planes, three sides
for(int i=0; i<3; i++)
    {
        PSN[0][i] = new Coordinate(); PSN[0][i].cross(P1Normal,P1S[i]);
        // PSN[0][i].WriteScr();
        PSN[1][i] = new Coordinate(); PSN[1][i].cross(P2Normal,P2S[i]);
        // PSN[1][i].WriteScr();
        PSO[0][i] = new Coordinate(cuttingPlanes); PSO[0][i].mult(-1.0);
        PSO[0][i].add(P1,1.0); PSO[0][i].add(P1.corner[i],1.0);
        // PSO[0][i].WriteScr();
        PSO[1][i] = new Coordinate(cuttingPlanes); PSO[1][i].mult(-1.0);
        PSO[1][i].add(P2,1.0); PSO[1][i].add(P2.corner[i],1.0);
        // PSO[1][i].WriteScr();
    }

double enter=0.0, exit=0.0;
// parameter of last entering and first exit
boolean enterDef=false, exitDef=false;
// has already a real value been assigned to the parameters

for(int i=0; i<3; i++)
    {
        for(int j=0; j<2; j++)
            {
                double div = PSN[j][i].dot(cuttingPlanes.dir);
```

```
                            //System.out.println(" div "+j+i+" is "+div);

                            if(div>0.0)   // ray is entering plane
                                {
                                    double par = (PSN[j][i].dot(PSO[j][i]))/div;
                                    //System.out.println(" value -> "+ par);
                                    if(par>enter || !enterDef){enter = par; enterDef = true;}
                                }
                            else if(div<0.0)   // ray is exiting plane
                                {
                                    double par = (PSN[j][i].dot(PSO[j][i]))/div;
                                    //System.out.println(" value -> "+ par);
                                    if(par<exit  || !exitDef ){exit  = par; exitDef  = true;}
                                }
                        }
                    }
                if(enter<exit)
                    {
                        result = true;
                        this.assign(cuttingPlanes); this.add(cuttingPlanes.dir,enter);
                        this.dir.assign(0.0,0.0,0.0);
                        this.dir.add(cuttingPlanes.dir,exit); this.dir.add(cuttingPlanes.dir,-enter);
                    }
                else
                    {
                        result = false;
                        this.assign(cuttingPlanes); this.dir.normalize();
                    }
            }
        else
            {
                MultiLine tempLine = new MultiLine();
                result = tempLine.cut(P1,P2);   // generally a multiple line is returned
                this.assign(tempLine);          // simplification with to a simple line
            }

        return result;
    }

//*********************************************************************************************

    public boolean equal(Line toCompare)
    {
        Coordinate P1O = new Coordinate(this), P1E = new Coordinate(P1O); P1E.add(this.dir,+1.0);
        Coordinate P2O = new Coordinate(toCompare), P2E = new Coordinate(P2O);
        P2E.add(toCompare.dir,+1.0);

        if( (P1O.equal(P2O) && P1E.equal(P2E)) || (P1O.equal(P2E) && P1E.equal(P2O)) )
            {return true;}
        else
            {return false;}
    }




//*********************************************************************************************


    /**
       Cuts two planes P1 and P2 and assigns the cutting line.
       @param P1 first plane to cut
       @param P2 second plane to cut
       @return boolean if P1 and P2 intersect or not, returns also false, if Polygons are parallel
       @deprecated use cut instead, this old version contains a bug
     */
    public boolean cutOld(Polygon P1, Polygon P2)
    {
        Line cutLine = new Line();
        Coordinate lineOrig = new Coordinate(), lineDir = new Coordinate();
        boolean result = false, resultTria1 = false, resultTria2 = false;

    nonplane:
        if(P1.topology!=4 && P2.topology!=4) // both Polygons being triangles
            {
                Coordinate DO = new Coordinate(P1, P2);
```

```
        Coordinate P1R = new Coordinate(P1.corner[0], P1.corner[1]); // P1R.WriteScr();
        Coordinate P1L = new Coordinate(P1.corner[0], P1.corner[2]); // P1L.WriteScr();
        Coordinate P2R = new Coordinate(P2.corner[0], P2.corner[1]); // P2R.WriteScr();
        Coordinate P2L = new Coordinate(P2.corner[0], P2.corner[2]); // P2L.WriteScr();


// Equation: (P1R * R1) + (PlL * L1) == DO + (P2R * R2) + (P2L * L2)  <-- intersection
//  --> R2 = f(L2) = (R2C + (R2L * L2))
//  --> R1 = f(L2) = (R1C + (R1L * L2))        // everything expresses as function of L2
//  --> L1 = f(L2) = (L1C + (L1L * L2))

        double nen =
            - P1L.x[2]*P1R.x[1]*P2R.x[0] + P1L.x[1]*P1R.x[2]*P2R.x[0]
            + P1L.x[2]*P1R.x[0]*P2R.x[1] - P1L.x[0]*P1R.x[2]*P2R.x[1]
            - P1L.x[1]*P1R.x[0]*P2R.x[2] + P1L.x[0]*P1R.x[1]*P2R.x[2];

        if(nen == 0.0)
        // maybe just the P2L axis is parallel to first plane, or two planes are parallel
            {
                Coordinate transposeRL = new Coordinate(P2R);
                // transpose two sides of second triangle
                P2R.assign(P2L);
                // orientation for this rutine not important
                P2L.assign(transposeRL);
                nen =
                    - P1L.x[2]*P1R.x[1]*P2R.x[0] + P1L.x[1]*P1R.x[2]*P2R.x[0]
                    + P1L.x[2]*P1R.x[0]*P2R.x[1] - P1L.x[0]*P1R.x[2]*P2R.x[1]
                    - P1L.x[1]*P1R.x[0]*P2R.x[2] + P1L.x[0]*P1R.x[1]*P2R.x[2];
                if(nen == 0)
                // test also failed for swiched second triangle --> planes are parallel
                    {
                        System.out.println("  ERROR -- the two triangles are parallel !!
                        Division by " + nen);
                        break nonplane;
                    }
            }

        // System.out.println(" division by: "+nen);

double R2C = (1.0/nen) * (
+ DO.x[2]*P1L.x[1]*P1R.x[0] - DO.x[1]*P1L.x[2]*P1R.x[0] - DO.x[2]*P1L.x[0]*P1R.x[1]
+ DO.x[0]*P1L.x[2]*P1R.x[1] + DO.x[1]*P1L.x[0]*P1R.x[2] - DO.x[0]*P1L.x[1]*P1R.x[2]);
double R2L = (1.0/nen) * (
+ P1L.x[2]*P1R.x[1]*P2L.x[0] - P1L.x[1]*P1R.x[2]*P2L.x[0] - P1L.x[2]*P1R.x[0]*P2L.x[1]
+ P1L.x[0]*P1R.x[2]*P2L.x[1] + P1L.x[1]*P1R.x[0]*P2L.x[2] - P1L.x[0]*P1R.x[1]*P2L.x[2]);

double R1C = (1.0/nen) * (
- DO.x[2]*P1R.x[1]*P2R.x[0] + DO.x[1]*P1R.x[2]*P2R.x[0] + DO.x[2]*P1R.x[0]*P2R.x[1]
- DO.x[0]*P1R.x[2]*P2R.x[1] - DO.x[1]*P1R.x[0]*P2R.x[2] + DO.x[0]*P1R.x[1]*P2R.x[2]);
double R1L = (1.0/nen) * (
- P1L.x[2]*P2L.x[1]*P2R.x[0] + P1L.x[1]*P2L.x[2]*P2R.x[0] + P1L.x[2]*P2L.x[0]*P2R.x[1]
- P1L.x[0]*P2L.x[2]*P2R.x[1] - P1L.x[1]*P2L.x[0]*P2R.x[2] + P1L.x[0]*P2L.x[1]*P2R.x[2]);

double L1C = (1.0/nen) * (
- DO.x[2]*P1R.x[1]*P2R.x[0] + DO.x[1]*P1R.x[2]*P2R.x[0] + DO.x[2]*P1R.x[0]*P2R.x[1]
- DO.x[0]*P1R.x[2]*P2R.x[1] - DO.x[1]*P1R.x[0]*P2R.x[2] + DO.x[0]*P1R.x[1]*P2R.x[2]);
double L1L = (1.0/nen) * (
+ P1R.x[2]*P2L.x[1]*P2R.x[0] - P1R.x[1]*P2L.x[2]*P2R.x[0] - P1R.x[2]*P2L.x[0]*P2R.x[1]
+ P1R.x[0]*P2L.x[2]*P2R.x[1] + P1R.x[1]*P2L.x[0]*P2R.x[2] - P1R.x[0]*P2L.x[1]*P2R.x[2]);

double U1, O1, U2, O2, U3, O3; // Temporary variables of lower and upper limits

        // cutting of second triangle with first infinit plane, values expressed for L2

        double L2UP2, L2OP2;
        // L2 Parameters for second triangle:   L2UP2 -> lower limit  ;  L2OP2 -> upper limit

        double R2_EQUAL_0          = (0.0-R2C)/(R2L) ;    //  R2      = 0
        double R2_EQUAL_1          = (1.0-R2C)/(R2L) ;    //  R2      = 1
        double R2_PLUS_L2_EQUAL_0  = (0.0-R2C)/(1+R2L);   //  R2 + L2 = 0
        double R2_PLUS_L2_EQUAL_1  = (1.0-R2C)/(1+R2L);   //  R2 + L2 = 1

        //System.out.println(R2_EQUAL_0);
        //System.out.println(R2_EQUAL_1);
         //System.out.println(R2_PLUS_L2_EQUAL_0);
        //System.out.println(R2_PLUS_L2_EQUAL_1);
```

```
if(R2_EQUAL_0 <= R2_EQUAL_1){U1=R2_EQUAL_0; O1=R2_EQUAL_1;}
else{U1=R2_EQUAL_1; O1=R2_EQUAL_0;}

U2 = 0.0; O2 = 1.0;

if(R2_PLUS_L2_EQUAL_0 <= R2_PLUS_L2_EQUAL_1){U3=R2_PLUS_L2_EQUAL_0;
O3=R2_PLUS_L2_EQUAL_1;}
else{U3=R2_PLUS_L2_EQUAL_1; O3=R2_PLUS_L2_EQUAL_0;}

if(U1<=U3 && U2<=U3){L2UP2=U3;}else{if(U1<=U2){L2UP2=U2;}else{L2UP2=U1;}}
if(O1>=O3 && O2>=O3){L2OP2=03;}else{if(O1>=O2){L2OP2=O2;}else{L2OP2=O1;}}

//System.out.println(L2UP2+" "+L2OP2);

if(L2UP2<=L2OP2){resultTria2 = true;}else{resultTria2 = false;}
// does second Triangle touch first Plane?


// cutting of first triangle with second infinit plane, values expressed for L2

double L2UP1, L2OP1;
// L2 Parameters for first triangle:   L2UP1 -> lower limit  ;  L2OP1 -> upper limit

double R1_EQUAL_0          = (0.0-R1C)/(R1L) ;
double R1_EQUAL_1          = (1.0-R1C)/(R1L);
double L1_EQUAL_0          = (0.0-L1C)/(L1L);
double L1_EQUAL_1          = (1.0-L1C)/(L1L);
double R1_PLUS_L1_EQUAL_0  = (0.0-(R1C+L1C))/(R1L+L1L);
double R1_PLUS_L1_EQUAL_1  = (1.0-(R1C+L1C))/(R1L+L1L);

if(R1_EQUAL_0 <= R1_EQUAL_1){U1=R1_EQUAL_0; O1=R1_EQUAL_1;}
else{U1=R1_EQUAL_1; O1=R1_EQUAL_0;}

if(L1_EQUAL_0 <= L1_EQUAL_1){U2=L1_EQUAL_0; O2=L1_EQUAL_1;}
else{U2=L1_EQUAL_1; O2=L1_EQUAL_0;}

if(R1_PLUS_L1_EQUAL_0 <= R1_PLUS_L1_EQUAL_1){U3=R1_PLUS_L1_EQUAL_0;
O3=R1_PLUS_L1_EQUAL_1;}
else{U3=R1_PLUS_L1_EQUAL_1; O3=R1_PLUS_L1_EQUAL_0;}

if(U1<=U3 && U2<=U3){L2UP1=U3;}else{if(U1<=U2){L2UP1=U2;}else{L2UP1=U1;}}
if(O1>=O3 && O2>=O3){L2OP1=O3;}else{if(O1>=O2){L2OP1=O2;}else{L2OP1=O1;}}

if(L2UP1<=L2OP1){resultTria1 = true;}else{resultTria1 = false;}
// does first Triangle touch second Plane?

//System.out.println(resultTria1);
//System.out.println(resultTria2);

// cutting of both triangles

double L2U_BOTH, L2O_BOTH;    // L2 limits for both triangles

if(L2UP1 >= L2UP2){L2U_BOTH = L2UP1;}else{L2U_BOTH = L2UP2;}
if(L2OP1 <= L2OP2){L2O_BOTH = L2OP1;}else{L2O_BOTH = L2OP2;}

if(L2U_BOTH<L2O_BOTH)
    {
        double R2U_BOTH = R2C + (R2L * L2U_BOTH);
        double R2O_BOTH = R2C + (R2L * L2O_BOTH);

        result=true;
        lineOrig.assign(P2); lineOrig.add(P2.corner[0]);
        lineOrig.add(P2L,L2U_BOTH); lineOrig.add(P2R,R2U_BOTH);
        lineDir.assign(P2); lineOrig.add(P2.corner[0]);
        lineDir.add(P2L,L2O_BOTH); lineDir.add(P2R,R2O_BOTH);
        lineDir.add(lineOrig,-1);
    }
else    // no cutting lines of triangles, but return cut line of infinite-planes
    {
        L2O_BOTH = L2U_BOTH + 1;     // just setting any value
        double R2U_BOTH = R2C + (R2L * L2U_BOTH);
        double R2O_BOTH = R2C + (R2L * L2O_BOTH);

        result=false;
        lineOrig.assign(P2); lineOrig.add(P2.corner[0]);
        lineOrig.add(P2L,L2U_BOTH); lineOrig.add(P2R,R2U_BOTH);
```

```
                        lineOrig.assign(P2); lineOrig.add(P2.corner[0]);
                        lineDir.add(P2L,L2O_BOTH); lineDir.add(P2R,R2O_BOTH);
                        lineDir.add(lineOrig,-1);
                        lineDir.normalize();
                    }

                cutLine.assign(lineOrig,lineDir);
            }
        this.assign(cutLine);
        return result;
    }

    //****************** WriteScr ******************************

    public void WriteScr()
    {
        System.out.print(" Line: | ");
        for(int i=0; i<3; i++){System.out.print(this.x[i]+" ");}
        System.out.print("| + t* | ");
        for(int i=0; i<3; i++){System.out.print(dir.x[i]+" ");}
        System.out.println("| ");
    }

}
```

C.2.5 The plane class

```
/*
 * Copyright 1996-2000 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

// ********************************************************************************************
// *********************************   Class Plane ********************************************
// ********************************************************************************************

/**
<i><b>Infinit plane:</b></i> here represented by a <i>point</i> on the plane and the <i>normal</i> to
the plane
 */
public class Plane extends Coordinate
{

    public Coordinate normal;

    //*******************  Constructors ************************

    /**
       Generates plane with origin and normal being (0,0,0)
     */
    public Plane()
    {
        super();
        normal = new Coordinate();
    }

    /**
       Generates a plane represented by an origin and two direction vectors:
       (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
       If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is
produced.
     */
    public Plane(double ox1, double ox2, double ox3,
                 double d1x1, double d1x2, double d1x3,
                 double d2x1, double d2x2, double d2x3,
                 boolean pointsOnPlane)
    {
        this();
        if(pointsOnPlane)  // input is three points on the plane (pointsOnPlane = true)
            {
                Coordinate oVec = new Coordinate(ox1,ox2,ox3);
                Coordinate e1Vec = new Coordinate(d1x1,d1x2,d1x3);
```

```
                Coordinate e2Vec = new Coordinate(d2x1,d2x2,d2x3);
                 Coordinate d1Vec = new Coordinate();
                 Coordinate d2Vec = new Coordinate();
                Coordinate nVec = new Coordinate();
                d1Vec.add(e1Vec,oVec,-1.0);
                d2Vec.add(e2Vec,oVec,-1.0);
                nVec.cross(d1Vec,d2Vec);
                nVec.normalize();
                this.assign(oVec);
                normal = new Coordinate(nVec);
            }
        else   // input is origin and direction
            {
                 Coordinate d1Vec = new Coordinate(d1x1,d1x2,d1x3);
                 Coordinate d2Vec = new Coordinate(d2x1,d2x2,d2x3);
                Coordinate nVec = new Coordinate();
                nVec.cross(d1Vec,d2Vec);
                nVec.normalize();
                this.assign(ox1,ox2,ox3);
                normal = new Coordinate(nVec);
            }
    }

    /**
       Generates a plane represented by an origin and two direction vectors:
       (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
    */
    public Plane(double ox1, double ox2, double ox3,
                 double d1x1, double d1x2, double d1x3,
                 double d2x1, double d2x2, double d2x3)
    {
        super();
        Coordinate d1Vec = new Coordinate(d1x1,d1x2,d1x3);
        Coordinate d2Vec = new Coordinate(d2x1,d2x2,d2x3);
        Coordinate nVec = new Coordinate();
        nVec.cross(d1Vec,d2Vec);
        nVec.normalize();
        super.assign(ox1,ox2,ox3);
        normal = new Coordinate(nVec);
    }

    /**
       Generates a plane represented by an origin and two direction vectors:
       o+u(d1)+v(d2).
       If "pointsOnPlane" is set to "true", then a plane going trough the
       coordinates (o, d1, d2) is produced.
     */
    public Plane(Coordinate o, Coordinate d1, Coordinate d2, boolean pointsOnPlane)
    {
        super();
        Plane P = new
Plane(o.x[0],o.x[1],o.x[2],d1.x[0],d1.x[1],d1.x[2],d2.x[0],d2.x[1],d2.x[2],pointsOnPlane);
        super.assign(P);
        normal = new Coordinate(P.normal);
    }

    /**
       Generates a plane represented by an origin and two direction vectors:
       o+u(d1)+v(d2).
    */
    public Plane(Coordinate o, Coordinate d1, Coordinate d2)
    {
        super();
        Plane P = new Plane(o.x[0],o.x[1],o.x[2],d1.x[0],d1.x[1],d1.x[2],d2.x[0],d2.x[1],d2.x[2]);
        super.assign(P);
        normal = new Coordinate(P.normal);
    }


    public Plane(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3,
                 boolean endPoints)
    {
        super();
        if(endPoints)
            {
                Coordinate oVec = new Coordinate(ox1,ox2,ox3);
                Coordinate eVec = new Coordinate(nx1,nx2,nx3);
```

```
            Coordinate nVec = new Coordinate();
            nVec.add(eVec,oVec,-1.0);
            super.assign(oVec);
            normal = new Coordinate(nVec);
        }
    else
        {
            super.assign(ox1,ox2,ox3);
            normal = new Coordinate(nx1,nx2,nx3);
        }
}

public Plane(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3)
{
    super(ox1,ox2,ox3);
    normal = new Coordinate(nx1,nx2,nx3);
}

public Plane(Coordinate o, Coordinate n, boolean pointsOnPlane)
{
    this(o.x[0],o.x[1],o.x[2],n.x[0],n.x[1],n.x[2], pointsOnPlane);
}

public Plane(Coordinate o, Coordinate n)
{
    super(o);
    normal = new Coordinate(n);
}

public Plane(Plane toCopy)
{
    super(toCopy);
    normal = new Coordinate(toCopy.normal);
}

/**
   Constructor of plane with Polygon as argument. In both cases, when the polygon
    is a triangle or a quadragle, the plane is defined by the first three corners.
   @param toTrsf polygon being transformed into a plane
 */
public Plane(Polygon toTrsf)
{
    super();
    super.add(toTrsf); super.add(toTrsf.corner[0]);

    Coordinate side[] = new Coordinate[2];
    side[0] = new Coordinate(toTrsf.corner[1]); side[0].add(toTrsf.corner[0],-1.0);
    side[1] = new Coordinate(toTrsf.corner[2]); side[1].add(toTrsf.corner[0],-1.0);

    Coordinate nor = new Coordinate();
    nor.cross(side[0],side[1]);
    nor.normalize();

    this.normal = new Coordinate(nor);
}

//*******************  assign  *******************************************

/**
   Assigns a plane represented by an origin and two direction vectors:
   (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
   If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is
    produced.
 */
public void assign(double ox1, double ox2, double ox3,
                   double d1x1, double d1x2, double d1x3,
                   double d2x1, double d2x2, double d2x3,
                   boolean pointsOnPlane)
{
    if(pointsOnPlane)  // input is three points on the plane (pointsOnPlane = true)
        {
            Coordinate oVec = new Coordinate(ox1,ox2,ox3);
            Coordinate e1Vec = new Coordinate(d1x1,d1x2,d1x3);
            Coordinate e2Vec = new Coordinate(d2x1,d2x2,d2x3);
             Coordinate d1Vec = new Coordinate();
             Coordinate d2Vec = new Coordinate();
            Coordinate nVec = new Coordinate();
```

```
            d1Vec.add(e1Vec,oVec,-1.0);
            d2Vec.add(e2Vec,oVec,-1.0);
            nVec.cross(d1Vec,d2Vec);
            nVec.normalize();
            super.assign(oVec);
            normal.assign(nVec);
        }
    else   // input is origin and direction
        {
            Coordinate d1Vec = new Coordinate(d1x1,d1x2,d1x3);
            Coordinate d2Vec = new Coordinate(d2x1,d2x2,d2x3);
           Coordinate nVec = new Coordinate();
           nVec.cross(d1Vec,d2Vec);
           nVec.normalize();
           super.assign(ox1,ox2,ox3);
           normal.assign(nVec);
        }
}

/**
   Assigns a plane represented by an origin and two direction vectors:
   (ox1,ox2,ox3)+u(d1x1,d1x2,d1x3)+v(d2x1,d2x2,d2x3).
*/
public void assign(double ox1, double ox2, double ox3,
                   double d1x1, double d1x2, double d1x3,
                   double d2x1, double d2x2, double d2x3)
{
    Coordinate d1Vec = new Coordinate(d1x1,d1x2,d1x3);
    Coordinate d2Vec = new Coordinate(d2x1,d2x2,d2x3);
    Coordinate nVec = new Coordinate();
    nVec.cross(d1Vec,d2Vec);
    nVec.normalize();
    super.assign(ox1,ox2,ox3);
    normal.assign(nVec);
}

/**
   Assigns a plane represented by an origin and two direction vectors:
   o+u(d1)+v(d2).
   If "pointsOnPlane" is set to "true", then a plane going trough the coordinates (o, d1, d2) is
    produced.
 */
public void assign(Coordinate o, Coordinate d1, Coordinate d2, boolean pointsOnPlane)
{
    Plane P =
   new Plane(o.x[0],o.x[1],o.x[2],d1.x[0],d1.x[1],d1.x[2],d2.x[0],d2.x[1],d2.x[2],pointsOnPlane);
    super.assign(P);
    normal.assign(P.normal);
}

/**
   Assigns a plane represented by an origin and two direction vectors:
   o+u(d1)+v(d2).
*/
public void assign(Coordinate o, Coordinate d1, Coordinate d2)
{
    Plane P = new Plane(o.x[0],o.x[1],o.x[2],d1.x[0],d1.x[1],d1.x[2],d2.x[0],d2.x[1],d2.x[2]);
    super.assign(P);
    normal.assign(P.normal);
}


public void assign(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3,
                   boolean endPoints)
{
    if(endPoints)
        {
            Coordinate oVec = new Coordinate(ox1,ox2,ox3);
            Coordinate eVec = new Coordinate(nx1,nx2,nx3);
            Coordinate nVec = new Coordinate();
            nVec.add(eVec,oVec,-1.0);
            super.assign(oVec);
            normal.assign(nVec);
        }
    else
        {
            super.assign(ox1,ox2,ox3);
```

```
                normal.assign(nx1,nx2,nx3);
            }
    }

    public void assign(double ox1, double ox2, double ox3, double nx1, double nx2, double nx3)
    {
        super.assign(ox1,ox2,ox3);
        normal.assign(nx1,nx2,nx3);
    }

    public void assign(Coordinate o, Coordinate n, boolean pointsOnPlane)
    {
        Plane P = new Plane(o.x[0],o.x[1],o.x[2],n.x[0],n.x[1],n.x[2], pointsOnPlane);
        super.assign(P);
        normal.assign(P.normal);
    }

    public void assign(Coordinate o, Coordinate n)
    {
        super.assign(o);
        normal.assign(n);
    }

    public void assign(Plane toCopy)
    {
        super.assign(toCopy);
        normal.assign(toCopy.normal);
    }

    //*******************  WriteScr()  ************************

    public void WriteScr()
    {
        System.out.print(" Plane: | ");
        for(int i=0; i<3; i++){System.out.print(super.x[i]+" ");}
        System.out.print("| + n* | ");
        for(int i=0; i<3; i++){System.out.print(normal.x[i]+" ");}
        System.out.println("| ");
    }

}
```

C.2.6 The Polygon class

```
package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.math.*;

//*****************************************************************************************
// *********************************   Class Polygon ***********************************
// *****************************************************************************************

/**
Polygon in &real<sup>3</sup>, it can be a triangle or a quadrangle, the quadrangle should be convex.
If not so, the order procedure will modifiy correctly.
*/
public class Polygon extends Coordinate
{
    // ****************************** Varables of the Class ************************************

    /**
       Corners of the polygon relative to the Origin (number of corners depends from Polygon type
defined in "type")
    */
    public Coordinate corner[];
    /**
       Type of polygon, 4 types are defined as follows (compartible to FEMAP neutral file):
<b>Triangle</b> - with 3 corners -> <b><em>2</em></b> - with 6 corners -> 3 , <b>Quadragle</b> with 4
corners -> <b><em>4</em></b> - with 8 corners -> 5 , both, Triangle and Quadrangle define a surface,
for the triangle in general two different surfaces, being trianles, can be defined.
    */
    public int topology;
    /**
```

```
       Surface splitting parameter: For a triangle this parameter is irrelevant, for the quadrangle
the subdivision into triangles can be performed in two ways, by calling the four corners c1, c2, c3,
c4 the splitting line can be defined from c1 to c3 which is default and is called right to left (lower
triangle is right, upper is left), in this case rt_lf is set to true, if splitting line is from c2 to
c4 the rt_lf parameter is set to false. The subdivision in necessary for the general case, where the
four corners are not in one plane and when one wants to know, wether a point lies above or under the
surface.
     */
    public boolean rt_lf;


   // ************************* Constructors of the Class *********************************

    /**
       Default constructor creates structure for triangle
     */
    public Polygon()
    {
        super();
        corner = new Coordinate[3];
        for(int i=0; i<3; i++){corner[i] = new Coordinate();}
        topology = 2;
        rt_lf = true;
    }

    /**
       Consturcts Polygon with "numCorn" corners, the "rt_lf" parameter is set to "true"
       @param numCorn defines the number of corners, only 3 and 4 are accepted for the moment,
        everythin else is set to 3
    */
    public Polygon(int numCorn)
    {
        super();

        int j;
        if (numCorn == 3)
            {
               j = 3;
               topology = 2;
               rt_lf = true;
            }
        else
            {
               if (numCorn == 4)
                   {
                      j =4;
                      topology = 4;
                      rt_lf = true;
                   }
               else
                   {
                      j = 3;   // Default-value
                      topology = 2;
                      rt_lf = true;
                   }
            }

        corner = new Coordinate[j];
        for(int i=0; i<j; i++){corner[i] = new Coordinate();}
    }

    /**
       Constructs a Quadrangle, because only here setting the surface splitting orientation parameter
        rt_lf makes sense!
       @param is_rt_lf sets the surface splitting orientation of the quadrangle, if set to true it is
        composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left
        and a upper-right triangle.
     */
    public Polygon(boolean is_rt_lf)
    {
        super();
        corner = new Coordinate[4];
        for(int i=0; i<4; i++){corner[i] = new Coordinate();}
        topology = 4;
        rt_lf = is_rt_lf;
    }
```

```
    /**
       Constructs a Triangle. Define corners counterclockwise, as given as example in the parameter
        explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back corner
       @param toOrder should polygon be controlled and reorganized when initialized
     */
    public Polygon(boolean toOrder, Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3)
    {
        super(ori);
        corner = new Coordinate[3];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        topology = 2;
        rt_lf = true;    // defined but not used
        if(toOrder){
            if(!this.order(false)){ System.out.println("   ERROR in Polygon initial., propably
        screwed of primitive");}}
    }

    /**
       Constructs a Triangle. Define corners counterclockwise, as given as example in the parameter
explanation. Control is performed.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back corner
     */
    public Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3)
    {
        this(true,ori,c1,c2,c3);
    }


    /**
       Constructs a Quadrangle with the parameter "lt-rt" set to default value "true", this means that
the surface is split into a lower-right and an upper-left triangle. Define corners counterclockwise,
as given as example in the parameter explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
       @param toOrder should polygon be controlled and reorganized when initialized
     */
    public Polygon(boolean toOrder, Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3,
Coordinate c4)
    {
        super(ori);
        corner = new Coordinate[4];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        corner[3] = new Coordinate(c4);
        topology = 4;
        rt_lf = true;    // set of default value
        if(toOrder){
            if(!this.order(false)){ System.out.println("   ERROR in Polygon initial., probably
                screwed of primitive");}}
    }

    /**
       Constructs a Quadrangle with the parameter "lt-rt" set to default falue "true", this means that
the surface is split into a lower-right and an upper-left triangle. Define corners counterclockwise,
as given as example in the parameter explanation. Control is performed.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
    */
    public Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4)
    {
        this(true,ori,c1,c2,c3,c4);
```

```
    }

    /**
       Constructs a Quadrangle. Define corners counterclockwise, as given as example in the parameter
explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
       @param is_rt_lf sets the surface splitting orientation of the quadrangle, if set to true it is
composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a
upper-right triangle.
       @param toOrder should polygon be controlled and reorganized when initialized
     */
    public Polygon(boolean toOrder, Coordinate ori, Coordinate c1,Coordinate c2,Coordinate
c3,Coordinate c4,boolean is_rt_lf)
    {
        super(ori);
        corner = new Coordinate[4];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        corner[3] = new Coordinate(c4);
        topology = 4;
        rt_lf = is_rt_lf;    // set of default value
        if(toOrder){
            if(!this.order(false)){ System.out.println("   ERROR in Polygon initial., probably
               screwed of primitive");}}
    }

    /**
       Constructs a Quadrangle. Define corners counterclockwise, as given as example in the
        parameter explanation. Control is perfomed.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
       @param is_rt_lf sets the surface splitting orientation of the quadrangle, if set to
        true it is composed of a lower-right and a upper-left triangle, otherwise it is
        composed of a lower-left and a upper-right triangle.
    */
    public Polygon(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4,
                      boolean is_rt_lf)
    {
        this(true,ori,c1,c2,c3,c4,is_rt_lf);
    }

    /**
       Copy constructor, assigns the given Polygon to a new created one.
       @param toCopy Polygon to be assigned to the new created one
     */
    public Polygon(Polygon toCopy, boolean toOrder)
    {
        super(toCopy);

        int j = toCopy.corner.length;

        corner = new Coordinate[j];
        for(int i=0; i<j; i++){corner[i] = new Coordinate(toCopy.corner[i]);}
        topology = toCopy.topology;
        rt_lf = toCopy.rt_lf;
        if(toOrder){
            if(!this.order(false)){ System.out.println("   ERROR in Polygon initial.,
               probably screwed of primitive");}}
    }

    public Polygon(Polygon toCopy)
    {
        this(toCopy, true);
    }

    // ************************************************** Assigns of the Class
*********************************
```

```
    /**
       Assigns a Triangle. Define corners counterclockwise,
        as given as example in the parameter explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back corner
     */
    public void assign(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3)
    {
        if(topology != 2)
            {
                System.out.println(" !!!!!!!!!!!!! ERROR, wrong surface type assigned, original type
is not triangle !!!!");
            }
        else
            {
                super.assign(ori);
                corner = new Coordinate[3];
                corner[0] = new Coordinate(c1);
                corner[1] = new Coordinate(c2);
                corner[2] = new Coordinate(c3);
                topology = 2;
                rt_lf = true;    // defined but not used
                if(!this.order(false)){ System.out.println("   ERROR in Polygon initial.,
                probably screwed of primitive");}
            }
    }

    /**
       Assigns a Quadrangle. The parameter "lt-rt" is kept as defined before.
        Define corners counterclockwise, as given as example in the parameter explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
     */
    public void assign(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4)
    {
        if(topology != 4)
            {
                System.out.println(" !!!!!!!!!!!!! ERROR, wrong surface type assigned,
                original type is not quadrangle !!!");
            }
        else
            {
                super.assign(ori);
                corner[0].assign(c1);
                corner[1].assign(c2);
                corner[2].assign(c3);
                corner[3].assign(c4);
                topology = 4;
                // rt_lf keep old surface splitting orientation type
                if(!this.order(false)){ System.out.println("   ERROR in Polygon initial.,
                probably screwed of primitive");}
            }
    }

    /**
       Assigns a Quadrangle. Define corners counterclockwise,
        as given as example in the parameter explanation.
       @param ori Origin and reference point of the triangle
       @param c1 e.g. front-left corner
       @param c2 e.g. front-right corner
       @param c3 e.g. back-right corner
       @param c4 e.g. back-left corner
       @param is_rt_lf sets the surface splitting orientation of the quadrangle, if set to true it is
composed of a lower-right and a upper-left triangle, otherwise it is composed of a lower-left and a
upper-right triangle.
     */
    public void assign(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4,
boolean is_rt_lf)
    {
        if(topology != 4)
            {
```

```
                System.out.println(" !!!!!!!!!!!!! ERROR, wrong surface type assigned, original type
is not quadrangle !!!");
            }
        else
            {
                super.assign(ori);
                corner[0].assign(c1);
                corner[1].assign(c2);
                corner[2].assign(c3);
                corner[3].assign(c4);
                topology = 4;
                rt_lf = is_rt_lf;
                if(!this.order(false)){ System.out.println("   ERROR in Polygon initial.,
                        probably screwed of primitive");}
            }
    }

    /**
       Copy constructor, assigns the given Polygon to a new created one.
       @param toCopy Polygon to be assigned to the new created one
     */
    public void assign(Polygon toCopy)
    {
        if(topology != toCopy.topology)
            {
                System.out.println(" !!!!!!!!!!!!! ERROR, wrong surface type assigned, original type
not equal to assigned one !!!");
            }
        else
            {
                int j = toCopy.corner.length;

                super.assign(toCopy);
                for(int i=0; i<j; i++){corner[i].assign(toCopy.corner[i]);}
                topology = toCopy.topology;
                rt_lf = toCopy.rt_lf;
                if(!this.order(false)){ System.out.println("   ERROR in Polygon initial.,
                        probably screwed of primitive");}
            }
    }


    // ***************************** order ********************************************************

    /**
       Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the
order.
       @param out Set to true if the routine should report corrections
       @return If the polygon was ok, or if sucessful corrections could be applied, the routine
returns true, if the quadrangle is irreparable screwed, the routine returns false.
     */
    public boolean order(boolean out)
    {
        boolean ok;

        Permutation toChange = new Permutation(this);

        ok = toChange.pointOrder(this, out);

        this.permute(toChange);

        return ok;
    }



    /**
       Takes a polygon and verifies whether the order of corners is valid, if not so, it corrects the
order.
       @param out Set to true if the routine should report corrections
       @return If the polygon was ok, or if sucessful corrections could be applied, the routine
returns true, if the quadrangle is irreparable screwed, the routine returns false.
     */
    public boolean orderOld(boolean out)
    {
        boolean ok = true;
```

```
    int numCorner = this.corner.length;

    if(numCorner == 3)
        {
            if(topology != 2)
               {if(out){System.out.println(" Error -- wrong polygon-topology detected
                                        (for triangle): "
                                       +this.topology+" instead of 2! <- corrected");}
                this.topology =2;}
            Coordinate nor = new Coordinate();
            nor.cross(this.corner[0],this.corner[1],this.corner[2]);
            if(nor.abs() == 0.0)
                {System.out.println(" Error -- polygon (in this case triangle) is degenerated");}
            // no other test is needed, triangle can't be screwed
        }

    if(numCorner == 4)
        {
            if(topology != 4)
               {if(out){System.out.println(" Error -- wrong polygon-topology detected
                                             (for triangle): "
                                       +this.topology+" instead of 4! <- corrected");}
                this.topology = 4;}
            Coordinate nor1 = new Coordinate();
            nor1.cross(this.corner[0],this.corner[1],this.corner[3]);
            Coordinate nor2 = new Coordinate();
            nor2.cross(this.corner[1],this.corner[2],this.corner[0]);
            Coordinate nor3 = new Coordinate();
            nor3.cross(this.corner[2],this.corner[3],this.corner[1]);
            Coordinate nor4 = new Coordinate();
            nor4.cross(this.corner[3],this.corner[0],this.corner[2]);

            double c2 = Coordinate.dot(nor1,nor2);
            double c3 = Coordinate.dot(nor1,nor3);
            double c4 = Coordinate.dot(nor1,nor4);

            if(c2<0 || c3<0 || c4<0)
                {
                   if(c2*c3*c4 < 0)
                       {
                           System.out.println("  Warning -- Convex Quadrangle
                                               <- no correction possible");
                       }
                   else
                       {
                           if(out){System.out.println(" Error –
                                       Quadrangle is screwed <- corrected");}
                           if(c2<0 && c3<0)
                               {Coordinate per = new Coordinate(this.corner[1]);
                               this.corner[1].assign(this.corner[2]);
                               this.corner[2].assign(per);}
                           else{if(c3<0 && c4<0)
                               {Coordinate per = new Coordinate(this.corner[2]);
                               this.corner[2].assign(this.corner[3]);
                               this.corner[3].assign(per);}
                           else{if(c2<0 && c4<0)
                               {Coordinate per = new Coordinate(this.corner[1]);
                               this.corner[1].assign(this.corner[3]);
                               this.corner[3].assign(per);}
                           else{System.out.println(" ERROR in polygon order,
                                       inpossible case in permutation rout");}}}
                       }
                }
            else
                {
                   // all same direction -> ok
                }
        }
    return ok;
}


//*********************** permute ***************************************************

/**
   Permutates any Polygon-corner entities with the given permutation instance, toApply.
   @param toApply Permutation instance to be applied to the Polygon-corner entities.
```

```
       @return True if permutation was successful. False if the length of <em>toApply</em> does not
correspond to the number of corners.
     */
    public boolean permute(Permutation toApply)
    {
        boolean ok = true;

        int N = this.corner.length;

        if(N == toApply.order.length)
            {
                int transNum = 0;
                for(int i=0; i<N; i++)
                    {if(toApply.order[i]!=i){transNum++;}}
                if(transNum>0)
                    {
                        int perms[] = new int[transNum];
                        Coordinate permsCont[] = new Coordinate[transNum];
                        {
                            int M=0;
                            for(int i=0; i<N; i++)
                                {if(toApply.order[i]!=i)
                                    {perms[M]=i;permsCont[M]=
                                        new Coordinate(this.corner[toApply.order[i]]);M++;}}
                        }
                        for(int i=0;i<transNum;i++)
                            {this.corner[perms[i]].assign(permsCont[i]);}
                    }
                ok = true;
            }
        else{ok = false;}

        return ok;
    }

    // ********************************************** convex
*****************************************************

    /**
       Checks whether polygon surface is convex or concave, triangle surface is allways convex, for
the quadrangle surface the inner side is defined by the right hand rule (-> inner side in direction of
cross product c1-c2 x c1-c4), the result also depends on the surface splitting oreintation!
       @return Reurns "true" if the surface is found to be convex, "false" otherwise, flat is defined
as convex (of course)!
     */
    public boolean convex()
    {
        boolean result = true;

        if(topology == 2)
            {
                result = true;
            }
        else
            {
                Coordinate right = new Coordinate(corner[0],corner[1]);
                Coordinate left = new Coordinate(corner[0],corner[3]);
                Coordinate middle = new Coordinate(corner[0],corner[2]);
                 double orient = Coordinate.spat(right, left, middle);
                if(orient==0)
                    {
                        result = true;
                    }
                else
                    {
                        if(orient > 0)
                            {
                                if(rt_lf){result = false;}else{result = true;}
                            }
                        else
                            {
                                if(rt_lf){result = true;}else{result = false;}
                            }
                    }
            }
        return result;
    }
```

