

Master Thesis

**Topology of the Deformation of a
Non-Uniform Grain Structure**

Chae, Jae yong (蔡 在 鎔)

Department of Ferrous Technology

(Computational Metallurgy)

Graduate Institute of Ferrous Technology

Pohang University of Science and Technology

2008

불균일 결정 구조의
가공에 관한 위상 기하학

**Topology of the deformation
of a Non-uniform Grain structure**

Topology of the deformation of a Non-Uniform Grain Structure

by


Chae, Jae Yong
Department of Ferrous Technology
(Computational Metallurgy)
Graduate Institute of Ferrous Technology
Pohang University of Science and Technology

A thesis submitted to the faculty of Pohang University of Science and Technology in partial fulfillments of the requirements for the degree of Master of Science in the Graduate Institute of Ferrous Technology (Computational Metallurgy)

Pohang, Korea
June 23th, 2008

Approved by

Prof. Lee, Hae Geon



Major Advisor

Prof. Bhadeshia, H. K. D. H.



Co-Advisor

Topology of the deformation of a Non-Uniform Grain Structure

Chae, Jae Yong

This dissertation is submitted for the degree of Master of Science at the Graduate Institute of Ferrous Technology of Pohang University of Science and Technology. The research reported herein was approved by the committee of Thesis Appraisal

June 23th, 2008

Thesis Review Committee

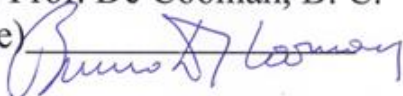
Chairman: Prof. Lee, Hae Geon

(signature) 

Member: Prof. Kim, In Gee

(signature) 

Member: Prof. De Cooman, B. C.

(signature) 

MFT

20062871

Chae, Jae Yong

Topology of the Deformation of a Non-Uniform Grain Structure, Department of Ferrous Technology (Computational Metallurgy) 2008

Advisor: Prof. Lee, Hae Geon; Prof. Bhadeshia, H.K.D.H.

Text in English

Abstract

The purpose of this work was to investigate a method for treating non-uniform grains which had a distribution of sizes and shapes. It was the extension of a previous work for the idealized tetrakaidecahedron-shaped grains.

First step of the work was generating the vertex coordinates of non-uniform grains from tetrakaidecahedra array, randomized vertex method (RVM) and rescale randomized vertex method (RRVM) were considered for that. Then several deformations, such as plane strain compression, axisymmetric compression, axisymmetric tension and simple shear, were applied to the grains, and the changes in the grain parameters (surface area and edge length per unit volume) were calculated. It is also possible to calculate the parameters for any other homogeneous deformations or for combinations of several deformations in the same way. It is shown that the results for non-uniform grains have little differences as compared with the results for idealized grains.

Contents

Abstract	I
Contents	II
Nomenclature	V
I . Introduction – Literature Review	1
1.1 Aim of the work.....	1
1.2 Basic crystallography	2
1.2.1 Lattice, Unit Cell, Basis.....	2
1.2.2 Point and direction.....	3
1.2.3 Planes.....	4
1.3 Vector	6
1.3.1 Representation of a vector	6
1.3.2 Addition and scalar multiplication.....	8
1.3.3 Dot product.....	9
1.3.4 Cross product.....	10
1.4 Deformation.....	11
1.4.1 Strain and stress	12
1.4.2 Type of deformation	14

1.5 The effect of plastic deformation on uniform grains.....	15
1.5.1 Tetrakaidecahedron.....	15
1.5.2 The application of the deformation	17
1.5.3 The calculation	20
1.5.4 Equivalent strain.....	22
1.5.5 The result of calculation	23
1.5.6 Improvement of model	25
II. Method.....	27
2.1 Definition of uniform grain structure	28
2.2 The number of the defined grain, odd & even grain	35
2.3 Conversion of uniform grain structure into non-uniform array.....	36
2.4 Extracting vertex coordinates	38
2.5 Deformation.....	40
III. Result.....	42
3.1 Non-uniform grain generation	42
3.2 Applying various deformations	47
3.3 Analysis of the results.....	53
IV. Summary and future work	54

References	56
Appendix A	60
Appendix B	67

Nomenclature

A_0	Original cross-sectional area of tensile test specimen
A	Instantaneous cross-sectional area of tensile test specimen
E	Young's modulus
F	Force
L	Size of tetrakaidecahedron grain
L_q	Quarter of the grain size L
L_0	Initial grain edge length
L_V	Grain edge length per unit volume
L_{V0}	Initial grain edge length per unit volume
L_{V0}^P	Primary edge length per unit volume
L_{V0}^S	Secondary edge length per unit volume
S_0	Initial grain boundary area
S_V	Grain boundary area per unit volume
S_{V0}	Initial grain boundary area per unit volume
R	Rotation matrix
S	Deformation matrix
S_{ij}	The elements of deformation matrix
a	Vector \vec{a}

b	Vector $\vec{\mathbf{b}}$
u	Edge vector before deformation
v	Resultant edge vector after deformation
e₁	Basis vector (1, 0, 0)
e₂	Basis vector (0, 1, 0)
e₃	Basis vector (0, 0, 1)
<i>l</i>	Instantaneous length of tensile test specimen
<i>l₀</i>	Original length of tensile test specimen
ϵ	Engineering strain
ϵ_T	True strain
<i>f</i>	Equivalent stress
<i>r</i>	Transformed vertex position
<i>r₀</i>	Initial vertex position
<i>r_{ref}</i>	Reference position inside the grain
σ	Engineering stress
σ_T	True stress
$\bar{\sigma}$	Tensile yield stress in a tensile test
ζ	Weight factor in RRVM
ξ	Random number, of $0 < \xi < 1$
ω	Weight of fluctuation

ϵ_{ij}	Normal components of strain
γ_{ij}	Shear components of strain
RRVM	Rescaled randomized vertex method
RVM	Randomized vertex method

I . Introduction – Literature Review

1.1 Aim of the work

Polycrystalline materials such as steel and aluminum are produced in very large quantities using plastic deformation, which changes their microstructure, properties and shape into the required form. One of the important factors influenced by the deformation at high temperatures is the grain structure. For that reason, calculation about the consequential change in the amount of grain boundary area per unit volume (S_V) and grain edge length per unit volume (L_V) after plastic deformation may be significant in determining the course of phase transformations and recrystallisation processes in general.

Underwood expressed these parameters as a function of the extent of deformation using stereological methods [Underwood, 1970], which have the advantage of avoiding assumptions about grain shape as long as space is filled. The method cannot however be adapted to complex combinations of deformations. Other approaches involve analytical equations numerical computations based on a variety of approximations of the three-dimensional shape of grains, for example, spheres, cubes and tetrakaidecahedra [Umemoto *et al.*, 1983; Bate and Hutchinson, 2005; Gil-Sevillano *et al.*, 1980; Knustad *et al.*, 1985; Vatne *et al.*, 1996; Singh and

Bhadeshia, 1998; Zhu *et al.*, 2007]. These methods can be adapted to combinations of deformations but they assume that all the grains are exactly identical in shape and size. The purpose of the present work is to develop a corresponding model for a non-uniform grain structure.

1.2 Basic crystallography

Polycrystalline materials have properties which depend on the nature of the each crystal, the size and shape distribution of the crystals and the crystallographic orientation of individual crystals in it. Knowledge of the relationship between crystals in a polycrystalline material is a major part of crystallography, in contrast to the knowledge of atomic arrangements in single crystals. The former concept is of particular importance in the present work.

1.2.1 Lattice, Unit cell, Basis

Materials may be classified according to the regularity of array of atomic arrangements in crystals. Crystals are identified by a regular array of points, which form a lattice with translational symmetry. This three-dimensional array of points

may or may not coincide with the positions of atoms. The entire repeating pattern of the lattice can be described in terms of a small repeat entity is called a unit cell. The unit cell may be a space-filling parallelepiped with vertices at lattice points, and with its edges defined by three non-coplanar basis vectors, each of which represents translation between two lattice points. The magnitudes of the basis vectors are the lattice parameters of unit cell. There can be many number of lattice vectors which can be used in defining the unit cell. Choosing smaller basis vectors to represent the shape of lattice is convenient with respect to the symmetry of the lattice.

1.2.2 Point and direction

The position of any point located within a unit cell may be specified in terms of its coordinates as fractional multiples of the unit cell edge lengths.

A direction is defined as a line between two points, or a vector in crystallography. Consider the determination of the three directional indices of a vector which passes through the origin of the coordinate system. The projection of the vector on each of the three axes is determined, and expressed as a fraction of each of the basis vectors. These three numbers are multiplied or divided by a common factor to reduce them to the smallest integer values. Any parallel vectors can be translated into same results and three indices are enclosed in square brackets. In addition, any negative index is represented by positive number with a bar over it. It follows that any vector

can be represented as:

$$\mathbf{u} = u_1\mathbf{a}_1 + u_2\mathbf{a}_2 + u_3\mathbf{a}_3 \quad (1-1)$$

Where $u_1, u_2,$ and u_3 are the components of the vector and $\mathbf{a}_1, \mathbf{a}_2$ and \mathbf{a}_3 are the basis vectors of the unit cell. The vector \mathbf{u} is thus represented in terms of its components as $[u_1 \ u_2 \ u_3]$.

For example, direction $[111]$ means the vector:

$$\mathbf{u} = 1\mathbf{a}_1 + 1\mathbf{a}_2 + 1\mathbf{a}_3 \quad (1-2)$$

and, direction $[\bar{1}\bar{1}1]$ means the vector:

$$\mathbf{u} = -1\mathbf{a}_1 - 1\mathbf{a}_2 + 1\mathbf{a}_3 \quad (1-3)$$

In some crystal structures, the spacing of atoms along each direction is the same. Thus, several nonparallel directions are actually equivalent and they can be grouped together into a *family*. Such equivalent directions are identified by enclosing them in angle brackets. In a cubic lattice, all the directions represented by the following indices are equivalent: $[100]$, $[\bar{1}00]$, $[010]$, $[0\bar{1}0]$, $[001]$ and $[00\bar{1}]$. This group can be represented as $\langle 100 \rangle$.

1.2.3 Planes

Crystallographic planes are specified by three *Miller indices* as (hkl) . Any two

planes parallel to each other are equivalent and have identical indices. The determination of the indices is as follows:

1. If the plane passes through the origin of unit cell, either another parallel plane should be considered by an appropriate translation, or a new origin should be established at the corner of another unit cell.
2. The plane would then intersect or be parallel to each of the three axes; the length of each intercept for each axis is determined in terms of the lattice parameters a , b , and c .
3. The reciprocals of the intercepts, expressed as multiples (whole or fractional) of the lattice parameters, form the indices of the plane into indices. If a plane is parallel to an axis, intercept is regarded as infinity, and its reciprocal is 0, which forms one of the three indices of that plane.
4. If necessary, the index numbers can be multiplied or divided by a common factor, and then the final results give the Miller indices; h, k, l .

The following is the brief example, where a , b , and c are the lattice parameters of unit cell :

	x	y	z
Intercepts	parallel	-b	c/2
Intercepts (relative to lattice parameter)	∞	-1	1/2
Reciprocals	0	-1	2
Miller indices		$(0 \bar{1} 2)$	

Table 1.1 Illustration of the determination of Miller indices.

1.3 Vector

A vector is the quantity which is characterized by magnitude and direction, whereas a scalar just has magnitude. Velocity, force, and displacement are vectors and speed, power, and time are scalars.

1.3.1 Representation of a vector

A vector is graphically represented by an arrow pointing in a particular direction, as illustrated below:

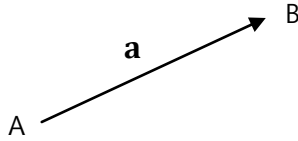


Figure 1.1 : Graphical representation of vector \overrightarrow{AB}

Here, the point A is called the initial point, and B is called endpoint of vector \overrightarrow{AB} . This vector can be identified by the symbol with arrow (e.g. \vec{a}) or underlining the lower-case symbol (e.g. \underline{a}). The length of the arrow represents the magnitude of the vector, denoted as $|\vec{a}|$. A vector is equal with another vector if they both point same direction, and have identical magnitude.

It is inconvenient to use the graphical representation for complicated problems. Thus, vector in an n -dimensional Euclidean space can be represented in a Cartesian coordinate system; it is identified by the coordinate of the endpoint, which is a list of n real numbers, with the origin as the initial point. As an example, in three-dimensional Euclidean space, the vector from the origin $O = (0, 0, 0)$ to the point $A = (1, 2, 3)$ is simply written as $\overrightarrow{OA} = (1, 2, 3)$. These coordinate numbers are often arranged into a column vector or row vector, particularly when dealing with matrices, as follows:

$$\vec{OA} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \text{ or } \vec{OA} = (1 \ 2 \ 3) \quad (1-4)$$

Another way to express a vector in three dimensions is to introduce basic coordinate vectors referred to as basis vectors:

$$\mathbf{e}_1 = (1, 0, 0), \mathbf{e}_2 = (0, 1, 0), \mathbf{e}_3 = (0, 0, 1) \quad (1-5)$$

In terms of these, any vector (a, b, c) in three dimensions can be expressed in the form:

$$\begin{aligned} (a, b, c) &= a(1, 0, 0) + b(0, 1, 0) + c(0, 0, 1) \\ &= a\mathbf{e}_1 + b\mathbf{e}_2 + c\mathbf{e}_3 \end{aligned} \quad (1-6)$$

The basis vectors can be any set of vectors which are *linearly independent*, i.e. not parallel but can express all the vectors in three dimensions.

1.3.2 Addition and scalar multiplication

Let $\mathbf{a} = a_1\mathbf{e}_1 + a_2\mathbf{e}_2 + a_3\mathbf{e}_3$ and $\mathbf{b} = b_1\mathbf{e}_1 + b_2\mathbf{e}_2 + b_3\mathbf{e}_3$. The sum of \mathbf{a} and \mathbf{b} is:

$$\mathbf{a} + \mathbf{b} = (a_1 + b_1)\mathbf{e}_1 + (a_2 + b_2)\mathbf{e}_2 + (a_3 + b_3)\mathbf{e}_3 \quad (1-7)$$

The addition of \mathbf{a} and \mathbf{b} may be also performed by graphical method.

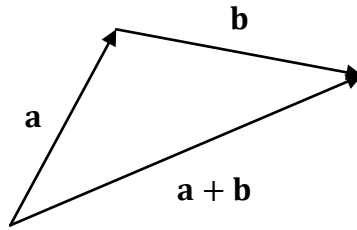


Figure 1.2 : The addition of vector **a** and **b**

The initial point of vector **b** is placed by parallel translation at the endpoint of **a**; the initial point of result, **a + b**, is then the initial point of **a** and its endpoint corresponds to the endpoint of **b**.

A vector can be multiplied by a real number r which is scalar object. The operation of multiplying a vector by a scalar is called scalar multiplication. The resulting vector of $r\mathbf{a}$ is:

$$r\mathbf{a} = (ra_1)\mathbf{e}_1 + (ra_2)\mathbf{e}_2 + (ra_3)\mathbf{e}_3 \quad (1-8)$$

Graphically, this can be represented by stretching a vector **a** out by a factor of r . If r is negative, then the resulting vector changes direction into an opposite direction.

1.3.3 Dot product

The dot product of two vectors **a** and **b** is also known as the inner or scalar product

and is defined as:

$$\mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos \theta \quad (1-9)$$

Where, $|\mathbf{a}|$ and $|\mathbf{b}|$ denote the magnitude of \mathbf{a} and \mathbf{b} , and θ is the angle between \mathbf{a} and \mathbf{b} .

It can also be defined as the sum of the products of components of each vector:

$$\begin{aligned} \mathbf{a} \cdot \mathbf{b} &= (a_1, a_2, \dots, a_n) \cdot (b_1, b_2, \dots, b_n) \\ &= a_1 b_1 + a_2 b_2 + \dots + a_n b_n \end{aligned} \quad (1-10)$$

Where, a_n and b_n are components of vectors \mathbf{a} and \mathbf{b} in n dimensions.

Since $|\mathbf{a}| \cos \theta$ is the scalar projection of \mathbf{a} onto \mathbf{b} , the dot product geometrically means the product of the length of projection and the length of \mathbf{b} .

The dot product of two orthogonal vectors is 0, because the cosine of 90° is 0. In this way, dot product can be used to test the orthogonality of two vectors. Moreover the angle between them can also be found by rearranging the formula:

$$\theta = \cos^{-1} \left(\frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}||\mathbf{b}|} \right) \quad (1-11)$$

1.3.4 Cross product

The cross product, also known as outer product or vector product, is a binary operation on two vectors in a three-dimensional space that results in another vector which is perpendicular to the two input vectors, while the result of dot product is a

scalar quantity. The cross product $\mathbf{a} \times \mathbf{b}$ is given by the formula:

$$\mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}| \sin \theta \mathbf{n} \quad (1-12)$$

where, \mathbf{n} is a unit vector which is perpendicular to \mathbf{a} and \mathbf{b} . The direction of the vector \mathbf{n} is given by the right-hand rule. The magnitude of the result of cross product is equal to the area of the parallelogram that the two input vectors span.