```
    // ********************** above ********************************************************

    /**
       Takes a coordinate "isAbove" and checks whether this is above the surface or not. The direction
is thereby given by the right hand rule with the surface being oriented conterclockwise with the
corners.
       @param isAbove Coordinates of the point being tested to be above suface
       @return Returns "true" if point "isAbove" is above the surface, "false" otherwise.
     */
    public boolean above(Coordinate isAbove)
    {
       Coordinate isAboveRel = new Coordinate(isAbove);  // will be modified further on

       boolean result = true;

       if (topology == 4)  // -> Quadrangle
           {
               Coordinate M, R, L; // defining the two subtriangles, right: R-M, left: M-L
               if(rt_lf)
                   {
                       isAboveRel.add(this,-1); isAboveRel.add(this.corner[0],-1);
                       // isAbove relative to c1
                       M = new Coordinate(corner[0],corner[2]);
                       R = new Coordinate(corner[0],corner[1]);
                       L = new Coordinate(corner[0],corner[3]);
                   }
               else
                   {
                       isAboveRel.add(this,-1); isAboveRel.add(this.corner[1],-1);
                       // isAbove relative to c2
                       M = new Coordinate(corner[1],corner[3]);
                       R = new Coordinate(corner[1],corner[2]);
                       L = new Coordinate(corner[1],corner[0]);
                   }
               double distR = Coordinate.spat(R, M, isAboveRel);
               double distL = Coordinate.spat(M, L, isAboveRel);
               if(this.convex())
                   {
                       if(distR >= 0 && distL >= 0){result = true;}else{result = false;}
                   }
               else
                   {
                       if(distR < 0 && distL < 0){result = false;}else{result = true;}
                   }
           }
       else   // supose topology = 2 ! -> Triangle
           {
               Coordinate R, L; // defining triangle R-L
               R = new Coordinate(corner[0],corner[1]);
               L = new Coordinate(corner[0],corner[2]);
               isAboveRel.add(this,-1); isAboveRel.add(corner[0],-1);
               // isAbove relative to c1
               double dist = Coordinate.spat(R, L, isAboveRel);
               if(dist >= 0){result = true;}else{result = false;}
           }
       return result;
    }


/**
       Takes a coordinate "isAbove" and checks whether this is above the surface or not. The direction
is thereby given by the right hand rule with the surface being oriented conterclockwise with the
corners.
       @param isAbove Coordinates of the point being tested to be above suface
       @param tol distance which is needed, that point is considered to be above.
       @return Returns "true" if point "isAbove" is above the surface, "false" otherwise.
     */
    public boolean above(Coordinate isAbove, double tol)
    {
       Coordinate isAboveRel = new Coordinate(isAbove);  // will be modified further on

       boolean result = true;

       if (topology == 4)  // -> Quadrangle
           {
               Coordinate M, R, L; // defining the two subtriangles, right: R-M, left: M-L
```

```
                if(rt_lf)
                    {
                        isAboveRel.add(this,-1); isAboveRel.add(this.corner[0],-1);
                        // isAbove relative to c1
                        M = new Coordinate(corner[0],corner[2]);
                        R = new Coordinate(corner[0],corner[1]);
                        L = new Coordinate(corner[0],corner[3]);
                    }
                else
                    {
                        isAboveRel.add(this,-1); isAboveRel.add(this.corner[1],-1);
                        // isAbove relative to c2
                        M = new Coordinate(corner[1],corner[3]);
                        R = new Coordinate(corner[1],corner[2]);
                        L = new Coordinate(corner[1],corner[0]);
                    }
                double distR = Coordinate.spat(R, M, isAboveRel);
                double distL = Coordinate.spat(M, L, isAboveRel);
                if(this.convex())
                    {
                        if(distR >= tol && distL >= tol){result = true;}else{result = false;}
                    }
                else
                    {
                        if(distR < tol && distL < tol){result = false;}else{result = true;}
                    }
            }
        else   // supose topology = 2 ! -> Triangle
            {
                Coordinate R, L; // defining triangle R-L
                R = new Coordinate(corner[0],corner[1]);
                L = new Coordinate(corner[0],corner[2]);
                isAboveRel.add(this,-1); isAboveRel.add(corner[0],-1);    // isAbove relative to c1
                double dist = Coordinate.spat(R, L, isAboveRel);
                if(dist >= tol){result = true;}else{result = false;}
            }
        return result;
    }

/**
       Takes a Polygon "isAbove" and checks whether this is above the calling Polygon or not. The
direction is thereby given by the right hand rule with the surface being oriented conterclockwise with
the corners. When all corners are above, then the whole surface is above.
       @param isAbove Polygon being tested to be above suface
       @param pos If set to false, the orientation of the calling surface can be temporarely canged
for this check, therefore the <em>isAbove</em> Polygon can be verified to be under the calling
polygon!
       @return Returns "true" if Polygon "isAbove" is above the surface, "false" otherwise.
     */
    public boolean above(Polygon isAbove, boolean pos)
    {
        int N = isAbove.corner.length;

        boolean ok = true;

        boolean itIs[] = new boolean[N];

        for(int i=0;i<N;i++)
            {
                Coordinate toCheck = new Coordinate(isAbove);
                toCheck.add(isAbove.corner[i]);
                itIs[i] = this.above(toCheck);
            }

        for(int i=0;i<N;i++)
            {
                if( (!itIs[i] && pos) || (itIs[i] && !pos) )
                    {ok = false;}
            }
        return ok;
    }

    // ************************** Cut ******************************************************

    /**
       Cuts a polygon with a second one. The procedure-object is modified in the following way: If the
two polygons don't interact at all, no modification, if the the Polygon given as parameter only partly
```

```
cuts trough the calling polygon, the calling polygon is cut as if the parameter polygon was an infinit
plane, this is also the case for a comletly cutting second polygon! The routine can change the
topology of the polygon, a triangle can become a quadrangle. If the resulting polygon would be a
pentagle (eg. when cutting a quadrangle with one corner lying outside), a simplification is performed,
the corner lying outside is projected to the cutting plane.
       @param cutting Polygon which cuts the first polygon. The part of the polygon being above the
plane is returned.
       @return True if the calling polygon was cut by the parameter-polygon.
      */
    public boolean cut(Polygon cutting)
    {
        Line cutLine = new Line();
        boolean isCut = false;

        Plane pla = new Plane(cutting);
        isCut = cutLine.cut(this,pla);

        if(isCut)
            {Plane polyPla = new Plane(this);
            boolean isCutPl = cutLine.cut(this,pla);

            Coordinate newCorns[] = new Coordinate[2];
                // new corners in polygon system (relative to its origin)
            newCorns[0] = new Coordinate(cutLine); newCorns[0].add(this,-1.0);
            newCorns[1] = new Coordinate(newCorns[0]); newCorns[1].add(cutLine.dir);


            if(this.topology!=4) // triangle
                {
                    Coordinate relCorns[] = new Coordinate[3];
                        // triangle-corners relativ to plane-origin
                    for(int i=0;i<3;i++)
                        {relCorns[i]=
                   new Coordinate(this);relCorns[i].add(this.corner[i]);relCorns[i].add(pla,-1.0);}

                    double dist[] = new double[3];
                        // distance of triangle-corners to plane
                    for(int i=0;i<3;i++){dist[i]=Coordinate.dot(pla.normal,relCorns[i]);}

                    int order[] = new int[3];
                        // distances in an incresing order (signum important -> not abs distance!)
                    for(int i=0;i<3;i++)  // finding smallest
                        {if(dist[i]<=dist[(i+1)%3] && dist[i]<=dist[(i+2)%3]){order[0]=i;}}
                    if(dist[(order[0]+1)%3]<=dist[(order[0]+2)%3])         // second and third order
                        {order[1]=(order[0]+1)%3;order[2]=(order[0]+2)%3;}
                    else{order[2]=(order[0]+1)%3;order[1]=(order[0]+2)%3;}

                    //for(int i=0;i<3;i++){System.out.println(" dist "+i+" is "+dist[order[i]]);}

                    if(dist[order[1]]>0.0) // two points above --> result is a convex quadrangle
                        {
                            // System.out.println("  to quadrangle!");
                            this.topology = 4; // change of topology

                            if(order[0]==0)
                                // outside element has index 0, change pattern: (.xx) -> (..xx)
                                {
                                    Coordinate quad[] = new Coordinate[4];
                                    for(int i=2;i<4;i++)
                                // last two triangle entities are copied to the last quadr. ent
                                        {quad[i] = new Coordinate(this.corner[i-1]);}

                                    Coordinate dirOld = new Coordinate(), dirNew = new Coordinate();
                                    dirOld.cross(this.corner[1],this.corner[2],newCorns[0]);
                                    dirNew.cross(this.corner[2],newCorns[0],newCorns[1]);

                                    double orient = Coordinate.dot(dirOld,dirNew);
                                    if(orient>=0){quad[0]=
                                new Coordinate(newCorns[0]);quad[1]=new Coordinate(newCorns[1]);}
                                    else{quad[0]=new Coordinate(newCorns[1]);quad[1]=
                                new Coordinate(newCorns[0]);}

                                    this.corner = new Coordinate[4];
                                    for(int i=0;i<4;i++){this.corner[i]=new Coordinate(quad[i]);}
                                }
                            else{if(order[0]==1) // outside element has index 1,
                                                 // change pattern: (x.x) -> (x..x)
```

```
                        {
                            Coordinate quad[] = new Coordinate[4];

                            quad[0] = new Coordinate(this.corner[0]);
                                      // see change pattern above!
                            quad[3] = new Coordinate(this.corner[2]);

                            Coordinate dirOld = new Coordinate(), dirNew = new Coordinate();
                            dirOld.cross(this.corner[0],newCorns[0],this.corner[2]);
                            dirNew.cross(this.corner[0],newCorns[0],newCorns[1]);

                            double orient = Coordinate.dot(dirOld,dirNew);
                            if(orient>=0){quad[1]=new Coordinate(newCorns[0]);quad[2]=
                                new Coordinate(newCorns[1]);}
                            else{quad[1]=new Coordinate(newCorns[1]);quad[2]=
                                new Coordinate(newCorns[0]);}

                            this.corner = new Coordinate[4];
                            for(int i=0;i<4;i++){this.corner[i]=
                                new Coordinate(quad[i]);}
                        }
                    else{if(order[0]==2)
                        // outside element has index 1, change pattern: (xx.) -> (xx..)
                        {
                            Coordinate quad[] = new Coordinate[4];

                            for(int i=0;i<2;i++)
                        // first two triangle entities are copied to first quadr. ent
                                {quad[i] = new Coordinate(this.corner[i]);}

                            Coordinate dirOld = new Coordinate(), dirNew = new Coordinate();
                            dirOld.cross(this.corner[0],this.corner[1],newCorns[0]);
                            dirNew.cross(this.corner[1],newCorns[0],newCorns[1]);

                            double orient = Coordinate.dot(dirOld,dirNew);
                            if(orient>=0){quad[2]=new Coordinate(newCorns[0]);
                                quad[3]=new Coordinate(newCorns[1]);}
                            else{quad[2]=new Coordinate(newCorns[1]);
                                quad[3]=new Coordinate(newCorns[0]);}

                            this.corner = new Coordinate[4];
                            for(int i=0;i<4;i++){this.corner[i]=new Coordinate(quad[i]);}
                        }}}

                }
            else        // one points above --> result is a triangle
                {
                     Coordinate dirOld = new Coordinate(), dirNew = new Coordinate();
                     dirOld.cross(this.corner[0],this.corner[1],this.corner[2]);
                     dirNew.cross(this.corner[order[2]],newCorns[0],newCorns[1]);
                     double orient = Coordinate.dot(dirOld,dirNew);

                     if(orient>=0)
                         {this.corner[(order[2]+1)%3].assign(newCorns[0]);
                         this.corner[(order[2]+2)%3].assign(newCorns[1]);}
                     else
                         {this.corner[(order[2]+1)%3].assign(newCorns[1]);
                         this.corner[(order[2]+2)%3].assign(newCorns[0]);}
                }
        }
    else
        {


            Coordinate relCorns[] = new Coordinate[4];
                // triangle-corners relativ to plane-origin
            for(int i=0;i<4;i++)
                {relCorns[i]=
        new Coordinate(this);relCorns[i].add(this.corner[i]);relCorns[i].add(pla,-1.0);}

            double dist[] = new double[4];    // distance of triangle-corners to plane
            for(int i=0;i<4;i++){dist[i]=Coordinate.dot(pla.normal,relCorns[i]);}

            int order[] = new int[4];
                // distances in an incresing order (signum important -> not abs distance!)
            for(int i=0;i<4;i++)  // finding smallest
                {if(dist[i]<=dist[(i+1)%4] &&
```

```
                        dist[i]<=dist[(i+2)%4] &&
                        dist[i]<=dist[(i+3)%4]){order[0]=i;}}
                for(int i=1;i<4;i++)  // finding second smallest
                    {if(dist[(order[0]+i)%4]<=dist[(order[0]+i+1)%4] &&
                        dist[(order[0]+i)%4]<=dist[(order[0]+i+2)%4])
                        {order[1]=(order[0]+i)%4;}}
                for(int i=0;i<4;i++)  // finding biggest
                    {if(dist[i]>=dist[(i+1)%3] &&
                        dist[i]>=dist[(i+2)%3] &&
                        dist[i]>=dist[(i+3)%3] &&
                        i != order[0] &&
                        i != order[1]){order[3]=i;}}
                for(int i=0;i<4;i++)  // find remaining
                    {if(i != order[0] && i != order[1] && i != order[3])
                        {order[2]=i;}}

                //for(int i=0;i<4;i++){System.out.println(" dist "+i+" is "+dist[order[i]]);}


                if(dist[order[3]]<=0) // biggest rel. distance under plane -> no interaction
                    {
                        isCut = false;
                    }
                else{if(dist[order[2]]<=0)
                            // one point is above plane -> new polygon is a quadrangle
                    {
                        isCut = true;

                        int newOrder[] = new int[3];
                        newOrder[2]=(order[3]+3)%4;    // last entity before biggest one
                        newOrder[0]=order[3];          // biggest entitiy
                        newOrder[1]=(order[3]+1)%4;    // next entity after biggest one


    if(newOrder[1]==0){newOrder[1]=newOrder[2];newOrder[2]=newOrder[0];newOrder[0]=0;}

    if(newOrder[2]==0){newOrder[2]=newOrder[1];newOrder[1]=newOrder[0];newOrder[2]=0;}

                        Polygon tri = new Polygon(this,
                                                  this.corner[newOrder[0]],
                                                  this.corner[newOrder[1]],
                                                  this.corner[newOrder[2]]);

                        boolean isReallyCut = tri.cut(cutting);

                        this.topology=2;
                        this.corner = new Coordinate[3];
                        for(int i=0;i<3;i++){this.corner[i]=new Coordinate(tri.corner[i]);}
                    }
                else{if(dist[order[1]]<=0)
                    {
                        isCut = true;

                        this.corner[order[0]] = newCorns[0];
                        this.corner[order[1]] = newCorns[1];

                        Polygon contr = new Polygon(this); boolean ok = contr.order(false);
                            //check with ordering routine
                        if(!this.corner[0].equal(contr.corner[0]) ||
                           !this.corner[1].equal(contr.corner[1]) ||
                           !this.corner[2].equal(contr.corner[2]) ||
                           !this.corner[3].equal(contr.corner[3]))
                            {
                            // if ordered poly is different -> reorder
                                this.corner[order[0]] = newCorns[1];
                                this.corner[order[1]] = newCorns[0];
                            }
                    }
                else{if(dist[order[0]]<=0)
                    {
                        isCut = true;
                        Coordinate toProj = new Coordinate(this.corner[order[0]]);
                                   toProj.add(this);
                        this.corner[order[0]].project(pla,toProj);
                        this.corner[order[0]].add(this,-1.0);
                    }
                else{isCut = false;}}}}
```

```
                }
            }
        else
            {
                isCut = false;
            }


        return isCut;
    }



//*********************************************Sides*********************************************
**********

    /**
       Sides  of the polygon, effectively the difference of the subceeding corners.
       @return sides of the polygon as pure vectors without origin.
     */
    public Coordinate[] sides()
    {
        int N = corner.length;

        Coordinate res[] = new Coordinate[N];

        for(int i=0;i<N;i++)
            {
                res[i] = new Coordinate(this.corner[i],this.corner[(i+1)%N]);
            }

        return res;
    }


//*********************************************split*********************************************
**********

    /**
       Splits any polygons into triangles.
       @return triangle fragments of a polygon
     */
        public Polygon[] split()
    {
        Polygon res[];

        int N = this.corner.length;

        if(N==3)
            {
                res = new Polygon[1];
                res[0] = new Polygon(this);
            }
        else{if(N==4)
            {
                res = new Polygon[2];

                if(rt_lf)
                    {
                        res[0] = new Polygon(false,this,this.corner[0],this.corner[1],this.corner[2]);
                        res[1] = new Polygon(false,this,this.corner[0],this.corner[2],this.corner[3]);
                    }
                else
                    {
                        res[0] = new Polygon(false,this,this.corner[1],this.corner[2],this.corner[3]);
                        res[1] = new Polygon(false,this,this.corner[1],this.corner[3],this.corner[0]);
                    }
            }
        else{res = new Polygon[0];
        System.out.println(" Error - Wrong topology in polygon, number of corners impossible!");}}

        return res;
    }

    // ********************************************* Write ************************************

    /**
```

```
   Verifies wether two polygons are equal.
 */
public boolean equal(Polygon toCompare)
{
    boolean result = false;

    if(this.topology == toCompare.topology)
        {
            int cornNum = 0;
            if(this.topology==2){cornNum=3;}else{if(this.topology==4){cornNum=4;}else{cornNum=3;}}

            Coordinate A[] = new Coordinate[cornNum];
            Coordinate B[] = new Coordinate[cornNum];

            for(int i=0; i<cornNum; i++)
                {A[i] = new Coordinate(this); A[i].add(this.corner[i]);
                B[i] = new Coordinate(toCompare); B[i].add(toCompare.corner[i]);}

            {int i[] = new int[cornNum];
            found:
            if(cornNum==3)
                {
                    for(int j=0; j<3; j++)
                        {i[0]=j;
                        for(int k=0; k<2; k++)
                            {i[1] = (i[0]+k+1)%3;
                            i[2] = (i[0]+1)%3;
                            if(i[2] == i[1]){i[2] = (i[1]+1)%3;}
                            //System.out.println(" Indizes: "+i[0]+" "+i[1]+" "+i[2]);
                            if(A[0].equal(B[i[0]]) &&
                               A[1].equal(B[i[1]]) &&
                               A[2].equal(B[i[2]]))
                                {result = true;
                                //System.out.println(" true at: "+i[0]+" "+i[1]+" "+i[2]);
                                break found;}
                            }
                        }
                }
            else
                {
                    for(int j=0; j<4; j++)
                        {i[0]=j;                                  // i[0]
                        for(int k=0; k<3; k++)
                            {i[1] = (i[0]+k+1)%4;                // i[1]
                            for(int l=0; l<2; l++)
                                {if(l==0)
                            {if((i[0]+1)%4 != i[1]){i[2]=(i[0]+1)%4;}else{i[2]=(i[0]+3)%4;}}
                                else
                            {if((i[0]+2)%4 != i[1]){i[2]=(i[0]+2)%4;}else{i[2]=(i[0]+3)%4;}}
                            // i[2]
                                i[3]=(i[0]+1)%4;
                                if(i[3] == i[2] || i[3] == i[1]){i[3]=(i[3]+1)%4;}
                                if(i[3] == i[2] || i[3] == i[1]){i[3]=(i[3]+1)%4;}
                            // i[3]
                        //System.out.println(" Indizes: "+i[0]+" "+i[1]+" "+i[2]+" "+i[3]);
                                if(A[0].equal(B[i[0]]) &&
                                   A[1].equal(B[i[1]]) &&
                                   A[2].equal(B[i[2]]) &&
                                   A[3].equal(B[i[3]]))
                                    {result = true;
                                    break found;}
                                }
                            }
                        }
                }
            }
        }
    else  // quadrangle to cut
        {
            result = false;
        }
    return result;
}

// ***************************************** Write ******************************
```

```
    /**
       Writes a Polygon to the Screen,
     */
    public void WriteScr()
    {
        int j;
        System.out.println();
        if(topology == 4)
            {
                j=4;
                System.out.print(" Polygon: Quadrangle type with the
                                    surface splitting orientation being: ");
                if(rt_lf)
                    {
                        System.out.println("Right-Lower and Left-Upper triangle");
                    }
                else
                    {
                        System.out.println("Left-Lower and Right-Upper triangle");
                    }
            }
        else
            {
                j=3;
                System.out.println(" Polygon: Triangle type");
                 topology = 2;
            }
        System.out.print("   Origin: |");
        System.out.print(this.x[0]+" "+this.x[1]+" "+this.x[2]+"| + Corners: ");
        for(int i=0;i<j;i++)
                {System.out.print("|"+corner[i].x[0]+" "+corner[i].x[1]+" "+corner[i].x[2]+"| ");}
        System.out.println();
    }

}
```

C.2.7 The Polyhedron

```
/*
 * Copyright 1996-2000 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.math.*;

// ********************************************************************************************
// *********************************   Class Polyhedron ***************************************
// ********************************************************************************************

/**
   Describes a cell in space, generally it is a sheared cubus or wedge, if type is true, then cubus (4
side prism), otherwise, if type is false, then it is a wedge (3 side prism). Origin is the base of the
three sides, describing the cell. dir[0] describes the elevation of the prism, and dir[1,2] the two
sides.
   @author Manuel Alexandre Pouchon - &copy JNC Tokai Works (JAPAN)
   @version First created 2001.01.18 - version 0
 */
public class Polyhedron extends Coordinate
{

    // ****************************** Varables of the Class ************************************

    /**
       Corners of the polyhedron relative to the Origin (number of corners depends from Polyhedron
type defined in "type")
    */
    public Coordinate corner[];
    /**
       Type of polyhedorn, 6 types are defined as follows (compartible to FEMAP neutral file):
&bull<b>Tetraeder</b> - with 4 corners -> <b><em>6</em></b> - with 10 corners -> 10 ,
&bull<b>Wedge</b> - with 6 corners -> <b><em>7</em></b> - with 15 corners -> 11 , &bull<b>Brick</b>
with 8 corners -> <b><em>8</em></b> - with 20 corners -> 12 &hellip   In the same layer (lower or
```

```
upper), define the corners counterclockwise, start with the same corner in the upper layer, as you did
in the lower layer (only for Wedge and Brick)
    */
    public int topology;

    // ****************************** Constructors of the Class *********************************

    /**
       Empty constructor
     */
    public Polyhedron()
    {
    }

    /**
       Constructor for type 6 (Tetraeder) Polyhedron with 4 corners
       @param ori Origin of the Tetraeder
       @param c1 lower-left-front - corner of Tetraeder
       @param c2 lower-right-front - corner of Teraeder
       @param c3 lower-back - corner of Tetraeder
       @param c4 upper - corner of Tetraeder
       @return no return, only constructs tetraeder
     */
    public Polyhedron(Coordinate ori, Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4)
    {
        super(ori);
        topology = 6;
        corner = new Coordinate[4];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        corner[3] = new Coordinate(c4);
    }

    /**
       Constructor for type 8 (Wedge) Polyhedron with 6 corners
       @param ori Origin of the Wedge
       @param c1 lower-left-front - corner of Wedge
       @param c2 lower-right-front - corner of Wedge
       @param c3 lower-back - corner of Wedge
       @param c4 upper-left-front - corner of Wedge
       @param c5 upper-right-front - corner of Wedge
       @param c6 upper-back - corner of Wedge
       @return no return, only constructs Wedge
     */
    public Polyhedron(Coordinate ori,
                      Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4, Coordinate c5,
                      Coordinate c6)
    {
        super(ori);
        topology = 7;
        corner = new Coordinate[6];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        corner[3] = new Coordinate(c4);
        corner[4] = new Coordinate(c5);
        corner[5] = new Coordinate(c6);
    }

    /**
       Constructor for type 8 (Brick) Polyhedron with 8 corners
       @param ori Origin of the Brick
       @param c1 lower-left-front - corner of Brick
       @param c2 lower-right-front - corner of Brick
       @param c3 lower-right-back  - corner of Brick
       @param c4 lower-left-back  - corner of Brick
       @param c5 upper-left-front - corner of Brick
       @param c6 upper-right-front - corner of Brick
       @param c7 upper-right-back  - corner of Brick
       @param c8 upper-left-back  - corner of Brick
       @return no return, only constructs Brick
     */
    public Polyhedron(Coordinate ori,
                      Coordinate c1, Coordinate c2, Coordinate c3, Coordinate c4,
                      Coordinate c5, Coordinate c6, Coordinate c7, Coordinate c8)
    {
```

```
        super(ori);
        topology = 8;
        corner = new Coordinate[8];
        corner[0] = new Coordinate(c1);
        corner[1] = new Coordinate(c2);
        corner[2] = new Coordinate(c3);
        corner[3] = new Coordinate(c4);
        corner[4] = new Coordinate(c5);
        corner[5] = new Coordinate(c6);
        corner[6] = new Coordinate(c7);
        corner[7] = new Coordinate(c8);
    }

    public Polyhedron(Coordinate ori, Coordinate corners[])
    {
        super(ori);

        int N = corners.length;

        if(N==4){topology=6;}else{if(N==6){topology=7;}else{if(N==8){topology=8;}else{topology=-1;}}}

        //System.out.println("\n\t The topology is "+topology);

        corner = new Coordinate[N];
        for(int i=0;i<N;i++){corner[i]=new Coordinate(corners[i]);}

        if(topology==-1)
        {System.out.println("\n\n\t ** ERROR ** Non valid number of
                            arguments in Polyhedr. initialis.");}
    }

    /**
       Constructor which takes an instance of the Cell-class to construct the polyhedron. This is a
special simple case of a Wedge, or Brick, with steight surfaces and oposit sides being parallel (of
course in case of Wedge the sides are not parallel, only the bottom and ceiling)
       @param toTrsf Cell to be transformed into the polyhedron.
     */
    public Polyhedron(Cell toTrsf)
    {
        super(toTrsf);

        Coordinate C1 = new Coordinate();
        Coordinate C2 = new Coordinate(); C2.add(toTrsf.dir[1]);
        Coordinate C4 = new Coordinate(); C4.add(toTrsf.dir[2]);
        Coordinate C5 = new Coordinate(); C5.add(toTrsf.dir[0]);
        Coordinate C6 = new Coordinate(); C6.add(toTrsf.dir[0]); C6.add(toTrsf.dir[1]);
        Coordinate C8 = new Coordinate(); C8.add(toTrsf.dir[0]); C8.add(toTrsf.dir[2]);

        if(toTrsf.type == 1)
            {
                Coordinate C3 = new Coordinate(); C3.add(toTrsf.dir[1]); C3.add(toTrsf.dir[2]);
                Coordinate C7 = new Coordinate(); C7.add(toTrsf.dir[0]); C7.add(toTrsf.dir[1]);
                C7.add(toTrsf.dir[2]);

                topology = 8;
                corner = new Coordinate[8];
                corner[0] = new Coordinate(C1);
                corner[1] = new Coordinate(C2);
                corner[2] = new Coordinate(C3);
                corner[3] = new Coordinate(C4);
                corner[4] = new Coordinate(C5);
                corner[5] = new Coordinate(C6);
                corner[6] = new Coordinate(C7);
                corner[7] = new Coordinate(C8);
            }
        else
            {
                topology = 7;
                corner = new Coordinate[6];
                corner[0] = new Coordinate(C1);
                corner[1] = new Coordinate(C2);
                corner[2] = new Coordinate(C4);
                corner[3] = new Coordinate(C5);
                corner[4] = new Coordinate(C6);
                corner[5] = new Coordinate(C8);
            }
    }
```

```
    /**
       Copy constructor
       @param toCopy Polyhedron to be copied into the new polyhedron
     */
    public Polyhedron(Polyhedron toCopy)
    {
        super(toCopy);

        this.topology = toCopy.topology;

        int N = toCopy.corner.length;
        this.corner = new Coordinate[N];
        for(int i=0;i<N;i++){this.corner[i]=new Coordinate(toCopy.corner[i]);}
    }

    //  ********************************** order ******************************************

    /**
       Takes a polygon and verifies whether the order of corners is valid, if not so,
        it corrects the order.
       @param out Set to true if the routine should report corrections
       @return If the polygon was ok, or if sucessful corrections could be applied,
        the routine returns true, if the quadrangle is irreparable screwed,
        the routine returns false.
     */
    public boolean order(boolean out)
    {
        boolean ok;

        Permutation toChange = new Permutation(this);
        //System.out.println(" Trivial ordering List: "); toChange.WriteScr();
        //System.out.println(" Polyhedron initial: "); this.WriteScr();
        ok = toChange.pointOrder(this, out);
        //System.out.println(" Polyhedron not ordered: "); this.WriteScr();
        this.permute(toChange);
        //System.out.println(" Ordering List: "); toChange.WriteScr();
        //System.out.println(" Polyhedron ordered: "); this.WriteScr();
        return ok;
    }


    /**
       Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls
the order of corners and also corrects if necessary. Error messages for degenerated polyhedrons will
also be printed.
       @param out Set to <em>true</em> if correction information should be printend to screen,
<em>false</em> otherwise.
     */
    public void orderOld(boolean out)
    {

        int numCorners = this.corner.length;

        if(numCorners == 4)
            {
                if(this.topology != 6)
                    {if(out){System.out.println(" Error -- wrong polyhedron-topology detected
                                                    (for tetraeder): "
                                            +this.topology+" instead of 6! <- corrected");}
                    this.topology =6;}

                double vol =
                        Coordinate.spat(this.corner[0],this.corner[1],this.corner[2],this.corner[3]);

                if(vol<0)
                    {if(out){System.out.println(" Error -- wrong polyhedron orientation detected
                                                    (for tetraeder)"+
                                            "<- corrected");}
                    Coordinate per = new Coordinate(this.corner[2]);
                    this.corner[2].assign(this.corner[1]); this.corner[1].assign(per);}

                if(vol==0.0)
                    {System.out.println(" Error -- polyhedron (in this case tetraeder)
                                            is degenerated (volume = 0)");}
            }
```

```
if(numCorners == 6)
    {
       if(this.topology != 7)
           {if(out){System.out.println(" Error -- wrong polyhedron-topology detected
                                          (for wedge): "
                                   +this.topology+" instead of 7! <- corrected");}
           this.topology =7;}

       double uab1 =
              Coordinate.spat(this.corner[0],this.corner[1],this.corner[2],this.corner[3]),
           uab2 =
              Coordinate.spat(this.corner[0],this.corner[1],this.corner[2],this.corner[4]),
           uab3 =
              Coordinate.spat(this.corner[0],this.corner[1],this.corner[2],this.corner[5]),
           oab1 =
              Coordinate.spat(this.corner[3],this.corner[5],this.corner[4],this.corner[0]),
           oab2 =
              Coordinate.spat(this.corner[3],this.corner[5],this.corner[4],this.corner[1]),
           oab3 =
              Coordinate.spat(this.corner[3],this.corner[5],this.corner[4],this.corner[2]);

       if(uab1==0.0 || uab2==0.0 || uab3==0.0 || oab1==0.0 || oab2==0.0 || oab3==0.0)
           {
              System.out.println(" Error -- polyhedron
                                    (in this case wedge) is degenerated");
           }
       else
           {
              {if(uab1<0 && uab2<0 && uab3<0)
                  {
                     if(out){System.out.println(" Error -- wrong polyhedron
                                                   orientation detected"+
                                              " (first three corners for wedge)
                                                   <- corrected");}
                     Coordinate per = new Coordinate(this.corner[2]);
                     this.corner[2].assign(this.corner[1]); this.corner[1].assign(per);
                  }
              else
                  {
                     if(uab1>0 && uab2>0 && uab3>0)
                         {
                            // is ok!
                         }
                     else
                         {
                            System.out.println(" Error -- polyhedron is screwed!!
                                                   no correction possible!");
                         }
                  }
              }
              {if(oab1<0 && oab2<0 && oab3<0)
                  {
                     if(out){System.out.println(" Error -- wrong polyhedron
                                                   orientation detected"+
                                              " (last three corners for wedge)
                                                   <- corrected");}
                     Coordinate per = new Coordinate(this.corner[2]);
                     this.corner[2].assign(this.corner[1]); this.corner[1].assign(per);
                  }
              else
                  {
                     if(oab1>0 && oab2>0 && oab3>0)
                         {
                            // is ok!
                         }
                     else
                         {
                            System.out.println(" Error -- polyhedron is screwed!!
                                                   no correction possible!");
                         }
                  }
              }
           }

    }
if(numCorners == 8)
    {
```

```
            if(this.topology != 8)
                {if(out){System.out.println(" Error -- wrong polyhedron-topology detected
                                                    (for brick): "
                                        +this.topology+" instead of 8! <- corrected");}
                this.topology =8;}

            Polygon base = new Polygon(this, this.corner[0], this.corner[1], this.corner[2],
                                                    this.corner[3]),
                top = new Polygon(this, this.corner[4], this.corner[7], this.corner[6],
                                         this.corner[5]);

            for(int i=0;i<4;i++){this.corner[i] = base.corner[i];
                                 this.corner[i+4] = top.corner[i];}
            //  <---- Correction of possible screwed base and top definitions

            boolean btt = base.above(top,true);
            boolean btf = base.above(top,false);
            boolean tbt = base.above(top,true);

            if(!btt && !btf)
                {System.out.println(" ERROR - Brick is screwed, no correction possible!");}
            else
                {
                   if(!btt)     // permutation of corner 1 & 3
                       {
                           Coordinate perm = new Coordinate(this.corner[1]);
                           this.corner[1].assign(this.corner[3]);
                           this.corner[3].assign(perm);
                       }
                   if(tbt)     // permutation of corner 5 & 7
                       {
                           Coordinate perm = new Coordinate(this.corner[5]);
                           this.corner[5].assign(this.corner[7]);
                           this.corner[7].assign(perm);
                       }
                }

        }
    }


    //********************************************** permute ******************************

    /**
       Permutates any Polyhedron-corner entities with the given permutation instance, toApply.
       @param toApply Permutation instance to be applied to the Polyhedron-corner entities.
       @return True if permutation was succesful. False if the length of <em>toApply</em> does not
correspond to the number of corners.
     */
    public boolean permute(Permutation toApply)
    {
        boolean ok = true;

       int N = this.corner.length;

       if(N == toApply.order.length)
           {
               int transNum = 0;
               for(int i=0; i<N; i++)
                   {if(toApply.order[i]!=i){transNum++;}}
               if(transNum>0)
                   {
                       int perms[] = new int[transNum];
                       Coordinate permsCont[] = new Coordinate[transNum];
                       {
                           int M=0;
                           for(int i=0; i<N; i++)
                               {if(toApply.order[i]!=i)
                                   {perms[M]=i;permsCont[M]=
                                       new Coordinate(this.corner[toApply.order[i]]);M++;}}
                       }
                       for(int i=0;i<transNum;i++)
                           {this.corner[perms[i]].assign(permsCont[i]);}
                   }
               ok = true;
           }
       else{ok = false;}
```

```
    return ok;
}


// ******************************** In Polyhedron ****************************************

/**
   Verifies wether coordinate lies within the polyhedron, returns boolean
   @param isIn Coordinate to be verified
   @return true when <b>isIn</b> is contained in the polyhedron, false otherwise
 */
public boolean within(Coordinate isIn)
{
    MultiPolygon faces = new MultiPolygon();
    faces = this.polygons();

    int N = faces.polygons.length;

    boolean in = true;

    for(int i=0; i<N; i++)
        {
           if(!faces.polygons[i].above(isIn,Coordinate.precision))
               {in = false;}
        }
        return in;
}

/**
   Verifies wether polygon lies within the polyhedron, returns boolean
   @param isIn Polygon to be verified
   @return true when <b>isIn</b> is contained in the polyhedron, false otherwise
 */
public boolean within(Polygon isIn)
{
    int M = isIn.corner.length;
    Coordinate allCorners[] = new Coordinate[M];
    for(int i=0;i<M;i++){allCorners[i] = new Coordinate(isIn);
                         allCorners[i].add(isIn.corner[i]);}

    MultiPolygon faces = new MultiPolygon();
    faces = this.polygons();

    int N = faces.polygons.length;

    boolean in = true;

    for(int j=0; j<M; j++)
        {
           for(int i=0; i<N; i++)
               {
                  if(!faces.polygons[i].above(allCorners[j]))
                      {in = false;}
               }
        }
    return in;
}

/**
   Verifies wether polyhedron lies within the polyhedron, returns boolean
   @param isIn Polyhedron to be verified
   @return true when <b>isIn</b> is contained in the polyhedron, false otherwise
 */
public boolean within(Polyhedron isIn)
{
    int M = isIn.corner.length;
    Coordinate allCorners[] = new Coordinate[M];
    for(int i=0;i<M;i++){allCorners[i] = new Coordinate(isIn);
                                 allCorners[i].add(isIn.corner[i]);}

    MultiPolygon faces = new MultiPolygon();
    faces = this.polygons();

    int N = faces.polygons.length;

    boolean in = true;
```

```
        for(int j=0; j<M; j++)
            {
                for(int i=0; i<N; i++)
                    {
                        if(!faces.polygons[i].above(allCorners[j]))
                            {in = false;}
                    }
            }
        return in;
    }

    /**
       Verifies wether corners of polyhedron lie outside the polyhedron, returns boolean
       @param cornAreOut Polyhedron whose corners are to be verified
       @return true when all corners of <b>cornsAreOut</b> are outside the polyhedron,
        false otherwise
     */
    public boolean cornerOutside(Polyhedron cornAreOut)
    {
        int M = cornAreOut.corner.length;
        Coordinate allCorners[] = new Coordinate[M];
        for(int i=0;i<M;i++){allCorners[i] = new Coordinate(cornAreOut);
                                                    allCorners[i].add(cornAreOut.corner[i]);}

        MultiPolygon faces = new MultiPolygon();
        faces = this.polygons();

        int N = faces.polygons.length;

        boolean out = true;

        for(int j=0; j<M; j++)
            {
                boolean in = true;
                for(int i=0; i<N; i++)
                    {
                        if(!faces.polygons[i].above(allCorners[j]))
                            {in = false;}
                    }
                if(in){out = false;break;}
            }
        return out;
    }


    /**
       Verifies wether corners of polygon lie outside the polyhedron, returns boolean
       @param cornAreOut Polygon whose corners are to be verified
       @return true when all corners of <b>cornsAreOut</b> are outside the polyhedron, false otherwise
     */
    public boolean cornerOutside(Polygon cornAreOut)
    {
        int M = cornAreOut.corner.length;
        Coordinate allCorners[] = new Coordinate[M];
        for(int i=0;i<M;i++){allCorners[i] = new Coordinate(cornAreOut);
allCorners[i].add(cornAreOut.corner[i]);}

        MultiPolygon faces = new MultiPolygon();
        faces = this.polygons();

        int N = faces.polygons.length;

        boolean out = true;

        for(int j=0; j<M; j++)
            {
                boolean in = true;
                for(int i=0; i<N; i++)
                    {
                        if(!faces.polygons[i].above(allCorners[j]))
                            {in = false;}
                    }
                if(in){out = false;break;}
            }
        return out;
    }
```

```
//  ********************************** disjunkt ********************************

/**
   Tests wether two polyhedrons are disjunkt or not.
   @param isDisj second polyhedron to be tested against the first one (the object-instance).
   @return true if the two polyhedrons are disjunct, false otherwise.
 */
public boolean disjunct(Polyhedron isDisj)
{
   boolean disj = true;
   if(!this.cornerOutside(isDisj) || !isDisj.cornerOutside(this)){disj = false;}

   if(disj)
       {
          //System.out.println(" Element has to bee investigated, none of the elements is
                                      within the other");

          MultiPolygon p1Surf = new MultiPolygon(); p1Surf = this.polygons();
          MultiPolygon p2Surf = new MultiPolygon(); p2Surf = isDisj.polygons();