It can also be defined using the coordinates of two vectors:

$$\mathbf{a} \times \mathbf{b} = (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1) \quad (1-13)$$

1.4 Deformation

Deformation leads to a change in the shape of material due to an applied stress such as tensile, compressive, shear, torsion, etc. There are of course special deformations, such as equichannel angular processing, which do not lead to a change in shape. This change may be temporary or permanent; a recoverable change is called elastic deformation and whereas a permanent change is called plastic deformation. To describe the deformations numerically, the concept of stress and strain will be handled at first. This will be followed by a description of the various kinds of stress and resulting deformations.

1.4.1 strain and stress

Strain is the numerical expression of deformation. It is calculated by measuring a change in dimension between the initial and the final states of the material, when stress is applied. Engineering stress σ and engineering strain ϵ can be defined by the following formula:

$$\sigma = \frac{F}{A_0}, \quad \epsilon = \frac{l - l_0}{l_0} = \frac{\Delta l}{l_0} \quad (1-14)$$

in which F is the instantaneous force applied to material normal to the original area A_0 , l_0 is the original length prior to deformation, and l is the length after the application of the force.

Strain is a dimensionless quantity. If strain is equal everywhere within the material, it is said to be homogeneous; otherwise, it is heterogeneous.

Sometimes, it is more meaningful to use a true stress and true strain scheme. The stress and strain represented in equation (1-14) have the adjective “engineering” in order to distinguish them from true stress and strain. True stress σ_T is defined as the load F divided by the instantaneous cross-sectional area A over which deformation is occurring:

$$\sigma_T = \frac{F}{A} \quad (1-15)$$

and similarly true strain ϵ_T is defined by

$$\epsilon_T = \ln\left(\frac{l}{l_0}\right) \quad (1-16)$$

The conversion of engineering stress and strain to true stress and strain is as follows:

$$\sigma_T = \sigma(1 + \epsilon), \quad \epsilon_T = \ln(1 + \epsilon) \quad (1-17)$$

The engineering stress and strain are on the basis of the original cross-sectional area before any deformation, where true stress and strain are defined with respect to the instantaneous dimensions.

Stress is a measure of an applied mechanical load or force, normalized to take into account the cross-sectional area. There are at least three principal ways in which a load may be applied; tension, compression, and shear. In addition, stress is torsion rather than simple shear in many engineering practices. Those four kinds of stress is illustrated in Figure 1.3.

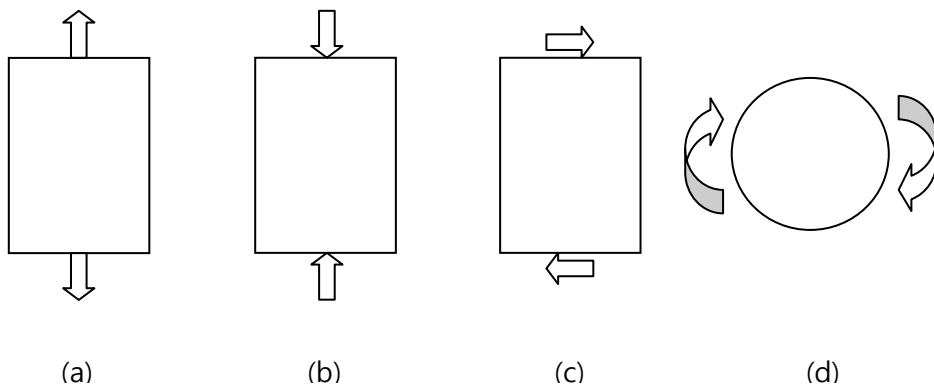


Figure 1.3 : Schematic illustration of (a) tensile stress, (b) compressive stress, (c) shear stress, and (d) torsional stress

1.4.2 Type of deformation

Elastic deformation is reversible. During such deformation the stress and strain are proportional to each other as follows:

$$\sigma = E\epsilon \quad (1-18)$$

The formula is known as Hooke's law, and the constant E is the modulus of elasticity, or *Young's modulus*, which can be experimentally determined from the slope of a stress-strain curve in the elastic region.

For most metallic materials, elastic deformation persists only to strains of about 0.005. As the material is deformed beyond this point, the stress and strain is no longer proportional and plastic deformation starts to occur. The transition from elastic to plastic is gradual for most metal material. The shape change of plastic deformation is not reversible and plastic deformation ends with the fracture of the material.

1.5 The effect of plastic deformation on uniform grains

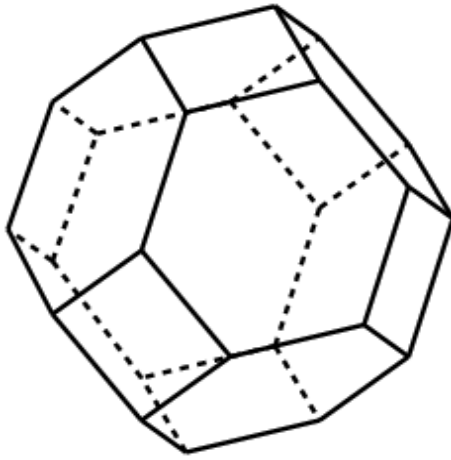
The effect of deformation on grain surface area per unit volume and edge length per unit volume is interesting from many points of view. These values can be used to determine the course of phase transformations or recrystallisation processes in steel. The calculation first started using stereological methods, in which it was assumed that the grains are space-filling and equiaxed, but do not have specific shape [Underwood, 1970]. A spherical grain model was also considered for the calculation in many succeeding investigations [Umemoto *et al.*, 1983; Bate and Hutchinson, 2005]. Cubes [Gil-Sevillano *et al.*, 1980; Knustad *et al.*, 1985; Vatne *et al.*, 1996] or tetrakaidecahedra [Underwood, 1970; Singh and Bhadeshia, 1998] have used to represent the undeformed grain shapes in more recent work. In these works, the deformations such as plane strain, axisymmetric tension, axisymmetric compression, and simple shear are applied to the grain vectors, and all of the deformations are regarded to be homogeneous.

1.5.1 Tetrakaidecahedron

A tetrakaidecahedron is polyhedron which has 36 edges of the same length and 14 faces consisting of eight hexagons and six squares. All the edges of length a can be described in terms of just six vectors, where the directions of two rectangular edges

of the base square and the perpendicular line of them are regarded as the basic orientation axes of the grain. Each vector includes 6 parallel vectors which have same orientation and magnitude, i.e. all the parallel edges are defined as one vector. The shape of a tetrakaidecahedron and those six vectors are described in the Figure 1.4.

This shape has been recently used to represent the undeformed grain because of the weak points of the earlier work; for example, a sphere is not space filling and has no edges, and a cube oversimplifies the real grain. A tetrakaidecahedron has a shape similar to real grains as observed metallographically, and also has proper angles which only require small changes to satisfy equilibrium of interfacial tensions between different grain faces. Furthermore, the array of tetrakaidecahedra of same size can fill the space and their 6 edge vectors are defined with proper orientations for easy derivation of the deformation equation. However this aspect is also a limitation because real grain structures may not have uniform grains.



(a)

Vector components
$[a, 0, 0]$
$[0, a, 0]$
$\left[-\frac{a}{2}, -\frac{a}{2}, -\frac{a}{\sqrt{2}}\right]$
$\left[\frac{a}{2}, -\frac{a}{2}, \frac{a}{\sqrt{2}}\right]$
$\left[\frac{a}{2}, \frac{a}{2}, \frac{a}{\sqrt{2}}\right]$
$\left[-\frac{a}{2}, \frac{a}{2}, \frac{a}{\sqrt{2}}\right]$

(b)

Figure 1.4 : (a) The shape of tetrakaidecahedron and (b) six basic vectors of tetrakaidecahedron of edge length a [Underwood, 1970; Singh *et al.*, 1998].

1.5.2 The application of the deformation

To give a deformation effect on the edges of an undeformed grain, the edge vectors \mathbf{u} are multiplied by the deformation matrices \mathbf{S} of 3 by 3 to generate the result vectors \mathbf{v} (Czinege, 1977; Singh, 1998; Bhadeshia, 2001) as follows:

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \quad (1-19)$$

The orientation of the grain may be changed in some scenarios. Suppose that we

need to deform the grains in various orientations relative to an external frame, then the rotation matrix \mathbf{R} can be used to rotate the objects relative to the axes defining \mathbf{S} .

$$\begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{pmatrix} \begin{pmatrix} R_{11} & R_{12} & R_{13} \\ R_{21} & R_{22} & R_{23} \\ R_{31} & R_{32} & R_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} \quad (1-20)$$

The elements S_{ij} of deformation matrix \mathbf{S} are determined by the type of deformation. In the case of plane strain deformation, as an example, all of the elements are 0 except the diagonal elements S_{11} , S_{22} , and S_{33} . For volume preserving deformations, the multiplication of diagonal elements S_{11} , S_{22} , and S_{33} should be 1. Since $S_{22} = 1$, $S_{11} \times S_{33} = 1$. The terms S_{11} , S_{22} , and S_{33} represent the principal distortions. The elements of four types of deformations are presented in table 1.2 (Zhu *et al.* 2007).

Type	S_{11}	S_{12}	S_{13}	S_{21}	S_{22}	S_{23}	S_{31}	S_{32}	S_{33}
Plane strain compression	≥ 1	0	0	0	1	0	0	0	$1/S_{11}$
Axisymmetric compression	$1/(S_{33})^{1/2}$	0	0	0	$1/(S_{33})^{1/2}$	0	0	0	≤ 1
Axisymmetric tension	≥ 1	0	0	0	$1/(S_{11})^{1/2}$	0	0	0	$1/(S_{11})^{1/2}$
Simple shear	1	0	+ve	0	1	0	0	0	1

Table 1.2 : volume preserving deformations (the convention used is that $S_{11} > S_{22} > S_{33}$)

From the set of vectors of tetrakaidecahedron in Figure 1.4 (b), the new vector set of deformed grain can be derived with those deformation vectors. The following is the result vectors of tetrakaidecahedron after plane strain or axisymmetric deformation without changing orientation.

Vector
$[aS_{11}, 0, 0]$
$[0, aS_{22}, 0]$
$\left[-\frac{aS_{11}}{2}, -\frac{aS_{22}}{2}, -\frac{aS_{33}}{\sqrt{2}}\right]$
$\left[\frac{aS_{11}}{2}, -\frac{aS_{22}}{2}, \frac{aS_{33}}{\sqrt{2}}\right]$
$\left[\frac{aS_{11}}{2}, \frac{aS_{22}}{2}, \frac{aS_{33}}{\sqrt{2}}\right]$
$\left[-\frac{aS_{11}}{2}, \frac{aS_{22}}{2}, \frac{aS_{33}}{\sqrt{2}}\right]$

Table 1.3 : Vectors of tetrakaidecahedron after plane strain or axisymmetric deformation

1.5.3 The calculation of edge length and surface area of tetrakaidecahedron

The change of tetrakaidecahedron grain after deformation is observed with the final to initial edge length (L/L_0) and surface area (S/S_0) ratio. Those values can be calculated mathematically using the vectors defined in Table 1.3 (Zhu *et al.*, 2007).

Consider plane strain deformation. The total edge length of undeformed tetrakaidecahedron of edge length a is $36a$ obviously, since all the 36 edges of that

polyhedron are of identical length. To calculate the total edge length of the grain after deformation, the magnitudes of the six deformed vectors in Table 1.3 are added up and multiplied by 6 which imply six parallel edges. So, the final to initial edge length (L/L_0) of tetrakaidecahedron about plane strain deformation is derived by the following equation:

$$L_0 = 36a \quad (1-21)$$

$$L = 6a\{1 + S_{11} + 2(1 + S_{11}^2 + 2S_{33}^2)^{1/2}\} \quad (1-22)$$

$$\frac{L}{L_0} = \frac{1 + S_{11} + 2(1 + S_{11}^2 + 2S_{33}^2)^{1/2}}{6} \quad (1-23)$$

The surface area can be calculated using vector cross products. As mentioned in 1.3.4, the magnitude of the cross product is equal to the area of the parallelogram that the two input vectors span. The surface area of an undeformed tetrakaidecahedron is the sum of 6 squares and 8 hexagons area; each hexagon consists of three squares which are made by combination of 3 kinds of edge vectors (6 vectors indeed. The area of each square is a^2 and of each hexagon $(3\sqrt{3}/2)a^2$ before the deformation. Then the final to initial surface areas of the tetrakaidecahedron are given:

$$S_0 = 6a^2 + 8(3\sqrt{3}/2)a^2 = 6a^2(1 + 2\sqrt{3}) \quad (1-24)$$

$$S = 2a^2[S_{11} + 3\{S_{11}(1 + 2S_{33}^2)^{1/2} + (S_{11}^2 + 2S_{33}^2)^{1/2}\} + S_{33}\{2(1 + 2S_{11}^2)\}^{1/2}] \quad (1-25)$$

$$\frac{S}{S_0} = \frac{S_{11} + 3\{S_{11}(1 + 2S_{33}^2)^{1/2} + (S_{11}^2 + 2S_{33}^2)^{1/2}\} + S_{33}\{2(1 + 2S_{11}^2)\}^{1/2}}{3(1 + 2\sqrt{3})} \quad (1-26)$$

The calculations about other deformations can also be done by the same process, with the deformation matrices appropriately changed.

1.5.4 Equivalent strain

A concept of the equivalent strain is necessary for the comparison of deformed materials using various imposed stresses. The yield condition of a material may generally be considered to be a function of stress and strain. When the yield condition is assumed to be a function of only stress and to be designated by $f = \bar{\sigma}$, this function f is called the equivalent stress, where $\bar{\sigma}$ is taken to be equal to the tensile yield stress in a tensile test. Corresponding to this, it is possible to define a function of plastic strain called the equivalent strain [Saito *et al.*, 1972]. It is generally considered that the results from the different tests are consistent when they are compared at the same equivalent strain. Equivalent strain can be defined by

the following equation:

$$\varepsilon = \sqrt{\frac{2}{3} \left(\varepsilon_{11}^2 + \varepsilon_{22}^2 + \varepsilon_{33}^2 + \frac{1}{2} \gamma_{13}^2 + \frac{1}{2} \gamma_{12}^2 + \frac{1}{2} \gamma_{23}^2 \right)^{1/2}} \quad 1-27$$

where ε_{11} , ε_{22} and ε_{33} are normal components and γ_{13} , γ_{12} and γ_{23} are shear components of strain (the tangents of the shear angles) [Zhu *et al.*, 2007].

For plane strain deformation, the diagonal matrix components S_{11} , S_{22} , and S_{33} represent the principal distortions; the ratios of the final to initial lengths of unit vectors along the principal axes. And true strains are given by $\varepsilon_{11} = \ln(S_{11})$, $\varepsilon_{22} = \ln(S_{22})$ and $\varepsilon_{33} = \ln(S_{33})$. Thus equivalent strain of homogeneous plane strain compression can be derived as $\varepsilon = (2/\sqrt{3})\varepsilon_{11}$.

However, for simple shear, there is some controversy about how shear strains should be converted to equivalent strains. So, the shear strain ($\gamma = S_{13}$) is used for comparison.

1.5.5 The result of calculation

The calculations about the final to the initial edge length ratio (L/L_0) and surface area ratio (S/S_0) are repeated to see the effects of change in deformation degree and in orientation of a tetrakaidecahedron grain. Equivalent strains and shear strains (in case of simple shear) can be regarded as characteristics of distortion caused by each deformation, where rotation matrix \mathbf{R} is generated to orient the grain randomly. The results of the final to initial edge length (L/L_0) ratio are illustrated in Figure 1.5.