          Line p1Lines[] = this.sides();
          Line p2Lines[] = isDisj.sides();

          int M = p1Surf.polygons.length;
          int MM = p1Lines.length;
          int N = p2Surf.polygons.length;
          int NN = p2Lines.length;

          Line test = new Line();


          for(int i=0;i<M;i++)
              {
                  for(int j=0;j<NN;j++)
                      {
                          Coordinate cutPt = new Coordinate();
                          if(cutPt.cut(p1Surf.polygons[i],p2Lines[j],true)){disj = false;}
                      }
              }

          for(int i=0;i<N;i++)
              {
                  for(int j=0;j<MM;j++)
                      {
                          Coordinate cutPt = new Coordinate();
                          if(cutPt.cut(p2Surf.polygons[i],p1Lines[j],true)){disj = false;}
                      }
              }

       }
   return disj;
}

/**
   Tests wether two polyhedrons are disjunkt or not.
   @param isDisj second polyhedron to be tested against the first one (the object-instance).
   @return true if the two polyhedrons are disjunct, false otherwise.
 */
public boolean disjunctOld(Polyhedron isDisj)
{
   boolean disj = true;
   if(!this.cornerOutside(isDisj) || !isDisj.cornerOutside(this)){disj = false;}

   if(disj)
       {
          //System.out.println(" Element has to bee investigated,
                                      none of the elements is within the other");

          MultiPolygon p1Surf = new MultiPolygon(); p1Surf = this.polygons();
          MultiPolygon p2Surf = new MultiPolygon(); p2Surf = isDisj.polygons();

          int M = p1Surf.polygons.length;
          int N = p2Surf.polygons.length;

          Line test = new Line();
```

```
            boolean cutting = false;
            {
                boolean cuttingLoc = false;
                for(int i=0;i<M;i++)
                    {
                        for(int j=0;j<N;j++)
                            {
                                cuttingLoc = test.cut(p1Surf.polygons[i],p2Surf.polygons[j]);
                                if(cuttingLoc){cutting = true;}
                            }
                    }
            }
            if(cutting)
                {disj = false;
                //System.out.println("\n\n\t special case, only sides corssing of polyhedrons");
                }
        }
    return disj;
}


/**
   Tests wether two polygon is disjunkt is disjunct with calling polyhedron.
   @param isDisj polygon to be tested against the polyhedron (the object-instance).
   @return true if the polygon is disjunct, false otherwise.
 */
public boolean disjunct(Polygon isDisj)
{
    boolean disj = true;
    if(!this.cornerOutside(isDisj)){disj = false;}

    if(disj)
        {
            //System.out.println(" Element has to be investigated,
                                 none of the elements is within the other");

            Line side[] = this.sides();
            int N = side.length;

            //System.out.println("    sides in Polyhedron "+N);

            for(int i=0;i<N;i++)
                {
                    Coordinate schnPkt = new Coordinate();
                    if(schnPkt.cut(isDisj,side[i],true)){disj = false;}
                }
        }

    return disj;
    }

/**
   Tests wether two polygon is disjunkt is disjunct with calling polyhedron.
   @param isDisj polygon to be tested against the polyhedron (the object-instance).
   @return true if the polygon is disjunct, false otherwise.
 */
public boolean disjunctOld(Polygon isDisj)
{
    boolean disj = true;
    if(!this.cornerOutside(isDisj)){disj = false;}

    if(disj)
        {
            //System.out.println(" Element has to bee investigated,
                                          none of the elements is within the other");

            MultiPolygon pSurf = new MultiPolygon(); pSurf = this.polygons();

            int N = pSurf.polygons.length;

            Line test = new Line();

            boolean cutting = false;
            {
                boolean cuttingLoc = false;
                for(int i=0;i<N;i++)
```

```
                    {
                        cuttingLoc = test.cut(pSurf.polygons[i],isDisj);
                        if(cuttingLoc){cutting = true;}
                    }
                }
                if(cutting)
                    {disj = false;
                    //System.out.println("\n\n\t special case, only sides corssing of polyhedrons");
                    }
            }
        return disj;
        }

    //  ********************************** polygons **********************

    /**
       Returns the <em>polygonal faces</em> of a polyhedron, with the faces being oriented towards the
cell (cross product of side-vectors).
       @return multiple polygon containing all faces of the polyhedron.
    */
    public MultiPolygon polygons()
    {
        int numFaces = 0;

        switch(this.topology) {
        case 8 : numFaces = 6 ; break; // Brick
        case 7 : numFaces = 5 ; break; // Wedge
        case 6 : numFaces = 4 ; break; // Tetraeder
        default: numFaces = 4;
        }

        Polygon face[] = new Polygon[numFaces];

        if(this.topology == 8)
            {
                face[0] =
        new Polygon(this,this.corner[0],this.corner[1],this.corner[2],this.corner[3]); // base
                face[1] =
        new Polygon(this,this.corner[0],this.corner[4],this.corner[5],this.corner[1]);
                face[2] =
        new Polygon(this,this.corner[1],this.corner[5],this.corner[6],this.corner[2]);
                face[3] =
        new Polygon(this,this.corner[2],this.corner[6],this.corner[7],this.corner[3]);
                face[4] =
        new Polygon(this,this.corner[3],this.corner[7],this.corner[4],this.corner[0]);
                face[5] =
        new Polygon(this,this.corner[4],this.corner[7],this.corner[6],this.corner[5]);  // cover
            }

        if(this.topology == 7)
            {
                face[0] = new Polygon(this,this.corner[0],this.corner[1],this.corner[2]); // base
                face[1] =
        new Polygon(this,this.corner[0],this.corner[3],this.corner[4],this.corner[1]);
                face[2] =
        new Polygon(this,this.corner[1],this.corner[4],this.corner[5],this.corner[2]);
                face[3] =
        new Polygon(this,this.corner[2],this.corner[5],this.corner[3],this.corner[0]);
                face[4] = new Polygon(this,this.corner[3],this.corner[5],this.corner[4]); // cover
            }

        if(this.topology == 6)
            {
                face[0] = new Polygon(this,this.corner[0],this.corner[1],this.corner[2]);
                face[1] = new Polygon(this,this.corner[0],this.corner[3],this.corner[1]);
                face[2] = new Polygon(this,this.corner[1],this.corner[3],this.corner[2]);
                face[3] = new Polygon(this,this.corner[2],this.corner[3],this.corner[0]);
            }

        MultiPolygon faceData = new MultiPolygon(face);

        return faceData;

    }

    //************************** DEGENERATED ****************************************
```

```
    /**
       Is the polyhedron degenerated? Only implemented for tetraeder now!
       @return true if tetraeder is degenerated
     */
    public boolean degenerated()
    {
        boolean res = false;

        if(this.topology==6)
            {
                int N = this.corner.length;
                for(int i=0;i<N-1;i++)
                    {
                        for(int j=i+1;j<N;j++)
                            {
                                if(this.corner[i].equal(this.corner[j]))
                                    {res = true;}
                            }
                    }
            }
        return res;
    }

    //************************** CUT ********************************************

   /**
       Cutting a polyhdron with a plane.
       @param pla plane cutting the polyhedon
       @return list of polyhedrons representing the cutten volume
     */
    public MultiPolyhedron cut(Plane pla)
    {
        boolean isCut = false;
        MultiPolyhedron res;
        int N = 0;
        Polyhedron newPh[];

        if(this.topology==6)    // tetraeder
            {
                Coordinate relCorns[] = new Coordinate[4];
                       // triangle-corners relativ to plane-origin
                for(int i=0;i<4;i++)
                    {relCorns[i]=
                new Coordinate(this);relCorns[i].add(this.corner[i]);relCorns[i].add(pla,-1.0);}

                double dist[] = new double[4];    // distance of triangle-corners to plane
                for(int i=0;i<4;i++){dist[i]=Coordinate.dot(pla.normal,relCorns[i]);}

                int order[] = new int[4];
                // distances in an incresing order (signum important -> not abs distance!)
                for(int i=0;i<4;i++)  // finding smallest
                    {if(dist[i]<=dist[(i+1)%4] &&
                        dist[i]<=dist[(i+2)%4] &&
                        dist[i]<=dist[(i+3)%4]){order[0]=i;}}
                for(int i=1;i<4;i++)  // finding second smallest
                    {if(dist[(order[0]+i)%4]<=dist[(order[0]+1+((i)%3))%4] &&
                        dist[(order[0]+i)%4]<=dist[(order[0]+1+((i+1)%3))%4])
                        {order[1]=(order[0]+i)%4;}}
                for(int i=0;i<4;i++)  // finding biggest
                    {if(dist[i]>=dist[(i+1)%3] &&
                        dist[i]>=dist[(i+2)%3] &&
                        dist[i]>=dist[(i+3)%3] &&
                        i != order[0] &&
                        i != order[1]){order[3]=i;}}
                for(int i=0;i<4;i++)  // find remaining
                    {if(i != order[0] && i != order[1] && i != order[3])
                        {order[2]=i;}}

                //System.out.print("\n\t Order: ");
                //for(int i=0;i<4;i++)
                        {System.out.print("  "+i+" / "+order[i]+": "+dist[order[i]]+"  | ");}
                //System.out.print("\n");

                if(dist[order[3]]<=0.0)
                        // biggest rel. distance under plane -> no interaction and empty result
                        {
                            //System.out.println("  Case 0 - all outside");
```

```
            isCut = false;
            newPh = new Polyhedron[0];
            //newPh[0] = new Polyhedron(this);
        }
    else{if(dist[order[2]]<=0.0)
        // one point (order[3]) is above plane -> new polyhedron is a tetraeder
        {
            //System.out.println("   Case 1 - three outside");
            isCut = true;

            Coordinate cutPoint[] = new Coordinate[4];
            cutPoint[order[3]]=this.corner[order[3]]; // iside-point doesen't change
            {Coordinate anf = new Coordinate(this); anf.add(this.corner[order[3]]);
            for(int i=0;i<3;i++)
                {
                    Coordinate end = new Coordinate(this);
                        end.add(this.corner[(order[3]+1+i)%4]);
                    Line cutSide=new Line(anf,end,true);
                    cutPoint[(order[3]+1+i)%4]=new Coordinate();
                    boolean lineCuts = cutPoint[(order[3]+1+i)%4].cut(pla,cutSide);
                    cutPoint[(order[3]+1+i)%4].add(this,-1.0);
                }
            }
            newPh = new Polyhedron[1];
            newPh[0] =
                new Polyhedron(this,cutPoint[0],cutPoint[1],cutPoint[2],cutPoint[3]);
        }
    else{if(dist[order[1]]<0.0)
        // two points (order[3] & order[2]) are above plane -> wedge as result
        {
            //System.out.println("   Case 2 - two outside");
            isCut = true;

            Coordinate cutPoint[] = new Coordinate[6];
            Coordinate dir;
            int oldNewInd[] = new int[2];

            // find out first entitiy and set it to same place in new order,
            // second remaining entitiy is set 3 places behind first.
            if(order[2]<order[3]) // -> order[2] <= 2 (in 0 1 2)
                {//System.out.println("case1");
                oldNewInd[0]=order[2]; oldNewInd[1]=(oldNewInd[0]+3);
                // +3 in second plane
                cutPoint[oldNewInd[0]]=new Coordinate(this.corner[order[2]]);
                cutPoint[oldNewInd[1]]=new Coordinate(this.corner[order[3]]);
                dir = new Coordinate(this.corner[order[2]],this.corner[order[3]]);}
            else                  // -> order[3] <= 2 (in 0 1 2)
                {//System.out.println("case2");
                oldNewInd[0]=order[3]; oldNewInd[1]=(oldNewInd[0]+3);
                // +3 in second plane
                cutPoint[oldNewInd[0]]=new Coordinate(this.corner[order[3]]);
                cutPoint[oldNewInd[1]]=new Coordinate(this.corner[order[2]]);
                dir = new Coordinate(this.corner[order[3]],this.corner[order[2]]);}

            for(int i=0;i<2;i++)
                      // counter over the starting points scecified with oldNewInd
                {
                      // i = 0 for lover triangle, i = 2 for upper triangle

                    Coordinate provCutPts[] = new Coordinate[2];
                    Coordinate anf=new Coordinate(this);
                        anf.add(cutPoint[oldNewInd[i]]);
                    //System.out.println();
                    for(int j=0;j<2;j++)
                        {
                            Coordinate end=new Coordinate(this);
                                end.add(this.corner[order[0+j]]);
                            Line cutSide=new Line(anf,end,true);
                            //System.out.print("\n\n");cutSide.WriteScr();
                            //this.WriteScr();
                            //System.out.print(" Start and End: ");
                            // anf.WriteScr();end.WriteScr();
                            //System.out.println();
                            provCutPts[j]=new Coordinate();
                            if(anf.equal(end))
                                {
                                    provCutPts[j].assign(anf);
```

```
                                // poits lay on cutting plane
                                }
                            else
                                {
                                    boolean lineCuts = provCutPts[j].cut(pla,cutSide);
                                }
                            provCutPts[j].add(this,-1.0);
                            //pla.WriteScr();

        //System.out.println("provCut");provCutPts[j].WriteScr();System.out.println();
                        }
                    //System.out.println("\n "+i+"\n\n");
                    Coordinate newSide[] = new Coordinate[2];
                    for(int j=0;j<2;j++){newSide[j] =
                                            new Coordinate(cutPoint[oldNewInd[i]],
                                                           provCutPts[j]);}
                    Coordinate dir2 = new Coordinate();
                        dir2.cross(newSide[0],newSide[1]);
                    //dir2.WriteScr();
                    if(Coordinate.dot(dir,dir2)>=0)
                        {for(int j=0;j<2;j++)
                            {cutPoint[((1+j+oldNewInd[i]-(i*3))%3)+(i*3)] =
                                 provCutPts[j];}}
                    else
                        {for(int j=0;j<2;j++)
                            {cutPoint[((1+j+oldNewInd[i]-(i*3))%3)+(i*3)] =
                                 provCutPts[(j+1)%2];}}
                }
            newPh = new Polyhedron[1];
            //System.out.println(" Points for Wedge: ");
            //for(int q=0;q<6;q++){cutPoint[q].WriteScr();}
            newPh[0] =
                new Polyhedron(this,cutPoint[0],cutPoint[1],cutPoint[2],
                               cutPoint[3],cutPoint[4],cutPoint[5]);
        }
    else{if(dist[order[1]]==0.0 && dist[order[0]]<0.0)
                // special case, result are two tetraeder
        {
            //System.out.println("   Case 3 - two inside and one on plane");
            Line conn[] = new Line[2];
            Coordinate cutPts[] = new Coordinate[2];
            Coordinate start = new Coordinate(this); start.add(this.corner[order[0]]);
            Coordinate end[] = new Coordinate[2];
            for(int i=0;i<2;i++){end[i] = new Coordinate(this);
            end[i].add(this.corner[order[2+i]]);}
            for(int i=0;i<2;i++){conn[i] = new Line(start,end[i],true);}
            for(int i=0;i<2;i++)
                {cutPts[i]=
                new Coordinate();cutPts[i].cut(pla,conn[i]);cutPts[i].add(this,-1.0);}
            newPh = new Polyhedron[2];
            Polygon ground =
        new Polygon(true,this,this.corner[order[2]],cutPts[0],
                        cutPts[1],this.corner[order[3]],true);
            newPh[0] = new Polyhedron(this,
                               ground.corner[0],ground.corner[1],ground.corner[2],
                                     this.corner[order[1]]);
            newPh[0].order(false);
            newPh[1] = new Polyhedron(this,
                                    ground.corner[0],ground.corner[2],ground.corner[3],
                                      this.corner[order[1]]);
            newPh[1].order(false);
        }
    else{if(dist[order[0]]<0.0)
        // three points (order[3] & order[2] & order[1]) are above plane
        {                        //       ->  wedge as result
            //System.out.println("  Case 4 - one outside");
            isCut = true;

            int newLayer = 0;
                // which layer is going to contain to cut-points (0 or 1)

            Coordinate cutPoint[] = new Coordinate[6];
            if(order[0]==0)
                {newLayer = 0;       // the existing points are placed in second layer
                for(int i=0;i<3;i++)
                        // direction is ok, because top layer -> normal looks up!
                    {cutPoint[3+i] = new Coordinate(this.corner[i+1]);}
```

```
                        }
                    else
                        {newLayer = 1;   // the existing points are placed in first layer
                        int flo[];       //fist layer order
                             if(order[0]==1){flo = new int[]{0,2,3};}
                             else{if(order[0]==2){flo = new int[]{0,3,1};}
                             else{flo = new int[]{0,1,2};}}
                        {for(int i=0;i<3;i++)
                            {cutPoint[i] = new Coordinate(this.corner[flo[i]]);}}
                        }

                    Coordinate anf = new Coordinate(this); anf.add(this.corner[order[0]]);
                        // only point under layer
                    for(int i=0;i<3;i++)
                        {
                            Coordinate end=new Coordinate(this);
                            end.add(cutPoint[(((newLayer+1)%2)*3)+i]);
                            Line cutSide=new Line(anf,end,true);
                            cutPoint[(newLayer*3)+i] = new Coordinate();
                            boolean lineCuts = cutPoint[(newLayer*3)+i].cut(pla,cutSide);
                            cutPoint[(newLayer*3)+i].add(this,-1.0);
                        }
                    newPh = new Polyhedron[1];
                    newPh[0] = new Polyhedron(this,cutPoint[0],cutPoint[1],cutPoint[2],
                                    cutPoint[3],cutPoint[4],cutPoint[5]);
                }
            else            // all points are above surface -> polyhedron remains unchanged
                {//System.out.println("  Case 5 - all inside");
                isCut = false;
                newPh = new Polyhedron[1];
                newPh[0] = new Polyhedron(this);}}}}}
        }
    else
        {
            MultiPolyhedron split = new MultiPolyhedron();
            split.reduceTopo(this);

            int NN=split.polyhedrons.length;

            Vector all = new Vector();
            for(int i=0;i<NN;i++)
                {
                    MultiPolyhedron cutSplit = new MultiPolyhedron();
                    cutSplit = split.polyhedrons[i].cut(pla);
                    int NNN = cutSplit.polyhedrons.length;
                    for(int k=0;k<NNN;k++)
                        {all.add(cutSplit.polyhedrons[k]);}
                }
            int M = all.size();

            newPh = new Polyhedron[M];
            for(int i=0;i<M;i++)
                {
                    newPh[i] = new Polyhedron((Polyhedron)all.get(i));
                }

        }

    res = new MultiPolyhedron(newPh);
    return res;

}

/**
   Cutting a polyhdron with a cell.
   @param cell cell cutting the polyhedon
   @return list of polyhedrons representing the cutten volume
   @deprecated
 */
public MultiPolyhedron cutOld(Polyhedron cell)
{
    Polyhedron res[];

    if(this.within(cell)) // this is conained in outer cell -> no cutting
        {//System.out.println(" Case 0");
        res = new Polyhedron[1];
        res[0] = new Polyhedron(this);}
```

```
    else{if(cell.within(this)) // cell is contained in this -> cell is equal to cutten this
        {//System.out.println(" Case 1");
        res = new Polyhedron[1];
        res[0] = new Polyhedron(cell);}
    else{if(this.disjunct(cell))
        {//System.out.println(" Case 2");
        res = new Polyhedron[0];}
    else
        {
           //System.out.println(" Case 3");
           Vector all = new Vector();
           all.add(this);
           MultiPolygon faces = new MultiPolygon();
           faces = cell.polygons();
           int N = faces.polygons.length;
           for(int i=0;i<N;i++)
               {
                  int M = all.size();
                  //System.out.println("   "+i+" "+M);
                  Vector allNew = new Vector();
                  Plane pla = new Plane(faces.polygons[i]);
                  for(int j=0;j<M;j++)
                      {
                         Polyhedron loc = new Polyhedron((Polyhedron)all.get(j));
                         MultiPolyhedron singleCut = new MultiPolyhedron();
                         singleCut = loc.cut(pla);
                         int O = singleCut.polyhedrons.length;

                         //pla.WriteScr();singleCut.WriteScr();
                                System.out.println(" this is dividied into:");

                         MultiPolyhedron primitives[] = new MultiPolyhedron[O];
                         for(int k=0;k<O;k++)
                             {primitives[k] = new MultiPolyhedron();
                             primitives[k].reduceTopo(singleCut.polyhedrons[k]);
                             int numPrims = primitives[k].polyhedrons.length;
                             for(int prims=0; prims<numPrims; prims++)
                                 {
                                     primitives[k].polyhedrons[prims].order(false);
                                 }

                             //for(int l=0;l<(primitives[k].polyhedrons.length);l++)
                             //{primitives[k].polyhedrons[l].WriteScr();}
                             }


                         for(int k=0;k<O;k++)
                             {
                                 int P = primitives[k].polyhedrons.length;
                                 for(int l=0;l<P;l++)
                                     {
                                         if(!primitives[k].polyhedrons[l].degenerated())
                                             {allNew.add(primitives[k].polyhedrons[l]);}
                                         else{/*primitives[k].polyhedrons[l].WriteScr();*/}
                                     }
                             }
                      }
                  all.setSize(0);
                  int O = allNew.size();
                  for(int j=0;j<O;j++)
                      {
                         all.add((Polyhedron)allNew.get(j));
                      }
                  allNew.setSize(0);
               }
           int M = all.size();

           res = new Polyhedron[M];
           for(int i=0;i<M;i++){res[i] = new Polyhedron((Polyhedron)all.get(i));}
        }}}

    MultiPolyhedron resu = new MultiPolyhedron(res);
    return resu;


}
```

```
/**
   Cutting a polyhdron with a cell.
   @param cell cell cutting the polyhedon
   @return list of polyhedrons representing the cutten volume
 */
public MultiPolyhedron cut(Polyhedron cell)
{
    Polyhedron res[];

    if(cell.within(this)) // this is conained in outer cell -> no cutting
        {//System.out.println(" Case - polyhedron completely contained in the limiting-cell");
        res = new Polyhedron[1];
        res[0] = new Polyhedron(this);}
    else{if(this.within(cell)) // cell is contained in this -> cell is equal to cutten this
        {//System.out.println(" Case - limiting-cell completely contained in the polyhedron");
        res = new Polyhedron[1];
        res[0] = new Polyhedron(cell);}
    else{if(this.disjunct(cell))
        {//System.out.println(" Case - limiting cell cuts the polyhedron");
        res = new Polyhedron[0];}
    else
        {
           //System.out.println(" Case 3");
           Vector all = new Vector();
           all.add(this);
           MultiPolygon faces = new MultiPolygon();
           faces = cell.polygons();
           int N = faces.polygons.length;
           for(int i=0;i<N;i++)
               {
                  int M = all.size();
                  //System.out.println("   "+i+" "+M);
                  Vector allNew = new Vector();
                  Polygon plaLim = new Polygon(faces.polygons[i]);
                  Plane pla = new Plane(plaLim);
                  for(int j=0;j<M;j++)
                      {
                          Polyhedron loc = new Polyhedron((Polyhedron)all.get(j));
                          MultiPolyhedron singleCut;

                          if(!loc.disjunct(plaLim))
                                  // does the polyhedron interact with polygon, or only plane?
                              {
                                  singleCut = new MultiPolyhedron();
                                  singleCut = loc.cut(pla);
                              }
                          else
                              {
                                  singleCut = new MultiPolyhedron(loc);
                              }
                          int O = singleCut.polyhedrons.length;

                          //pla.WriteScr();singleCut.WriteScr();
                          System.out.println(" this is dividied into:");

                          MultiPolyhedron primitives[] = new MultiPolyhedron[O];
                          for(int k=0;k<O;k++)
                              {primitives[k] = new MultiPolyhedron();
                              primitives[k].reduceTopo(singleCut.polyhedrons[k]);
                              int numPrims = primitives[k].polyhedrons.length;
                              for(int prims=0; prims<numPrims; prims++)
                                  {
                                      primitives[k].polyhedrons[prims].order(false);
                                  }

                              //for(int l=0;l<(primitives[k].polyhedrons.length);l++)
                              //{primitives[k].polyhedrons[l].WriteScr();}
                              }


                          for(int k=0;k<O;k++)
                              {
                                  int P = primitives[k].polyhedrons.length;
                                  for(int l=0;l<P;l++)
                                      {
                                          if(!primitives[k].polyhedrons[l].degenerated())
                                              {allNew.add(primitives[k].polyhedrons[l]);}
```

```
                                        else{/*primitives[k].polyhedrons[l].WriteScr();*/}
                                    }
                                }
                            }
                        all.setSize(0);
                        int O = allNew.size();
                        for(int j=0;j<O;j++)
                            {
                                all.add((Polyhedron)allNew.get(j));
                            }
                        allNew.setSize(0);
                    }
                int M = all.size();

                res = new Polyhedron[M];
                for(int i=0;i<M;i++){res[i] = new Polyhedron((Polyhedron)all.get(i));}
            }}}

        MultiPolyhedron resu = new MultiPolyhedron(res);
        return resu;


    }


    // ************************************* primList ************************************

    /**
       Gives back a entity list of the primitive elements (tetraeder) of the defined polyhedrons
       @return entity list in the form [elements][tetraeder entities], each row contains a different
element, each element is described by the number of the original corner-points of the polyhedron.
     */
    public int[][] primList()
    {
        int numCorns = this.corner.length;

        int N=0;

        if(this.topology==6){N=1;}
        if(this.topology==7){N=3;}
        if(this.topology==8){N=5;}

        int res[][] = new int[N][4];

        if(this.topology==6 && numCorns==4)
            {
                for(int i=0;i<4;i++){res[0][i]=i;}
            }
        else{if(this.topology==7 && numCorns==6)
            {
                res[0][0] = 0; res[0][1] = 4; res[0][2] = 5; res[0][3] = 3;
                res[1][0] = 0; res[1][1] = 4; res[1][2] = 2; res[1][3] = 5;
                res[2][0] = 0; res[2][1] = 1; res[2][2] = 2; res[2][3] = 4;
            }
        else{if(this.topology==8 && numCorns==8)
            {
                res[0][0] = 0; res[0][1] = 2; res[0][2] = 7; res[0][3] = 5;
                res[1][0] = 0; res[1][1] = 2; res[1][2] = 3; res[1][3] = 7;
                res[2][0] = 0; res[2][1] = 5; res[2][2] = 7; res[2][3] = 4;
                res[3][0] = 0; res[3][1] = 1; res[3][2] = 2; res[3][3] = 5;
                res[4][0] = 2; res[4][1] = 7; res[4][2] = 5; res[4][3] = 6;
            }
        else
            {
                System.out.println(" ******************* Error, in primList, topology not correct!");
            }}}
        return res;
    }

    //*********************************************************************************************

    /**
       Sides of a Polyhedron
       @return Sides of the polyhedron as Line-Type in a list
     */
    public Line[] sides()
    {
```

```
        int N = this.corner.length;

        Line res[];

        if((this.topology==6) && (N==4))
            {
                res = new Line[6];

                for(int i=0;i<3;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[(i+1)%3]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i] = new Line(orig,side);}
                for(int i=0;i<3;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[3]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i+3] = new Line(orig,side);}
            }
        else{if((this.topology==7) && (N==6))
            {
                res = new Line[9];

                for(int i=0;i<3;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[(i+1)%3]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i] = new Line(orig,side);}
                for(int i=0;i<3;i++)
                    {Coordinate side = new Coordinate(this.corner[i+3],this.corner[((i+1)%3)+3]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i+3]);
                    res[i+3] = new Line(orig,side);}
                for(int i=0;i<3;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[i+3]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i+6] = new Line(orig,side);}
            }
        else{if((this.topology==8) && (N==8))
            {
                res = new Line[12];

                for(int i=0;i<4;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[(i+1)%4]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i] = new Line(orig,side);}
                for(int i=0;i<4;i++)
                    {Coordinate side = new Coordinate(this.corner[i+4],this.corner[((i+1)%4)+4]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i+4]);
                    res[i+4] = new Line(orig,side);}
                for(int i=0;i<4;i++)
                    {Coordinate side = new Coordinate(this.corner[i],this.corner[i+4]);
                    Coordinate orig = new Coordinate(this);orig.add(corner[i]);
                    res[i+8] = new Line(orig,side);}
            }
        else{res = new Line[0];
        System.out.println("   Error - something wrong with Polyhedron-Type");}}}


        return res;
    }


    //*************************************************************************

    public void WriteScr()
    {
        System.out.print("\t Polyhedron of type "+this.topology);
        int N = this.corner.length;
        if(N==4){System.out.println(" (= tetraeder)");}
        if(N==6){System.out.println(" (= wedge)");}
        if(N==8){System.out.println(" (= brick)");}
        System.out.print("\t\t The reference point is: "); super.WriteScr();
        System.out.print("\n\t\t and the corners are:\n\t\t\t");
        for(int i=0;i<N;i++)
            {
                this.corner[i].WriteScr();
                if(N==6 && i==2){System.out.print("\n\t\t\t");}
                if(N==8 && i==3){System.out.print("\n\t\t\t");}
            }
        System.out.println();
```

```
    }
}
```

C.2.8 The sphere class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

// ********************************************************************************************
// *********************************   Class Sphere  ******************************************
// ********************************************************************************************

/**
Defines sphere in R3.
 */
public class Sphere extends Coordinate
{
    /**
       Radius of the sphere
     */
    public double r;

    //********************************** CONSTRUCTORS ***************************************

    /**
       Constructor with the three coordinates and the radius as argument.
       @param xi three cooridnates of the sphere-center
       @param r radius of the sphere
     */
    public Sphere(double x0, double x1, double x2, double radius)
    {
        //this.r = radius;
        //this.x[0]=x0; this.x[1]=x1; this.x[2]=x2;
        super(x0,x1,x2);
        this.r = radius;

    }

    /**
       Constructor with no argument. Constructs a degenerated sphere (radius = 0.0) in the origin.
     */
    public Sphere()
    {
        this(0.0,0.0,0.0,0.0);
    }

   /**
      Constructor with the coordinate and the radius as argument.
      @param loc coordinate of the sphere-center location
      @param r radius of the sphere
     */
    public Sphere(Coordinate loc, double radius)
    {
        super(loc);
        this.r = radius;
    }

    /**
      Copy-constructor with a sphere as argument
      @param toCopy Shpere which is subject to be copied
     */
    public Sphere(Sphere toCopy)
    {
        this(toCopy.x[0],toCopy.x[1],toCopy.x[2],toCopy.r);
    }

    //*************************************** ASSIGN *********************************

    /**
       Assigns sphere with center-coordinates and radius.
```

```
       @param xi three cooridnates of the sphere-center
       @param r radius of the sphere
     */
    public void assign(double x0, double x1, double x2, double radius)
    {
        super.assign(x0,x1,x2);
        this.r = radius;
    }

    /**
       Assigns degenerated sphere (Radius = 0.0) in the origin
     */
    public void assign()
    {
        this.assign(0.0,0.0,0.0,0.0);
    }

    /**
      Assign operator with the coordinate and the radius as argument.
      @param loc coordinate of the sphere-center location
      @param r radius of the sphere
     */
    public void assign(Coordinate loc, double radius)
    {
        super.assign(loc);
        this.r = radius;
    }

    /**
      Copy-operator with a sphere as argument
      @param toCopy Shpere which is subject to be copied
     */
    public void assign(Sphere toCopy)
    {
        this.assign(toCopy.x[0],toCopy.x[1],toCopy.x[2],toCopy.r);
    }

    //****************************************** WriteScr ************************************

    public void WriteScr()
    {
        super.WriteScr();
        System.out.print(" - "+this.r+"  ");
    }

    public void WritelnScr()
    {
        System.out.print("\n  Sphere with Coordinate ");
        super.WriteScr();
        System.out.print(" and radius "+this.r+" \n ");
    }
}
```

C.2.9 The Face class

```
/*
 * Copyright 2000-2002 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

// ***************************************************************************************
// *********************************   Class Face  ***************************************
// ***************************************************************************************

public class Face
{

    public Coordinate orig;
    public Coordinate dir[];
    int type; // 0 if triangle, 1 if square (sheared)

    public Face()
```

```
    {
        orig = new Coordinate();
        dir = new Coordinate[2];
        for(int i=0; i<2; i++){dir[i] = new Coordinate();}
        type = 1;
    }

    public Face(Coordinate origin, Coordinate dir0, Coordinate dir1, int faceType)
    {
        orig = new Coordinate(origin);
        dir = new Coordinate[2];
        dir[0] = new Coordinate(dir0); dir[1] = new Coordinate(dir1);
        type = faceType;
    }

    public Face(Face F)
    {
        orig = new Coordinate(F.orig);
        dir = new Coordinate[2];
        dir[0] = new Coordinate(F.dir[0]); dir[1] = new Coordinate(F.dir[1]);
        type = F.type;
    }

    public void assign(Coordinate origin, Coordinate dir0, Coordinate dir1, int faceType)
    {
        orig.assign(origin);
        dir[0].assign(dir0); dir[1].assign(dir1);
        type = faceType;
    }

    public void WriteScr()
    {
        System.out.print(" Face: |");
        System.out.print(orig.x[0]+" "+orig.x[1]+" "+orig.x[2]+"| + sides ");
        for(int i=0;i<2;i++){System.out.print("|"+dir[i].x[0]+" "+dir[i].x[1]+" "+dir[i].x[2]+"| ");}
        System.out.println();
    }
}
```

C.2.10 The matrix class

```
/*
 * Copyright 1996-2000 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

//
**********************************************************************************************************
**********
// *********************************   Class Matrix
***********************************************
//
**********************************************************************************************************
**********

/**
   Representing a Matrix; mainly for vector transformations

   @version 1.00 2001/01/10
   @author Manuel Alexandre POUCHON - International Fellow of JNC Tokai Works (JAPAN)
*/
public class Matrix
{
    public Matrix(double x11, double x12, double x13, double x21, double x22, double x23, double x31,
double x32, double x33)
    {
        x = new double[3][3];
        x[0][0]=x11; x[0][1]=x21; x[0][2]=x31;
        x[1][0]=x12; x[1][1]=x22; x[1][2]=x32;
        x[2][0]=x13; x[2][1]=x23; x[2][2]=x33;
    }
```

```
    public Matrix()
    {
        x = new double[3][3];
         for(int i=0; i<3; i++){for(int j=0; j<3;j++){x[i][j]=0.0;}}
    }

    public void assign(double x11,double x12,double x13,double x21,double x22,double x23,double
x31,double x32,double x33)
    {
        x[0][0]=x11; x[0][1]=x21; x[0][2]=x31;
        x[1][0]=x12; x[1][1]=x22; x[1][2]=x32;
        x[2][0]=x13; x[2][1]=x23; x[2][2]=x33;
    }

    public Matrix(Coordinate v1, Coordinate v2, Coordinate v3)
    {
        x = new double[3][3];
        assign(v1.x[0], v1.x[1], v1.x[2], v2.x[0], v2.x[1], v2.x[2], v3.x[0], v3.x[1], v3.x[2]);
    }

    public void assign(Coordinate v1, Coordinate v2, Coordinate v3)
    {
        assign(v1.x[0], v1.x[1], v1.x[2], v2.x[0], v2.x[1], v2.x[2], v3.x[0], v3.x[1], v3.x[2]);
    }

    public Matrix(Matrix M)
    {
        x = new double[3][3];
        for(int i=0; i<3; i++){for(int j=0; j<3; j++){x[i][j]=M.x[i][j];}}
    }

    public void assign(Matrix M)
    {
        for(int i=0; i<3; i++){for(int j=0; j<3; j++){x[i][j]=M.x[i][j];}}
    }

    public void mult(Matrix M, boolean first)
    {
        // first = True if Calss Member is fist Marix in Multiplication

        Matrix A = new Matrix();

        if (first)
            {for (int i=0; i<3; i++)
               { for (int j=0; j<3; j++)
                   { for (int k=0; k<3; k++)
                      { A.x[i][j] += x[i][k]*M.x[k][j];
                      };
                   };
               };
            }
        else
            {for (int i=0; i<3; i++)
               { for (int j=0; j<3; j++)
                   { for (int k=0; k<3; k++)
                      { A.x[i][j] += x[k][j]*M.x[i][k];
                      };
                   };
               };
            };

        for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = A.x[i][j];}; };
    }

    public void mult(Matrix M0, Matrix M2)
    {
        Matrix M1 = new Matrix(M0);
        M1.mult(M2,true);
        for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = M1.x[i][j];}; };
    }

    public void inv()
    {
        double A[][] = new double[3][3];

        A[0][0] =
               ((-x[1][2])*x[2][1] + x[1][1]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
```

```
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[0][1] = (x[0][2]*x[2][1] - x[0][1]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[0][2] = ((-x[0][2])*x[1][1] + x[0][1]*x[1][2])/((-x[0][2])* x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);

        A[1][0] = (x[1][2]*x[2][0] - x[1][0]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[1][1] = ((-x[0][2])*x[2][0] + x[0][0]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[1][2] = (x[0][2]*x[1][0] - x[0][0]*x[1][2])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);

        A[2][0] = ((-x[1][1])*x[2][0] + x[1][0]*x[2][1])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[2][1] = (x[0][1]*x[2][0] - x[0][0]*x[2][1])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);
        A[2][2] = ((-x[0][1])*x[1][0] + x[0][0]*x[1][1])/((-x[0][2])*x[1][1]*x[2][0] +
                x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] -
                x[0][1]*x[1][0]*x[2][2] + x[0][0]*x[1][1]*x[2][2]);

        for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = A[i][j]; }; };
    }

    public void inv(Matrix mIn)
    {
        Matrix M = new Matrix(mIn);
        M.inv();
        for (int i=0; i<3; i++){ for (int j=0; j<3; j++) {x[i][j]=M.x[i][j]; }; };
    }

    public void WriteScr()
    {
        for(int i=0; i<3; i++){for(int j=0; j<3; j++)
                                {System.out.print(x[j][i]+" ");}System.out.println();}
    }


    public double x[][];
}
```