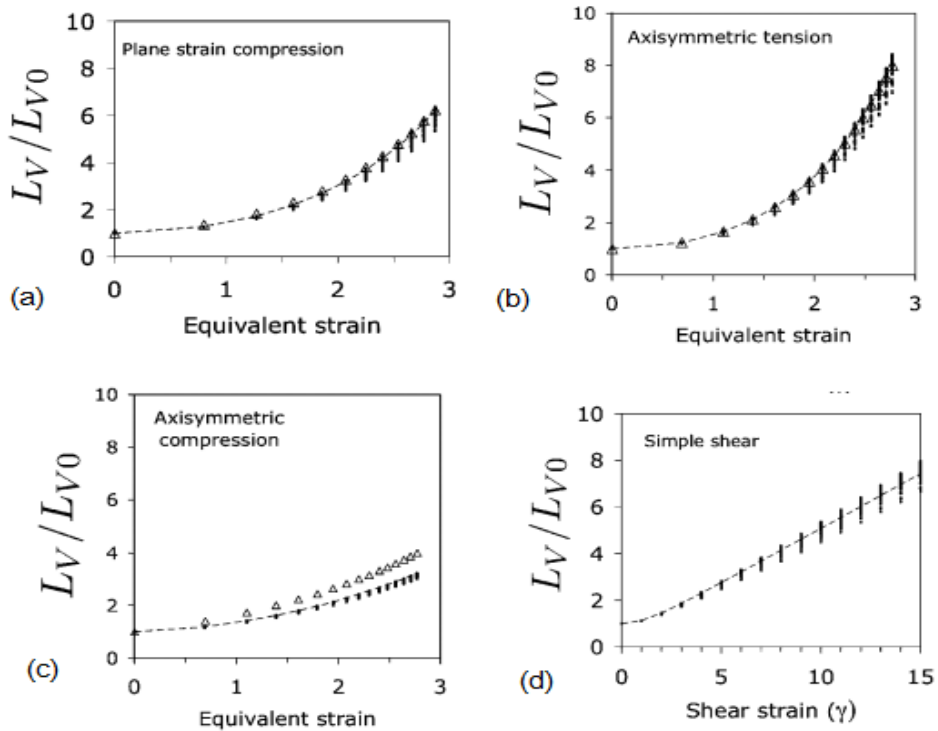


Figure 1.5 : Calculation about changes in length for (a) plane strain deformation, (b) axisymmetric tension, (c) axisymmetric compression and (d) simple shear deformation (Zhu *et al.*, 2007).

In Figure 1.5, the results are plotted against the equivalent strain, but it should be noted that the shear strain is a simple shear deformation. The dashed lines represent the result for the basic orientation of the tetrakaidecahedron defined relative to the coordinate system listed in. And the points are for the 99 other results of randomly oriented grains.

The results for surface area (S/S_0) are also illustrated in Figure 1.6.

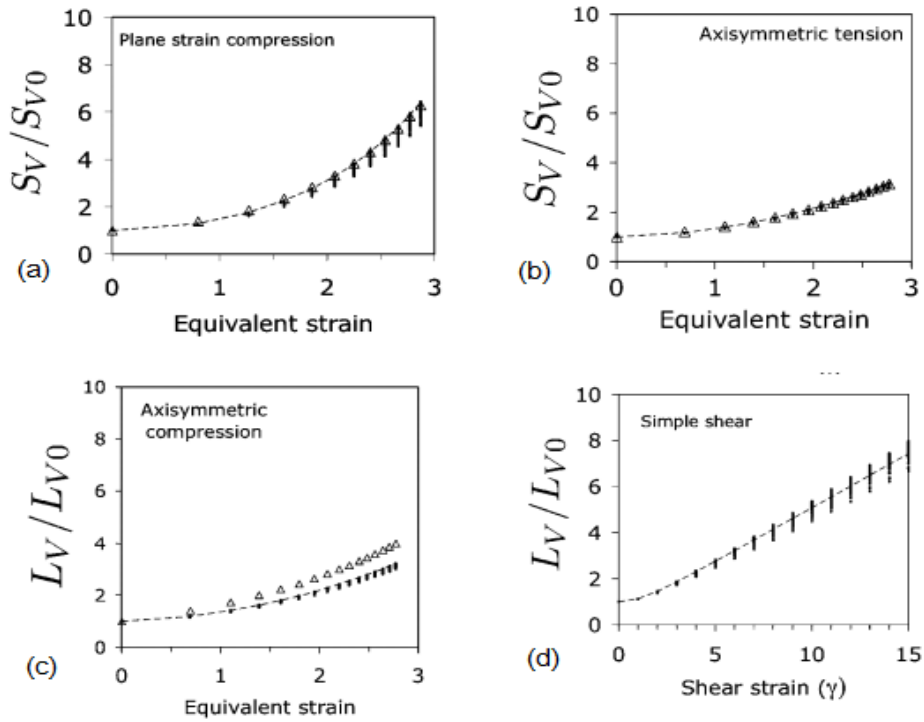


Figure 1.6 : Calculation about changes in surface area for (a) plane strain deformation, (b) axisymmetric tension, (c) axisymmetric compression and (d) simple shear deformation (Zhu *et al.*, 2007).

1.5.6 Improvement of model

In the calculations using the deformation matrices, the grains have been assumed to be uniform shape and size. This kind of assumption can make the calculations simple and easily fill the three dimensional space; cube or tetrakaidecahedron shaped grains are used for those reasons. However, in a real material, there will be a

distribution in the grain size and various shapes of grains (Figure 1.7).

To make a more accurate grain model, new approaches should be considered and they will be treated in this thesis.

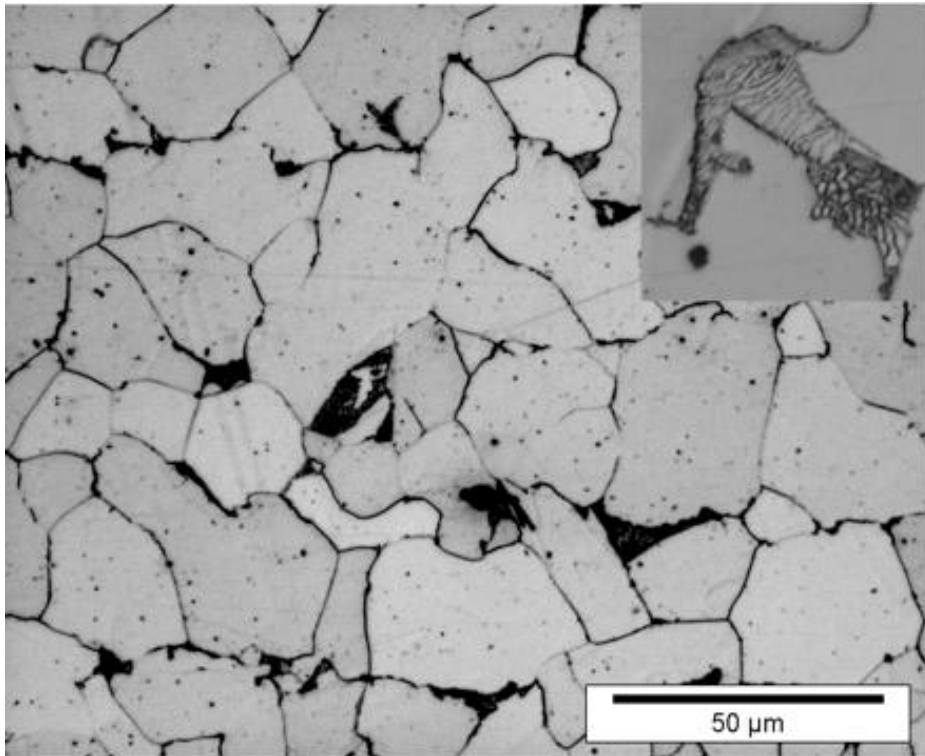


Figure 1.7 : Optical microscopy images of the steel (0.21 wt % C, 0.51 wt % Mn and 0.20 wt % Si) at room temperature. There is a distribution in the grain size and various shapes of the grains. The light regions correspond to ferrite and the dark regions to pearlite (Offerman *et al.*, 2002).

II. Method

In previous work [Zhu *et al.*, 2007], the shape of a grain was represented as a tetrakaidecahedron which can be defined by just six vectors parallel to the edges of the polyhedron. The deformation is then implemented by operating the deformation matrices on those vectors, allowing the resultant vectors of the deformed grain to be deduced. The deformation effect is numerically calculated by observing changes in the surface area (S_v) and edge length (L_v) of the grain using the initial vectors and resultant vectors. Only one tetrakaidecahedron needs to be treated in this way, because all grains are of same shape and size. However this simplification may not work when the grain structure is not uniform like a real grain structure. The edges in the grain cannot be represented by six vectors any longer; each edge needs to be treated separately, a process which is computationally tedious. A different approach is therefore adopted to calculate S_v and L_v in non-uniform grain structure.

In the new method, a uniform array of identical tetrakaidecahedra, Figure 2.1, is perturbed at all the vertices using a stochastic process to generate a non-uniform grain structure; vertices in the uniform grain are transformed to random direction by random degree. This process can simply alter the uniform shaped grains into non-uniform shaped grains. However, to keep our space full-filled with grains, the transformation of vertices should be performed in concurrency with that of

neighboring grains. Each surface is shared with by two grains, each edge by three and each vertex by four grains; that is, the right side vertex in a grain is the left side vertex of the grain which is in the right. So, to handle vertices of one grain, the changes in neighboring grains should be also considered.