C.2.11 The MultiLine class

```
/*
 * Copyright 1996-2000 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;


// ********************************************************************
// *********************************   Class MultLine  ****************
// ********************************************************************

/**
Class which represents multiple lines, these can be a radom collection of single lines, possible
connections at the endpoints are detected and declared.
 */
public class MultiLine
{
    /**
       Collection of single points describing the multiple line.
     */
    public Coordinate[] points;
    /**
```

```
       Gives a list for each point, which describes possible connections to other points.
     */
    public int[][] connect;
    /**
       Possible path along the lines. All lines should be included. Single connected multiple lines
start and end with "-1", if there are several line-fragments, this is then indicated with additional
"-1" entities.
     */
    public int[] path;
    /**
       Is true if the multiple line connects to one single path.
     */
    public int numPaths;
    /**
       Is true if the multiple line connects to one single closed path.
     */
    public boolean circular;

    //*******************  Constructors *********************************
    public MultiLine()
    {
    }

    /**
       Generates a multiple line from a list of single lines,
                    automatic endpoint-connection detection.
     */
    public MultiLine(Line in[])
    {
       int N = in.length;
       //for(int i=0;i<N;i++){in[i].WriteScr();}

       boolean coinc[][][][] = new boolean[N][2][N][2];        // Coincidence Matrix of LinePoints
       for(int i=0; i<N; i++){for(int j=0; j<N; j++)   // initialisation of all values to false
           {coinc[i][0][j][0]=false; coinc[i][0][j][1]=false;
                    coinc[i][1][j][0]=false; coinc[i][1][j][1]=false;}}

       int numCoinc[][] = new int[N][2];
                    // number of coincidences for each line-end
       for(int i=0; i<N; i++){numCoinc[i][0]=0; numCoinc[i][1]=0;}

       for(int i=0; i<N; i++)                            // finding coincidences of points
           {
               Coordinate tgt0 = new Coordinate(in[i]);
               // startpoint of line we compare with
               Coordinate tgt1 = new Coordinate(tgt0); tgt1.add(in[i].dir);
               // endpoint of line we compare with
               for(int j=0; j<N; j++)
                   {
                       if(i!=j)
                           {
                               Coordinate toCo0 = new Coordinate(in[j]);     // Points to copare to
                               Coordinate toCo1 = new Coordinate(toCo0); toCo1.add(in[j].dir);
                               if(tgt0.equal(toCo0)){numCoinc[i][0]++; coinc[i][0][j][0] = true;}
                               if(tgt0.equal(toCo1)){numCoinc[i][0]++; coinc[i][0][j][1] = true;}
                               if(tgt1.equal(toCo0)){numCoinc[i][1]++; coinc[i][1][j][0] = true;}
                               if(tgt1.equal(toCo1)){numCoinc[i][1]++; coinc[i][1][j][1] = true;}
                           }
                   }
           }

       /*for(int a=0; a<2; a++){for(int b=0;b<2;b++){System.out.println();for(int i=0; i<N; i++)
         {System.out.println();for(int j=0; j<N; j++)
               {System.out.print(" | "+coinc[i][a][j][b]);}}}}*/

       double numSepPointsReal = 0.0;
                           // number of separated points describing the whole multiple line
                            // defined as double because of fraction adition below
       for(int i=0; i<N; i++)
           {
               numSepPointsReal += 1.0/(1.0+(double)(numCoinc[i][0]));
               // add both line sides with their importance
               numSepPointsReal += 1.0/(1.0+(double)(numCoinc[i][1]));
           }
       int numSepPoints = (int)(numSepPointsReal+Coordinate.precision);
       if((Math.abs((double)numSepPoints-numSepPointsReal)>Coordinate.precision))
       {System.out.println(" Error in Different point evaluation. Not an integer");}
```

```
this.points = new Coordinate[numSepPoints];
Coordinate pointList[] = new Coordinate[numSepPoints];
for(int i=0; i<numSepPoints; i++){pointList[i] = new Coordinate(-1. , -1. , -1.);}

int preConn[][] = new int[N][2];
for(int i=0; i<N; i++){for(int j=0; j<2; j++){preConn[i][j]=-1;}}
        // initialisation with -1 -values

{               // assigning numbers to Coordinates, later this numbers are used as reference
    int k=0;    // If coordinate is double, only one number is assigned!
for(int i=0; i<N; i++)
    {
        for(int j=0; j<2; j++)
            {
                if(preConn[i][j]==-1)
                    {
                        preConn[i][j]=k;                        // k'th point assigned
                        //System.out.println(" indizes :"+k+" "+i);
                        pointList[k]=new Coordinate(in[i]);// assigning values to points
                        if(j==1){pointList[k].add(in[i].dir);}
                        // if endpoint, than add line-length
                        for(int l=0; l<N; l++)
                            {
                                // look for all entrys if there are some coincidences
                                for(int m=0; m<2; m++)
                                // and assign same point identity if so
                                    {
                                        if(coinc[i][j][l][m])
                                            {
                                                preConn[l][m]=k;
                                            }
                                    }
                            }
                        k++;
                    }
            }
    }
}

//      for(int i=0;i<2;i++){System.out.println();
                for(int j=0;j<N;j++){System.out.print(" |"+preConn[j][i]);}}

int[][] pathConn = new int[numSepPoints][];

Vector tempConn = new Vector();

for(int i=0; i<numSepPoints; i++)
        // Finds all connections and writes them to a Vector
    {
        for(int j=0; j<N; j++)
            {
                if(preConn[j][0]==i || preConn[j][1]==i)
                    {
                        if(preConn[j][0]==i)
                            {tempConn.add(new Integer(preConn[j][1]));}
                        else
                            {tempConn.add(new Integer(preConn[j][0]));}
                    }
            }
        // double path erase, seek for souble entities and erase them
        int iN = tempConn.size();
        for(int ii=0;ii<iN-1;ii++)
        {int locComp =((Integer)tempConn.get(ii)).intValue() ;for(int jj=ii+1;jj<iN;jj++)
            {if((((Integer)tempConn.get(jj)).intValue())==locComp)
                {tempConn.remove(jj); jj--; iN--;}}}
        // end of double path erase
        tempConn.trimToSize();
        iN = tempConn.size();
        pathConn[i] = new int[iN];   // writing Vector-elements to an Array
        for(int k=0; k<iN; k++){pathConn[i][k] = ((Integer)tempConn.get(k)).intValue();}
        tempConn.setSize(0);    // kill elements in Vector

        //System.out.println();for(int k=0;k<iN;k++)
                {System.out.print(" |"+pathConn[i][k]+"|");}System.out.println();
    }
```

```
// maybe add some kind of order feature ...................

for(int i=0; i<numSepPoints; i++){this.points[i] = new Coordinate(pointList[i]);}

this.connect = new int[numSepPoints][];
for(int i=0; i<numSepPoints; i++)
    {int NI=pathConn[i].length; this.connect[i] = new int[NI];
         for(int j=0; j<NI; j++)
             {this.connect[i][j] = pathConn[i][j];}}

// Path-detection

Vector remPath[] = new Vector[numSepPoints]; // Vector-Array with the remaining path
for(int i=0; i<numSepPoints; i++)
    {int NI=pathConn[i].length; remPath[i] = new Vector(NI);
         for(int j=0; j<NI; j++)
             {remPath[i].add(new Integer(pathConn[i][j]));}}

Vector multiplePath = new Vector();
boolean noLeft = false;

while(!noLeft)
    {
       int startPt = -1;
       {int i = 0;
       while((startPt == -1) && (i<numSepPoints))
           {
              if(remPath[i].size()==1){startPt = i;}  // Edge point
              i++;
           }
       }
       if(startPt==-1)
           {
              {int i = 0;
              while((startPt == -1) && (i<numSepPoints))
                  {
                     if(remPath[i].size()>0){startPt = i;}  // Edge point
                     i++;
                  }
              }
           }
       if(startPt==-1)
{System.out.println(" ***** ERROR: Somethin wrong with remaining points vector!!!");break;}

       multiplePath.add(new Integer(-1));    // start of new partial path
       multiplePath.add(new Integer(startPt)); // first entry
       int nextInd = ((Integer)remPath[startPt].get(0)).intValue();
       int lastInd = startPt;
       remPath[lastInd].remove(0);

       while(remPath[nextInd].size()>0)   // look for aditional points
           {
              int entityToLast = -1, actLength = remPath[nextInd].size();
              for(int i=0; i<actLength; i++)
              // finding back-connection to last pt
            {if((((Integer)remPath[nextInd].get(i)).intValue())==lastInd){entityToLast=i;}}
              if(entityToLast==-1)
              {System.out.println(" ***** ERROR: Somethin wrong with rem pt vec!!!");break;}
              remPath[nextInd].remove(entityToLast);  // and kill it

              multiplePath.add(new Integer(nextInd));
              lastInd = nextInd;

              if(remPath[lastInd].size()>0)    // are there elements left in Vector
                  {
                     nextInd = ((Integer)remPath[lastInd].get(0)).intValue();
                     remPath[lastInd].remove(0);
                  }
              else
                  {
                     multiplePath.add(new Integer(-1)); // close this partial path
                  }
           }

       {int bigLength=0;
       for(int i=0; i<numSepPoints; i++)
           {int locLength = remPath[i].size(); if(locLength>bigLength){bigLength=locLength;}}
```

```
                if(bigLength==0){noLeft=true;}else{noLeft=false;}
                }
            }

        int NP = multiplePath.size();
        this.path = new int[NP];
        for(int i=0; i<NP; i++){this.path[i] = ((Integer)multiplePath.get(i)).intValue();}

        int numPathsLoc = 0;
        for(int i=0; i<NP; i++)
            {
                if(this.path[i]==-1){numPathsLoc++;}
            }
        if(numPathsLoc%2!=0){System.out.println("  ** Error at path initialisation, number of -1 not
even");}
        this.numPaths = numPathsLoc/2;

        if(this.numPaths == 1 && (this.path[1] == this.path[NP-2]))
            {circular = true;}
        else{circular = false;}
    }

    //*******************  assign
*********************************************************************************************

    public void assign(MultiLine toCopy)
    {
        {
            int N = toCopy.points.length;
            this.points = new Coordinate[N];
            for(int i=0; i<N; i++){this.points[i] = new Coordinate(toCopy.points[i]);}
        }
        {
            int N = toCopy.connect.length;
            this.connect = new int[N][];
            for(int i=0; i<N; i++)
                {int M = toCopy.connect[i].length;
                this.connect[i] = new int[M];
                for(int j=0; j<M; j++)
                    {this.connect[i][j] = toCopy.connect[i][j];}}
        }
        {
            int N = toCopy.path.length;
            this.path = new int[N];
            for(int i=0; i<N; i++){this.path[i] = toCopy.path[i];}
        }
        this.numPaths = toCopy.numPaths;
        this.circular = toCopy.circular;
    }

    //******************* Cut Polygon and plane
*************************************************************************

    /**
       Cuts a polygons with a plane and assigns the cutting multiple-line.
       @param pol polygon to cut
       @param pla plane to cut
     */
    public boolean cut(Polygon pol, Plane pla)
    {
        boolean isCut = false;
        MultiLine allLines;

        if(pol.topology!=4) // polygon is triangle
            {
                Line singleLine[] = new Line[1];
                singleLine[0] = new Line();
                isCut = singleLine[0].cut(pol,pla);
                allLines = new MultiLine(singleLine);
            }
        else        // is quadragle
            {
                Line toCutLines[] = new Line[2]; for(int i=0;i<2;i++){toCutLines[i] = new Line();}
                Polygon pol1 = new Polygon(), pol2 = new Polygon();

                if(pol.rt_lf)
                    {
```

```
                    pol1.assign(pol, pol.corner[0], pol.corner[1], pol.corner[2]);
                    pol2.assign(pol, pol.corner[0], pol.corner[2], pol.corner[3]);
                }
            else
                {
                    pol1.assign(pol, pol.corner[1], pol.corner[2], pol.corner[3]);
                    pol2.assign(pol, pol.corner[1], pol.corner[3], pol.corner[0]);
                }

            boolean isCut1 = false, isCut2 = false;

            isCut1 = toCutLines[0].cut(pol1,pla);
            isCut2 = toCutLines[1].cut(pol2,pla);

            if(isCut1 && isCut2)
                {
                    isCut = true;
                    Line cuttingLines[] = new Line[2];
                    cuttingLines[0] = new Line(toCutLines[0]);
                    cuttingLines[1] = new Line(toCutLines[1]);
                    allLines = new MultiLine(cuttingLines);
                }
            else{if(isCut1 || isCut2)
                {
                    isCut = true;
                    Line singleLine[] = new Line[1];
                    if(isCut1){singleLine[0] = new Line(toCutLines[0]);}
                    else{singleLine[0] = new Line(toCutLines[1]);}
                    allLines = new MultiLine(singleLine);
                }
            else
                {
                    isCut = false;
                    Line noLine[] = new Line[1];
                    noLine[0] = new Line();
                    allLines = new MultiLine(noLine);
                }}

        }
    this.assign(allLines);
    return isCut;
}


    //******************* Cut two Polygons
*****************************************************************************

    public boolean cut(Polygon P1, Polygon P2)
    {
        boolean result = false;
        MultiLine allLines;

        if(P1.topology!=4 && P2.topology!=4) // both Polygons being triangles
            {
                Line singleLine[] = new Line[1];
                singleLine[0] = new Line();
                result = singleLine[0].cut(P1,P2);
                allLines = new MultiLine(singleLine);
            }
        else
            {
                Line cuttingLines[];
                if((P1.topology==4 && P2.topology!=4) || (P1.topology!=4 && P2.topology==4))
                    {
                        Polygon P1C, P2C;
                        if(P1.topology==4)
                            {
                                P1C = new Polygon(P1); P2C = new Polygon(P2);
                            }
                        else
                            {
                                P1C = new Polygon(P2); P2C = new Polygon(P1);
                            }
                        Polygon P11 = new Polygon();
                        Polygon P12 = new Polygon();
                        Line cutP11P2 = new Line();
                        Line cutP12P2 = new Line();
```

```
            boolean isCutP11P2 = false;
            boolean isCutP12P2 = false;
            if(P1C.rt_lf)
                {
                    P11.assign(P1C, P1C.corner[0], P1C.corner[1], P1C.corner[2]);
                    P12.assign(P1C, P1C.corner[0], P1C.corner[2], P1C.corner[3]);
                }
            else
                {
                    P11.assign(P1C, P1C.corner[1], P1C.corner[2], P1C.corner[3]);
                    P12.assign(P1C, P1C.corner[1], P1C.corner[3], P1C.corner[0]);
                }
            //P11.WriteScr();P12.WriteScr();
            if(cutP11P2.cut(P11,P2C)){isCutP11P2 = true;}else{isCutP11P2 = false;}
            if(cutP12P2.cut(P12,P2C)){isCutP12P2 = true;}else{isCutP12P2 = false;}
            //System.out.println("  "+isCutP11P2+"  "+isCutP12P2);

            if(isCutP11P2 && isCutP12P2)
                {                                   // assigns both cutting lines
                    cuttingLines = new Line[2];
                    cuttingLines[0] = new Line(cutP11P2); //cuttingLines[0].WriteScr();
                    cuttingLines[1] = new Line(cutP12P2); //cuttingLines[1].WriteScr();
                    result = true;
                }
            else if(isCutP11P2 || isCutP12P2)
                {
                                        // assigns the cutting line
                    cuttingLines = new Line[1];
                    if(isCutP11P2){cuttingLines[0]=
                    new Line(cutP11P2);}else{cuttingLines[0]=new Line(cutP12P2);}
                    result = true;
            // cuttingLines[0].WriteScr();
                }
            else
                {
                    // assigns both cutting lines of infinit planes
                    cuttingLines = new Line[2];
                    cuttingLines[0] = new Line(cutP11P2);  //cuttingLines[0].WriteScr();
                    cuttingLines[1] = new Line(cutP12P2);  //cuttingLines[1].WriteScr();
                    result = false;
                }
        }
    else if(P1.topology==4 && P2.topology==4)
        {
            Polygon PT[][] = new Polygon[2][2];
            // 1'st index: # of quadrangle --- 2'nd index: # of subtriangle
            Line cutPT[][] = new Line[2][2];
            //     1'st index: # of subtrianle in 1'st quadrangle
            boolean isCutPT[][] = new boolean[2][2];
            // and 2'nd index: # of subtrianle in 2'nd quadrangle
            for(int i=0; i<2; i++){for(int j=0; j<2; j++)
                {
                    isCutPT[i][j] = false;
                    PT[i][i] = new Polygon();
                    cutPT[i][i] = new Line();
                }}

            if(P1.rt_lf)
            // orientation of first quadrangle -> creation of two new triangle
                {
                    PT[0][0].assign(P1, P1.corner[0], P1.corner[1], P1.corner[2]);
                    PT[0][1].assign(P1, P1.corner[0], P1.corner[2], P1.corner[3]);
                }
            else
                {
                    PT[0][0].assign(P1, P1.corner[1], P1.corner[2], P1.corner[3]);
                    PT[0][1].assign(P1, P1.corner[0], P1.corner[1], P1.corner[3]);
                }
            if(P2.rt_lf)
            // orientation of second quadrangle -> creation of two new triangle
                {
                    PT[1][0].assign(P2, P2.corner[0], P2.corner[1], P2.corner[2]);
                    PT[1][1].assign(P2, P2.corner[0], P2.corner[2], P2.corner[3]);
                }
            else
                {
                    PT[1][0].assign(P2, P2.corner[1], P2.corner[2], P2.corner[3]);
```

```
                        PT[1][1].assign(P2, P2.corner[0], P2.corner[1], P2.corner[3]);
                    }
                int numCuttingLines = 0;
                for(int i=0; i<2; i++){for(int j=0; j<2; j++)
                    {
                        if(cutPT[i][j].cut(PT[0][i],PT[1][j]))
                            {isCutPT[i][j]=true; numCuttingLines++ ;}
                        else{isCutPT[i][j]=false;}
                    }}
                if(numCuttingLines>0)  // there is at leaste one cutting line
                    {
                        cuttingLines = new Line[numCuttingLines];
                        int k = 0;
                        for(int i=0; i<2; i++){for(int j=0; j<2; j++)
                            {
                                if(isCutPT[i][j]){cuttingLines[k].assign(cutPT[i][j]);}
                                k++;
                            }}
                        result = true;
                    }
                else            // two polygons do not interact, assign infinit planes
                    {
                        cuttingLines = new Line[4];
                        int k = 0;
                        for(int i=0; i<2; i++){for(int j=0; j<2; j++)
                            {
                                cuttingLines[k].assign(cutPT[i][j]);
                                k++;
                            }}
                        result = false;
                    }
                if(result)
                    {
                    }

            }
        else
            {
                cuttingLines = new Line[1]; cuttingLines[0] = new Line();
                System.out.println(" Error * Not identified case  ");
            }
        allLines = new MultiLine(cuttingLines);
      }

    this.assign(allLines);
    return result;
  }

  //********************  WriteScr ************************************************************

  /**
     Writes a multiple line to the screen. It declares all points, all connections between them and
the detected path.
   */
  public void WriteScr()
  {
    int N = this.points.length;
    System.out.println("\n MultiLine:\n\t The independent point-coordinates are:");
    for(int i=0; i<N; i++){System.out.print("\t\t Point - "+i+": ");this.points[i].WriteScr();}
    System.out.print("\t The point-connections are:
                       (each point connects to the tabulated points) ");
    for(int i=0; i<N; i++)
        {System.out.print("\n\t\t Point - "+i+":    "); int NI = this.connect[i].length;
        for(int j=0; j<NI; j++)
            {System.out.print(this.connect[i][j]); if(j<(NI-1)){System.out.print(" <-> ");}}}
    int NP = this.path.length;
    System.out.print("\n\t Possible path along the multiple line:
                       (-1 signifies a line-start or stop)\n\t\t");
    for(int i=0; i<NP; i++){System.out.print(" "+this.path[i]);}
    if(this.numPaths == 1)
        {System.out.print("\n\t There is exacly one path which is ");
        if(this.circular){System.out.println("circular (closed path)");}
        else{System.out.println("not circular (open path)");}}
    else{System.out.println("\n\t The multiple line is not a single path,
                             there are "+this.numPaths+" separate paths");}
    System.out.println();
```

```
    }
}
```

C.2.12 The MultiPolyhedron class

```
package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;

// ********************************************************************************
// ***********************************   Class Polygon ****************************
// ********************************************************************************

public class MultiPolygon
{
    // ****************************** Varables of the Class ****************************

    /**
       List of independent polygons
     */
    public  Polygon[] polygons;
    /**
       Connection between the polygons, only perfect connections (coincidence of a side) are listed.
    */
    public  int[][] connect;
    /**
    */
    public boolean[] fullyConnected;
    /**
     */
    public boolean closed;


   // ******************** Constructors of the Class *********************************

    public MultiPolygon()
    {
    }

    public MultiPolygon(Polygon in[])
    {
        int N = in.length; //System.out.println(N);

        Vector inTemp = new Vector(N);

        for(int i=0; i<N; i++){inTemp.add(in[i]);}

        for(int i=0;i<(N-1);i++){for(int j=(i+1);j<N;j++)
            {//System.out.println(" "+i+" "+j);
            Polygon A = new Polygon((Polygon)inTemp.get(i)), B = new Polygon((Polygon)inTemp.get(j));
            if(A.equal(B)){inTemp.remove(j);j--;N--;/*System.out.println("kill"+i+" "+j+" "+N);*/}}}

        //System.out.println(" "+N);System.out.println(" "+inTemp.size());
        //for(int i=0;i<N;i++){Polygon A = new Polygon((Polygon)inTemp.get(i)); A.WriteScr();}

        Polygon inFin[] = new Polygon[N];
        for(int i=0;i<N;i++){inFin[i] = new Polygon((Polygon)inTemp.get(i));}

        this.polygons = new Polygon[N]; //System.out.println(N);
        for(int i=0;i<N;i++){this.polygons[i] = new Polygon(inFin[i]);}

        boolean coinc[][][][] = new boolean[N][4][N][4];
        // Coincidence Matrix of Polygon-Lines, there are 3 or 4 lines
        for(int i=0; i<N; i++){for(int j=0; j<N; j++){for(int k=0; k<4; k++){for(int l=0; l<4; l++)
            {coinc[i][k][j][l]=false;}}}}

        int numCoinc[][] = new int[N][4];
        // number of coincidences for each line-end
        for(int i=0; i<N; i++){for(int j=0; j<4; j++){numCoinc[i][j]=0;}}

        for(int i=0; i<N; i++)                          // finding coincidences of points
            {
               int tgtDim = 0;
               if(inFin[i].topology==4){tgtDim = 4;}else{tgtDim = 3;}
                Line tgt[] = new Line[tgtDim];
```

```
            for(int a=0; a<tgtDim; a++)
                {int b = (a+1)%tgtDim;
                Coordinate start = new Coordinate(inFin[i]), end = new Coordinate(start);
                start.add(inFin[i].corner[a]); end.add(inFin[i].corner[b]);
                tgt[a] = new Line(start,end,true);}    // lines of polygon we compare with

            for(int j=0; j<N; j++)
                {
                   if(i!=j)
                       {
                           int toCoDim = 0;
                           if(inFin[j].topology==4){toCoDim = 4;}else{toCoDim = 3;}
                           Line toCo[] = new Line[toCoDim];
                           for(int a=0; a<toCoDim; a++)
                               {int b = (a+1)%toCoDim;
                               Coordinate start = new Coordinate(inFin[j]),
                                   end = new Coordinate(start);
                               start.add(inFin[j].corner[a]); end.add(inFin[j].corner[b]);
                               toCo[a] = new Line(start,end,true);}
                                   // lines of comparing polygon

                           for(int a=0; a<tgtDim; a++){for(int b=0; b<toCoDim; b++)
                               {if(tgt[a].equal(toCo[b]))
                                   {numCoinc[i][a]++;
                                   coinc[i][a][j][b] = true;}}}
                       }
                }
        }

    //for(int a=0; a<4; a++){for(int b=0;b<4;b++){System.out.println();for(int i=0; i<N; i++)
    //    {System.out.println();for(int j=0; j<N; j++){System.out.print(" |
"+coinc[i][a][j][b]);}}}}

    int[][] pathConn = new int[N][];

    Vector tempConn = new Vector();

    for(int i=0; i<N; i++)
        {
            for(int j=0; j<N; j++)
                {
                   for(int k=0; k<4; k++){for(int l=0;l<4;l++)
                       {
                           if(coinc[i][k][j][l])
                               {tempConn.add(new Integer(j));}
                       }}
                }

            tempConn.trimToSize();
            int iN = tempConn.size();
            pathConn[i] = new int[iN];   // writing Vector-elements to an Array
            for(int k=0; k<iN; k++){pathConn[i][k] = ((Integer)tempConn.get(k)).intValue();}
            tempConn.setSize(0);    // kill elements in Vector

            //System.out.println();for(int k=0;k<iN;k++)
                   {System.out.print(" |"+pathConn[i][k]+"|");}System.out.println();

        }

    this.connect = new int[N][];
    for(int i=0; i<N; i++)
        {int NI=pathConn[i].length; this.connect[i] = new int[NI];
        for(int j=0; j<NI; j++)
            {this.connect[i][j] = pathConn[i][j];}}

    fullyConnected = new boolean[N];
    for(int i=0;i<N;i++){fullyConnected[i] = true;}

    for(int i=0; i<N; i++)
        {int NN = this.polygons[i].corner.length;
        boolean sideConn[] = new boolean[NN];
        for(int m=0;m<NN;m++){sideConn[m] = false;}
        for(int k=0;k<NN;k++)
            {boolean oneSideConn = false;
            boolean oneSideMultConn = false;
            for(int j=0;j<N;j++){for(int l=0;l<4;l++)
                {if(coinc[i][k][j][l])
```

```
                        {if(oneSideConn){oneSideMultConn = true;}
                        else{oneSideConn = true;}}}}
                    if(oneSideConn){sideConn[k]=true;}
                    // please modify if you want to exclude muliple connections
                    }
                for(int m=0;m<NN;m++){if(!sideConn[m]){fullyConnected[i] = false;}}
                }

            this.closed = true;
            for(int i=0;i<N;i++){if(!fullyConnected[i]){this.closed = false;}}
        }

       // ************************** WriteRoutine *********************************

        public void WriteScr()
        {
            // Write Polygons
            int N = this.polygons.length;
            System.out.println("  Multiple Polygon: \n\t The multiple polygon has the following "
                               +N+" polygon-atoms: ");
            for(int i=0;i<N;i++)
                {int NN = this.polygons[i].corner.length;
                Coordinate koor = new Coordinate(this.polygons[i]);
                System.out.print("\t    Polygon "+i+":\t Origin: "); koor.WriteScr();
                System.out.print("\n\t\t\t Corners:");
                for(int j=0; j<NN; j++){this.polygons[i].corner[j].WriteScr();}
                System.out.println();}
            // Write Connections
            System.out.print("\t The polygon-connections are:
                               (each Polygon connects to the tabulated polygons) ");
            for(int i=0; i<N; i++)
                {System.out.print("\n\t\t Polygon - "+i+":    "); int NI = this.connect[i].length;
                for(int j=0; j<NI; j++)
                    {System.out.print(this.connect[i][j]); if(j<(NI-1)){System.out.print(" & ");}}
                    System.out.print("     -> fully connected: "+fullyConnected[i]);}
                    System.out.print("\n\t The polygons form a
                               closed body (all sides connect): "+this.closed+"\n\n");
        }


}
```

C.2.13 The MultiPolyhedron class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.geometry;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.math.*;
import java.text.*;

// ********************************************************************************************
// *********************************   Class Polyhedron ***************************************
// ********************************************************************************************

/**
   Class representing a collection of polyhedrons, originally created to reduce wedge and brick
polyhedrons to tetraedes.
   @author Manuel Alexandre Pouchon - &copy JNC Tokai Works (JAPAN)
   @version First created 2001.03.09 - Version 0
 */
public class MultiPolyhedron
{
    /**
       Collection of polyhedrons.
     */
    public Polyhedron[] polyhedrons;

    //************************************* consturctors ********************************

    public MultiPolyhedron()
    {
```

```
    }

    public MultiPolyhedron(Polyhedron ph1)
    {
        this.polyhedrons = new Polyhedron[1];
        this.polyhedrons[0] = new Polyhedron(ph1);
    }

    public MultiPolyhedron(Polyhedron ph1, Polyhedron ph2)
    {
        polyhedrons = new Polyhedron[2];
        polyhedrons[0] = new Polyhedron(ph1);
        polyhedrons[1] = new Polyhedron(ph2);
    }

    public MultiPolyhedron(Polyhedron ph1, Polyhedron ph2, Polyhedron ph3)
    {
        this.polyhedrons = new Polyhedron[3];
        this.polyhedrons[0] = new Polyhedron(ph1);
        this.polyhedrons[1] = new Polyhedron(ph2);
        this.polyhedrons[2] = new Polyhedron(ph3);
    }

    public MultiPolyhedron(Polyhedron ph1, Polyhedron ph2,
                           Polyhedron ph3, Polyhedron ph4)
    {
        this.polyhedrons = new Polyhedron[4];
        this.polyhedrons[0] = new Polyhedron(ph1);
        this.polyhedrons[1] = new Polyhedron(ph2);
        this.polyhedrons[2] = new Polyhedron(ph3);
        this.polyhedrons[3] = new Polyhedron(ph4);
    }

    public MultiPolyhedron(Polyhedron ph1, Polyhedron ph2,
                           Polyhedron ph3, Polyhedron ph4, Polyhedron ph5)
    {
        this.polyhedrons = new Polyhedron[5];
        this.polyhedrons[0] = new Polyhedron(ph1);
        this.polyhedrons[1] = new Polyhedron(ph2);
        this.polyhedrons[2] = new Polyhedron(ph3);
        this.polyhedrons[3] = new Polyhedron(ph4);
        this.polyhedrons[4] = new Polyhedron(ph5);
    }

    public MultiPolyhedron(Polyhedron ph1, Polyhedron ph2,
                           Polyhedron ph3, Polyhedron ph4, Polyhedron ph5, Polyhedron ph6)
    {
        this.polyhedrons = new Polyhedron[6];
        this.polyhedrons[0] = new Polyhedron(ph1);
        this.polyhedrons[1] = new Polyhedron(ph2);
        this.polyhedrons[2] = new Polyhedron(ph3);
        this.polyhedrons[3] = new Polyhedron(ph4);
        this.polyhedrons[4] = new Polyhedron(ph5);
        this.polyhedrons[5] = new Polyhedron(ph6);
    }

    public MultiPolyhedron(Polyhedron[] ph)
    {
        int N = ph.length;
        this.polyhedrons = new Polyhedron[N];
        for(int i=0;i<N;i++){this.polyhedrons[i]=new Polyhedron(ph[i]);}
    }

    public MultiPolyhedron(MultiPolyhedron mph, Polyhedron ph)
    {
        int N = mph.polyhedrons.length;
        this.polyhedrons = new Polyhedron[N+1];
        for(int i=0;i<N;i++){this.polyhedrons[i]=new Polyhedron(mph.polyhedrons[i]);}
        this.polyhedrons[N]=new Polyhedron(ph);
    }

    //*********************************** reduce topology ************************************

    /**
       Gives back a list of polyhedrons (tetraeder) from the reduction of a single polyhedron of any
defined type.
       @see jp.go.jnc.tokai.pouchon.geometry.Polyhedron#primList()
```

```
       @param toReduce Polyhedron from which the tetraeder list is derived.
     */
    public void reduceTopo(Polyhedron toReduce)
    {
        int prims[][]; prims = toReduce.primList();

        int N = prims.length;

        Polyhedron polys[] = new Polyhedron[N];
        for(int i=0;i<N;i++)
            {polys[i] =
                 new Polyhedron(toReduce,
                                toReduce.corner[prims[i][0]],
                                toReduce.corner[prims[i][1]],
                                toReduce.corner[prims[i][2]],
                                toReduce.corner[prims[i][3]]);}

        this.polyhedrons = new Polyhedron[N];
        for(int i=0;i<N;i++){this.polyhedrons[i]=new Polyhedron(polys[i]);}
    }

    // ************************************WriteScr*************************************

    public void WriteScr()
    {
        int N = this.polyhedrons.length;


        System.out.println(" Multiple polyhedron:");
        for(int i=0;i<N;i++)
            {
                System.out.println(" ======================== The polyhedron "+i+" is:");
                this.polyhedrons[i].WriteScr();
            }
    }

    /**
       Writes the multiple polyhedorn to a file with the format understood by the (external) Java-
Applet ThreeD.java
       @param fileName file name which should be used for the object to be stored.
       @param double a zooming factor to all coordinates
     */
    public void graphicFile(File fileName, double zoom)
    {
        try
            {
                PrintWriter out = new PrintWriter(new FileWriter(fileName));
                int count = 1;
                int N = this.polyhedrons.length;
                Vector coordinates = new Vector();
                Vector surfaces = new Vector();
                int lines[][][]= new int[N][][];
                // Print all coordinates
                for(int i=0;i<N;i++)
                    {
                        if(this.polyhedrons[i].topology==6)  // Write Tetraeder
                            {
                                for(int j=0;j<4;j++)
                                    {
                                        coordinates.add(this.polyhedrons[i].corner[j]);
                                    }
                                lines[i] = new int[6][];
                                lines[i][0] = new int[] {count,count+1};
                                lines[i][1] = new int[] {count,count+2};
                                lines[i][2] = new int[] {count,count+3};
                                lines[i][3] = new int[] {count+1,count+2};
                                lines[i][4] = new int[] {count+2,count+3};
                                lines[i][5] = new int[] {count+1,count+3};
                                count += 4;
                            }
                        if(this.polyhedrons[i].topology==7)   // Write Wedge
                            {
                                for(int j=0;j<6;j++)
                                    {
                                        coordinates.add(this.polyhedrons[i].corner[j]);
                                    }
                                lines[i] = new int[9][];
```

```
                        lines[i][0] = new int[] {count,count+1};   // bottom
                        lines[i][1] = new int[] {count+1,count+2};
                        lines[i][2] = new int[] {count+2,count};
                        lines[i][3] = new int[] {count+3,count+4}; // top
                        lines[i][4] = new int[] {count+4,count+5};
                        lines[i][5] = new int[] {count+5,count+3};
                        lines[i][6] = new int[] {count,count+3}; // sides
                        lines[i][7] = new int[] {count+1,count+4};
                        lines[i][8] = new int[] {count+2,count+5};
                        count += 6;
                    }
                if(this.polyhedrons[i].topology==8)   // Write Brick
                    {
                        for(int j=0;j<8;j++)
                            {
                                coordinates.add(this.polyhedrons[i].corner[j]);
                            }
                        lines[i] = new int[12][];
                        lines[i][0] = new int[] {count,count+1};   // bottom
                        lines[i][1] = new int[] {count+1,count+2};
                        lines[i][2] = new int[] {count+2,count+3};
                        lines[i][3] = new int[] {count+3,count};
                        lines[i][4] = new int[] {count+4,count+5}; // top
                        lines[i][5] = new int[] {count+5,count+6};
                        lines[i][6] = new int[] {count+6,count+7};
                        lines[i][7] = new int[] {count+7,count+4};
                        lines[i][8] = new int[] {count,count+4};   // sides
                        lines[i][9] = new int[] {count+1,count+5};
                        lines[i][10]= new int[] {count+2,count+6};
                        lines[i][11]= new int[] {count+3,count+7};
                        count += 8;
                    }
            }
        int M = coordinates.size();
        for(int i=0;i<M;i++)
            {
                Coordinate toWr = new Coordinate((Coordinate)coordinates.get(i));
                toWr.mult(zoom);
                String x0 = nonScNot(toWr.x[0]);
                String x1 = nonScNot(toWr.x[1]);
                String x2 = nonScNot(toWr.x[2]);
                out.print("v "+x0+" "+x1+" "+x2+"\n");
            }
        for(int i=0;i<N;i++)
            {
                int NN = lines[i].length;
                for(int j=0;j<NN;j++)
                    {
                        int NNN = lines[i][j].length;
                        out.print("l ");
                        for(int k=0;k<NNN;k++)
                            {
                                out.print(lines[i][j][k]+" ");
                            }
                        out.print("\n");
                    }
            }
        out.close();
    }
catch(IOException e)
    {
        System.out.print(" - Fehler: (beim Schreiben von " +
                fileName + ")--> " + e);
        System.exit(1);
    }
}

    //*********************************


    static String nonScNot(double number)
        {
            String ret;

            NumberFormat nf = NumberFormat.getNumberInstance();
            nf.setMaximumFractionDigits(20);
            nf.setMinimumFractionDigits(0);
```

```
            ret = nf.format(number);

            return(ret);
        }

    //************************************
}
```

C.3 The math package

C.3.1 The permutation class

```
/*
 * Copyright 2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.math;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;

/**
Class describing permutations, eg for initialisation of geometrical bodies, where an input order of
points may want to be changed because of possible problems.
 */
public class Permutation
{
    public int[] order;

    //********************************* CONSTRUKTOR ************************************

    /**
       Basic initialisation with a trivial order of entities. [0,1,2,3,4, &#8230 num]
       @param num Number of entities to initialise.
     */
    public Permutation(int num)
    {
        order = new int[num];
        for(int i=0; i<num; i++)
            {order[i]=i;}
    }

    /**
       Basic initialisation with a trivial order of entities. [0,1,2,3,4, &#8230 Num] where Num is
derived from the given Polygon num.
       @param num Polygon with which corner-number the permutation should be initialised.
     */
    public Permutation(Polygon num)
    {
        int NUM = num.corner.length;
        order = new int[NUM];
        for(int i=0; i<NUM; i++)
            {order[i]=i;}
    }

    /**
       Basic initialisation with a trivial order of entities. [0,1,2,3,4, &#8230 Num] where Num is
derived from the given Polyhedorn num.
       @param num Polyhedron with which corner-number the permutation should be initialised.
     */
    public Permutation(Polyhedron num)
    {
        int NUM = num.corner.length;
        order = new int[NUM];
        for(int i=0; i<NUM; i++)
            {order[i]=i;}
    }
```

```
    //********************************** TRANSPOSE ************************************

    /**
       Tranposes two entities in a permutation-vector.
       @param t1 first element to be transposed
       @param t2 second element to be transposed
       @return True if the two elements existed, false if one (or both) of the elements were ouside
the range.
     */
    public boolean transpose(int t1, int t2)
    {
       boolean ok;
       int N = this.order.length;
       if(t1 < N && t2 < N && t1 >= 0 && t2 >= 0)
           {
              int trs = this.order[t1];
              this.order[t1] = this.order[t2];
              this.order[t2] = trs;
              ok = true;
           }
       else{ok = false;}
       return ok;
    }

    // ********************************** pointOrdering routines *************************


    /**
       Takes a polygon and checks wether it is valid or not. If possible it makes permutations to
validate. The instance is then modified to represent this correction.
       @param toOrder Input polygon to be checked for validity
       @param out Set to true if permutation informations should be printed
       @return True if Polygon is valid, if some irrecoverable errors apeared, the routine retruns
false!
     */
    public boolean pointOrder(Polygon toOrder, boolean out)
    {
       boolean ok = true;

       int numCorner = toOrder.corner.length;

       if(numCorner == 3)       // permutation is never necessary !!!
           {
              if(toOrder.topology != 2)
                 {if(out){System.out.println(" Error -- wrong polygon-topology
                                                 detected (for triangle): "
                                          +toOrder.topology+" instead of 2! <- corrected");}
                  toOrder.topology =2;}
              Coordinate nor = new Coordinate();
              nor.cross(toOrder.corner[0],toOrder.corner[1],toOrder.corner[2]);
              if(nor.abs() == 0.0)
                  {System.out.println(" Error -- polygon (in toOrder case triangle)
                                          is degenerated");}
              // no other test is needed, triangle can't be screwed
           }

       if(numCorner == 4)
           {
              if(toOrder.topology != 4)
                 {if(out){System.out.println(" Error -- wrong polygon-topology detected
                                           (for triangle): "
                                          +toOrder.topology+" instead of 4! <- corrected");}
                  toOrder.topology = 4;}
              Coordinate nor1 = new Coordinate();
              nor1.cross(toOrder.corner[0],toOrder.corner[1],toOrder.corner[3]);
              Coordinate nor2 = new Coordinate();
              nor2.cross(toOrder.corner[1],toOrder.corner[2],toOrder.corner[0]);
              Coordinate nor3 = new Coordinate();
              nor3.cross(toOrder.corner[2],toOrder.corner[3],toOrder.corner[1]);
              Coordinate nor4 = new Coordinate();
              nor4.cross(toOrder.corner[3],toOrder.corner[0],toOrder.corner[2]);

              double c2 = Coordinate.dot(nor1,nor2);
              double c3 = Coordinate.dot(nor1,nor3);
              double c4 = Coordinate.dot(nor1,nor4);

              if(c2<0 || c3<0 || c4<0)
```

```
                {
                    if(c2*c3*c4 < 0)
                        {
                            System.out.println("  Warning -- Convex Quadrangle <-
                                                no correction possible");
                        }
                    else
                        {
                            if(out){System.out.println(" Error -- Quadrangle is screwed <-
                                        corrected");}
                            if(c2<0 && c3<0)
                                {ok=this.transpose(1,2);}
                            else{if(c3<0 && c4<0)
                                {ok=this.transpose(2,3);}
                            else{if(c2<0 && c4<0)
                                {ok=this.transpose(1,3);}
                            else{System.out.println(" ERROR , inpossible case in permutation
                                                routine");}}}
                        }
                }
            else
                {
                    // all same direction -> ok
                }
            //this.WriteScr();

        }
    return ok;
}




/**
       Reads in a polyhedron, if necessary corrects the topology (by number of corners) and controls
the order of corners and also corrects if necessary. Error messages for degenerated polyhedrons will
also be printed.
       @param out Set to <em>true</em> if correction information should be printend to screen,
<em>false</em> otherwise.
     */
    public boolean pointOrder(Polyhedron toOrder, boolean out)
    {

        boolean ok = true;
        //Polyhedron toOrderCopy = new Polyhedron(toOrder);

        int numCorners = toOrder.corner.length;

        if(numCorners == 4)
            {
                if(toOrder.topology != 6)
                    {if(out){System.out.println(" Error -- wrong polyhedron-topology detected
                                                (for tetraeder): "
                                        +toOrder.topology+" instead of 6! <- corrected");}
                    toOrder.topology =6;}

                double vol =
                        Coordinate.spat(toOrder.corner[0],toOrder.corner[1],
                                        toOrder.corner[2],toOrder.corner[3]);

                if(vol<0)
                    {if(out){System.out.println(" Error -- wrong polyhedron orientation detected
                                         (for tetraeder)"+
                                        "<- corrected");}
                    ok=this.transpose(1,2);}

                if(vol==0.0)
                    {System.out.println(" Error -- polyhedron (in this case tetraeder) is degenerated
                                                        (volume = 0)");}
            }

        if(numCorners == 6)
            {
                if(toOrder.topology != 7)
                    {if(out){System.out.println("\n\n Error -- wrong polyhedron-topology detected
                                                                    (for wedge): "
```