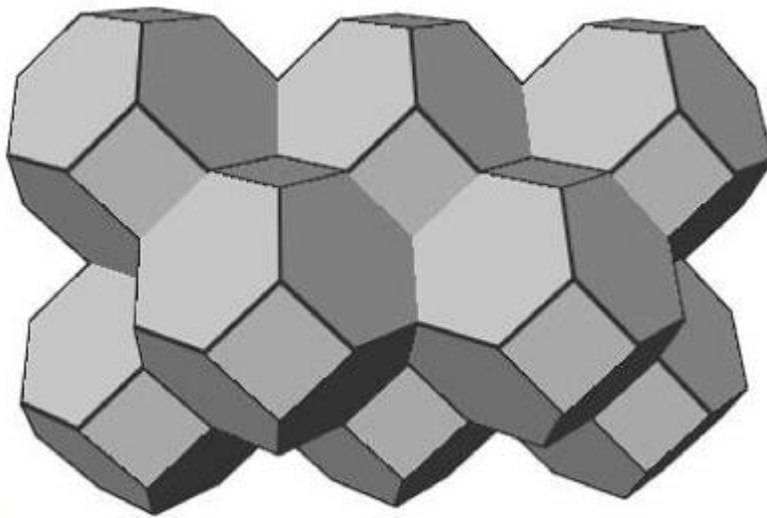


Figure 2.1 : Array of eight identical tetrakaidecahedra which are uniform shape and size.

2.1 Definition of uniform grain structure

Each grain is represented as a set of vertices not a set of edges in this method, because demanded number of components is small; just 24 vertices are needed to define a single tetrakaidecahedron, where 36 edges are needed. Thus first step of the

new calculation is the determination of vertex coordinates of the grain array, which is still a uniform structure.

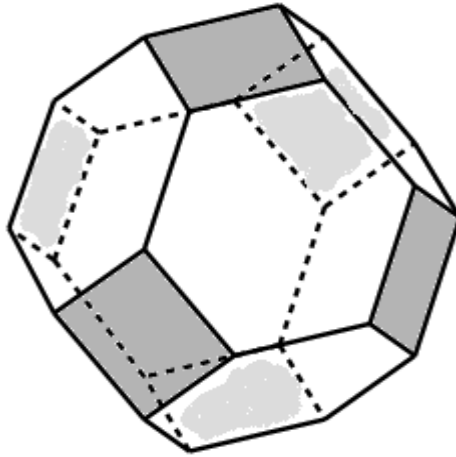


Figure 2.2 : A single tetrakaidecahedron. All 24 vertices are included in the six squares.

Starting from a single tetrakaidecahedron grain, six square faces of the polyhedron do not share any vertex, but they include all vertices (Figure 2.2). The vertices which consist of eight hexagonal faces are already included in them. Briefly a tetrakaidecahedron can be represented by defining its six square faces. But things are different when considering arrays; just three faces need to be defined, because the remaining three belong to neighboring grains and hence are defined later. This situation can be explained more easily in a simplified example. In Figure 2.3, squares are arranged in an array. To define a single square, it is necessary to define

four edges should be defined. However to define repeated squares, just two edges are needed. In the case of square 1 in Figure 2.3, edges c and d define square 3 and square 2 later, because edges c and d of square 1 are exactly edges a and b of square 3 and 2 respectively. In the same way, a tetrakaidecahedron in an array can be defined by just three squares which represent half of the entire polyhedron.

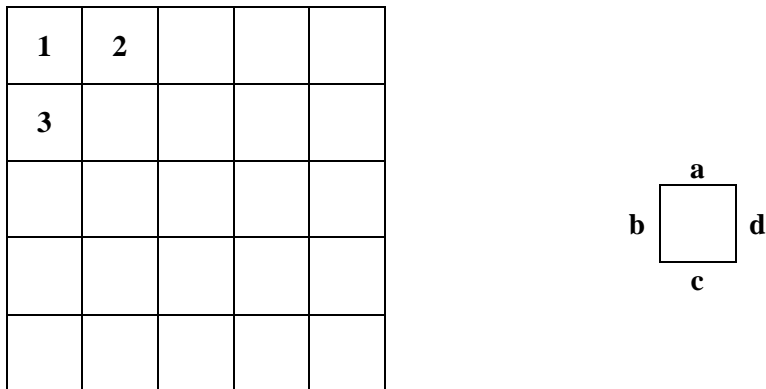


Figure 2.3 : Simplified example of grain array in 2D. Each square in the Figure can correspond to a tetrakaidecahedron grain in 3D.

Now the work to define a uniform grain structure is simplified to define repetitively three squares for grains in an array. Consider a single tetrakaidecahedron grain of edge length a ; the distance between opposing square faces is designated as a size of grain, $L = 2\sqrt{2}a$ in Figure 2.4(a). The orthogonal coordinates x , y and z axes are defined normal to the square faces. The grain is then represented by three square faces placed at the bottom, back and left of the polyhedron. Figure 2.4(b) is the

sectional diagram of tetrakaidecahedron observed in an orthogonal position of each square, in which the shape is an octagon with a regular square in the center. The edge length of each dotted square in Figure 2.4(b) is the quarter of grain size, $L_q = L/4$, and it is standard to determine the coordinates of square vertices. The coordinates of 12 vertices of 3 squares are described in table 2.1. They represent just relative positions in each tetrakaidecahedron grain, not absolute coordinates in the whole space.

Square position	Coordinate
Bottom (x-y plane)	$(L_q, 2L_q, 0)$
	$(2L_q, L_q, 0)$
	$(2L_q, 3L_q, 0)$
	$(3L_q, 2L_q, 0)$
Left (x-z plane)	$(L_q, 0, 2L_q)$
	$(2L_q, 0, L_q)$
	$(2L_q, 0, 3L_q)$
	$(3L_q, 0, 2L_q)$
Back (y-z plane)	$(0, L_q, 2L_q)$
	$(0, 2L_q, L_q)$
	$(0, 2L_q, 3L_q)$
	$(0, 3L_q, 2L_q)$

Table 2.1 : The relative positions of square vertices in a grain, where $L_q = L/4$.

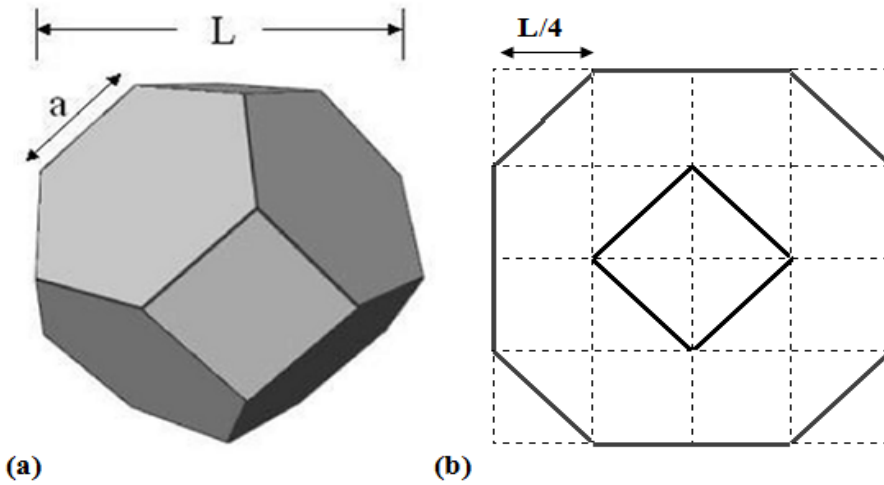


Figure 2.4 : (a) A single tetrakaidecahedron of edge length a . (b) The sectional diagram of tetrakaidecahedron observed in orthogonal position of each square.

To obtain the absolute coordinates of the vertices, those of the starting positions in each grain should be calculated in advance. The coordinates of the vertices are then calculated simply by adding the relative positions of each vertex, as illustrated in table 2.1, to the resultant starting coordinates. The starting coordinate of the (i, j, k) th grain represented as:

$$(i * L, j * L, k * L) \tag{2-1}$$

where, L is the size of the grain and i, j and k are the sequences of the grain in each axis.

The vertex coordinates of three squares can be acquired directly using the equations in table 2.2. The remaining work to define uniform shapes of the grains is the determination of the numbers of grains through the axes, and to calculate the coordinates of the vertices for each grain repetitively. Since just three squares are defined for one tetrakaidecahedron grain, the final grains in each axis will not be defined completely. They do not have the next neighboring grains which can generate their top, front and right square faces. For that reason, an extra row of grains has to be considered for each axis.

Square position	Coordinate
Bottom (x-y plane)	$(i*L+L_q, j*L+2L_q, k*L)$
	$(i*L+2L_q, j*L+L_q, k*L)$
	$(i*L+2L_q, j*L+3L_q, k*L)$
	$(i*L+3L_q, j*L+2L_q, k*L)$
Left (x-z plane)	$(i*L+L_q, j*L, k*L+2L_q)$
	$(i*L+2L_q, j*L, k*L+L_q)$
	$(i*L+2L_q, j*L, k*L+3L_q)$
	$(i*L+3L_q, j*L, k*L+2L_q)$
Back (y-z plane)	$(i*L, j*L+L_q, k*L+2L_q)$
	$(i*L, j*L+2L_q, k*L+L_q)$
	$(i*L, j*L+2L_q, k*L+3L_q)$
	$(i*L, jL+3L_q, kL+2L_q)$

Table 2.2 : The coordinates of square vertices of (i, j, k) th grain, where $L_q = L/4$ and i, j , and k is the sequence of the grain in array.

Notice that just half of vertices are known for each grain. The remainder must be extracted from other grains at a later stage in the calculation. It is even possible to calculate all the 24 vertices of six squares not just half of them. However that would cause the duplication of definition, and all the defined vertices should be affected simultaneously by deformation. That is the reason why we do not define all the vertices for each grain.

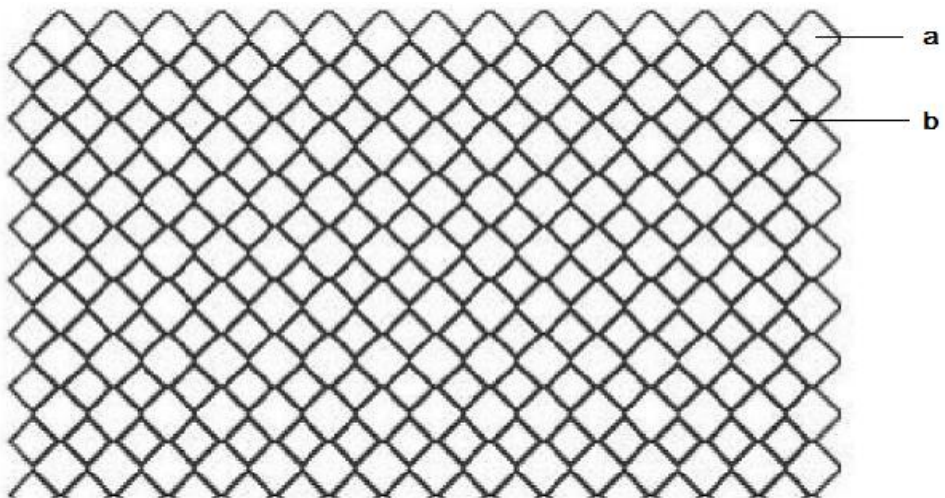


Figure 2.5 : Section through a regular stack of uniform tetrakaidecahedra, where grain **a** is one of the odd grains and grain **b** of the even grains.

2.2 The number of the defined grain, odd & even grain

The number of grains which are defined is not equal to the number of grains which exist in that structure. Additional grains exist at the crevice position of every eight tetrakaidecahedra, which are located at the eight summit positions of cube. Figure 2.5 is a section through a regular stack of uniform tetrakaidecahedra. **a** grains are defined ones and can be observed in the surface of total structure, while **b** grains are not defined but obviously exist among **a** grains. **a** grains are referred to as *odd grains*, **b** grains are *even grains* in this study. Those two kinds of grains will not be in the same layer.

Approximately, double the number of grains will be generated than intended. If we define vertices for $I \times J \times K$ grains, which are odd grains, $(I - 1) \times (J - 1) \times (K - 1)$ additional even grains will be generated. The volume considered in this study has the dimensions $20L \times 20L \times 15L$ with $L = 5\mu m$, enclosing a total of 12,000 grains. For this uniform array of grains, the edges are henceforth referred to as the *primary grain edges*.

2.3 Conversion of uniform grain structure into non-uniform array

A non-uniform but space-filling grain structure is then generated by a topological transformation in which the vertices are randomly perturbed to new positions. The resulting non-uniform grain still has 24 vertices, a maximum of 44 triangular surfaces and a maximum 66 edges of various lengths; 66 vectors are therefore needed to describe each grain. The additional edges are henceforth referred to as the *secondary grain edges* and connect just two grains; they occur in real grain structure, appearing as protrusions in two dimensional sections (Figure 2.6). Note that the geometry of each planar surface can be any polygon, not simply triangular.

There are two topological transformations to achieve the non-uniform grain structure. The first is designated the *randomized vertex method* (RVM), in which the vertex position r_0 is transformed into the new location r according to

$$r = r_0 + \frac{\omega}{4}(\xi - 0.5)L \quad (2-2)$$

where $\omega \leq 1$ is a weight which defines the extent of the fluctuation, and $0 < \xi < 1$ is a random number.

Figure 2.6 shows a grain generated using the random perturbations and the corresponding set of eight grains similarly generated. A comparison them with Figures 2.1 and 2.2 may help understand the process.

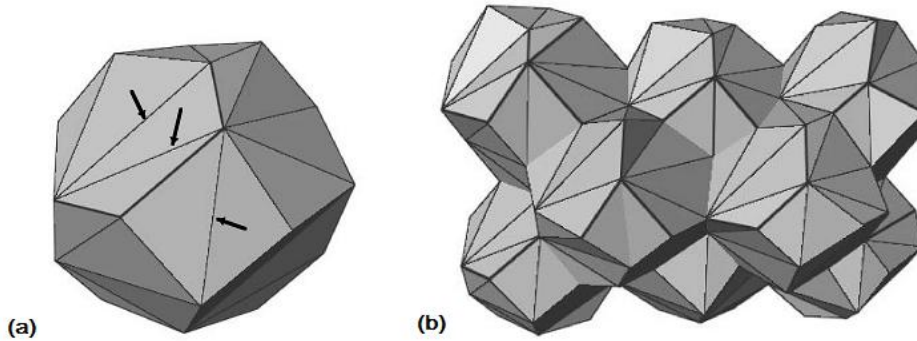


Figure 2.6 : (a) A non-uniform grain generated using RVM. Note that additional secondary edges on the faces of grain, some of which are identified by the arrows. (b) A space-filling of eight non-uniform grains similarly generated. Notice that each grain is different.

The second topological transformation is the *rescale randomized vertex method* (RRVM) in which the vertex is modified as follows:

$$|r - r_{ref}| = \zeta |r_0 - r_{ref}| \quad (2-3)$$

where r_{ref} is a reference point inside the grain, $0 \leq \zeta \leq 1$ is a weight factor which can be defined in a number of ways. For the present purpose, four cases are considered:

$$\text{Case 1 : } \zeta = \xi$$

$$\text{Case 2 : } \zeta = \sqrt{\xi}$$

$$\text{Case 3 : } \zeta = \xi^{2.5}$$

$$\text{Case 4 : } \zeta = \exp[-2(\xi - 1)^2]$$

(2-4)

In this, ς is a fixed number which is taken from ξ where the latter is equivalent to white noise which is evenly distributed between 0 and 1.

2.4 Extracting vertex coordinates

All the vertices in uniform grain structures were defined and then perturbed into the positions of non-uniform grain structures in previous step. Then the vertices coordinates are required to calculate the surface area and edge length of a grain. Every grain was regarded as a tetrakaidecahedron shape and then changed into non-uniform shapes, so all the grains (even random-shaped grains) have 24 vertices as a tetrakaidecahedron. Therefore the work to do in this step is to find out those 24 vertices for each grain. Odd grains and even grains will be handled separately, e.g., (0, 0, 0)th odd grain and (0, 0, 0)th even grain are distinguished. Notice that all the vertices have already been defined in a previous step, and they will be just extracted in this step.

It is relatively easy to obtain the coordinates of odd grains, because the 12 vertices for each odd grain were already known. Those 12 vertices represent the 3 squares which are in the bottom, left and back of a tetrakaidecahedron shape. The remaining 12 vertices for the top, right and front squares can be obtained from neighboring grains (Table 2.3).

Square position	Neighboring Grain
Top (x-y plane)	Bottom square of $(i, j, k+1)th$ grain
Right (x-z plane)	Left square of $(i, j+1, k)th$ grain
Front (y-z plane)	Rear square of $(i+1, j, k)th$ grain

Table 2.3 : The relationship between not defined square faces and faces of neighboring grains.

Every even grain is surrounded by eight neighboring odd grains, so all the vertices of an even grain are also the vertices of those eight odd grains. Therefore, to obtain the vertices coordinates of an even grain, the 8 odd grains and their vertices should be considered. Actually just 7 of 8 neighboring odd grains are related in this extracting process, because not all vertices were defined for each grain. The number of vertices which are shared with neighboring odd grains is listed in table 2.4. As mentioned above, $(i, j, k)th$ odd grain is different with $(i, j, k)th$ even grain.