```
                                        +toOrder.topology+" instead of 7! <- corrected");}
            toOrder.topology =7;}

        double uab1 =
Coordinate.spat(toOrder.corner[0],toOrder.corner[1],toOrder.corner[2],toOrder.corner[3]),
            uab2 =
Coordinate.spat(toOrder.corner[0],toOrder.corner[1],toOrder.corner[2],toOrder.corner[4]),
            uab3 =
Coordinate.spat(toOrder.corner[0],toOrder.corner[1],toOrder.corner[2],toOrder.corner[5]),
            oab1 =
Coordinate.spat(toOrder.corner[3],toOrder.corner[5],toOrder.corner[4],toOrder.corner[0]),
            oab2 =
Coordinate.spat(toOrder.corner[3],toOrder.corner[5],toOrder.corner[4],toOrder.corner[1]),
            oab3 =
Coordinate.spat(toOrder.corner[3],toOrder.corner[5],toOrder.corner[4],toOrder.corner[2]);

        if(uab1==0.0 || uab2==0.0 || uab3==0.0 || oab1==0.0 || oab2==0.0 || oab3==0.0)
            {
                System.out.println(" Error -- polyhedron (in this case wedge) is
                                                degenerated");
            }
        else
            {
                {if(uab1<0 && uab2<0 && uab3<0)
                    {
                        if(out){System.out.println("\n\n Error -- wrong polyhedron
                                                orientation detected"+
                                            " (first three corners for wedge)
                                                <- corrected");}
                        ok=this.transpose(1,2);
                    }
                else
                    {
                        if(uab1>0 && uab2>0 && uab3>0)
                            {
                                // is ok!
                            }
                        else
                            {
                                System.out.println("\n\n Error -- polyhedron is screwed!!
                                                no correction possible!");
                            }
                    }
                }
                {if(oab1<0 && oab2<0 && oab3<0)
                    {
                        if(out){System.out.println("\n\n Error -- wrong polyhedron
                                                        orientation detected"+
                                            " (last three corners for wedge) <-
                                                        corrected");}
                        ok=this.transpose(4,5);
                    }
                else
                    {
                        if(oab1>0 && oab2>0 && oab3>0)
                            {
                                // is ok!
                            }
                        else
                            {
                                System.out.println(" Error -- polyhedron is screwed!!
                                                no correction possible!");
                            }
                    }
                }
            }

    }
if(numCorners == 8)
    {
        if(toOrder.topology != 8)
            {if(out){System.out.println(" Error -- wrong polyhedron-topology detected
                                            (for brick): "
                                        +toOrder.topology+" instead of 8! <- corrected");}
            toOrder.topology =8;}

        Polygon
```

```
                    base=
                        new Polygon(false,toOrder,toOrder.corner[0],toOrder.corner[1],
                                    toOrder.corner[2],toOrder.corner[3]),
                    top =
                        new Polygon(false,toOrder,toOrder.corner[4],toOrder.corner[5],
                                    toOrder.corner[6],toOrder.corner[7]);

                //System.out.println("\n\n Boden: ");base.WriteScr();
                //System.out.println("\n\n Deckel: ");top.WriteScr();

                boolean blind;
                Permutation baseOrd = new Permutation(base), topOrd = new Permutation(top);
                blind = baseOrd.pointOrder(base,false); blind = base.permute(baseOrd);
                blind = topOrd.pointOrder(top,false); blind = top.permute(topOrd);

                if(this.order.length == 8 && baseOrd.order.length == 4 && topOrd.order.length == 4)
                    {for(int i=0; i<4; i++)
                        {this.order[i]=baseOrd.order[i];this.order[i+4]=topOrd.order[i]+4;}}

                //this.WriteScr();

                //for(int i=0;i<4;i++){toOrder.corner[i] = base.corner[i];
                                    toOrder.corner[i+4] = top.corner[i];}
                //  <---- Correction of possible screwed base and top definitions

                boolean btt = base.above(top,true);
                boolean btf = base.above(top,false);
                boolean tbt = top.above(base,true);

                if(!btt && !btf)
                    {System.out.println(" ERROR - Brick is screwed, no correction possible!");}
                else
                    {
                        if(!btt)     // permutation of corner 1 & 3
                            {
                                ok=this.transpose(1,3);
                                        //System.out.println(" 1-3 Permutation");
                            }
                        if(tbt)     // permutation of corner 5 & 7
                            {
                                ok=this.transpose(5,7); //System.out.println(" 5-7 Permutation");

                            }
                    }

            }
        //this.WriteScr();
        return ok;
    }

    // ********************************* WRITE SCR *********************************

    public void WriteScr()
    {
        int N = this.order.length;
        System.out.print(" | ");
        for(int i=0;i<N;i++)
            {
                System.out.print(this.order[i]+" | ");
            }
    }

}
```

C.4 The finitele package

C.4.1 The element class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;

// ************************************************************************************
// *********************************   Class Element    *********************************
// ************************************************************************************

public class Element
{
    /**
       id of the element
     */
    public long id;
    /**
       Topology of the element, same topology-notation like for the geometry-class Polyhedron!
       @see jp.go.jnc.tokai.pouchon.geometry.Polyhedron
     */
    public int topology; // 6: Tetra4, 7: Wedge6, 8: Brick8, 10: Tetra10, 11: Wedge15, 12: Brick20
    public long[] nodeId;

    public Element()
    {
        id = 0;
        nodeId = new long[20];
        for(int i=0; i<20; i++){nodeId[i]=0;}
        topology = 0;
    }

    /**
       Constructor for <b>Tetraeder</b> with 4 nodes.
       @param idIn identity of this tetraeder
       @param node<sub>i=0&#8230;3</sub> nodes needed for initialisation of tetraeder.
     */
    public Element(long idIn, Node node0, Node node1, Node node2,
                   Node node3) // read in tetragonal element with 4 nodes
    {
        this.id = idIn;   // assigning id
        this.topology = 6;

        this.nodeId = new long[20];
        for(int i=0; i<20; i++){this.nodeId[i]=0;}
        Coordinate zero = new Coordinate(0.0,0.0,0.0);
        Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3);

        Entities ids = new Entities(node0.id, node1.id, node2.id, node3.id);

        Permutation perm = new Permutation(tetra);
        boolean ok = perm.pointOrder(tetra,false);
        ids.permute(perm);

        for(int i=0; i<3; i++){this.nodeId[i] = ids.id[i];}
        this.nodeId[4] = ids.id[3];
    }

    /**
       Constructor for <b>Wedge</b> with 6 nodes.
       @param idIn identity of this wedge
       @param node<sub>i=0&#8230;5</sub> nodes needed for initialisation of wedge.
     */
    public Element(long idIn, Node node0, Node node1, Node node2,
                   Node node3, Node node4, Node node5)
                        // read in Wedge element with 6 nodes
    {
```

```
        this.id = idIn;   // assigning id
        this.topology = 7;

        this.nodeId = new long[20];
        for(int i=0; i<20; i++){this.nodeId[i]=0;}
        Coordinate zero = new Coordinate(0.0,0.0,0.0);
        Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3,node4,node5);

        Entities ids = new Entities(node0.id,node1.id,node2.id,node3.id,node4.id,node5.id);

        Permutation perm = new Permutation(tetra);
        boolean ok = perm.pointOrder(tetra,false);
        ids.permute(perm);

        for(int i=0; i<3; i++){nodeId[i] = ids.id[i]; nodeId[i+4] = ids.id[i+3];}
    }

    /**
       Constructor for <b>Brick</b> with 8 nodes.
       @param idIn identity of this Brick
       @param node<sub>i=0&#8230;7</sub> nodes needed for initialisation of brick.
     */
    public Element(long idIn, Node node0, Node node1, Node node2, Node node3,
                   Node node4, Node node5, Node node6, Node node7)
                // read in Brick element with 8 nodes
    {
        this.id = idIn;   // assigning id
        this.topology = 8;

        this.nodeId = new long[20];
        for(int i=0; i<20; i++){this.nodeId[i]=0;}
        Coordinate zero = new Coordinate(0.0,0.0,0.0);
        Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3,node4,node5,node6,node7);

        Entities ids =
                new Entities(node0.id,node1.id,node2.id,node3.id,node4.id,node5.id,node6.id,node7.id);

        Permutation perm = new Permutation(tetra);
        boolean ok = perm.pointOrder(tetra,false);
        ids.permute(perm);

        for(int i=0; i<8; i++){this.nodeId[i] = ids.id[i];}
    }

    /**
       Copy constructor.
       @param toCopy Element being copied to new Element.
     */
    public Element(Element toCopy)
    {
        if(toCopy.topology==6 || toCopy.topology==7 || toCopy.topology==8)
            {
                this.nodeId = new long[20];
                for(int i=0; i<20; i++){this.nodeId[i]=toCopy.nodeId[i];}
                this.topology = toCopy.topology;
                this.id = toCopy.id;
            }
        else
            {System.out.println("*****  !!!!!!!!!!!! NOT IMPLEMENTED !!!!!!!!!!!!! ********");}
    }




    /**
       Assign operator for <b>Tetraeder</b> with 4 nodes.
       @param idIn identity of this tetraeder
       @param node<sub>i=0&#8230;3</sub> nodes needed for initialisation of tetraeder.
     */
    public void assign(long idIn, Node node0, Node node1, Node node2, Node node3)
                // read in tetragonal element with 4 nodes
    {
        this.id = idIn;   // assigning id
        this.topology = 6;
```

```
    for(int i=0; i<20; i++){this.nodeId[i]=0;}
    Coordinate zero = new Coordinate(0.0,0.0,0.0);
    Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3);

    Entities ids = new Entities(node0.id, node1.id, node2.id, node3.id);

    Permutation perm = new Permutation(tetra);
    boolean ok = perm.pointOrder(tetra,false);
    ids.permute(perm);

    for(int i=0; i<3; i++){this.nodeId[i] = ids.id[i];}
    this.nodeId[4] = ids.id[3];
}


/**
   Assign operator for <b>Wedge</b> with 6 nodes.
   @param idIn identity of this wedge
   @param node<sub>i=0&#8230;5</sub> nodes needed for initialisation of wedge.
 */
public void assign(long idIn, Node node0, Node node1, Node node2,
                   Node node3, Node node4, Node node5) // read in Wedge element with 6 nodes
{
    this.id = idIn;   // assigning id
    this.topology = 7;

    for(int i=0; i<20; i++){this.nodeId[i]=0;}
    Coordinate zero = new Coordinate(0.0,0.0,0.0);
    Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3,node4,node5);

    Entities ids = new Entities(node0.id,node1.id,node2.id,node3.id,node3.id,node4.id,node5.id);

    Permutation perm = new Permutation(tetra);
    boolean ok = perm.pointOrder(tetra,false);
    ids.permute(perm);

    for(int i=0; i<3; i++){this.nodeId[i] = ids.id[i]; nodeId[i+4] = ids.id[i+3];}
}

/**
   Assign operator for <b>Brick</b> with 8 nodes.
   @param idIn identity of this Brick
   @param node<sub>i=0&#8230;7</sub> nodes needed for initialisation of brick.
 */
public void assign(long idIn, Node node0, Node node1, Node node2, Node node3,
                   Node node4, Node node5, Node node6, Node node7)
            // read in Brick element with 8 nodes
{
    this.id = idIn;   // assigning id
    this.topology = 8;

    for(int i=0; i<20; i++){this.nodeId[i]=0;}
    Coordinate zero = new Coordinate(0.0,0.0,0.0);
    Polyhedron tetra = new Polyhedron(zero,node0,node1,node2,node3,node4,node5,node6,node7);

    Entities ids =
     new Entities(node0.id,node1.id,node2.id,node3.id,node4.id,node5.id,node6.id,node7.id);

    Permutation perm = new Permutation(tetra);
    boolean ok = perm.pointOrder(tetra,false);
    ids.permute(perm);

    for(int i=0; i<8; i++){this.nodeId[i] = ids.id[i];}
}

/**
   Assigns a copy to the existing element.
   @param toCopy Element being copied to element.
 */
public void assign(Element toCopy)
{
    this.id = toCopy.id;
    for(int i=0; i<20; i++){this.nodeId[i]=toCopy.nodeId[i];}
    this.topology = toCopy.topology;
}
```

```
    /**
       Procedure to find out the coordinates of an element. The procedure seeks all the nodes given in
a list if they are contained in the element and returns the corresponting list of coordinates. The
order of the returned coordinates corresponds to the id order.
       @param listOfNodes List of nodes where the element should be contained.
       @return list of Coordinates of the element-nodes.
     */
    public Coordinate[] coordinates(Node[] listOfNodes)
    {
        // where do the corner ID's sit in the list? This depends on the topology, of course!
        int places[];
        if(this.topology==6){places=new int[4];places[0]=0;places[1]=1;places[2]=2;places[3]=4;}
        else{if(this.topology==7){places=
                new int[6];places[0]=0;places[1]=1;places[2]=2;places[3]=4;places[4]=5;places[5]=6;}
        else{if(this.topology==8)
            {places=new int[8];places[0]=0;places[1]=1;places[2]=2;places[3]=3;
            places[4]=4;places[5]=5;places[6]=6;places[7]=7;}
        else{places=new int[0];}}}  // not identified case

        //System.out.println("\n\t The topology is " + topology);

        int numOfNodes = listOfNodes.length;
        int nodesInEle = places.length;
        long idsInEle[] = new long[nodesInEle];
        for(int i=0;i<nodesInEle;i++){idsInEle[i]=this.nodeId[places[i]];}

        boolean isFound[] = new boolean[nodesInEle];
        for(int i=0;i<nodesInEle;i++){isFound[i]=false;}
        boolean allFound = true;

        long idList[][] = new long[nodesInEle][2];
                        // First Column for FemapIds and second for local numbering
        Coordinate koorList[] = new Coordinate[nodesInEle];
        for(int i=0; i<nodesInEle; i++){koorList[i] = new Coordinate();}
        for(int i=0; i<nodesInEle; i++){idList[i][0] = idsInEle[i]; idList[i][1] = i;}
        for(int i=0; i<numOfNodes; i++)
            {
                long idToComp = listOfNodes[i].id;
                for(int j=0; j<nodesInEle; j++)
                    {
                        if(idList[j][0] == idToComp)   // Record found <- same id's
                            {
                                isFound[j] = true;

                                koorList[(int)idList[j][1]].assign(listOfNodes[i]);
                                // assigning Coordinate
                                for(int k=j; k<nodesInEle-1; k++)
                                // at location given by local numbering
                                    {
                                        idList[k][0]=idList[k+1][0];
                                        // Found record is not compared anymore
                                        idList[k][1]=idList[k+1][1];
                                        // -> Upper records are copied downwards
                                    }
                                idList[nodesInEle-1][0]=-1; // Not anymore used record
                                idList[nodesInEle-1][1]=-1;
                                nodesInEle--;  // one record less to compare in future
                                break;   // leave this for statement for increase in listOfNodes
                            }
                        else{}
                    }
            }
        for(int i=0; i<nodesInEle; i++){if(!isFound[i]){allFound = false;}}
        if(!allFound)
        {System.out.println("\n\n\t ERROR - corresponding Coordinate not found for Node!!!!");}

        return koorList;
    }


    /**
       Procedure to find out the nodes of an element. The procedure seeks all the nodes given in a
list if they are contained in the element and returns the corresponting list of nodes. The order of
the returned nodes corresponds to the id order.
       @param listOfNodes List of nodes where the element should be contained.
       @return list of Nodes contained in Element.
     */
```

```
public Node[] nodes(Node[] listOfNodes)
{
    // where do the corner ID's sit in the list? This depends on the topology, of course!
    int places[];
    if(this.topology==6){places=new int[]{0,1,2,4};}
    else{if(this.topology==7){places=new int[]{0,1,2,4,5,6};}
    else{if(this.topology==8){places=new int[]{0,1,2,3,4,5,6,7};}
    else{places=new int[0];}}}  // not identified case

    //System.out.println("\n\t The topology is " + topology);

    int numOfNodes = listOfNodes.length;
    int nodesInEle = places.length;
    long idsInEle[] = new long[nodesInEle];
    for(int i=0;i<nodesInEle;i++){idsInEle[i]=this.nodeId[places[i]];}

    boolean isFound[] = new boolean[nodesInEle];
    for(int i=0;i<nodesInEle;i++){isFound[i]=false;}
    boolean allFound = true;

    long idList[][] = new long[nodesInEle][2];
           // First Column for FemapIds and second for local numbering
    Node koorList[] = new Node[nodesInEle];
    for(int i=0; i<nodesInEle; i++){koorList[i] = new Node();}
    for(int i=0; i<nodesInEle; i++){idList[i][0] = idsInEle[i]; idList[i][1] = i;}
    for(int i=0; i<numOfNodes; i++)
        {
            long idToComp = listOfNodes[i].id;
            for(int j=0; j<nodesInEle; j++)
                {
                    if(idList[j][0] == idToComp)   // Record found <- same id's
                        {
                            isFound[j] = true;

                            koorList[(int)idList[j][1]].assign(listOfNodes[i]);
                            // assigning Node
                            for(int k=j; k<nodesInEle-1; k++)
                            // at location given by local numbering
                                {
                                    idList[k][0]=idList[k+1][0];
                                    // Found record is not compared anymore
                                    idList[k][1]=idList[k+1][1];
                                    // -> Upper records are copied downwards
                                }
                            idList[nodesInEle-1][0]=-1; // Not anymore used record
                            idList[nodesInEle-1][1]=-1;
                            nodesInEle--;  // one record less to compare in future
                            break;   // leave this for statement for increase in listOfNodes
                        }
                    else{}
                }
        }

    for(int i=0; i<nodesInEle; i++){if(!isFound[i]){allFound = false;}}
    if(!allFound)
    {System.out.println("\n\n\t ERROR - corresponding Coordinate not found for Node!!!!");}

    return koorList;
}

/**
   modifies the object instance to the modification
   @param modification the modification to be applied,
    negative node index means the cration of a new node!
   @param ids Object counter
   @return nodes with new ids, the old ids are set to 0!
 */
public Node[] modify(ElementCoordinate modification, ObjectCounter ids)
{
    long modIds[] = modification.ids();   // identities of modification

    int N = modIds.length;
    System.out.println("/n/n/t There are "+N+" Nodes in this Element!");

    Node res[] = new Node[N];

    int objId = ids.getObjPlace(this.nodeId[0]);
```

```
        this.assign(modification);    // assigning topology

        for(int i=0;i<N;i++)
            {
               if(modIds[i]<0)  // new node to initialise
                   {
                       long newId = ids.newElementInObj(objId);
                       System.out.print("\n "+i);modification.coords[i].WriteScr();
                       res[i] = new Node(newId,modification.coords[i]);
                   }
               else
                   {
                       res[i] = new Node(-modIds[i],modification.coords[i]);
                   }
            }

        for(int i=0;i<N;i++)    // apply abs value of ids to element
            {
               this.nodeId[i] = Math.abs(res[i].id);
            }

        for(int i=0;i<N;i++)   // set negative ids to 0
            {
               res[i].id = (res[i].id+Math.abs(res[i].id))/2;
            }
        return res;
    }

 /**
       modifies the object instance to the modification
       @param modification the modification to be applied,
       negative node index means the cration of a new node!
 */
    public void modify(ElementCoordinate modification)
    {
        long oldNewId = this.id;
        Element toAss = new Element(modification);
        this.assign(toAss);
        this.id = oldNewId;
    }

    /**
       Gives back the ids contained in an Element, the ids are normally contained in a list of 20
long-numbers. But only 4, 6 or 8 of these are actually used. The information is returned depending on
the topology.
       @return List of used ids
     */
    public long[] ids()
    {
        long ret[];

        if(this.topology==6)
            {ret=new long[]{this.nodeId[0],this.nodeId[1],this.nodeId[2],this.nodeId[4]};}
        else{if(this.topology==7)
            {ret=
               new long[]{this.nodeId[0],this.nodeId[1],this.nodeId[2],
               this.nodeId[4],this.nodeId[5],this.nodeId[6]};}
        else{if(this.topology==8)
            {ret=new long[]{this.nodeId[0],this.nodeId[1],this.nodeId[2],this.nodeId[3],
                     this.nodeId[4],this.nodeId[5],this.nodeId[6],this.nodeId[7]};}
        else{ret = new long[0]; System.out.println("\n  unknown topology "+ this.topology);}}}

        return ret;
    }

    /**
       Puts a new Id in the element. Because each element can contain 20 ids, this procedure is needed
to put the new Id at the right place.
       @param newId The new id to be set in the existing element
       @param numNewId The number of the new id.
     */
    public void putId(long newId,int numNewId)
    {
        int placeToPut;

        if(this.topology==6)
```

```
            {if(numNewId<3){placeToPut = numNewId;}else{placeToPut = numNewId+1;}}
        else{if(this.topology==7)
            {if(numNewId<3){placeToPut = numNewId;}else{placeToPut = numNewId+1;}}
        else{if(this.topology==8)
            {placeToPut = numNewId;}
        else{placeToPut = 0; System.out.println("\n  unknown topology "+ this.topology);}}}

        this.nodeId[placeToPut] = newId;
    }


    /**
       Writes an element to the screen.
     */
    public void WriteScr()
    {
        System.out.println();
        System.out.println(" - The element with Id " + id + " contains the nodes with the Ids: " );
        for(int i=0; i<20; i++)
            {
                System.out.print(" * " + nodeId[i]);
            }
        System.out.println();
    }


}
```

C.4.2 The ElementList class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;
import jp.go.jnc.tokai.pouchon.spherearr.*;


// ********************************************************************************************
// *********************************   Class ElementList  *********************************
// ********************************************************************************************

public class ElementList
{
    public Vector elements = new Vector();



    public void GetElements(SphereMesh newElements)
    {
        for(int i=0; i<newElements.elements.length; i++)
            {
                if(newElements.elements[i].id > -1)
                    {
                        elements.add(newElements.elements[i]);
                    }
            }
    }

    public void WriteFile(PrintWriter os) throws IOException
    {
        Node nullNode = new Node();
        Element auszugeben = new Element(0,nullNode,nullNode,nullNode,nullNode);
        int l = elements.size();
        for(int i=0; i<l; i++)
            {
                auszugeben.assign((Element)elements.get(i));
                os.print(auszugeben.id);
                os.print(", 124, 1, 25,");
                 os.print(auszugeben.topology);
                os.println(", 1, 0, 0, 0, 0, 0, 0,");
```

```
                for(int j=0; j<2; j++)
                    {
                        for(int k=0; k<10; k++)
                            {
                                os.print(auszugeben.nodeId[(j*10)+k]+", ");
                            }
                        os.println();
                    }
                for(int j=0;j<3;j++){for (int k=0; k<3; k++){os.print("0., ");}os.println();}
                for(int j=0;j<16;j++){os.print("0, ");} os.println();
            }
    }

}
```

C.4.3 The node class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;

public class Node extends Coordinate
{

    public long id;
    public Vector inEle = new Vector(3);

    public Node(Node toCopy)
    {
        super(toCopy);
        id = toCopy.id;
    }

    public Node(long idIn, Coordinate loc)
    {
        Coordinate A = new Coordinate(loc);
        for(int i=0;i<3;i++){x[i]=A.x[i];}
        id = idIn;
    }

    public Node()
    {
        new Node(0,0.0,0.0,0.0);
    }


    public Node(long idIn, double x0, double x1, double x2)
    {
        x[0]=x0; x[1]=x1; x[2]=x2;
        id = idIn;
    }

    public void assign(long idIn, Coordinate loc)
    {
        assign(loc);
        id = idIn;
    }

    public void assign(long idIn, double x0, double x1, double x2)
    {
        assign(x0,x1,x2);
        id = idIn;
    }

    public void assign(Node nToAss)
    {
        assign(nToAss.x[0],nToAss.x[1],nToAss.x[2]);
        id = nToAss.id;
    }
```

```
    public void WriteScr()
    {
        Coordinate wr = new Coordinate(this);
        System.out.print("\n - Node Nr. "+ id +" has the coordinate: ");
        wr.WriteScr();
    }

}
```

C.4.4 The NodeList class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;
import jp.go.jnc.tokai.pouchon.spherearr.*;



public class NodeList
{
    public Vector nodes = new Vector();



    public void GetNodes(SphereMesh newNodes)
    {
        for(int i=0; i<newNodes.nodes.length; i++)
            {
                nodes.add(newNodes.nodes[i]);
            }
    }

    public void WriteFile(PrintWriter os) throws IOException
    {
        Node auszugeben = new Node();
        int k = nodes.size();
        for(int i=0; i<k; i++)
            {
                auszugeben.assign((Node)nodes.get(i));
                os.print(auszugeben.id+", ");
                os.print("0, 0, 1, 46, 0, 0, 0, 0, 0, 0, "); // set to default values, 1: Layer, 46:
color
                for(int j=0; j<3; j++)
                    {
                        os.print(auszugeben.x[j]+", ");
                    }
                os.println();
            }
    }


}
```

C.4.5 The Entities class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;

/**
```

```
General Class for representing a number of entities, for example the node-entities of a emlement, or
an element collection!
 */
public class Entities
{
    /**
       List of entities.
     */
    public long id[];




    /** Constructor for 2 entities
        @param e<sub>i=0&#8230;1</sub> list of entities*/
    public Entities(long e0, long e1)
    {
        id = new long[2];
        id[0] = e0; id[1] = e1;
    }

    /** Constructor for 3 entities
        @param e<sub>i=0&#8230;2</sub> list of entities*/
    public Entities(long e0, long e1, long e2)
    {
        id = new long[3];
        id[0] = e0; id[1] = e1; id[2] = e2;
    }

    /** Constructor for 4 entities
        @param e<sub>i=0&#8230;3</sub> list of entities*/
    public Entities(long e0, long e1, long e2, long e3)
    {
        id = new long[4];
        id[0] = e0; id[1] = e1; id[2] = e2; id[3] = e3;
    }

    /** Constructor for 5 entities
        @param e<sub>i=0&#8230;4</sub> list of entities*/
    public Entities(long e0, long e1, long e2, long e3, long e4)
    {
        id = new long[5];
        id[0] = e0; id[1] = e1; id[2] = e2; id[3] = e3; id[4] = e4;
    }

    /** Constructor for 6 entities
        @param e<sub>i=0&#8230;5</sub> list of entities*/
    public Entities(long e0, long e1, long e2, long e3, long e4, long e5)
    {
        id = new long[6];
        id[0] = e0; id[1] = e1; id[2] = e2; id[3] = e3; id[4] = e4; id[5] = e5;
    }

    /** Constructor for 7 entities
        @param e<sub>i=0&#8230;6</sub> list of entities*/
    public Entities(long e0, long e1, long e2, long e3, long e4, long e5, long e6)
    {
        id = new long[7];
        id[0] = e0; id[1] = e1; id[2] = e2; id[3] = e3; id[4] = e4; id[5] = e5; id[6] = e6;
    }

    /** Constructor for 8 entities
        @param e<sub>i=0&#8230;7</sub> list of entities*/
    public Entities(long e0, long e1, long e2, long e3, long e4, long e5, long e6, long e7)
    {
        id = new long[8];
        id[0] = e0; id[1] = e1; id[2] = e2; id[3] = e3; id[4] = e4; id[5] = e5; id[6] = e6;
        id[7] = e7;
    }

    /** Constructor for n entities
        @param e[] list of entities */
    public Entities(long e[])
    {
        int n = e.length;
```

```
        id = new long[n];
        for(int i=0;i<n;i++){id[i]=e[i];}
    }


    /**
       Permutates any entity-list with the given permutation instance, toApply.
       @param toApply Permutation instance to be applied to the entitiy-list.
       @return True if permutation was succesfull. False if the length of <em>toApply</em> does not
correspond to the number of entities.
     */
    public boolean permute(Permutation toApply)
    {
        boolean ok = true;

        int N = this.id.length;

        if(N == toApply.order.length)
            {
                int transNum = 0;
                for(int i=0; i<N; i++)
                    {if(toApply.order[i]!=i){transNum++;}}
                if(transNum>0)
                    {
                        int perms[] = new int[transNum];
                        long permsCont[] = new long[transNum];
                        {
                            int M=0;
                            for(int i=0; i<N; i++)
                                {if(toApply.order[i]!=i)
                                    {perms[M]=i;permsCont[M]=this.id[toApply.order[i]];M++;}}
                        }
                        for(int i=0;i<transNum;i++)
                            {this.id[perms[i]]=permsCont[i];}
                    }
                ok = true;
            }
        else{ok = false;}

        return ok;
    }

}
```

C.4.6 The ElementCoordinateClass

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;

// ********************************************************************************************
// *********************************   Class ElementCoordinate ********************************
// ********************************************************************************************

/**
   Additional Class to <b>Element</b>. The class <b>Element</b> only contains the Node-Number
information, <b>id</b>. This class completes this information with the corresponding coordinates.
*/
public class ElementCoordinate extends Element
{
    public Coordinate[] coords;

    /**
       Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes,
        contained in the Element, are specified.
       @param toComplete the Element information which is going to be completed with the
        corresponding coordinates.
       @param toLookIn the list of nodes, where the information (here, the coordinates) about the
        single nodes is seeked.
```

```
    public ElementCoordinate(Element toComplete, Node[] toLookIn)
    {
        Coordinate coordinates[];
        coordinates = toComplete.nodes(toLookIn);
    }
    */

    /**
       Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes,
        contained in the Element, are specified.
       @param toComplete the Element information which is going to be completed with the
        corresponding coordinates.
       @param toLookIn the list of nodes, where the information (here, the coordinates) about the
        single nodes is seeked.
     */
    public ElementCoordinate(Element toComplete, NodeList toLookIn)
    {
        int N = toLookIn.nodes.size();
        Node nodeList[] = new Node[N];
        for(int i=0;i<N;i++)
            {nodeList[i] = new Node((Node)toLookIn.nodes.get(i));}
        Coordinate coordinates[];
        coordinates = toComplete.nodes(nodeList);
    }

/**
       Constructor of ElementCoordinate, it takes an Element and a NodeList where the nodes,
        contained in the Element, are specified.
       @param toComplete the Element information which is going to be completed with the
        corresponding coordinates.
       @param nodeList the list of nodes, where the information (here, the coordinates) about the
        single nodes is seeked.
     */
    public ElementCoordinate(Element toComplete, Node[] nodeList)
    {
        super(toComplete);

        Coordinate coordinates[];
        coordinates = toComplete.nodes(nodeList);

        int N = coordinates.length;

        this.coords = new Coordinate[N];

        for(int i=0;i<N;i++)
            {
                this.coords[i] = new Coordinate(coordinates[i]);
            }
    }

    /**
       Constructor of ElementCoordinate, it takes an a NodeList representing the element in the
        correct order.
       @param node the list of nodes, where the information (here, the coordinates) about the single
        nodes is seeked.
     */
    public ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4)
    {
        super(id, node1,node2,node3,node4);
        this.coords = new Coordinate[4];
        this.coords[0] = new Coordinate(node1);
        this.coords[1] = new Coordinate(node2);
        this.coords[2] = new Coordinate(node3);
        this.coords[3] = new Coordinate(node4);
    }

    /**
       Constructor of ElementCoordinate, it takes an a NodeList representing the element in the
        correct order.
       @param node the list of nodes, where the information (here, the coordinates) about the single
        nodes is seeked.
     */
    public ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4,
                             Node node5, Node node6)
    {
        super(id, node1,node2,node3,node4,node5,node6);
```

```
        this.coords = new Coordinate[6];
        this.coords[0] = new Coordinate(node1);
        this.coords[1] = new Coordinate(node2);
        this.coords[2] = new Coordinate(node3);
        this.coords[3] = new Coordinate(node4);
        this.coords[4] = new Coordinate(node5);
        this.coords[5] = new Coordinate(node6);
    }

/**
       Constructor of ElementCoordinate, it takes an a NodeList representing the element in the
correct order.
       @param node the list of nodes, where the information (here, the coordinates) about the single
nodes is seeked.
     */
    public ElementCoordinate(long id, Node node1, Node node2, Node node3, Node node4,
                             Node node5, Node node6, Node node7, Node node8)
    {
        super(id, node1,node2,node3,node4,node5,node6,node7,node8);
        this.coords = new Coordinate[8];
        this.coords[0] = new Coordinate(node1);
        this.coords[1] = new Coordinate(node2);
        this.coords[2] = new Coordinate(node3);
        this.coords[3] = new Coordinate(node4);
        this.coords[4] = new Coordinate(node5);
        this.coords[5] = new Coordinate(node6);
        this.coords[6] = new Coordinate(node7);
        this.coords[7] = new Coordinate(node8);
    }



    /*
    public ElementCoordiante(Polyhedron toComplete, int nodeSart)
    {
        this.topology = toComplete.topology;
        this.coords = new Coordinate
    }
    */

    /**
       Takes an element with the coordinate-information and froms the geometrical object Polyhedron.
       @return polyhedron from the infomation on the element
     */
    public Polyhedron polyhedron()
    {
        int N = this.coords.length;

        Polyhedron res;
        Coordinate zero = new Coordinate(0.0,0.0,0.0);

        if(this.topology == 6 && N == 4) // tetraeder
            {res = new Polyhedron(zero,this.coords[0],this.coords[1],this.coords[2],this.coords[3]);}
        else{if(this.topology == 7 && N == 6) // Wedge
            {res = new Polyhedron(zero,this.coords[0],this.coords[1],this.coords[2],
                                  this.coords[3],this.coords[4],this.coords[5]);}
        else{if(this.topology == 8 && N == 8) // Brick
            {res = new Polyhedron(zero,this.coords[0],this.coords[1],this.coords[2],this.coords[3],
                                  this.coords[4],this.coords[5],this.coords[6],this.coords[7]);}
        else
            {res = new Polyhedron(zero,zero,zero,zero,zero);
            System.out.println(" ERROR in polyhedron generator of ElementCoordinate,
                                     topology wrong!");}}}
        return res;

    }

    public Node[] nodes()
    {
        // where do the corner ID's sit in the list? This depends on the topology, of course!
        int places[];
        if(this.topology==6){places=new int[]{0,1,2,4};}
        else{if(this.topology==7){places=new int[]{0,1,2,4,5,6};}
        else{if(this.topology==8){places=new int[]{0,1,2,3,4,5,6,7};}
        else{places=new int[0];}}}  // not identified case

        int nodesInEle = places.length;
```

```
        Node nodesBack[] = new Node[nodesInEle];
        for(int i=0;i<nodesInEle;i++){nodesBack[i]=new Node(this.nodeId[places[i]],this.coords[i]);}

        return nodesBack;
    }

    /**
       Routine to cut an Element with a polyhedron. It uses a cutting procedure of the Polyhedron-
class. New indizes are marked with a negative index, for recongnition! Starting from 1. These are not
the definitive indizes, but have to be initialised later.
       @see jp.go.jnc.tokai.pouchon.geometry.Polyhedron#cut(Polyhedron)
       @param cutting cell which defines the cutting limits.
       @return list of elements with the coordinates, representing the cut-volume. This might contain
new elements and nodes. These must be written to the general element- and node-list!
     */
    public ElementCoordinate[] adjust(Polyhedron cutting)
    {

        Polyhedron ph;
        ph = this.polyhedron();


        MultiPolyhedron phs;
        phs = ph.cut(cutting);

        int N = phs.polyhedrons.length;    // Definition of total amount of polyhedrons after cutting
        //System.out.println(" Adjusting procedure for elementCoordinate started! "
                             +N+" Polyhedrons result");

        ElementCoordinate res[] = new ElementCoordinate[N];

        Node nodesBefore[] = this.nodes();

        Vector newCoords = new Vector();

        Node nodesAfter[][] = new Node[N][];               // [polyhedrons][corners]

        for(int i=0;i<N;i++)                               // counter over all polyhedrons
            {
                //phs.polyhedrons[i].WriteScr();

                int M = phs.polyhedrons[i].corner.length;
                nodesAfter[i] = new Node[M];
                for(int j=0;j<M;j++)
                        // counter over the corners of single polyhedrons
                    {Coordinate comp = phs.polyhedrons[i].corner[j];
                    int O = nodesBefore.length;
                    boolean hasEqual = false;
                    for(int k=0;k<O;k++)
                        // counter over old corners of uncuten polyhedron
                        {                   // -> did the node already exist before?
                            if(nodesBefore[k].equal(comp))
                                {
                //System.out.println(" Node has already equivalent at "+k+" at "+i+" "+j);
                                    nodesAfter[i][j] = new Node(nodesBefore[k].id,comp);
                                    hasEqual = true;      // node did already exist

                                }
                            else{}
                        }
                    if(!hasEqual)                          // Node didn't exist bafore
                        {
                            boolean already = false;
                            int newInd = 0;

                            int P = newCoords.size();   // Coordiante already contained in new list?
                            for(int l=0;l<P;l++)
                                {
                                    Coordinate comp1 = new Coordinate((Coordinate)newCoords.get(l));
                                    if(comp1.equal(comp))
                                        {
                                            newInd = -(l+1);   // take the old index
                                            already = true;
                                        }
                                }
                            if(!already)               // not contained !
                                {
```

```
                                newCoords.add(comp);      // -> therefore add to list
                                newInd = -(P+1);          // and set the new index
                            }
                        //System.out.println(" Node "+i+" "+j+" has no equivalent!");
                        nodesAfter[i][j] = new Node(newInd,comp);
                    }
                }
            }

        for(int i=0;i<N;i++)
            {
                //nodesAfter[i][0].WriteScr();
                //nodesAfter[i][1].WriteScr();
                //nodesAfter[i][2].WriteScr();
                //nodesAfter[i][3].WriteScr();
                int top = nodesAfter[i].length;
                //System.out.println("\n  result from adjusting procedure for "+
                                    i+" has topology: "+top);
                if(top == 4)
                    {res[i] =
                                new ElementCoordinate(i,nodesAfter[i][0],nodesAfter[i][1],
                                                      nodesAfter[i][2],nodesAfter[i][3]);}
                else{if(top == 6)
                    {res[i] =
                                new ElementCoordinate(i,nodesAfter[i][0],nodesAfter[i][1],
                                                      nodesAfter[i][2],nodesAfter[i][3],
                                                      nodesAfter[i][4],nodesAfter[i][5]);}
                else{if(top == 8)
                    {res[i] =
                                new ElementCoordinate(i,nodesAfter[i][0],nodesAfter[i][1],
                                                      nodesAfter[i][2], nodesAfter[i][3],
                                                      nodesAfter[i][4],nodesAfter[i][5],
                                                      nodesAfter[i][6],nodesAfter[i][7]);}}}

            }

        return res;
    }

}
```

C.4.7 The ObjectIds class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import java.lang.Integer.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;


/**
   Element and Node ids, the next vacant id is specified as double value, a vector specifies the
vacant ids, if elements or nodes have been errased.
 */
public class ObjectIds
{
    /**
       The next vacant Element in the list
     */
    public int nextVacEle;

    /**
       The next vacant Node in the list
     */
    public int nextVacNode;

    /**
       Vector of vacant element entities, which have been released due to erasing of an element.
     */
    public Vector vacEle = new Vector();
```

```
    /**
       Vector of vacant node entities, which have been released due to erasing of an elements.
     */
    public Vector vacNode = new Vector();

    public ObjectIds()
    {
        nextVacEle = 0;
        nextVacNode = 0;
    }

    /**
       Gives back the next possible id and icreases the <b>nextVacEle</b> entity or, if possible,
deletes this entity from the vacant list <b>vacEle</b>.
     */
    public int nextEle()
    {
        int nextVac;

        int N = this.vacEle.size();
        if(N>0)
            {
                nextVac = ((Integer)this.vacEle.get(N-1)).intValue();
                this.vacEle.setSize(N-1);
            }
        else
            {
                nextVac = this.nextVacEle;
                this.nextVacEle++;
            }

        return nextVac;
    }

    /**
       Gives back the next new id (does not take any elements from the vacant list  <b>vacEle</b>!)
and icreases the <b>nextVacEle</b> entity by num, useful for the creation of a new Object where a
large, connected amound of elements is created.
     */
    public int nextEle(int num)
    {
        int nextVac;

        nextVac = this.nextVacEle;
        this.nextVacEle += num;

        return nextVac;
    }

    /**
       When an element is deleted, its id is given free to be used again, therefore it is added to the
vacEle list, or, if it is the last element in the list, the counter nextVacEle is just lowered.
       @param toKill id of the element to be killed, this id is beween 0 and 9999, it is only the
element part of the complete id and does not contain the object-number.
       @return gives back true, if the element could be erased, false, if the element id given was
bigger than <b>nextVacEle</b> or was contained in the list of already killed elements.
     */
    public boolean killEle(int toKill)
    {
        boolean killed = false;
        boolean cantBeKilled = false;

        int N = this.vacEle.size();
        for(int i=0;i<N;i++)
            {
                int toCompare = ((Integer)vacEle.get(i)).intValue();
                if(toCompare==toKill){cantBeKilled = true;}   // element has already been killed !!
            }

        if(toKill>=this.nextVacEle)
            {
                cantBeKilled = true;
            }

        if(!cantBeKilled)
            {
```

```
                killed = true;
                if(toKill == (this.nextVacEle-1)){this.nextVacEle--;}
                // if last entity -> only decreasing index
                else{this.vacEle.add(new Integer(toKill));}
                // if in middel, adding entity id to list of vacancies
            }
        return killed;
    }


    /**
       Gives back the next possible id and icreases the <b>nextVacNode</b> entity or, if possible,
deletes this entity from the vacant list <b>vacEle</b>.
     */
    public int nextNode()
    {
        int nextVac;

        int N = this.vacNode.size();
        if(N>0)
            {
                nextVac = ((Integer)this.vacNode.get(N-1)).intValue();
                this.vacNode.setSize(N-1);
            }
        else
            {
                nextVac = this.nextVacNode;
                this.nextVacNode++;
            }
        return nextVac;
    }

    /**
       Gives back the next new id (does not take any elements from the vacant list  <b>vacNode</b>!)
and icreases the <b>nextVacNode</b> entity by num, useful for the creation of a new Object where a
large, connected amound of nodes is created.
     */
    public int nextNode(int num)
    {
        int nextVac;

        nextVac = this.nextVacNode;
        this.nextVacNode += num;

        return nextVac;
    }

    /**
       When an node is deleted, its id is given free to be used again, therefore it is added to the
vacNode list, or, if it is the last element in the list, the counter nextVacNode is just lowered.
       @param toKill id of the element to be killed, this id is beween 0 and 9999, it is only the
element part of the complete id and does not contain the object-number.
       @return gives back true, if the element could be erased, false, if the element id given was
bigger than <b>nextVacNode</b> or was contained in the list of already killed nodes
     */
    public boolean killNode(int toKill)
    {

        System.out.println(" Killing procedure started!");

        boolean killed = false;
        boolean cantBeKilled = false;

        int N = this.vacNode.size();
        for(int i=0;i<N;i++)
            {
                int toCompare = ((Integer)this.vacNode.get(i)).intValue();
                if(toCompare==toKill){cantBeKilled = true;}   // element has already been killed !!
            }

        if(toKill>=this.nextVacNode)
            {
                cantBeKilled = true;
            }

        if(!cantBeKilled)
            {
```

```
                killed = true;
                if(toKill == (nextVacNode-1)){nextVacNode--;}
                        // if last entity -> only decreasing index
                else{this.vacNode.add(new Integer(toKill));}
                        // if in middel, adding entity id to list of vacancies
            }

        System.out.println(" Killing procedure proceeded!");

        return killed;
    }

    /**
Gives back the actual ids (last initialised ids) of the node (as first integer) and the element (as
second integer) collection.
@return Two integer, first: id of last node, second: id of last element.
     */
    public int[] actIds()
    {
        int res[] = new int[2];
        res[0] = this.nextVacNode-1;
        res[1] = this.nextVacEle-1;

        return res;
    }

    public void WriteScr()
    {
        int N = this.vacEle.size();
        int M = this.vacNode.size();

        System.out.print("\n\t\t\t The last element is: "+
                                        (nextVacEle-1)+" & the Vacant Element List:\n\t\t\t");
        for(int i=0;i<N;i++)
            {
                int id = ((Integer)this.vacEle.get(i)).intValue();
                System.out.print(" | "+id);
            }

        System.out.print("\n\t\t\t The last node is: "+
                                            (nextVacNode-1)+" & the Vacant Node List:\n\t\t\t");
        for(int i=0;i<M;i++)
            {
                int id = ((Integer)this.vacNode.get(i)).intValue();
                System.out.print(" | "+id);
            }

    }

}
```

C.4.8 The ObjectCounter class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.finitele;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.math.*;

/**
 */
public class ObjectCounter
{
    public Vector nodesAndElements;
    public Vector objectId;

    private static final int maxNumElePerObj = 100000;
```

```
    public ObjectCounter()
    {
        nodesAndElements = new Vector();
        objectId = new Vector();
    }

    /**
       Just initialises a new Object, assignes a new Object-id and opens a placehoder for the node and
element ids.
     */
    public void newObject()
    {
        int M;

         int N = objectId.size();
        if(N>0)
            {
               M = 1+((Integer)objectId.get(N-1)).intValue();
            }
        else
            {
               M = 0;
            }
        objectId.add(new Integer(M));

        ObjectIds contains = new ObjectIds();

        this.nodesAndElements.add(contains);
    }

    /**
        Takes a whole id and determines the Object number and returns it.
        @param num whole id of any object entity.
        @return the object number.
     */
    public int getObjPlace(long num)
    {
        int objNum = (int)(num/maxNumElePerObj);

        int res = -1;

        int N = this.objectId.size();
        for(int i=0;i<N;i++)
            {int objNumFromList = ((Integer)(this.objectId.get(i))).intValue();
            if(objNumFromList == objNum)
               {res = i;}}

        if(res == -1){System.out.println(" Object-id not found !!!!");}

        return res;

    }


    /**
        Takes a whole id and determines the element or node number and returns it.
        @param num whole id of any object entity.
        @return the element of node number.
     */
    public int getEntityNr(long num)
    {
        int entNr = (int)(num%maxNumElePerObj);

        return entNr;

    }


    /**
       Adds a new element to the last Object
       @return new complete Id of the Element.
     */
    public long newElement()
    {
        int ret;
        int N = this.nodesAndElements.size();
```

```
    ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
    nodesAndElements.setSize(N-1);

    ret = toModify.nextEle();

    nodesAndElements.add(toModify);

    int objId = ((Integer)this.objectId.get(N-1)).intValue();
    long res = maxNumElePerObj*(long)objId+(long)ret;
    return res;
}



/**
   Adds num elements to the last Object
   @param num number of elements to be added!
   @return new complete Id of the fist Element.
 */
public long newElement(int num)
{
    int ret;
    int N = this.nodesAndElements.size();

    ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
    nodesAndElements.setSize(N-1);

    ret = toModify.nextEle(num);

    nodesAndElements.add(toModify);

    int objId = ((Integer)this.objectId.get(N-1)).intValue();
    long res = maxNumElePerObj*(long)objId+(long)ret;
    return res;
}

/**
   Creates a new element in the specified element, given by the number.
   @param num object number in List (to be found with getObjPlace(long num) - function)
   @see jp.go.jnc.tokai.pouchon.finitele.ObjectCounter#getObjPlace(long)
   @return complete id of the created element
 */
public long newElementInObj(int num)
{
    int ret = -1;
    int N = this.nodesAndElements.size();

    long res;

    if(num > N-1)
        {
            System.out.println("  Error - Object where Element should be added doesen't exist!");
            res = -1;
        }
    else
        {
            ObjectIds toModify = (ObjectIds)nodesAndElements.remove(num);

            ret = toModify.nextEle();

            nodesAndElements.add(num, toModify);

            int objId = ((Integer)this.objectId.get(num)).intValue();
            res = maxNumElePerObj*(long)objId+(long)ret;
        }
    return res;
}

/**
   Killst an element
 */
public void killEle(int toKill)
{
    boolean ret;
    int N = this.nodesAndElements.size();
```

```
        ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
        nodesAndElements.setSize(N-1);

        ret = toModify.killEle(toKill);

        nodesAndElements.add(toModify);

    }

    /**
       Kills an element specified by the total id-number.
       @param eleNr Number of the Element
       @param objNr Number of the Object
     */
    public void killElementInObj(int eleNr, int objNr)
    {
        int N = this.nodesAndElements.size();

        if(objNr > N-1)
            {
               System.out.println("  Error - Object where Node should be added doesen't exist!");
            }
        else
            {
               ObjectIds toModify = (ObjectIds)nodesAndElements.remove(objNr);

               boolean ret = toModify.killEle(eleNr);

               nodesAndElements.add(objNr,toModify);
            }
    }


   /**
       Adds a nodes to the last Object
       @return new complete Id of the Element.
     */
    public long newNode()
    {
        int ret;
        int N = this.nodesAndElements.size();

        ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
        nodesAndElements.setSize(N-1);

        ret = toModify.nextNode();

        nodesAndElements.add(toModify);

        int objId = ((Integer)this.objectId.get(N-1)).intValue();
        long res = maxNumElePerObj*(long)objId+(long)ret;
        return res;
    }

    /**
       Adds num nodes to the last Object
       @param num number of nodes to be added!
       @return new complete Id of the fist Element.
     */
    public long newNode(int num)
    {
        int ret;
        int N = this.nodesAndElements.size();

        ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
        nodesAndElements.setSize(N-1);

        ret = toModify.nextNode(num);

        nodesAndElements.add(toModify);

        int objId = ((Integer)this.objectId.get(N-1)).intValue();
        long res = maxNumElePerObj*(long)objId+(long)ret;
        return res;
    }
```

```
/**
   Creates a new node in the specified object, given by the number.
   @param num object number in List (to be found with getObjPlace(long num) - function)
   @see jp.go.jnc.tokai.pouchon.finitele.ObjectCounter#getObjPlace(long)
   @return complete id of the created node
 */
public long newNodeInObj(int num)
{
    int ret = -1;
    int N = this.nodesAndElements.size();

    long res;

    if(num > N-1)
        {
            System.out.println("  Error - Object where Node should be added doesen't exist!");
            res = -1;
        }
    else
        {
            ObjectIds toModify = (ObjectIds)nodesAndElements.remove(num);

            ret = toModify.nextNode();

            nodesAndElements.add(num, toModify);

            int objId = ((Integer)this.objectId.get(num)).intValue();
            res = maxNumElePerObj*(long)objId+(long)ret;
        }
    return res;
}

/**
   Kills Node in the actual Object.
   @param toKill number of the node to be killed.
 */
public void killNode(int toKill)
{
    boolean ret;
    int N = this.nodesAndElements.size();

    ObjectIds toModify = (ObjectIds)nodesAndElements.get(N-1);
    nodesAndElements.setSize(N-1);

    ret = toModify.killNode(toKill);

    nodesAndElements.add(toModify);
}

/**
   Kills a node specified by the total id-number.
   @param nodeNr Number of the Node
   @param objNr Number of the Object
 */
public void killNodeInObj(int nodeNr, int objNr)
{
    int N = this.nodesAndElements.size();

    if(objNr > N-1)
        {
            System.out.println("  Error - Object where Node should be added doesen't exist!");
        }
    else
        {
            ObjectIds toModify = (ObjectIds)nodesAndElements.remove(objNr);

            boolean ret = toModify.killNode(nodeNr);

            nodesAndElements.add(objNr,toModify);
        }
}



public void WriteScr()
{
    int N = this.objectId.size();
```

```
        if(N == this.nodesAndElements.size())
            {
                System.out.println("\n\n\t Object Counter:");
                for(int i=0;i<N;i++)
                    {int id = ((Integer)this.objectId.get(i)).intValue();
                    System.out.println("\n\t\t Object "+id+":");
                    ObjectIds toWrite = (ObjectIds)this.nodesAndElements.get(i);
                    toWrite.WriteScr();}
            }

    }

}
```

C.5 The spherearr package

C.5.1 The CellSphreres class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.spherearr;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.finitele.*;
import jp.go.jnc.tokai.pouchon.math.*;

/**
   Class describing a collection of spheres in the cell. In mainly provides utils to import sphere
data from a file.
 */
public class CellSpheres
{
    /**
       List of spheres, the basic type is CellSphereData, which contains many information about the
sphere, like number of neighbouts, ... .
     */
    public CellSphereData Spheres[];
    /**
       Number of spheres in the collection.
     */
    public int Size;

    /**
       Constructor which reads in two text files, one for the fine, and one for the coarse fraction.
The amount in each fraction is specified here.
       This function calls a constructor of CellSphereData with a Buffered Reader as parameter, which
extracts each sphere with its specifications from the file-lines
       @param fineName name of the text-file containing the fine fraction.
       @param coarseName name of the text-file containing the coarse fraction.
       @param nFine number of fine spheres in the collection
       @param nCoarse number of coarse spheres in the collection
       @see jp.go.jnc.tokai.pouchon.spherearr.CellSphereData#CellSphereData(java.io.BufferedReader,
boolean)
     */
    public CellSpheres(String fineName, String coarseName, int nFine, int nCoarse)
    {
        Spheres = new CellSphereData[nFine+nCoarse];
        int Size = nFine+nCoarse;
        int SizeCoarse = nCoarse;
        try
            {
                BufferedReader in = new BufferedReader(new FileReader(fineName));
                System.out.println();
                System.out.println(" - Initialisation of fine sphere-package with data from file: "+
                                   fineName+". The "+nFine +" sphere id's are: ");
                for(int i=0; i<nFine; i++)
                    {
                        Spheres[i] = new CellSphereData(in,false);
                        System.out.println( "  Sphere-id: "+ Spheres[i].id);
                    }
                in.close();
```

```
                System.out.println();
            }
        catch(IOException e)
            {
                System.out.print(" **** Fehler **** (Datei nicht gefunden) "+ e);
                System.exit(1);
            }
        try
            {
                BufferedReader in = new BufferedReader(new FileReader(coarseName));
                System.out.println();
                System.out.println(" - Initialisation of coarse sphere-package with data from file: "+
                                   coarseName+". The "+nCoarse +" sphere id's are: ");
                for(int i=0; i<nCoarse; i++)
                    {
                        Spheres[i+nFine] = new CellSphereData(in,true);
                        System.out.println( "  Sphere-id: "+ Spheres[i+nFine].id);
                    }
                in.close();
                System.out.println();
            }
        catch(IOException e)
            {
                System.out.print(" **** Fehler **** (Datei nicht gefunden) "+ e);
                System.exit(1);
            }
    }

    /**
       Writing all id's of the sphere-collection to the screen.
     */
    void WriteScr()
    {
        System.out.println(" - This sphere-package contains "+ Size +" spheres with the id's:");
        for(int i=0; i<Size; i++)
            {
                System.out.print(" * "+Spheres[i].id);
            }
    }

    /**
       Initialisation of a created instance (already contains Sphere information) with the direction
of the touching points given in spherical coordinates.
     */
    void initialize()
    {
        for(int i=0; i<Size; i++)
            {
                for(int j=0; j<12; j++)
                    {
                        for(int k=0; k<Size; k++)
                            {
                                if((Spheres[k].id == Spheres[i].neigId[j]) &&
                                   (Spheres[i].neigDist[j] < Spheres[i].r/10))
                                    {
                                        Coordinate RelTouchPoint;
                                        RelTouchPoint = new Coordinate(Spheres[i]);
                                        RelTouchPoint.add(Spheres[k], -1.0);
                                        Spheres[i].direction[j].assign(RelTouchPoint);
                                        RelTouchPoint = null;
                                    }
                            }
                    }
            }
    }


}
```

C.5.2 The CellSphereData class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */
```

```
package jp.go.jnc.tokai.pouchon.spherearr;
import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.finitele.*;
import jp.go.jnc.tokai.pouchon.math.*;

/**
Complete infromation about the sphere. Additional infromations are used for the meshing routine and
later connection between the spheres.
 */
public class CellSphereData extends Sphere
{
    /**
id of the sphere, this is not a entity for the elements but just the original appellation of each
sphere.
     */
    public long id;
    /**
       Number of neighbors for each sphere, when the distance is smaller than a critical value, the
spheres are considered to be touching.
     */
    public long NONeig;
    /**
id's of the next 12 neighbours. They don't have to touch, just the first NONeig's are touching.
     */
    public long neigId[];
    /**
       Distance of each neighbour.
     */
    public double neigDist[];
    /**
       The direction of the next neichbours, this is given relative to the midpoint in spherical
coordinates.
     */
    public SphereCoordinate direction[];

    /**
Construictor of an empty sphere, empty means here: in the origin (0,0,0) with a negative radius (of
course impossible), radius -1 is therefore a sign for not being initialized.
     */
    public CellSphereData()
    {
        super();

        neigId = new long[12];
        neigDist = new double[12];
        direction = new SphereCoordinate[12];

        id = -1;
        r = -1;
        NONeig = 0;
    }


    /**
       Constructor with the basic values.
       @param idIn id of the new sphere.
       @param OriginIn input of the sphere mid-point (origin)
       @param rIn input of radius
       @param NONeigIn number of neighbours (distance smaller than a critical value)
       @param neigIdIn<sub>i=0&#8230;12</sub> input of the neighbour id's
       @param neigDistIn<sub>i=0&#8230;12</sub> input of the neighbour distances
     */
    public CellSphereData(long idIn, Coordinate OriginIn, double rIn,
                          long NONeigIn, long neigIdIn[], double neigDistIn[])
    {
        super(OriginIn,rIn);

        neigId = new long[12];
        neigDist = new double[12];
        direction = new SphereCoordinate[12];

        id = idIn;
        NONeig = NONeigIn;
        neigId = neigIdIn;
        neigDist = neigDistIn;
```

```
    }

    /**
       Copy constructor.
     */
    public CellSphereData(CellSphereData in)
    {
        super(in);

        neigId = new long[12];
        neigDist = new double[12];
        direction = new SphereCoordinate[12];

        id = in.id;
        r = in.r;
        NONeig = in.NONeig;
        for(int i=0; i<12; i++)
            {
                neigId[i] = in.neigId[i];
                neigDist[i] = in.neigDist[i];
            }
    }

    /**
       Constructor with a buffered reader as argument, this contains the ASCII-text Stream whose lines
contain the sphere data, one line is taken at each call. The boolean variable tells if the fine or
coarse fraction is read in.
       @param is ASCII-text stream which contains all sphere data (of all spheres).
       @param coarse true if the coarse fraction is read in, false otherwise.
     */
    public CellSphereData(BufferedReader is, boolean coarse) throws IOException
    {
        super();

         String EmptyToken;
        neigId = new long[12];
        neigDist = new double[12];
        direction = new SphereCoordinate[12];

        String s = is.readLine();
        StringTokenizer t = new StringTokenizer(s, ",");
        id           = Long.parseLong(t.nextToken().trim());
        this.x[0]  = Double.parseDouble(t.nextToken().trim());
        this.x[1]  = Double.parseDouble(t.nextToken().trim());
        this.x[2]  = Double.parseDouble(t.nextToken().trim());
        r            = Double.parseDouble(t.nextToken().trim());
        NONeig       = Long.parseLong(t.nextToken().trim());
        if(!coarse)
            {
                //System.out.print(" This is a fine fraction particle");
                EmptyToken   = t.nextToken();
                for(int i=0; i<12; i++)
                    {
                        neigId[i]   = Long.parseLong(t.nextToken().trim());
                        neigDist[i] = Double.parseDouble(t.nextToken().trim());
                    }
            }
        else
            {
                //System.out.print(" This is a coarse fraction particle");
            }
    }

    /**
       Writes the complete infromation of one sphere to the screen.
     */
    public void WriteScr()
    {
        System.out.println( "* Id " + id );
        System.out.print( "   Origin " ); super.WriteScr();
        System.out.print( "   (" );
        for(int i=0; i<12; i++)
            {
                System.out.print( " n" + i + "=" + neigId[i] );
            }
        System.out.println( " )" );
    }
```

```
}
```

C.5.3 The SphrereMesh class

```
/*
 * Copyright 2000-2001 JNC Japan, Inc. All Rights Reserved.
 */

package jp.go.jnc.tokai.pouchon.spherearr;

import java.io.*;
import java.util.*;
import jp.go.jnc.tokai.pouchon.geometry.*;
import jp.go.jnc.tokai.pouchon.finitele.*;
import jp.go.jnc.tokai.pouchon.math.*;


// ********************************************************************************
// *********************************   Class SphereMesh   *************************
// ********************************************************************************

/**
   Class to generate, represent and modify meshes for spheres.
 */
public class SphereMesh extends CellSphereData
{
    /**
       Node coordinates in sherical system, this information is kept for later easy transfromations,
this infromation is more natural for the system than the cartesian description, but is not compartible
with all the transformation and notation tools.
     */
    public SphereCoordinate[] nodesSpherical; // information kept for later transformations
    /**
       Node coordinates in cartesian system.
     */
    public Node[] nodes;
    /**
       Elements in this sphere mesh.
     */
    public Element[] elements;

    public Vector newNodes = new Vector();

    public Vector newElements = new Vector();

    // ***************************************** Constructors **********************************

    /**
       Simple copy constructor.
       @param toCopy mesh of a shere to be copied.
     */
    public SphereMesh(SphereMesh toCopy)
    {
        super(toCopy);

        int N = toCopy.nodesSpherical.length;
        nodesSpherical = new SphereCoordinate[N];
        for(int i=0; i<N; i++)
            {
                nodesSpherical[i] = new SphereCoordinate(toCopy.nodesSpherical[i]);
            }

        N = toCopy.nodes.length;
        nodes = new Node[N];
        for(int i=0; i<N; i++)
            {
                nodes[i] = new Node(toCopy.nodes[i]);
            }

        N = toCopy.elements.length;
        elements = new Element[N];
        for(int i=0; i<N; i++)
            {
                elements[i] = new Element(toCopy.elements[i]);
```

```
            }
    }

    /**
       Procedure wich takes ObjectCounter instead of direct integer, otherwise see procedure with
integer argument for new ids. The startId of the  nodes and elements is the same here.
       @see jp.go.jnc.tokai.pouchon.spherearr.SphereMesh#SphereMesh(CellSphereData,long,int)
       @param sphIn sphere to be meshed
       @param ids Object counter for determination of the ids.
       @param degreeSf subdivision degree of the sphere-surface.
    */
    public SphereMesh(CellSphereData shpIn, ObjectCounter ids, int stufe)
    {
        this(shpIn, findStart(ids,stufe), stufe);
    }

    /**
     */
    private static long findStart(ObjectCounter ids, int stufe)
    {
        int nodeNum = (int)(Math.pow(stufe,2)*4)+3;
        int eleNum = (int)((Math.pow(stufe,2)*4)*2);

        long startIdNode = ids.newNode(nodeNum);
        long startIdEle  = ids.newElement(eleNum);
        long startId = 0;

        if(startIdNode > startIdEle)
            {long i = ids.newElement((int)(startIdNode-startIdEle));
            startId = startIdNode;}
        else{if(startIdNode < startIdEle)
            {long i = ids.newNode((int)(startIdEle-startIdNode));
            startId = startIdEle;}
        else
            {startId = startIdNode;}}

        return startId;
    }



    /**
       Basic constructor of a sphere mesh. It takes the sphere information as input and generates the
nodes and elements to represent it.
       @param sphIn sphere to be meshed
       @param startId id of the first entry in the new mesh.
       @param degreeSf subdivision degree of the sphere-surface.
       @param degreeRa radial subdivisions of the sphere.
     */
    public SphereMesh(CellSphereData sphIn, long startId,  int stufe)
    {
        super(sphIn);

        long newId = startId;
         int nodeNum = (int)(Math.pow(stufe,2)*4)+3;
        int eleNum = (int)((Math.pow(stufe,2)*4)*2);

        nodes = new Node[nodeNum];
        for(int i=0; i<nodeNum; i++){nodes[i] = new Node();}
        nodesSpherical = new SphereCoordinate[nodeNum];
        for(int i=0; i<nodeNum; i++){nodesSpherical[i] = new SphereCoordinate();}
        elements = new Element[eleNum];
        for(int i=0; i<eleNum; i++){elements[i] = new Element();}
        Coordinate midPt = new Coordinate(sphIn);
        Coordinate stack1Pt = new Coordinate();
        Coordinate stack2Pt = new Coordinate();
        SphereCoordinate stack1PPt = new SphereCoordinate();
        SphereCoordinate stack2PPt = new SphereCoordinate();

        nodes[0].assign(newId,midPt); newId++; //Mid-Point

        stack2PPt.assign(0,Math.acos(0),sphIn.r); stack2Pt.assign(stack2PPt); // Vector to pole
        stack1Pt.add(midPt, stack2Pt); nodes[1].assign(newId,stack1Pt); newId++; // North-Pole
        nodesSpherical[1].assign(stack2PPt); // assign North-Pole in spherical coordinates
        stack1Pt.add(midPt, stack2Pt, -1); nodes[2].assign(newId,stack1Pt); newId++; // South-Pole
        stack2PPt.scale(-1.0); nodesSpherical[2].assign(stack2PPt);
        // assign South-Pole in spherical coordinates
```

```
     int index = 2;
    for(int i=0; i<((stufe-1)*2+1); i++)
    // equator: 0 , first row Norh: 1, first row South: 2, .... Poles: later
        {
            int l = (int)((i+1)/2); // counter in one direction: 1: first row, 2: second row, ....
            int m = (int)Math.pow(-1,(1+l-(int)(i/2))); // direction: 1: North, -1: South, ....
            int k = (stufe-l)*4; // Number of divisions around z-axis
            for(int j=0;j<k;j++)
                {
                    index++;
                    double zDegree = (4*Math.acos(0)/k)*j;
                    double xyDegree = (Math.acos(0)/stufe)*l*m;
                    stack2PPt.assign(zDegree, xyDegree, sphIn.r); stack2Pt.assign(stack2PPt);
                    stack1Pt.add(midPt,stack2Pt);
                    nodes[index].assign(newId,stack1Pt); newId++;
                    // assign Node in normal coordinates
                    //nodes[index].WriteScr();
                    nodesSpherical[index].assign(stack2PPt);
                    // assign Node in spherical coordinates
                }
        }

    index = 0;
    int index2 = 3;
    newId = startId;

    for(int i=0; i<2*stufe; i++)
        {
            int ii, iii; // row indizes, see implementation (equator:0, ....)
            if(i<1){ii=0;}else{ii=i-1;}
            iii=i+1;

            int locInVec11, locInVec12;
            // determination of the matix(vector) start and end index for each row (here ii)
            if(ii<=2*(stufe-1)) // not one of the poles
                {
                    locInVec11=3;
                    for(int j=0;j<ii;j++)
                        {locInVec11 += (stufe-(int)((j+1.0)/2.0))*4;}
                    locInVec12 = locInVec11+((stufe-(int)((ii+1.0)/2.0))*4)-1;
                }
            else // poles
                {
                    locInVec11 = 2+ii-(2*stufe); locInVec12 = locInVec11;
                }

            int locInVec21, locInVec22;
            // determination of the matix(vector) start and end index for each row (here iii)
            if(iii<=2*(stufe-1)) // not one of the poles
                {
                    locInVec21=3;
                    for(int j=0;j<iii;j++)
                        {locInVec21 += (stufe-(int)((j+1.0)/2.0))*4;}
                    locInVec22 = locInVec21+((stufe-(int)((iii+1.0)/2.0))*4)-1;
                }
            else // poles
                {
                    locInVec21 = 2+iii-(2*stufe); locInVec22 = locInVec21;
                }

            long modInd = BeltMesh(locInVec11,locInVec12,locInVec21,locInVec22,newId,index);

            newId = modInd;
            if((locInVec12-locInVec11)<1 || locInVec22-locInVec21<2)
                {index += (locInVec12-locInVec11)+(locInVec22-locInVec21)+1;}
            else{index += (locInVec12-locInVec11)+(locInVec22-locInVec21)+2;}
        }
}

private long BeltMesh(int lowI1, int lowI2, int upI1, int upI2, long idStart, int intIdStart)
                    // returns next free Id
{
    Node midN = new Node(nodes[0]);
    int uNodeN = ((upI2-upI1)+1);
    int lNodeN = ((lowI2-lowI1)+1);
```

```
        Node lowerNodes[] = new Node[lNodeN];
        for(int i=0; i<lNodeN;i++){lowerNodes[i]=nodes[i+lowI1];}
        Node upperNodes[] = new Node[uNodeN];
        for(int i=0; i<uNodeN;i++){upperNodes[i]=nodes[i+upI1];}

        long newId = idStart;
        int newIntId = intIdStart;

        if(uNodeN > 1){
        for(int i=0; i<uNodeN; i++)
            {
                int ii;
                double midCon;
                if((i+1)<uNodeN){ii=(i+1);}else{ii=0;}
                if(ii < 1){midCon=(double)
                         (-1.0/(2.0*uNodeN));}else{midCon=(double)(i+ii)/(2.0*uNodeN);}

                double diff;
                double diffComp = 1; int opoInd = -1;
                for(int j=0; j<lNodeN; j++)
                    {
                        double lowPt = (double)(j*1.0/lNodeN);
                        if(Math.abs(lowPt-midCon)>.5)
                        {diff=(1-Math.abs(lowPt-midCon));}else{diff=Math.abs(lowPt-midCon);}
                        if(diff < diffComp){opoInd = j;diffComp=diff;}
                    }

                Element returnElement =
                new Element(newId, upperNodes[i],upperNodes[ii],lowerNodes[opoInd], midN); newId++;

                elements[newIntId]=(returnElement); newIntId++;
            }}

        if(lNodeN > 1){
        for(int i=0; i<lNodeN; i++)
            {
                int ii;
                double midCon;
                if(i+1<lNodeN){ii=i+1;}else{ii=0;}
                if(ii < 1){midCon=(-1.0/(2.0*lNodeN));}
                        else{midCon=(double)(i+ii)/(2.0*lNodeN);}
                double diff;
                double diffComp = midCon; int opoInd = 0;
                for(int j=0; j<uNodeN; j++)
                    {
                        double lowPt = (double)(1.0*j/uNodeN);
                        if(Math.abs(lowPt-midCon)>.5)
                        {diff=(1-Math.abs(lowPt-midCon));}else{diff=Math.abs(lowPt-midCon);}
                        if(diff < diffComp){opoInd = j; diffComp=diff;}
                    }


                Element returnElement = new Element(newId,lowerNodes[ii],lowerNodes[i],
                                                    upperNodes[opoInd], midN); newId++;
                elements[newIntId]=(returnElement); newIntId++;
            }}


        return newId;

    }

    /**
       Transformes sphere-mesh, so that the closest node is shifted to the Coordinate "toConn", the
rest of the nodes are transfored spherecally in the same direction but with decrasing distance
(decrasing transformation factor). The farest point to be transformed is given with the parameter
region, where 1 would just contain the whole sphere, if the whole mesh should be trasformed
identically, region should be set to infinit, or just a large number (of course).
       @param toConn point to be connected with the sphere-mesh
       @param region region of the sphere mesh to be adjusted, 0 &#8658; only one mesh node is
displaced, 1 &#8658; the whole sphere-mesh is transformed, the farest point with a factor of 0,
&#8594;&#8734; &#8658; the whole sphere-mesh is transformed with the same factor.
     */
    public void connectPoint(Coordinate toConn, double region)
    {
        // find closest Point
        int nodeNum = nodes.length;
```

```
      int closestInd = -1;
      double radius = r;
      Coordinate midPt = new Coordinate(this);
      double transf[] = new double[nodeNum]; for(int i=0; i<nodeNum; i++){transf[i]=0.0;}
      double closestDist = Double.MAX_VALUE;
      for(int i=1; i<nodeNum; i++) // no test for Midpoint -> no 0
          {
              if (nodes[i].distance(toConn) < closestDist)
                  {
                      closestDist = nodes[i].distance(toConn);
                      closestInd = i;
                  }
          }
      // transforation determination
      Coordinate closestPt = new Coordinate(nodes[closestInd]);
      Coordinate vecToConn = new Coordinate(toConn); vecToConn.add(nodes[0],-1);
      Coordinate vecToClNode = new Coordinate(nodes[closestInd]); vecToClNode.add(nodes[0],-1);
      SphereCoordinate vecToConnS = new SphereCoordinate(vecToConn);
      SphereCoordinate vecToClNodeS = new SphereCoordinate(vecToClNode);
      SphereCoordinate korrVec = new SphereCoordinate();
      korrVec.toTransform(vecToClNodeS,vecToConnS);
      //korrVec.WriteScr();
      for(int i=1; i<nodeNum; i++)
          {
              if(i!=closestInd)
                  {
                      double korr =
                       (2*region*radius-closestPt.distance(nodes[i]))/(2*region*radius);
                      if(korr>0)
                          {
                              Coordinate trsferd = new Coordinate(nodes[i]);
                              trsferd.add(midPt,-1);
                              trsferd.rotate(korrVec.phi[1],0,0,korr);
                              trsferd.rotate(0,0,korrVec.phi[0],korr);
                              Coordinate scaledKorr = new Coordinate(trsferd);
                               scaledKorr.mult(korrVec.r*korr);
                              scaledKorr.add(midPt);

                              nodes[i].assign(scaledKorr);
                          }
                  }
              else
                  {
                      nodes[closestInd].assign(toConn);
                  }
          }
      regenerate(true);
   }


public long connectPoint(Coordinate toConn, double region, boolean onlyIds)
   {
      // find closest Point
      int nodeNum = nodes.length;
      int closestInd = -1;
      double radius = r;
      Coordinate midPt = new Coordinate(this);
      double transf[] = new double[nodeNum]; for(int i=0; i<nodeNum; i++){transf[i]=0.0;}
      double closestDist = Double.MAX_VALUE;
      for(int i=1; i<nodeNum; i++) // no test for Midpoint -> no 0
          {
              if (nodes[i].distance(toConn) < closestDist)
                  {
                      closestDist = nodes[i].distance(toConn);
                      closestInd = i;
                  }
          }
      // transforation determination
      Coordinate closestPt = new Coordinate(nodes[closestInd]);
      Coordinate vecToConn = new Coordinate(toConn); vecToConn.add(nodes[0],-1);
      Coordinate vecToClNode = new Coordinate(nodes[closestInd]); vecToClNode.add(nodes[0],-1);
      SphereCoordinate vecToConnS = new SphereCoordinate(vecToConn);
      SphereCoordinate vecToClNodeS = new SphereCoordinate(vecToClNode);
      SphereCoordinate korrVec = new SphereCoordinate();
      korrVec.toTransform(vecToClNodeS,vecToConnS);
      //korrVec.WriteScr();
      for(int i=1; i<nodeNum; i++)
```

```
        {
            if(i!=closestInd)
                {
                    double korr =
                            (2*region*radius-closestPt.distance(nodes[i]))/(2*region*radius);
                    if(korr>0)
                        {
                            Coordinate trsferd = new Coordinate(nodes[i]);
                            trsferd.add(midPt,-1);
                            trsferd.rotate(korrVec.phi[1],0,0,korr);
                            trsferd.rotate(0,0,korrVec.phi[0],korr);
                            Coordinate scaledKorr = new Coordinate(trsferd);
                             scaledKorr.mult(korrVec.r*korr);
                            scaledKorr.add(midPt);

                            if(!onlyIds){nodes[i].assign(scaledKorr);}
                        }
                }
            else
                {
                    if(!onlyIds){nodes[closestInd].assign(toConn);}
                }
        }
    if(!onlyIds){regenerate(true);}

    return closestInd;
}

void regenerate(boolean toSpherical) // otherwise: Spherical to Normal
{
    if(toSpherical)
        {
            SphereCoordinate stackPt = new SphereCoordinate();
            Coordinate midPt = new Coordinate(this),  stack1Pt = new Coordinate();
            int nodeNum = nodes.length;
            for(int i=1; i<nodeNum; i++)
            //Midpoint is not subject of any modification routine->counter from 1 and not 0
                {
                    stack1Pt.add(nodes[i],midPt,-1); // substract Midpoint
                    nodesSpherical[i].assign(stack1Pt);
                }
        }
    else
        {
            Coordinate stackPt = new Coordinate(),stack1Pt = new Coordinate(),
            midPt = new Coordinate(this);
            int nodeNum = nodes.length;
            for(int i=1; i<nodeNum; i++)
            // Midpoint is not subject of any modification routine->counter from 1 and not 0
                {
                    stackPt.assign(nodesSpherical[i]); // spherical to normal coordinates
                    stack1Pt.add(midPt,stackPt); // add Midpoint
                    nodes[i].assign(stack1Pt);
                }
        }
}

/*
public void cut(Cell cuttingCell)
{
    int elementsNum = elements.length;
    int nodesNum = nodes.length;
    for(int i=0; i<elementsNum; i++)
        {
            int nodeInEle = elements[i].nodeId.length;
            int l[] = new int[nodeInEle];
            int k=0;
            int isout=1;
            for(int j=0;j<nodesNum;j++)
                {
                    for(int m=0;m<nodeInEle;m++)
                        {
                            if(elements[i].nodeId[m]==nodes[j].id)
                                {l[k]=j; k++;}
                            else{}
                        }
                }
```

```
            for(int n=0;n<nodeInEle;n++)
                {
                    if(cuttingCell.within(nodes[l[n]].Koor))
                        {isout = 0;}
                    else {}
                }
            if(isout==1)
                {
                    elements[i].id = -1;
                }
        }
}
*/

public void cut(Polyhedron cuttingCell)
{
    int elementsNum = elements.length;
    int nodesNum = nodes.length;
    Coordinate origin = new Coordinate(0.0,0.0,0.0);
    for(int i=0;i<elementsNum;i++)
        {
            //System.out.println("\n\t "+i+"'th element of sphere "+id+" is considered now!");
            Coordinate nodesInEle[] = elements[i].nodes(nodes);
            // read out the coordinates by knowing the id's
            Polyhedron eleToTest = new Polyhedron(origin, nodesInEle);
            // creation of polyhedron for testing
            if(cuttingCell.disjunct(eleToTest)){elements[i].id=-1;}
            // if disjunct then cancel by giving id -1!
        }
}


public void cut(Cell cuttingCell)
{
    int elementsNum = elements.length;
    int nodesNum = nodes.length;
    for(int i=0; i<elementsNum; i++)
        {
            Coordinate nodesInEle[] = elements[i].nodes(nodes);
            if(cuttingCell.within(elements[i], nodesInEle))
                {}
            else
                {elements[i].id = -1;}
        }
};

/**
   Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.
 */
public void adjust1(Polyhedron adjustingCell, ObjectCounter ids)
{
    int elementsNum = this.elements.length;
    int nodesNum = this.nodes.length;

    int objNr = -1;

    if(nodesNum>0)
        {long id = this.nodes[0].id;
        objNr = ids.getObjPlace(id);}

    for(int i=0;i<elementsNum;i++)
        {
            ElementCoordinate fullEle = new ElementCoordinate(this.elements[i],this.nodes);
            ElementCoordinate allEle[] = fullEle.adjust(adjustingCell);

            int eleNum = ids.getEntityNr(this.elements[i].id);

            int parts = allEle.length;

            long idAssoc[][][] = newNodesInd(allEle);

            if(parts==0)       // no element is coming back -> no reamining after cutting
                {
                    ids.killElementInObj(eleNum,objNr);
                    elements[i].id = -1;
                }
            else
```

```
                    // at leaste one remaining, element has same identity as original one
                {
                    Node newNods[];
                    newNods = elements[i].modify(allEle[0],ids);
                    int nNewNods = newNods.length; // == allEle[0].nodeId.length

                    for(int j=0;j<nNewNods;j++)    // add all the new nodes to the vector-list
                        {if(newNods[j].id!=0)
                            {Node toAdd = new Node(newNods[j]);
                            this.newNodes.add(toAdd);}}

                    for(int j=1;j<parts;j++)
                    // check if same identities can be initilised for nodes
                        {

                            for(int k=0;k<nNewNods;k++)
                                {
                                    int nNextNods = allEle[j].nodeId.length;
                                    for(int l=0;l<nNextNods;l++)
                                        {
                                            if((allEle[0].nodeId[k]==allEle[j].nodeId[l])
                                            && (allEle[0].nodeId[k]<0))
                                                {allEle[j].nodeId[l] = newNods[k].id;}
                                        }
                                }
                        }

                    for(int j=1;j<parts;j++)    // initialisation of new elements
                        {
                            Node newNods2[];
                            long newEle = ids.newElement(eleNum);

                            Element toAdd = new Element(allEle[j]);

                            System.out.println("   Fehlerstelle bei Obj "+objNr+" "+i+" "+j);
                            if(objNr==26){ids.WriteScr();}

                            newNods2 = toAdd.modify(allEle[j],ids);
                            int nNewNods2 = newNods2.length;

                            for(int k=0;k<nNewNods2;k++)
                                    // add all the new elements to the vector-list
                                {if(newNods2[k].id!=0)
                                    {Node nodeToAdd = new Node(newNods2[k]);
                                    this.newNodes.add(nodeToAdd);}}

                            for(int jj=j;jj<parts;jj++)
                                    // check if same identities can be initilised for nodes
                                {

                                    for(int k=0;k<nNewNods;k++)
                                        {
                                            int nNextNods = allEle[jj].nodeId.length;
                                            for(int l=0;l<nNextNods;l++)
                                                {
                                                    if((allEle[0].nodeId[k]==allEle[jj].nodeId[l])
                                                       && (allEle[0].nodeId[k]<0))
                                                        {allEle[jj].nodeId[l] = newNods2[k].id;}
                                                }
                                        }
                                }

                        }
                }
            }

        this.vectorToMatrix();
        //return ids;
    }

    /**
       Adjusts a single sphere mesh to the adjusting cell, given as Polyhedron.
     */
    public void adjust(Polyhedron adjustingCell, ObjectCounter ids)
    {
        int elementsNum = this.elements.length;
        int nodesNum = this.nodes.length;
```

```
    int objNr = -1;

    if(nodesNum>0)
        {long id = this.nodes[0].id;
        objNr = ids.getObjPlace(id);}

    for(int i=0;i<elementsNum;i++)
        {
           ElementCoordinate fullEle = new ElementCoordinate(this.elements[i],this.nodes);
           ElementCoordinate allEle[] = fullEle.adjust(adjustingCell);    // node ids meaningless
           // if new Node -> id < 0;

           int eleNum = ids.getEntityNr(this.elements[i].id);

           int parts = allEle.length;

           long idAssoc[][][]= SphereMesh.newNodesInd(allEle);
                  // what are the new nodes and where are they to be placed
           int numNewNodes = idAssoc.length;        // in the element-list

           for(int j=0;j<numNewNodes;j++)
                  // creation of new nodes and assigning ids to allEle[]
               {
                  // new id number <- found by using the object counter ids
                  long newId = ids.newNodeInObj(objNr);
                  System.out.println(" The new Node-Id is: "+newId);
                  // what is the corresponding coordinate (only look at first entry)
                  // System.out.println(" assoc: "+idAssoc[j][0][0]+" "+idAssoc[j][0][1]);
                  Coordinate newLoc =
                  new Coordinate(allEle[(int)idAssoc[j][0][0]].coords[(int)idAssoc[j][0][1]]);
                  // add the new node to the list
                  Node toAdd = new Node(newId,newLoc);
                  this.newNodes.add(toAdd);
                  // the new node id numbers are applied to the element list allEle
                  int numIsInEle = idAssoc[j].length;
                  for(int k=0;k<numIsInEle;k++)
                  // all points with same Coordinate -> same node
                      {
                          System.out.println(" assoc: "+idAssoc[j][k][0]+" "+idAssoc[j][k][1]);
                          allEle[(int)idAssoc[j][k][0]].putId(newId, (int)idAssoc[j][k][1]);
                          //allEle[(int)idAssoc[j][k][0]].nodeId[(int)idAssoc[j][k][1]] = newId;
                      }

               }

           if(parts==0)
                         // no element is coming back -> no reamining after cutting
               {
                  ids.killElementInObj(eleNum,objNr);
                  elements[i].id = -1;
               }
           else
                         // at leaste one remaining, element has same identity as original one
               {
                  elements[i].modify(allEle[0]);

                  for(int j=1;j<parts;j++)
                      {
                          long newEle = ids.newElementInObj(objNr);
                          System.out.println(" The new Element-Id is: "+newEle);
                          allEle[j].id = newEle;
                          this.newElements.add(allEle[j]);
                      }


               }
        }

    this.vectorToMatrix();
    //return ids;
}



public void adjustOld(Cell adjCell)
```

```
{

    System.out.println("Sphere with first element No. "+nodes[0].id); // KILL

    int elementsNum = elements.length;
    int nodesNum = nodes.length;
    Node change[] = new Node[nodesNum];
    for(int i=0; i<nodesNum; i++){change[i] = new Node(nodes[i]);}

    Face P[] = adjCell.faces();
    int plNum = P.length;

    // All outer nodes adjusted
    for(int outCase=1; outCase<4; outCase++){
        for(int i=0; i<elementsNum; i++)
           {
               int nodeInEle = 4;//elements[i].nodeId.length;
               int l[] = new int[nodeInEle];
               int k=0;
               int isout=0;
               for(int j=1;j<nodesNum;j++)
                   {
                       for(int m=0;m<nodeInEle;m++)
                          {
                              if(elements[i].nodeId[m]==nodes[j].id)
                                  {l[k]=j; k++;}
                              else{}
                          }
                   }
               for(int n=0;n<nodeInEle;n++)
                   {
                       if(!(adjCell.within(nodes[l[n]])))
                          {isout += 1;}
                       else {}
                   }
               // Runtime-Information
               //System.out.print("   ElementID. "+elements[i].id);
               //System.out.print(" -- Nodes "+nodes[l[0]].id+" "+nodes[l[1]].id+"
                                  "+nodes[l[2]].id+" "+nodes[l[3]].id);
               //System.out.print(" -- Nodes In Element: "+nodeInEle);
               //System.out.println(" -- Nodes Outside: "+isout);
               // Runtime-Information
               if(isout>0)
                   {
                       if(isout==3 && isout==outCase)
                          {
                              int isIn=-1;
                              for(int n=0;n<nodeInEle;n++){if(adjCell.within(nodes[l[n]]))
                                  {isIn = n;}}
                              for(int n=0;n<nodeInEle;n++)
                                  {
                                      Coordinate cuttingPtFinal = new Coordinate();
                                      if(n!=isIn)
                                          {
                                              for(int faNum=0; faNum<plNum; faNum++)
                                                  {
                                                      boolean isCutting;
                                                      Coordinate cuttingPt = new Coordinate();
                                                      Line cuttingLine =
                                                          new Line(nodes[l[isIn]],
                                                                   nodes[l[n]],true);
                                                      isCutting = cuttingPt.cut(P[faNum],
                                                                   cuttingLine,true);
                                                      if(isCutting)
                                                          {cuttingPtFinal.assign(cuttingPt);}
                                                  }
                                              change[l[n]].assign(cuttingPtFinal);
                                              //change[l[n]].Koor.WriteScr(); // Runtime Info
                                          }
                                  }
                          }
                       if(isout==2 && outCase==2)
                          {
                              int isIn[] = new int[2];
                              int nn=0;
                              for(int n=0;n<nodeInEle;n++)
                                  {if(adjCell.within(nodes[l[n]])){isIn[nn] = n; nn++;}}
```

```
                for(int n=0;n<nodeInEle;n++)
                    {
                        Coordinate cuttingPtFinal1 = new Coordinate();
                        Coordinate cuttingPtFinal2 = new Coordinate();
                        if(n!=isIn[0] && n!=isIn[1])
                            {
                                for(int faNum=0; faNum<plNum; faNum++)
                                    {
                                        boolean isCutting1, isCutting2;
                                        Coordinate cuttingPt1 = new Coordinate();
                                        Coordinate cuttingPt2 = new Coordinate();
                                        Line cuttingLine1=
                                            new Line(nodes[l[isIn[0]]],
                                                     nodes[l[n]],true);
                                        Line cuttingLine2=
                                            new Line(nodes[l[isIn[1]]],
                                                     nodes[l[n]],true);
                                        isCutting1 = cuttingPt1.cut(P[faNum],
                                                     cuttingLine1,true);
                                        isCutting2 = cuttingPt2.cut(P[faNum],
                                                     cuttingLine2,true);
                                        if(isCutting1)
                                            {cuttingPtFinal1.assign(cuttingPt1);}
                                        if(isCutting2)
                                            {cuttingPtFinal2.assign(cuttingPt2);}
                                    }
                                cuttingPtFinal1.add(cuttingPtFinal2);
                                cuttingPtFinal1.mult(.5);
                                change[l[n]].assign(cuttingPtFinal1);
                                //change[l[n]].Koor.WriteScr(); // Runtime Info
                            }
                    }
            }
        if(isout==1 && outCase == 1)
            {
                int isIn[] = new int[3];
                int nn=0;
                for(int n=0;n<nodeInEle;n++){if(adjCell.within(nodes[l[n]]))
                                    {isIn[nn] = n; nn++;}}
                for(int n=0;n<nodeInEle;n++)
                    {
                        Coordinate cuttingPtFinal1 = new Coordinate();
                        Coordinate cuttingPtFinal2 = new Coordinate();
                        Coordinate cuttingPtFinal3 = new Coordinate();
                        if(n!=isIn[0] && n!=isIn[1] && n!=isIn[2])
                            {
                                for(int faNum=0; faNum<plNum; faNum++)
                                    {
                                        boolean isCutting1, isCutting2, isCutting3;
                                        Coordinate cuttingPt1 = new Coordinate();
                                        Coordinate cuttingPt2 = new Coordinate();
                                        Coordinate cuttingPt3 = new Coordinate();
                                        Line cuttingLine1=
                                            new Line(nodes[l[isIn[0]]],
                                                    nodes[l[n]],true);
                                        Line cuttingLine2=
                                            new Line(nodes[l[isIn[1]]],
                                                    nodes[l[n]],true);
                                        Line cuttingLine3=
                                            new Line(nodes[l[isIn[2]]],
                                                    nodes[l[n]],true);
                                        isCutting1 = cuttingPt1.cut(P[faNum],
                                                    cuttingLine1,true);
                                        isCutting2 = cuttingPt2.cut(P[faNum],
                                                    cuttingLine2,true);
                                        isCutting3 = cuttingPt3.cut(P[faNum],
                                                    cuttingLine3,true);
                                        if(isCutting1)
                                            {cuttingPtFinal1.assign(cuttingPt1);}
                                        if(isCutting2)
                                            {cuttingPtFinal2.assign(cuttingPt2);}
                                        if(isCutting3)
                                            {cuttingPtFinal3.assign(cuttingPt3);}
                                    }
                                cuttingPtFinal1.add(cuttingPtFinal2);
                                cuttingPtFinal1.add(cuttingPtFinal3);
                                cuttingPtFinal1.mult(1.0/3.0);
```

```
                                            change[l[n]].assign(cuttingPtFinal1);
                                            //change[l[n]].Koor.WriteScr(); // Runtime Info
                                        }

                                    }
                                }

                            }
                        }
                    }
                    for(int i=1; i<nodesNum; i++){nodes[i].assign(change[i]);}

                    // Midpoints projected to Cell surface
                    if(!(adjCell.within(nodes[0])))
                        {
                            Coordinate projMidpt = new Coordinate(), projMidptDef = new Coordinate();
                            int faceOut = 0;
                            for(int i=0; i<plNum; i++)
                                {
                                    Coordinate oRelFace = new Coordinate(nodes[0]); oRelFace.add(P[i].orig,-1.0);
                                    if((Coordinate.spat(P[i].dir[0],P[i].dir[1],oRelFace))<0)
                                        {faceOut++; System.out.println("         "+i);}
                                }
                            if(faceOut == 1)
                                {
                                    Coordinate newMid = new Coordinate(nodes[0]);
                                    for(int i=0; i<plNum; i++)
                                        {
                                            Coordinate oRelFace = new Coordinate(nodes[0]);
                                            oRelFace.add(P[i].orig,-1.0);
                                            Coordinate projection = new Coordinate();
                                            boolean isProj = projection.project(P[i],nodes[0]);
                                            if((Coordinate.spat(P[i].dir[0],P[i].dir[1],oRelFace)<0))
                                                {newMid.assign(projection);}
                                        }
                                    nodes[0].assign(newMid);
                                }
                            if(faceOut == 2)
                                {
                                    Line cuttingL = new Line();
                                    Face toCut[] = new Face[2];
                                    Coordinate projection = new Coordinate();
                                    int ind = 0;
                                    for(int i=0; i<plNum; i++)
                                        {
                                            Coordinate oRelFace = new Coordinate(nodes[0]);
                                            oRelFace.add(P[i].orig,-1.0);

                                            if((Coordinate.spat(P[i].dir[0],P[i].dir[1],oRelFace)<0))
                                                {toCut[ind] = new Face(P[i]); ind++;}
                                        }
                                    Plane p1 = new Plane(toCut[0].orig, toCut[0].dir[0], toCut[0].dir[1]);
                                    Plane p2 = new Plane(toCut[1].orig, toCut[1].dir[0], toCut[1].dir[1]);
                                    cuttingL.cut(p1,p2);
                                    projection.project(cuttingL,nodes[0]);
                                    nodes[0].assign(projection);
                                }

                            //System.out.println("   "+faceOut+"  "+nodes[0].id);
                        }
                }


                public void vectorToMatrix()
                {
                    int numNodes = this.newNodes.size();
                    int numElements = this.newElements.size();

                    if(numNodes>0)
                        {
                            int N = this.nodes.length;
                            Node nodesAfter[] = new Node[N+numNodes];
                            for(int i=0;i<N;i++)
                                {nodesAfter[i] = new Node(this.nodes[i]);}
                            for(int i=0;i<numNodes;i++)
                                {nodesAfter[N+i]=new Node((Node)newNodes.get(i));}
                            this.nodes = new Node[N+numNodes];
```

```
            for(int i=0;i<(N+numNodes);i++)
                {this.nodes[i] = new Node(nodesAfter[i]);}

        }
    if(numElements>0)
        {
            int N = this.elements.length;
            Element elementsAfter[] = new Element[N+numElements];
            for(int i=0;i<N;i++)
                {elementsAfter[i] = new Element(this.elements[i]);}
            for(int i=0;i<numElements;i++)
                {elementsAfter[N+i]=new Element((Element)newElements.get(i));}
            this.elements = new Element[N+numElements];
            for(int i=0;i<(N+numElements);i++)
                {this.elements[i] = new Element(elementsAfter[i]);}
        }
}

/**
   @param from list of elements (with Coordiante information), which necesites new nodes;
   @return list of new nodes which schould be initialised
 */
public static long[][][] newNodesInd(ElementCoordinate in[])
{
    int divis = in.length;

    Vector differents = new Vector();

    // how many new ids do we need ????????????????????????
    for(int i=0;i<divis;i++)
            // CHECK of the new entity (i over all elements and j over the nodes in each element)
        {
            long in_i_Nodes[] = in[i].ids();
            int numNodes = in_i_Nodes.length;
            for(int j=0;j<numNodes;j++)
                {
                    long newId = in_i_Nodes[j];
                    if(newId<0){
                            // not already contained in the old NodeList
                        boolean already = false;   // has this id already been identified
                        for(int ii=0;ii<i;ii++){   // Comparison with all nodes before
                            long in_ii_Nodes[] = in[ii].ids();
                            int numNodesBef = in_ii_Nodes.length;
                            for(int jj=0;jj<numNodesBef;jj++){
                                if(newId==in_ii_Nodes[jj]){already = true;} }}
                                    // all to last row
                        {int ii=i;for(int jj=0;jj<j;jj++){
                            long in_ii_Nodes[] = in[ii].ids();
                            if(newId==in_ii_Nodes[jj]){already = true;}}
                                    // last row
                        if(!already){differents.add(new Long(newId));}
                                    //if no similar found before -> one entitiy more
                        }
                    }
                }
        }

    int numNewNodes = differents.size();
    //System.out.println("  The amount of new nodes is: "+numNewNodes);


    long res[][][] = new long[numNewNodes][][];

    Vector ele = new Vector();
    Vector nod = new Vector();

    for(int l=0;l<numNewNodes;l++)
        {
            //System.out.println(" The counter is at "+l);
            long actId = ((Long)differents.get(l)).longValue();
            for(int i=0;i<divis;i++)
                {
                    long in_i_Nodes[] = in[i].ids();
                    int numNodes = in_i_Nodes.length;
                    //System.out.println(" the number of nodes in this element is "+numNodes);
                    for(int j=0;j<numNodes;j++)
                        {
```

```
                            if(actId == in_i_Nodes[j])
                                {
                                    System.out.println(" the new ids sit in "+i+" "+j);
                                    ele.add(new Long(i));nod.add(new Long(j));
                                }
                        }
                    }
                int actLength = ele.size();
                res[l] = new long[actLength][2];

                for(int i=0;i<actLength;i++)
                    {res[l][i][0] = ((Long)ele.get(i)).longValue();
                     res[l][i][1] = ((Long)nod.get(i)).longValue();}

                ele.setSize(0);
                nod.setSize(0);
            }

        return res;

    }

    public void WriteScr()
    {
        System.out.println(" - Nodes in SphereMesh:");
        for(int i = 0; i<nodes.length; i++)
            {
                Coordinate a = new Coordinate(nodes[i]);
                System.out.print("\n\t - Node "+nodes[i].id+" has coordinate: "); a.WriteScr();
            }
    }



}
```

D Source code for PovRay graphics

The following source was used to generate Fig. 1 on page 1.

```
default {
  texture {
    pigment { rgb <1,0,0> }
  }
}


global_settings {
  adc_bailout 0.003922
  ambient_light <1.0,1.0,1.0>
  assumed_gamma 1.9
  hf_gray_16 off
  irid_wavelength <0.247059,0.176471,0.137255>
  max_intersections 64
  max_trace_level 10
  number_of_waves 10
  radiosity {
    brightness       3.3
    count            100
    distance_maximum 0.0
    error_bound      0.4
    gray_threshold   0.5
    low_error_factor 0.8
    minimum_reuse    0.015
    nearest_count    6
    recursion_limit  1
  }
}

background { color <1.000,1.000,1.000> }

camera {  //  Camera StdCam
  location  <    -0.000,     -0.200,      0.100>
  sky       <    0.00000,     0.00000,     1.00000> // Use right handed-system
  up        <        0.0,         0.0,         1.0> // Where Z is up
  right     <    1.35836,         0.0,         0.0> // Right Vector is adjusted to compensate for
spherical (Moray) vs. planar (POV-Ray) aspect ratio
  angle         1.5000    // Vertical      30.000
  look_at   <      0.000,       0.000,       0.0015>
}

//
// *******  L I G H T S *******
//




/*

light_source {   // Light1
  <0.0, 0.0, 0.0>
  color rgb <1.000, 1.000, 1.000>
  translate  <0.0, -40.0, 20.0>
}  */

#declare Lookfrom =
        <-0.000,     -0.200,      0.100>

light_source {   // Light1
  <0.0, 0.0, 0.0>
  color rgb <1.000, 1.000, 1.000>
  translate  Lookfrom
}


light_source {   // Arealight Arealight001
  <0.0, 0.0, 5.0>
```

```
  color rgb <1.000, 1.000, 1.000>
  area_light <2.000, 0.000, 0.000>, <0.000, 2.000, 0.000>, 3, 3
  adaptive 5
  jitter
}

light_source {   // Arealight Arealight001
  Lookfrom
  color rgb <1.000, 1.000, 1.000>
  area_light <2.000, -2.000, -1.000>, <0.000, 2.000, 0.000>, 3, 3
  adaptive 5
  jitter
}



//
// ********  MATERIALS  *******
//

#include "testfile.inc"
#include "colors.inc"


//
// ********  REFERENCED OBJECTS  *******
//



//
// ********  OBJECTS  *******
//




#declare WinkelNeigung = 0.0;
#declare WinkelDrehung = -1.61;
#declare AbstandEbeneO = 0.000063+0.004;
#declare OnlyLayer = 0;

#declare NormalenVector =
<cos(WinkelDrehung)*sin(WinkelNeigung),sin(WinkelDrehung)*sin(WinkelNeigung),cos(WinkelNeigung)>;



#declare CoarseFrac = union{
#fopen Koords "KoordinatenCoarse.txt" read
#while (defined(Koords))
   #read (Koords, Number, KoorX, KoorY, KoorZ, Radius, Distance)
   #local DistFromCut = vdot(<KoorX, KoorY, KoorZ>-<0,0,AbstandEbeneO>,NormalenVector);
   #if (DistFromCut<-Radius)
     #if (OnlyLayer)
     #else
     sphere {<KoorX, KoorY, KoorZ>,Radius
     material {Coarse}}
     #end
   #else
    #if (DistFromCut>+Radius)
    #else
       #if (OnlyLayer)
       intersection {
          sphere {<KoorX, KoorY, KoorZ>,Radius}
          plane {NormalenVector, AbstandEbeneO}
          plane {-NormalenVector, -AbstandEbeneO}
          material {Coarse}
          }
       #else
       intersection {
          sphere {<KoorX, KoorY, KoorZ>,Radius}
          plane {NormalenVector, AbstandEbeneO}
          material {Coarse}
          }
       #end
    #end
   #end
```

```
#end
}

#declare FineFrac = union{
#fopen Koords "KoordinatenFine.txt" read
#while (defined(Koords))
   #read (Koords, Number, KoorX, KoorY, KoorZ, Radius, Distance)
   #local DistFromCut = vdot(<KoorX, KoorY, KoorZ>-<0,0,AbstandEbeneO>,NormalenVector);
   #if (DistFromCut<-Radius)
     #if (OnlyLayer)
     #else
     sphere {<KoorX, KoorY, KoorZ>,Radius
     material {Fine}}
     #end
   #else
    #if (DistFromCut>+Radius)
    #else
       #if (OnlyLayer)
       intersection {
          sphere {<KoorX, KoorY, KoorZ>,Radius}
          plane {NormalenVector, AbstandEbeneO}
          plane {-NormalenVector, -AbstandEbeneO}
          material {Fine}
          }
       #else
       intersection {
          sphere {<KoorX, KoorY, KoorZ>,Radius}
          plane {NormalenVector, AbstandEbeneO}
          material {Fine}
          }
       #end
    #end
   #end
#end
}

disc { <0,0,0.000063+0.00025>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }

disc { <0,0,0.000063+0.0005>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }

disc { <0,0,0.000063+0.00075>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }

disc { <0,0,0.000063+0.001>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }

disc { <0,0,0.000063+0.00125>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }

disc { <0,0,0.000063+0.0015>, <0,0,1>, 0.0015
 pigment {
            color rgbt <0.0, 0.00, 1.0, 0.3>
          }  }


CoarseFrac
FineFrac
```

E Arrangement coordinates

E.1 Selected cell

The selected cell was is characterized by the following wedge. The list should be regarded as 4 entries of vectors, containing 3 numbers each. The first coordinate is the origin of the wedge, the second and the third entry represent the triangle, which is extruded to a wedge, and the forth entry is the extrusion vector.

-0.0006006 , -0.0008853 , 0.001201 , -0.0004474 , 0.0006629 , -2.2e-05 , 0 , 1.32677e-05 , 0.00039978 , 0.000191455 , 0.000142341 , 0.000395496 , 0 ,

E.2 C++ program to select spheres within cell

Program to extract the spheres interacting of lying within the wedge cell. The purpose of this program is to reduce the load on the Java program, which performs the meshing.

```
//
//
// Program Zelle.cc
//
// Program for extraction of spheres within a given
// pyramide.
//

#include <iostream>
#include <fstream>
#include <cmath>

class Koordinate;
class Vector;
class Matrix;
class Line;

//************************************************
//*          Class: Koordinate                   *
//************************************************

class Koordinate
{
public:

  Koordinate(double, double, double);
  Koordinate(const Koordinate&);
  ~Koordinate();

  double x;
  double y;
  double z;

  void Set(double, double, double);
  void Set(const Koordinate&);
  void WriteOut();
  void Add(Koordinate, Koordinate);
  void Add(Koordinate, Vector);
  void Add(Vector, Koordinate);
  void Mult(double);

  void Mult(Matrix);
  void Mult(Matrix,Vector);
  void Mult(Matrix,Koordinate);

  bool IsInCubus(Koordinate, Vector, Vector, Vector, Koordinate&);
  bool IsAboveSurface(Koordinate, Vector, Vector, Vector, double&);

  double DistanceFromLine(Line Linie);
};
```

```
//*************************************************
//*          Class: Vector                        *
//*************************************************

class Vector : public Koordinate
{
public:

  Vector(double, double, double);
  Vector(const Vector&);
  ~Vector();

  void Set(double, double, double);
  void Set(const Vector&);
  void WriteOut();
  void Make(Koordinate, Koordinate);
  void Cross(Vector, Vector);
  double Dot(Vector);
  double Length();
  void Add(Vector, Vector);
  void Add(Koordinate, Vector);
  void Add(Vector, Koordinate);
  void Add(Koordinate, Koordinate);
  //void Mult(double);
  void Normalize();
};


//*************************************************
//*          Class: Matrix                        *
//*************************************************

class Matrix
{
public:

  Matrix(double, double, double, double, double, double, double, double, double);
  Matrix(Vector, Vector, Vector);
  Matrix(const Matrix&);
  ~Matrix();

  double x[3][3];

  void Set(double, double, double, double, double, double, double, double, double);
  void Set(Vector, Vector, Vector);
  void Set(Matrix);

  void Mult(Matrix, bool);
  void Mult(Matrix,Matrix);

  void Inv(Matrix);
  void Inv();

  void WriteOut();
};

//*************************************************
//*          Class: Line                          *
//*************************************************

class Line
{
public:

  Line(double, double, double, double, double, double);
  Line(const Line&);
  ~Line();

  Koordinate Origin;
  Vector Direction;

  void Set(double, double, double, double, double, double);
  void Set(const Line&);
  void WriteOut();

};
```

```
//************************************************
//*          Class: Plane                        *
//************************************************

class Plane
{
public:

  Plane(double, double, double, double, double, double);
  Plane(Koordinate, Vector);
  Plane(Koordinate, Vector, Vector);
  Plane(const Plane&);
  ~Plane();

  double t;

  Koordinate Origin;
  Vector Normal;

  void Set(double, double, double, double, double, double);
  void Set(Koordinate, Vector);
  void Set(Koordinate, Vector, Vector);
  void Set(const Plane&);
  void WriteOut();
  void Cut3Planes(Plane, Plane, Koordinate&);
  void CutPlaneLine(Line, Koordinate&);
  void CutPlanePlane(Plane, Line&);
};



//************************************************
//*          Class: SphereKoor                   *
//************************************************

class SphereKoor
{
public:

  SphereKoor();
  ~SphereKoor();

  int Num;
  double x;
  double y;
  double z;
  double r;

  double Distance(SphereKoor);
  void ReadIn(std::fstream&);
};

//************************************************
//*          Class: SphereDaten                  *
//************************************************

class SphereDaten : public SphereKoor
{
public:

  SphereDaten();
  ~SphereDaten();

  int NachbAnz;
  double NaGesAbst;
  int NachbNum[12];
  double NachbAbst[12];
  void Naeher(SphereKoor);
  void NachbZaehlen();
  void Initialisierung(SphereKoor);
  void WriteOut(std::fstream&);
  void ReadIn(std::fstream&);
};

//************************************************
//*          Class: Cell                         *
```

```
//************************************************

class Cell
{

  // Defines a cell containing speres:
  //
  // There are two cases:
  //  *  The cell is a three-side prism: (Prisma4 = false)
  //        Side[0] defines the elevation of the prism
  //        Side[1] and Side [2] define the base triangle
  //  *  The cell is a four-side prism: (Prisma4 = true)
  //        All three sides define a generally sheared cubus
  //
  //  The inner cell defines the space within all sphere-midpoints
  //  are located, which define spheres wich laz completely
  //  in the given cell.
  //  The outer cell defines the space within all sphere-midpoints
  //  are located, which define spheres which lay copletely or
  //  parially in the given cell.


public:

  bool Prisma4; // True if 4-side prisma, otherwise 3-side prisma

  Cell();
  ~Cell();

  Koordinate Origin;
  Koordinate OuterOrigin;
  Koordinate InnerOrigin;

  Vector Side[3];
  Vector InnerSide[3];
  Vector OuterSide[3];


  void Create(Koordinate, Vector, Vector, Vector, bool, double);
  void WriteOut();
  void ReadIn(std::fstream&);
};




//************************************************
//*          Class: CellSpheres                  *
//************************************************

class CellSpheres : SphereDaten
{
public:

  double CuttingPlane[6];

  void Initialisierung(SphereDaten);
  bool InCellTest(Cell, SphereDaten);
  void WriteOut(std::fstream&);

};


//************************************************
//*          ClassFunction: Con-/De-structors    *
//************************************************


Koordinate::Koordinate(double cx = 0.0, double cy = 0.0, double cz = 0.0)
{
  x = cx; y = cy; z = cz;
}

Koordinate::Koordinate(const Koordinate& Koor)
{
  this->x = Koor.x;
  this->y = Koor.y;
```

```
  this->z = Koor.z;
}

Koordinate::~Koordinate()
{
}

Vector::Vector(double cx = 0.0, double cy = 0.0, double cz = 0.0)
{
  x = cx; y = cy; z = cz;
}

Vector::Vector(const Vector& Vec)
{
  this->x = Vec.x;
  this->y = Vec.y;
  this->z = Vec.z;
}

Vector::~Vector()
{
}

Line::Line(double ox = 0.0, double oy = 0.0, double oz = 0.0, double dx = 0.0, double dy = 0.0, double
dz = 0.0)
{
 Origin.x = ox; Origin.y = oy; Origin.z = oz;
 Direction.x = dx; Direction.y = dy; Direction.z = dz;
}

Line::Line(const Line& Lin)
{
  this->Origin.x = Lin.Origin.x;
  this->Origin.y = Lin.Origin.y;
  this->Origin.z = Lin.Origin.z;

  this->Direction.x = Lin.Direction.x;
  this->Direction.y = Lin.Direction.y;
  this->Direction.z = Lin.Direction.z;
}

Line::~Line()
{
}

Plane::Plane(double ox = 0.0, double oy = 0.0, double oz = 0.0, double nx = 0.0, double ny = 0.0,
double nz = 0.0)
{
 Origin.x = ox; Origin.y = oy; Origin.z = oz;
 Normal.x = nx; Normal.y = ny; Normal.z = nz;
}

Plane::Plane(Koordinate O, Vector N)
{
 Origin.x = O.x; Origin.y = O.y; Origin.z = O.z;
 Normal.x = N.x; Normal.y = N.y; Normal.z = N.z;
}

Plane::Plane(Koordinate O, Vector Side1, Vector Side2)
{
 Origin.x = O.x; Origin.y = O.y; Origin.z = O.z;
 Normal.Cross(Side1, Side2);
}

Plane::Plane(const Plane& Pla)
{
  this->Origin.x = Pla.Origin.x;
  this->Origin.y = Pla.Origin.y;
  this->Origin.z = Pla.Origin.z;

  this->Normal.x = Pla.Normal.x;
  this->Normal.y = Pla.Normal.y;
  this->Normal.z = Pla.Normal.z;
}

Plane::~Plane()
{
```

```
}


SphereKoor::SphereKoor():
  Num(0), x(0.0), y(0.0), z(0.0), r(0.0)
{
}

SphereKoor::~SphereKoor()
{
}

SphereDaten::SphereDaten()
{
 for (int i = 0; i < 12; ++i)
    {
      NachbAbst[i] = -1.0;
      NachbNum[i] = 0;
    };
 NaGesAbst = -1.0;
 NachbAnz = -1;
}

SphereDaten::~SphereDaten()
{
}

Cell::Cell()
{
}

Cell::~Cell()
{
}

Matrix::Matrix(double x11 = 0.0, double x12 = 0.0, double x13 = 0.0, double x21 = 0.0, double x22 =
0.0, double x23 = 0.0, double x31 = 0.0, double x32 = 0.0, double x33 = 0.0)
{
  x[0][0] = x11; x[0][1] = x21; x[0][2] = x31;
  x[1][0] = x12; x[1][1] = x22; x[1][2] = x32;
  x[2][0] = x13; x[2][1] = x23; x[2][2] = x33;
}

Matrix::Matrix(Vector v1, Vector v2, Vector v3)
{
  Set(v1.x, v1.y, v1.z, v2.x, v2.y, v2.z, v3.x, v3.y, v3.z);
}

Matrix::Matrix(const Matrix& M)
{
  this->x[0][0] = M.x[0][0];
  this->x[0][1] = M.x[0][1];
  this->x[0][2] = M.x[0][2];

  this->x[1][0] = M.x[1][0];
  this->x[1][1] = M.x[1][1];
  this->x[1][2] = M.x[1][2];

  this->x[2][0] = M.x[2][0];
  this->x[2][1] = M.x[2][1];
  this->x[2][2] = M.x[2][2];
}

Matrix::~Matrix()
{
}


//************************************************
//*           Set Function for different calsses    *
//************************************************


void Koordinate::Set(double cx = 0.0f, double cy = 0.0f, double cz = 0.0f)
{
  x = cx; y = cy; z = cz;
}
```

```
void Koordinate::Set(const Koordinate& Koor)
{
  this->x = Koor.x;
  this->y = Koor.y;
  this->z = Koor.z;
}

void Vector::Set(double cx = 0.0f, double cy = 0.0f, double cz = 0.0f)
{
  x = cx; y = cy; z = cz;
}

void Vector::Set(const Vector& Vec)
{
  this->x = Vec.x;
  this->y = Vec.y;
  this->z = Vec.z;
}

void Line::Set(double ox = 0.0f, double oy = 0.0f, double oz = 0.0f, double dx = 0.0f, double dy =
0.0f, double dz = 0.0f)
{
 Origin.x = ox; Origin.y = oy; Origin.z = oz;
 Direction.x = dx; Direction.y = dy; Direction.z = dz;
}


void Line::Set(const Line& Lin)
{
  this->Origin.x = Lin.Origin.x;
  this->Origin.y = Lin.Origin.y;
  this->Origin.z = Lin.Origin.z;

  this->Direction.x = Lin.Direction.x;
  this->Direction.y = Lin.Direction.y;
  this->Direction.z = Lin.Direction.z;
}

void Plane::Set(double ox = 0.0f, double oy = 0.0f, double oz = 0.0f, double nx = 0.0f, double ny =
0.0f, double nz = 0.0f)
{
 Origin.x = ox; Origin.y = oy; Origin.z = oz;
 Normal.x = nx; Normal.y = ny; Normal.z = nz;
}

void Plane::Set(Koordinate O, Vector N)
{
 Origin.x = O.x; Origin.y = O.y; Origin.z = O.z;
 Normal.x = N.x; Normal.y = N.y; Normal.z = N.z;
}

void Plane::Set(Koordinate O, Vector Side1, Vector Side2)
{
 Origin.x = O.x; Origin.y = O.y; Origin.z = O.z;
 Normal.Cross(Side1, Side2);
}

void Plane::Set(const Plane& Pla)
{
  this->Origin.x = Pla.Origin.x;
  this->Origin.y = Pla.Origin.y;
  this->Origin.z = Pla.Origin.z;

  this->Normal.x = Pla.Normal.x;
  this->Normal.y = Pla.Normal.y;
  this->Normal.z = Pla.Normal.z;
}

void Matrix::Set(double x11, double x12, double x13, double x21, double x22, double x23, double x31,
double x32, double x33)
{
  x[0][0] = x11; x[0][1] = x21; x[0][2] = x31;
  x[1][0] = x12; x[1][1] = x22; x[1][2] = x32;
  x[2][0] = x13; x[2][1] = x23; x[2][2] = x33;
}
```

```
void Matrix::Set(Vector v1, Vector v2, Vector v3)
{
  Set(v1.x, v1.y, v1.z, v2.x, v2.y, v2.z, v3.x, v3.y, v3.z);
}

void Matrix::Set(Matrix M)
{
  this->x[0][0] = M.x[0][0];
  this->x[0][1] = M.x[0][1];
  this->x[0][2] = M.x[0][2];

  this->x[1][0] = M.x[1][0];
  this->x[1][1] = M.x[1][1];
  this->x[1][2] = M.x[1][2];

  this->x[2][0] = M.x[2][0];
  this->x[2][1] = M.x[2][1];
  this->x[2][2] = M.x[2][2];
}

//*************************************************
//*          ClassProzedur: MakeVector            *
//*************************************************

void Vector::Make(Koordinate Anfang, Koordinate Ende)
{
  x = Ende.x - Anfang.x;
  y = Ende.y - Anfang.y;
  z = Ende.z - Anfang.z;
}

//*************************************************
//*          ClassProzedur: Add                   *
//*************************************************

void Vector::Add(Vector Vec1, Vector Vec2)
{
  x = Vec1.x + Vec2.x;
  y = Vec1.y + Vec2.y;
  z = Vec1.z + Vec2.z;
}

void Vector::Add(Koordinate Koo, Vector Vec)
{
  x = Koo.x + Vec.x;
  y = Koo.y + Vec.y;
  z = Koo.z + Vec.z;
}

void Vector::Add(Vector Vec, Koordinate Koo)
{
  x = Koo.x + Vec.x;
  y = Koo.y + Vec.y;
  z = Koo.z + Vec.z;
}

void Vector::Add(Koordinate Koo1, Koordinate Koo2)
{
  x = Koo1.x + Koo2.x;
  y = Koo1.y + Koo2.y;
  z = Koo1.z + Koo2.z;
}

void Koordinate::Add(Koordinate Koo1, Koordinate Koo2)
{
  x = Koo1.x + Koo2.x;
  y = Koo1.y + Koo2.y;
  z = Koo1.z + Koo2.z;
}

void Koordinate::Add(Koordinate Koo, Vector Vec)
{
  x = Koo.x + Vec.x;
  y = Koo.y + Vec.y;
  z = Koo.z + Vec.z;
}
```

```
void Koordinate::Add(Vector Vec, Koordinate Koo)
{
  x = Koo.x + Vec.x;
  y = Koo.y + Vec.y;
  z = Koo.z + Vec.z;
}


//*************************************************
//*          ClassProzedur: Mult                  *
//*************************************************

/*void Vector::Mult(double Multiplier)
{
  x *= Multiplier;
  y *= Multiplier;
  z *= Multiplier;
}*/

void Koordinate::Mult(double Multiplier)
{
  x *= Multiplier;
  y *= Multiplier;
  z *= Multiplier;
}

void Koordinate::Mult(Matrix M)
{
  double a[3], b[3];

  a[0]=x; a[1]=y; a[2]=z;

  for (int i=0; i<3; i++)
    {
      for (int j=0; j<3; j++)
        {
          b[i] += M.x[i][j] * a[j];
        };
    };

  x=b[0]; y=b[1]; z=b[2];
}

void Koordinate::Mult(Matrix M, Vector V)
{
  Vector Vl(V);
  Vl.Mult(M);
  x=Vl.x; y=Vl.y; z=Vl.z;
}

void Koordinate::Mult(Matrix M, Koordinate K)
{
  Koordinate Kl(K);
  Kl.Mult(M);
  x=Kl.x; y=Kl.y; z=Kl.z;
}

void Matrix::Mult(Matrix M, bool First)
{

  // First = True if Calss Member is fist Marix in Multiplication

  Matrix A;

  if (First)
    {for (int i=0; i<3; i++)
      { for (int j=0; j<3; j++)
        { for (int k=0; k<3; k++)
          { A.x[i][j] += x[i][k]*M.x[k][j];
          };
        };
      };
    }
  else
    {for (int i=0; i<3; i++)
      { for (int j=0; j<3; j++)
        { for (int k=0; k<3; k++)
```

```
         { A.x[i][j] += x[k][j]*M.x[i][k];
         };
       };
      };
    };

  for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = A.x[i][j];}; };
}

void Matrix::Mult(Matrix M1, Matrix M2)
{
  M1.Mult(M2,true);
  for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = M1.x[i][j];}; };
}

//*************************************************
//*          ClassProzedur: Inverse               *
//*************************************************

void Matrix::Inv()
{
  double A[3][3];
  for (int i=1; i<3; i++){for (int j=1; j<3; j++){A[i][j]=0.0;};};

  A[0][0] = ((-x[1][2])*x[2][1] + x[1][1]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2]
+ x[0][0]*x[1][1]*x[2][2]);
  A[0][1] = (x[0][2]*x[2][1] - x[0][1]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] + x[0][1]*x[1][2]*x[2][0]
+ x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2] +
x[0][0]*x[1][1]*x[2][2]);
  A[0][2] = ((-x[0][2])*x[1][1] + x[0][1]*x[1][2])/((-x[0][2])* x[1][1]*x[2][0] +
x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2]
+ x[0][0]*x[1][1]*x[2][2]);

  A[1][0] = (x[1][2]*x[2][0] - x[1][0]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] + x[0][1]*x[1][2]*x[2][0]
+ x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2] +
x[0][0]*x[1][1]*x[2][2]);
  A[1][1] = ((-x[0][2])*x[2][0] + x[0][0]*x[2][2])/((-x[0][2])*x[1][1]*x[2][0] +
x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2]
+ x[0][0]*x[1][1]*x[2][2]);
  A[1][2] = (x[0][2]*x[1][0] - x[0][0]*x[1][2])/((-x[0][2])*x[1][1]*x[2][0] + x[0][1]*x[1][2]*x[2][0]
+ x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2] +
x[0][0]*x[1][1]*x[2][2]);

  A[2][0] = ((-x[1][1])*x[2][0] + x[1][0]*x[2][1])/((-x[0][2])*x[1][1]*x[2][0] +
x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2]
+ x[0][0]*x[1][1]*x[2][2]);
  A[2][1] = (x[0][1]*x[2][0] - x[0][0]*x[2][1])/((-x[0][2])*x[1][1]*x[2][0] + x[0][1]*x[1][2]*x[2][0]
+ x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2] +
x[0][0]*x[1][1]*x[2][2]);
 A[2][2] = ((-x[0][1])*x[1][0] + x[0][0]*x[1][1])/((-x[0][2])*x[1][1]*x[2][0] +
x[0][1]*x[1][2]*x[2][0] + x[0][2]*x[1][0]*x[2][1] - x[0][0]*x[1][2]*x[2][1] - x[0][1]*x[1][0]*x[2][2]
+ x[0][0]*x[1][1]*x[2][2]);

  for (int i=0; i<3; i++) { for (int j=0; j<3; j++) { x[i][j] = A[i][j]; }; };
}

void Matrix::Inv(Matrix M)
{
  M.Inv();
  for (int i=0; i<3; i++){ for (int j=0; j<3; j++) {x[i][j]=M.x[i][j]; }; };
}

//*************************************************
//*          ClassProzedur: Normalize             *
//*************************************************

void Vector::Normalize()
{
  double laenge;

  laenge = sqrt(pow(x,2)+pow(y,2)+pow(z,2));
  Mult(1/laenge);
}

//*************************************************
//*          ClassProzedur: Cross                 *
```

```
//*************************************************

void Vector::Cross(Vector Vec1, Vector Vec2)
{
  x = (Vec1.y * Vec2.z) - (Vec1.z * Vec2.y);
  y = (Vec1.z * Vec2.x) - (Vec1.x * Vec2.z);
  z = (Vec1.x * Vec2.y) - (Vec1.y * Vec2.x);
}

//*************************************************
//*          ClassProzedur: Dot                   *
//*************************************************

double Vector::Dot(Vector Vec)
{
  return ((x * Vec.x) + (y * Vec.y) + (z * Vec.z));
}

//*************************************************
//*          ClassProzedur: Length                *
//*************************************************

double Vector::Length()
{
  return sqrt(x*x + y*y + z*z);
}


//*************************************************
//*          Function: CutPlaneLine               *
//*************************************************

void Plane::CutPlaneLine(Line Linie, Koordinate& Res)
{
  Vector pl1, pl2;
  double t;
  double Koo[3], Nor[3];
  double Gross = 0.0f;
  int GrossNum, ZweitNum, DrittNum;

  Koo[0] = Normal.x; Koo[1] = Normal.y; Koo[2] = Normal.z;

  for (int i = 0; i < 3; i++) {if ( abs(Koo[i]) > abs(Gross) ) { Gross = Koo[i]; GrossNum = i; }; };
  ZweitNum = int(floor((fmod((GrossNum+1),3))+0.1));
  DrittNum = int(floor((fmod((GrossNum+2),3))+0.1));

  Nor[GrossNum] = - Koo[ZweitNum];
  Nor[ZweitNum] = Koo[GrossNum];
  Nor[DrittNum] = 0;

  pl1.x = Nor[0]; pl1.y = Nor[1]; pl1.z = Nor[2];

  pl2.Cross(Normal,pl1);

  t = (Linie.Origin.z*pl1.y*pl2.x - Origin.z*pl1.y*pl2.x - Linie.Origin.y*pl1.z*pl2.x +
Origin.y*pl1.z*pl2.x - Linie.Origin.z*pl1.x*pl2.y + Origin.z*pl1.x*pl2.y + Linie.Origin.x*pl1.z*pl2.y
- Origin.x*pl1.z*pl2.y + Linie.Origin.y*pl1.x*pl2.z - Origin.y*pl1.x*pl2.z -
Linie.Origin.x*pl1.y*pl2.z + Origin.x*pl1.y*pl2.z)/(-Linie.Direction.z*pl1.y*pl2.x +
Linie.Direction.y*pl1.z*pl2.x + Linie.Direction.z*pl1.x*pl2.y - Linie.Direction.x*pl1.z*pl2.y -
Linie.Direction.y*pl1.x*pl2.z + Linie.Direction.x*pl1.y*pl2.z);

    Res.x = Linie.Origin.x + t*Linie.Direction.x;
    Res.y = Linie.Origin.y + t*Linie.Direction.y;
    Res.z = Linie.Origin.z + t*Linie.Direction.z;
}

//*************************************************
//*          Function: CutPlanePlane              *
//*************************************************

void Plane::CutPlanePlane(Plane Ebene, Line& Schnitt)
{
  Vector a, b;

  Koordinate Midpt;

  if (Origin.x != Ebene.Origin.x || Origin.y != Ebene.Origin.y || Origin.z != Ebene.Origin.z)
```

```
    {
      Line c;
      a.Make(Origin, Ebene.Origin);
      if ( abs(a.Dot(Normal)) < abs(a.Dot(Ebene.Normal)))
      {
       b.Cross(Normal, a);
       c.Origin.Set(Origin);
       c.Direction.Cross(Normal, b);
       Ebene.CutPlaneLine(c, Midpt);
      }
      else
      {
       b.Cross(Normal, a);
       c.Origin.Set(Ebene.Origin);
       c.Direction.Cross(Ebene.Normal, b);
       CutPlaneLine(c, Midpt);
      };
    }
    else
      {
       Midpt.Set(Origin);
      };
  Schnitt.Origin.Set(Midpt);
  a.Cross(Normal,Ebene.Normal);
  Schnitt.Direction.Set(a);
}

//************************************************
//*          Function: Cut3Planes                *
//************************************************

void Plane::Cut3Planes(Plane Ebene2, Plane Ebene3, Koordinate& SchnPkt)
{
  Line SchnGerade;

  CutPlanePlane(Ebene2, SchnGerade);
  Ebene3.CutPlaneLine(SchnGerade, SchnPkt);
}


//************************************************
//*       WriteOut Functions for different classes  *
//************************************************

void Koordinate::WriteOut()
{
  cout << " ( " << x << " , " << y << " , " << z << " ) " ;
}

void Vector::WriteOut()
{
  cout << " ( " << x << " , " << y << " , " << z << " ) " ;
}

void Plane::WriteOut()
{
  Origin.WriteOut(); cout << " + t x ";  Normal.WriteOut();
}

void Line::WriteOut()
{
  Origin.WriteOut(); cout << " + t * ";  Direction.WriteOut();
}

void Cell::WriteOut()
{
  cout << " \n Cell-Specifications: \n\n";
  cout << "  Origins: Given: "; Origin.WriteOut(); cout << "\n";
  cout << "           Outer: "; OuterOrigin.WriteOut(); cout << "\n";
  cout << "           Inner: "; InnerOrigin.WriteOut(); cout << "\n\n";
  cout << "  Side-    Given: "; for (int i = 0; i < 3; i++) {Side[i].WriteOut();}; cout << "\n";
  cout << "  Vectors: Outer: "; for (int i = 0; i < 3; i++) {OuterSide[i].WriteOut();}; cout << "\n";
  cout << "           Inner: "; for (int i = 0; i < 3; i++) {InnerSide[i].WriteOut();}; cout <<
"\n\n";
}

void SphereDaten::WriteOut(std::fstream& AufSchr)
```

```
{
  AufSchr << Num << "," << x << "," << y << "," << z << "," << r << ",\t";
  AufSchr << NachbAnz << "," << NaGesAbst << ",\t";
  for (int i = 0; i < 12; ++i)
    {
      AufSchr << NachbNum[i] << "," << NachbAbst[i] << ",";
    };
  AufSchr << std::endl;
}

void CellSpheres::WriteOut(std::fstream& AufSchr)
{
  AufSchr << Num << "," << x << "," << y << "," << z << "," << r << ",\t";
  AufSchr << NachbAnz << "," << NaGesAbst << ",\t";
  for (int i = 0; i < 12; ++i)
    {
      AufSchr << NachbNum[i] << "," << NachbAbst[i] << ",";
    };
  for (int i = 0; i < 6; ++i)
    {
      AufSchr << CuttingPlane[i] << "," ;
    };
  AufSchr << std::endl;
}

void Matrix::WriteOut()
{
  cout << "\n";
  for (int i=0; i<3; i++)
    {
      cout << " | ";
      for (int j=0; j<3; j++)
        {
          cout << " " << x[j][i] << "  ";
        };
      cout << " | \n";
    };
}



//**************************************************
//*          Cell::Create                          *
//**************************************************

void Cell::Create(Koordinate Ursprung, Vector Seite1, Vector Seite2, Vector Seite3, bool Prism4,
double Radius)
{
  Plane Ebene[3];
  Plane EbeneA[3];
  Plane EbeneI[3];
  Plane SeitenSchnittEbeneA, SeitenSchnittEbeneI;
  Line SeitenLinieA, SeitenLinieI;
  Koordinate SeitenSchnittPunktA, SeitenSchnittPunktI;
  Koordinate Pkt1, Pkt2, Pkt3;
  double ToSide;
  Vector Norm, Differenz, NegSeite;
  int i2,i3;

  Prisma4 = Prism4;

  Origin.Set(Ursprung);    // Initialisierung der Grundzelle
  Side[0].Set(Seite1);
  Side[1].Set(Seite2);
  Side[2].Set(Seite3);

  for (int i = 0 ; i>3 ; i++ ) { Ebene[i].Origin.Set(Ursprung); };    // Zuweisung des Ursprungs zur
Grundebenen

  for (int i = 0 ; i<3 ; i++ )
    {
      Ebene[i].Origin.Set(Origin);
      i2 = int(floor((fmod((i+1),3))+0.1));
      i3 = int(floor((fmod((i+2),3))+0.1));
      Norm.Cross(Side[i],Side[i2]); Norm.Normalize(); Ebene[i].Normal.Set(Norm); // Zuweisung des
Normalenvectors
```

```
      if (Norm.Dot(Side[i3]) > 0)                                          // Aussere -- Innere
Ebene
        {
          Norm.Mult(-1);
        };
      Norm.Mult(Radius);
      EbeneA[i].Origin.Add(Ebene[i].Origin, Norm); EbeneA[i].Normal.Set(Ebene[i].Normal);
      Norm.Mult(-1);
      EbeneI[i].Origin.Add(Ebene[i].Origin, Norm); EbeneI[i].Normal.Set(Ebene[i].Normal);
    }

  EbeneA[0].Cut3Planes(EbeneA[1], EbeneA[2], OuterOrigin);
  EbeneI[0].Cut3Planes(EbeneI[1], EbeneI[2], InnerOrigin);


  if (Prism4 == true)
    {
      for(int i = 0; i < 3; i++)
        {
          SeitenLinieA.Origin.Set(OuterOrigin); SeitenLinieA.Direction.Set(Side[i]);
          SeitenLinieI.Origin.Set(InnerOrigin); SeitenLinieI.Direction.Set(Side[i]);
          Pkt1.Add(Origin,Side[i]);
          i2 = int(floor((fmod((i+1),3))+0.1));
          i3 = int(floor((fmod((i+2),3))+0.1));
          Norm.Cross(Side[i2],Side[i3]);
          SeitenSchnittEbeneA.Normal.Set(Norm);
          SeitenSchnittEbeneI.Normal.Set(Norm);
          if ( Norm.Dot(Side[i]) < 0 )
            {
              Norm.Mult(-1.0);
            };
          Norm.Normalize();
          Norm.Mult(Radius);
          SeitenSchnittEbeneA.Origin.Add(Pkt1,Norm);
          Norm.Mult(-1.0);
          SeitenSchnittEbeneI.Origin.Add(Pkt1,Norm);
          SeitenSchnittEbeneA.CutPlaneLine(SeitenLinieA, Pkt2);
          SeitenSchnittEbeneI.CutPlaneLine(SeitenLinieI, Pkt3);
          OuterSide[i].Make(OuterOrigin, Pkt2);
          InnerSide[i].Make(InnerOrigin, Pkt3);
        }
    }
  else
    {
      SeitenLinieA.Origin.Set(OuterOrigin); SeitenLinieA.Direction.Set(Side[0]);
      SeitenLinieI.Origin.Set(InnerOrigin); SeitenLinieI.Direction.Set(Side[0]);
      Pkt1.Add(Origin,Side[0]);
      Norm.Cross(Side[1],Side[2]);
      SeitenSchnittEbeneA.Normal.Set(Norm);
      SeitenSchnittEbeneI.Normal.Set(Norm);
      if ( Norm.Dot(Side[0]) < 0 )
        {
          Norm.Mult(-1.0);
        };
      Norm.Normalize();
      Norm.Mult(Radius);
      SeitenSchnittEbeneA.Origin.Add(Pkt1,Norm);
      Norm.Mult(-1.0);
      SeitenSchnittEbeneI.Origin.Add(Pkt1,Norm);
      SeitenSchnittEbeneA.CutPlaneLine(SeitenLinieA, Pkt2);
      SeitenSchnittEbeneI.CutPlaneLine(SeitenLinieI, Pkt3);
      OuterSide[0].Make(OuterOrigin, Pkt2);
      InnerSide[0].Make(InnerOrigin, Pkt3);

      NegSeite.Set(Side[2]);
      NegSeite.Mult(-1.0);
      Differenz.Add( Side[1], NegSeite );
      Norm.Cross(Side[0],Differenz);
      SeitenSchnittEbeneA.Normal.Set(Norm);
      SeitenSchnittEbeneI.Normal.Set(Norm);
      if ( Norm.Dot(Side[1]) < 0 )
        {
          Norm.Mult(-1.0);
        };
      Norm.Normalize();
      Norm.Mult(Radius);
      Pkt1.Add(Origin,Side[2]);
```

```
      SeitenSchnittEbeneA.Origin.Add(Pkt1,Norm);
      Norm.Mult(-1.0);
      SeitenSchnittEbeneI.Origin.Add(Pkt1,Norm);
      for(int i = 1; i < 3; i++)
        {
          i2 = int(floor((fmod((i+1),3))+0.1));
          i3 = int(floor((fmod((i+2),3))+0.1));
          SeitenLinieA.Origin.Set(OuterOrigin); SeitenLinieA.Direction.Set(Side[i]);
          SeitenLinieI.Origin.Set(InnerOrigin); SeitenLinieI.Direction.Set(Side[i]);
          SeitenSchnittEbeneA.CutPlaneLine(SeitenLinieA, Pkt2);
          SeitenSchnittEbeneI.CutPlaneLine(SeitenLinieI, Pkt3);
          OuterSide[i].Make(OuterOrigin, Pkt2);
          InnerSide[i].Make(InnerOrigin, Pkt3);
        }
    };
}


//*************************************************
//*          ClassFunction: Distance              *
//*************************************************

double SphereKoor::Distance(SphereKoor Relat)
{
  return sqrt(pow(Relat.x - x, 2)+pow(Relat.y - y, 2)+pow(Relat.z - z, 2)) - (Relat.r + r);
};



//*************************************************
//*          ClassProzedur: Naeher                *
//*************************************************

void SphereDaten::Naeher(SphereKoor ZuVergleichen)
{
  double LocalAbst;
  int i;

  LocalAbst = Distance(ZuVergleichen);

  i = 12;

  while (((LocalAbst < NachbAbst[i-1]) || (NachbAbst[i-1] == -1.0)) && i > 0 && Num !=
ZuVergleichen.Num)
    {
      --i;
    };
  if (i < 12)
    {
      for (int j = 11; j > i; --j)
       {
        NachbNum[j] = NachbNum[j-1];
        NachbAbst[j] = NachbAbst[j-1];
       };
     NachbNum[i] = ZuVergleichen.Num;
     NachbAbst[i] = LocalAbst;

    };
};

//*************************************************
//*          ClassProzedur: NachbZaehlen          *
//*************************************************

void SphereDaten::NachbZaehlen()
{
  double Toleranz = 5.0e-6;
  int j = 0;
  double GesAbst = 0.0f;

  for (int i = 0; i < 12; ++i)
    {
      if (NachbAbst[i] < Toleranz)
        {
          ++j;
        }
      GesAbst += NachbAbst[i];
```

```
    };
  NachbAnz = j;
  NaGesAbst = GesAbst;
}

//************************************************
//*          ClassProzedur: Initialisierung      *
//************************************************

void SphereDaten::Initialisierung(SphereKoor ZuUeberladen)
{
  Num = ZuUeberladen.Num;
  x = ZuUeberladen.x;
  y = ZuUeberladen.y;
  z = ZuUeberladen.z;
  r = ZuUeberladen.r;

  for (int i = 0; i < 12; ++i)
    {
      NachbAbst[i] = -1.0f;
      NachbNum[i] = 0;
    };

  NachbAnz = -1;
  NaGesAbst = -1.0f;
}



//************************************************
//*          ReadIn Prozeduren                   *
//************************************************

void SphereKoor::ReadIn(std::fstream& VonLesen)
{
  char c;
  double x1, x2;

  VonLesen >> Num >> c >> x >> c >> y >> c >> z >> c >> r >> c >> x2  >> c;
}

void SphereDaten::ReadIn(std::fstream& VonLesen)
{
  char c;

  VonLesen >> Num >> c >> x >> c >> y >> c >> z >> c >> r >> c >> NachbAnz >> c >> NaGesAbst >> c;
  for (int i = 0; i < 12; ++i)
    {
      VonLesen >> NachbNum[i] >> c >> NachbAbst[i] >> c;
    };
}

void Cell::ReadIn(std::fstream& VonLesen)
{
  char c;

  VonLesen >> Origin.x >> c >> Origin.y >> c >> Origin.z >> c ;
  for (int i = 0; i < 3; ++i)
    {
      VonLesen >> Side[i].x >> c >> Side[i].y >> c >> Side[i].z >> c;
    };
  VonLesen >> Prisma4 >> c;
}

//************************************************
//*          CellSpheres :: Initialisierung      *
//************************************************

void CellSpheres::Initialisierung(SphereDaten ZuUeberladen)
{
  Num = ZuUeberladen.Num;
  x = ZuUeberladen.x;
  y = ZuUeberladen.y;
  z = ZuUeberladen.z;
  r = ZuUeberladen.r;

  for (int i = 0; i < 12; ++i)
```

```
    {
      NachbAbst[i] = ZuUeberladen.NachbAbst[i];
      NachbNum[i] = ZuUeberladen.NachbNum[i];
    };

  NachbAnz = ZuUeberladen.NachbAnz;
  NaGesAbst = ZuUeberladen.NaGesAbst;

  for (int i = 0; i < 6; ++i)
    {
      CuttingPlane[i] = -1;
    };
}

//************************************************
//*         IsInCubus and IsAboveSurface          *
//************************************************

bool Koordinate::IsInCubus(Koordinate CellOrig, Vector Side1, Vector Side2, Vector Side3, Koordinate&
Wo)
{
  Koordinate OriginCopy(x,y,z);
  Matrix Transf(Side1, Side2, Side3);
  Transf.Inv();

  CellOrig.Mult(-1);
  Wo.Add(OriginCopy, CellOrig);

  Wo.Mult(Transf);

if (Wo.x >= 0 && Wo.x <= 1 &&
    Wo.y >= 0 && Wo.y <= 1 &&
    Wo.z >= 0 && Wo.z <= 1)
  {return true;}
 else {return false;};
}

bool Koordinate::IsAboveSurface(Koordinate CellOrig, Vector Side1, Vector Side2, Vector Side3, double&
Distance)
{
  Vector NormalenVec;
  double Richtung;
  Koordinate SphereLocation(x,y,z), EinheitsKoor;

  NormalenVec.Cross(Side1, Side2);
  NormalenVec.Normalize();
  Richtung = Side2.Dot(NormalenVec);
  if (abs(Richtung) < 1e-15)
    {Richtung = 1;}
  else
    {Richtung *= 1/abs(Richtung);};
  NormalenVec.Mult(Richtung);

  SphereLocation.IsInCubus(CellOrig, Side1, Side2, NormalenVec, EinheitsKoor);

  Distance = -EinheitsKoor.z;

  if (EinheitsKoor.x >= 0 && EinheitsKoor.x <= 1 &&
      EinheitsKoor.y >= 0 && EinheitsKoor.y <= 1 )
    {return true;}
  else {return false;};
}

double Koordinate::DistanceFromLine(Line Linie)
{
  Koordinate NegUrspr(Linie.Origin), ProjektionsPkt;
  Vector KooBzglOrigin(x,y,z), KooBzglNull, Richt(Linie.Direction);
  double Projektion;

  NegUrspr.Mult(-1.0);
  KooBzglNull.Add(KooBzglOrigin,NegUrspr);

  Richt.Normalize();
  Projektion = KooBzglNull.Dot(Richt);
  Richt.Mult(Projektion);
  ProjektionsPkt.Add(Linie.Origin, Richt);
```

```
  return sqrt(pow(ProjektionsPkt.x-x,2)+pow(ProjektionsPkt.y-y,2)+pow(ProjektionsPkt.z-z,2));
}



//*************************************************
//*          CellSpheres::InCellTest               *
//*************************************************

bool CellSpheres::InCellTest(Cell InZelle, SphereDaten Kugel)
{
  bool Zustand[3][6], // index: [0]: within inner Cell, [1]: within Outer Cell, [2]: under limited
Cellplane (Projektion)
       FinalOK = true, TempOK = false;
  int i2, i3, B[2], C[3], D[3];
  double SollUnterInnerEbene, SollUnterOuterEbene, IstUnterInnerEbene, IstUnterOuterEbene,
        InvSollUnterInnerEbene, InvSollUnterOuterEbene, InvIstUnterInnerEbene, InvIstUnterOuterEbene,
        Distanzen[6], Abstand;
  Vector InnerZelleKugel, InvInnerZelleKugel, OuterZelleKugel, InvOuterZelleKugel,
         InvInnerSide[3], InvOuterSide[3], InvSide[3],
         VectorUnterInnerEbene, VectorUnterOuterEbene, InvVectorUnterInnerEbene,
InvVectorUnterOuterEbene;
  Koordinate KugelKoordinate(Kugel.x, Kugel.y, Kugel.z),
             A[2],
             InvInnerOrigin, InvOuterOrigin, InvOrigin,
             Ort(x,y,z);
  Plane Ebene[6];


  // Determination of opposit (to cell origin) cell point
  // Determination of complement cell sides (Complement to original sides)
  // Together with the complement origin all cell sides can now be described

   if (InZelle.Prisma4)
    {
      // COPLEMENT ORIGIN: ORIGIN + ALL SIDES
      A[0].Add(InZelle.OuterOrigin,InZelle.OuterSide[0]); A[1].Add(A[0],InZelle.OuterSide[1]);
      InvOuterOrigin.Add(A[1],InZelle.OuterSide[2]);
      A[0].Add(InZelle.InnerOrigin,InZelle.InnerSide[0]); A[1].Add(A[0],InZelle.InnerSide[1]);
      InvInnerOrigin.Add(A[1],InZelle.InnerSide[2]);
      A[0].Add(InZelle.Origin,InZelle.Side[0]); A[1].Add(A[0],InZelle.Side[1]);
      InvOrigin.Add(A[1],InZelle.Side[2]);
      // COMPLEMENT SIDES: INVERSE OF ORIGINAL SIDES
      for (int i = 0; i < 3; i++) {InvInnerSide[i].Set(InZelle.InnerSide[i]); InvInnerSide[i].Mult(-
1.0);};
      for (int i = 0; i < 3; i++) {InvOuterSide[i].Set(InZelle.OuterSide[i]); InvOuterSide[i].Mult(-
1.0);};
      for (int i = 0; i < 3; i++) {InvSide[i].Set(InZelle.Side[i]); InvSide[i].Mult(-1.0);};
    }
  else
    {
      // COMPLEMENT ORIGIN: ORIGIN + ELEVATION SIDE + FIRST BASE-SIDE
      A[0].Add(InZelle.OuterOrigin,InZelle.OuterSide[0]);
InvOuterOrigin.Add(A[0],InZelle.OuterSide[1]);
      A[0].Add(InZelle.InnerOrigin,InZelle.InnerSide[0]);
InvInnerOrigin.Add(A[0],InZelle.InnerSide[1]);
      A[0].Add(InZelle.Origin,InZelle.Side[0]); InvOrigin.Add(A[0],InZelle.Side[1]);
      // COMPLEMENT SIDES: INVERSE OF ELEVATION AND FIRST BASE SIDE, DIFFERENCE OF SECOND TO FIRST
BASE-SIDE
      for (int i = 0; i < 2; i++) {InvInnerSide[i].Set(InZelle.InnerSide[i]); InvInnerSide[i].Mult(-
1.0);};
      A[0].Set(InZelle.InnerSide[1]); A[0].Mult(-1.0); InvInnerSide[2].Add(InZelle.InnerSide[2],A[0]);
      for (int i = 0; i < 2; i++) {InvOuterSide[i].Set(InZelle.OuterSide[i]); InvOuterSide[i].Mult(-
1.0);};
      A[0].Set(InZelle.OuterSide[1]); A[0].Mult(-1.0); InvOuterSide[2].Add(InZelle.OuterSide[2],A[0]);
      for (int i = 0; i < 2; i++) {InvSide[i].Set(InZelle.Side[i]); InvSide[i].Mult(-1.0);};
      A[0].Set(InZelle.Side[1]); A[0].Mult(-1.0); InvSide[2].Add(InZelle.Side[2],A[0]);
    };

   // Vectorformation of Sphere-midpoint relative to Inner, Outer and Inverse Cell Origins

   A[0].Set(InZelle.OuterOrigin); A[0].Mult(-1.0); OuterZelleKugel.Add(KugelKoordinate,A[0]);
   A[0].Set(InZelle.InnerOrigin); A[0].Mult(-1.0); InnerZelleKugel.Add(KugelKoordinate,A[0]);

   A[0].Set(InvOuterOrigin); A[0].Mult(-1.0); InvOuterZelleKugel.Add(KugelKoordinate,A[0]);
   A[0].Set(InvInnerOrigin); A[0].Mult(-1.0); InvInnerZelleKugel.Add(KugelKoordinate,A[0]);
```

```
   for (int i = 0; i < 3; i++)
     {
      i2 = int(floor((fmod((i+1),3))+0.1));
      i3 = int(floor((fmod((i+2),3))+0.1));

      // Here: i: index of first side vector  - for plane
      //      i2: index of second side vector - for plane
      //      i3: index of direction vector - in direction of body

      // Plane definition - is going to be used later, for edge/corner calculation
      // in distance test.

      Ebene[i].Set(InZelle.Origin, InZelle.Side[i], InZelle.Side[i2]);
      Ebene[i+3].Set(InvOrigin, InvSide[i], InvSide[i2]);

      // Calculation of Sphere-Center above each Plane (limited), and test wheter Porjetion on layer

      Zustand[2][i] = Ort.IsAboveSurface(InZelle.Origin, InZelle.Side[i], InZelle.Side[i2],
InZelle.Side[i3], Distanzen[i]);
      Zustand[2][i+3] = Ort.IsAboveSurface(InvOrigin, InvSide[i], InvSide[i2], InvSide[i3],
Distanzen[i+3]);

      // Parallelepipedal product for testing if center fo sphere is under or over a surface
      // Product formation always for plane vectors and third cell vector ->  comparison
      // with product of plane vectors to relative vector to spherecenter

      VectorUnterInnerEbene.Cross(InZelle.InnerSide[i],InZelle.InnerSide[i2]);
      SollUnterInnerEbene = VectorUnterInnerEbene.Dot(InZelle.InnerSide[i3]);
      IstUnterInnerEbene = VectorUnterInnerEbene.Dot(InnerZelleKugel);

      VectorUnterOuterEbene.Cross(InZelle.OuterSide[i],InZelle.OuterSide[i2]);
      SollUnterOuterEbene = VectorUnterOuterEbene.Dot(InZelle.OuterSide[i3]);
      IstUnterOuterEbene = VectorUnterOuterEbene.Dot(OuterZelleKugel);

      InvVectorUnterInnerEbene.Cross(InvInnerSide[i],InvInnerSide[i2]);
      InvSollUnterInnerEbene = InvVectorUnterInnerEbene.Dot(InvInnerSide[i3]);
      InvIstUnterInnerEbene = InvVectorUnterInnerEbene.Dot(InvInnerZelleKugel);

      InvVectorUnterOuterEbene.Cross(InvOuterSide[i],InvOuterSide[i2]);
      InvSollUnterOuterEbene = InvVectorUnterOuterEbene.Dot(InvOuterSide[i3]);
      InvIstUnterOuterEbene = InvVectorUnterOuterEbene.Dot(InvOuterZelleKugel);

      // Test if spherecenter under cellsurface, if yes -> same signum for
      // reference- and spherecenter- parallelepipedal product

      if ( (SollUnterInnerEbene/IstUnterInnerEbene) > 0) { Zustand[0][i] = true; }
      else { Zustand[0][i] = false; };
      if ( (SollUnterOuterEbene/IstUnterOuterEbene) > 0) { Zustand[1][i] = true; }
      else { Zustand[1][i] = false; };

      if ( (InvSollUnterInnerEbene/InvIstUnterInnerEbene) > 0) { Zustand[0][i+3] = true; }
      else { Zustand[0][i+3] = false; };
      if ( (InvSollUnterOuterEbene/InvIstUnterOuterEbene) > 0) { Zustand[1][i+3] = true; }
      else { Zustand[1][i+3] = false; };
    };

   // is Sphere within all outer surfaces? -> also within outer cell (Top Condition)
   if (Zustand[1][0] && Zustand[1][1] && Zustand[1][2] && Zustand[1][3] && Zustand[1][4] &&
Zustand[1][5])
     {
       // Case determination: How many planes are cutten by sphere? And if plane (infinit) is cutten
       // is the sphere above limited surface (Projection)?
       B[0] = 0; B[1] = 0; // [0]: How many planes cutten / [1]: Not above how many surfaces
       int j = 0, k = 0;  // Counter for C[j] and D[j], for index increment
       C[0] = -1; C[1] = -1; C[2] = -1; // Contains the cutten and not above surface indixes, # is <=3
       D[0] = -1; D[1] = -1; D[2] = -1; // Contains the cutten surface indixes, # is <=3
       for (int i = 0; i < 6; i++)
         {
           if (InZelle.Prisma4)
             {
               if (!Zustand[0][i]) {B[0]++; D[k]=i; k++; if (!Zustand[2][i]){B[1]++ ; C[j]=i; j++;};};
             }
           else
             {
               // Because of double definition of one plane -> exclusion of Plane 4 (i=3)
```

```
              if (!Zustand[0][i] && i != 3) {B[0]++; D[k]=i; k++; if (!Zustand[2][i]){B[1]++ ;
C[j]=i; j++;};};
            };
         };

       switch (B[0])
         {
         case 0: // no interaction with any surface
           FinalOK = true;
           break;
         case 1: // cutting of one cubus surface, return of cutting depth
           FinalOK = true;
           CuttingPlane[D[0]] = Distanzen[D[0]];
           break;
         case 2: // cutting of two cubus surfaces, or edge exclusion
           if (D[0] != -1) {CuttingPlane[D[0]] = Distanzen[D[0]];};
           if (D[1] != -1) {CuttingPlane[D[1]] = Distanzen[D[1]];};
           if (B[1] == 2)
             {
               // testing of Distance (Edge condition)
               Line Seitenlinie;
               Ebene[C[0]].CutPlanePlane(Ebene[C[1]], Seitenlinie);
               Abstand = Ort.DistanceFromLine(Seitenlinie);
               if (Abstand <= Kugel.r)
                 {
                   FinalOK = true;
                   cout << " -- Sphere # " << Kugel.Num << " cutten by plane # ";
                   cout << C[0] << " (by " << Distanzen[C[0]] << ") & " << C[1]  << " (by " <<
Distanzen[C[1]] << ")";
                   cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n";
                 }
               else
                 {
                   FinalOK = false;
                   cout << " -- Sphere # " << Kugel.Num << " REJECTED 22 ";
                   cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n" ;
                 };
             }
           else
             {
               FinalOK = true;
             }
           break;
         case 3: // cutting of two cubus surfaces, or edge exclusion
           if (D[0] != -1) {CuttingPlane[D[0]] = Distanzen[D[0]];};
           if (D[1] != -1) {CuttingPlane[D[1]] = Distanzen[D[1]];};
           if (D[2] != -1) {CuttingPlane[D[2]] = Distanzen[D[2]];};
           if (B[1] == 3)
             {
               // testing of Distance (Corner condition)
               Line Seitenlinie;
               Koordinate Schnittpkt;
               Ebene[C[0]].CutPlanePlane(Ebene[C[1]], Seitenlinie);
               Ebene[C[2]].CutPlaneLine(Seitenlinie, Schnittpkt);
               Abstand = sqrt( pow(Ort.x-Schnittpkt.x,2)+pow(Ort.y-Schnittpkt.y,2)+pow(Ort.z-
Schnittpkt.z,2));
               if (Abstand <= Kugel.r)
                 {
                   FinalOK = true;
                   cout << " -- Sphere # " << Kugel.Num << " cutten by plane # ";
                   cout << C[0] << " (by " << Distanzen[C[0]] << ") & " << C[1]  << " (by " <<
Distanzen[C[1]] << ")";
                   cout << C[2]  << " (by " << Distanzen[C[2]] << ") ";
                   cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n";
                 }
               else
                 {
                   FinalOK = false;
                   cout << " -- Sphere # " << Kugel.Num << " REJECTED 33 ";
                   cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n";
                 }
             };
           if (B[1] == 2)
             {
               // testing of Distance (Edge condition)
               Line Seitenlinie;
               Ebene[C[0]].CutPlanePlane(Ebene[C[1]], Seitenlinie);
```

```
              Abstand = Ort.DistanceFromLine(Seitenlinie);
              if (Abstand <= Kugel.r)
                {
                  FinalOK = true;
                  cout << " -- Sphere # " << Kugel.Num << " cutten by plane # ";
                  cout << C[0] << " (by " << Distanzen[C[0]] << ") & " << C[1]  << " (by " <<
Distanzen[C[1]] << ")";
                  cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n";
                }
              else
                {
                  FinalOK = false;
                  cout << " -- Sphere # " << Kugel.Num << " REJECTED 32 ";
                  cout << " Cooriante: " << Kugel.x << "," << Kugel.y << "," << Kugel.z << "\n";
                };
            }
          else
            {
              FinalOK = true;
            };
          break;
        default:
          {
            FinalOK = false;
            cout << " -- Sphere # " << Kugel.Num << " EERROORR ";
          }
        };

       if (FinalOK)
         {
           return true;
         }
       else
         {
           return false;
         };
     }
   else
     {
       return false;
     };
}

//*************************************************
//*          HAUPTPROGRAMM                        *
//*************************************************



void main()
{
  // Zellendefinition ***************************************************************************

  Cell Zelle;

  std::fstream ZellenDaten;

  ZellenDaten.open("Cell.txt", std::ios::in);
  if (!ZellenDaten.is_open()) return(0);

  while (!ZellenDaten.eof())
    {
      Zelle.ReadIn(ZellenDaten);
    }

  ZellenDaten.close();

  double Radius = 5e-05;

  Zelle.Create(Zelle.Origin, Zelle.Side[0], Zelle.Side[1], Zelle.Side[2], Zelle.Prisma4, Radius);
  Zelle.WriteOut();

  // Datenstrom einlesen und testen *************************************************************

  SphereDaten EinzelneKugel;
  CellSpheres KugelBzglZelle;
  std::fstream AlleKugeln, InDerZelle;
```

```
  InDerZelle.open("InDerZelle.txt", std::ios::out);
  if (!InDerZelle.is_open()) return(0);

  AlleKugeln.open("DatenFine.txt", std::ios::in);
  if (!AlleKugeln.is_open()) return(0);


  while (!InDerZelle.eof())
    {
      EinzelneKugel.ReadIn(AlleKugeln);
      KugelBzglZelle.Initialisierung(EinzelneKugel);
      if (KugelBzglZelle.InCellTest(Zelle,EinzelneKugel))
        {
          KugelBzglZelle.WriteOut(InDerZelle);
        }
    };

  AlleKugeln.close();
  InDerZelle.close();

  cout << "ende";
}
```

E.3 Coarse spheres in the selected cell

The coarse spheres within the cell are given by:
identification numbers, followed by the tree coordinates and the sphere radius. These coordinates were selected by a C++ program being close enough to the cell to be taken into consideration.

4,4.75E-05,-0.000257,0.001155,0.0004,
1,-0.0001113,0.0005437,0.001146,0.0004,
18,0.0002287,0.0002019,0.001784,0.0004,
16,-0.0004851,-0.0001755,0.001746,0.0004,
11,0.0002919,6.22E-05,0.000463,0.0004,
2,-0.0004958,0.0001109,0.000594,0.0004,
27,0.0008061,-0.0005451,0.001221,0.0004,
8,0.0001829,-0.001045,0.00118,0.0004,
5,-0.001048,-0.0002224,0.001179,0.0004,
23,-0.0006006,-0.0008853,0.001201,0.0004,
7,-0.0001292,-0.0006976,0.000463,0.0004,
3,0.0006561,-0.0008499,0.000463,0.0004,
19,0.00055,-0.0009208,0.00188,0.0004,

E.4 Fine spheres in the selected cell

The fine spheres within the cell are given by:
identification numbers, followed by the tree coordinates and the sphere radius.

8517,-0.0005266,-0.0005606,0.001504,5e-05
4872,-0.0003822,-0.0006739,0.00164,5e-05
4797,-0.0005079,-0.0006132,0.001637,5e-05
6252,-0.0006207,-0.0005344,0.001482,5e-05
6762,-0.0004609,-0.0006845,0.001579,5e-05
6012,-0.0005856,-0.000585,0.001581,5e-05
5757,-0.0007009,-0.0005108,0.001537,5e-05
8546,-0.0006755,-0.0005629,0.001619,5e-05

7533,-0.0007322,-0.000486,0.001362,5e-05
5967,-0.00089,-7.818e-05,0.001575,5e-05
8366,-0.0005396,-0.0007081,0.001636,5e-05
7888,-0.000469,-0.0007764,0.001618,5e-05
7343,-0.000782,-0.0004692,0.001578,5e-05
6802,-0.0007409,-0.0005371,0.001449,5e-05
4866,-0.000626,-0.0006748,0.001598,5e-05
8677,-0.0008555,-0.000355,0.001564,5e-05
6649,-0.0009129,-0.0001784,0.001606,5e-05
7438,-0.0009064,-0.0002781,0.001603,5e-05
8639,-0.0007101,-0.0006354,0.001559,5e-05
7831,-0.0008618,-0.0004459,0.001523,5e-05
6946,-0.0008177,-0.0005523,0.001536,5e-05
6692,-0.000825,-0.0005475,0.001396,5e-05
8750,-0.00062,-0.0007676,0.001635,5e-05
6862,-0.000889,-0.0005266,0.00147,5e-05
6690,-0.0008037,-0.000651,0.001527,5e-05
8514,-0.0007211,-0.0007355,0.001608,5e-05
6134,-0.0009505,-0.0004303,0.001566,5e-05