Odd grain position	Number of vertices affected
$(i, j, k)th$	0
$(i+1, j, k)th$	2
$(i, j+1, k)th$	2
$(i+1, j+1, k)th$	4
$(i, j, k+1)th$	2
$(i+1, j, k+1)th$	4
$(i, j+1, k+1)th$	4
$(i+1, j+1, k+1)th$	6

Table 2.4 : The vertices of $(i, j, k)th$ even grain, all of them are from the neighboring odd grains.

2.5 Deformation

Once the grain structure is defined and its vertex coordinates of it are extracted, it is possible to homogeneously deform it by applying an appropriate mathematical deformation matrix S to each vertex u to generate the result deformed vertex v :

$$\begin{pmatrix} S_{11} & S_{12} & S_{13} \\ S_{21} & S_{22} & S_{23} \\ S_{31} & S_{32} & S_{33} \end{pmatrix} \begin{pmatrix} u_1 \\ u_2 \\ u_3 \end{pmatrix} = \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} \quad (2-2)$$

It is the same method which is described in the previous study [Zhu *et al.*, 2007],

but rotation matrix \mathbf{R} is not necessary because the grain shape is not uniform any more. The matrix representations of a variety of common deformations are listed in table 2.5, and it is emphasized that this method can be used for arbitrary homogeneous deformations. The details are not repeated here.

Type	S_{11}	S_{12}	S_{13}	S_{21}	S_{22}	S_{23}	S_{31}	S_{32}	S_{33}
Plain strain compression	≥ 1	0	0	0	1	0	0	0	$1/S_{11}$
Axisymmetric compression	$1/(S_{33})^{1/2}$	0	0	0	$1/(S_{33})^{1/2}$	0	0	0	≤ 1
Axisymmetric tension	≥ 1	0	0	0	$1/(S_{11})^{1/2}$	0	0	0	$1/(S_{11})^{1/2}$
Simple shear	1	0	+ve	0	1	0	0	0	1

Table 2.5 : volume preserving deformations (the convention used is that $S_{11} > S_{22} > S_{33}$)

III. Results

A total of 12,000 grains within a block $20L \times 20L \times 15L$ with $L = 5 \mu\text{m}$ were defined for the calculations presented here. Initially the shapes of those grains are all identical and space-filling, and then converted into a non-uniform structure by randomly perturbing the vertices. The resulting non-uniform but space-filling grains are homogeneously deformed by applying a deformation matrix to their vertices. Then the changes in the edge length and surface area are calculated mathematically to derive L_V and S_V . To compare them with the result of previous study [Zhu *et al.*, 2007] is the main purpose of this study. All of these processes have been progressed by compiling a computer program and the results will be described in this step.

3.1 Non-uniform grain generation

In the randomized vertex method (RVM, equation 2-2), the vertices of tetrakaidehedra grains are transformed into random positions to generate non-uniform grain. So the grain size distribution is affected by the transformation. Figure 3.1 shows how the distribution of grains changes from a uniform grain

volume of $62.5 \mu\text{m}^3$ as a function of the weight ω . The method clearly is successful in producing a three-dimensional distribution of grain sizes.

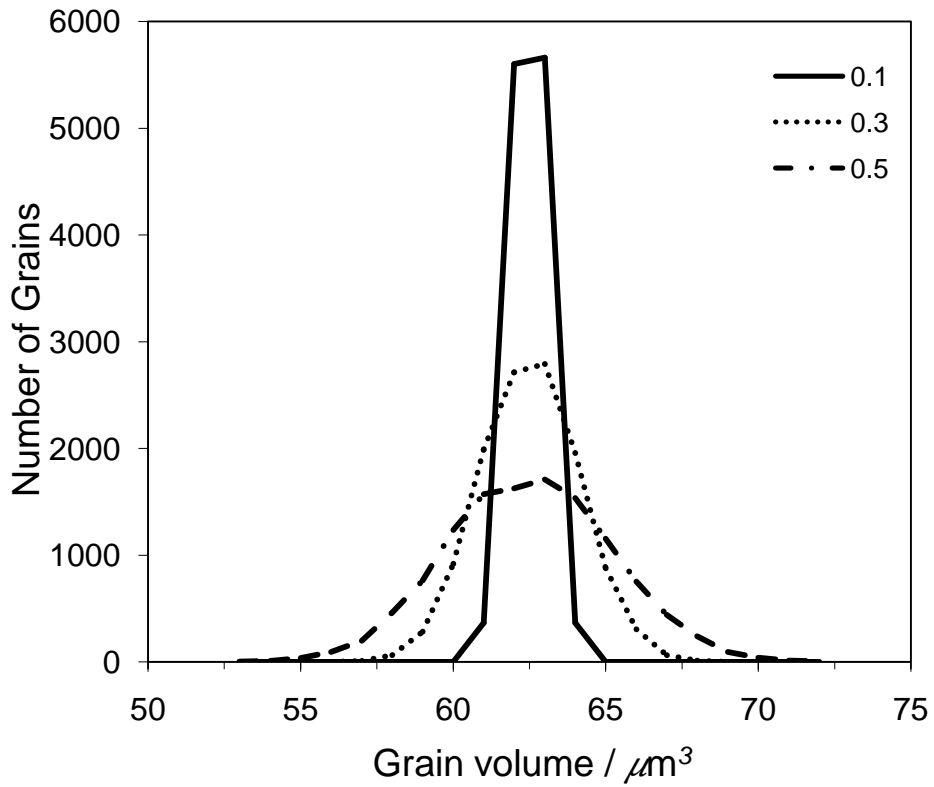


Figure 3.1 : Grain volume distribution using the RVM and a variety of weights. The total number of grains is 12,000. The number plotted on the vertical axis represents the value for $x \pm 0.5 \mu\text{m}^3$.

The grain volume distribution after the rescale randomized vertex method (RRVM, equation 2-3) transformation of the four cases in equation 2-4 is also presented in Figure 3.2. It can be seen that some values are more probable than others. Eight kinds of non-uniform grains are defined, which are of cases 1~4 in RRVM with ω at 0 and 0.5 in RVM.

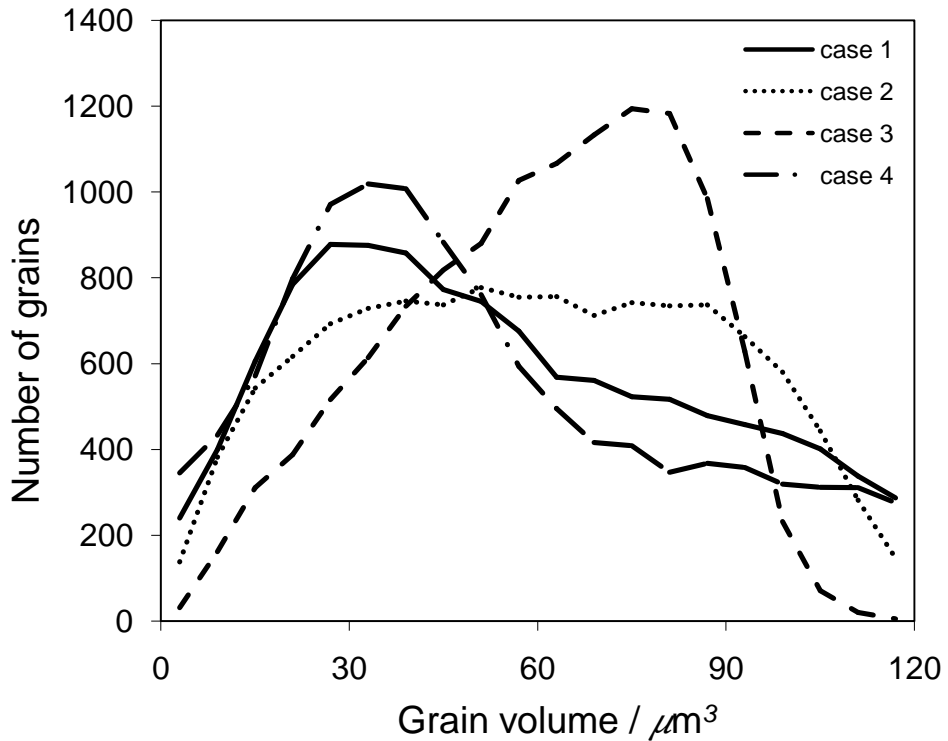


Figure 3.2 : Grain volume distribution using RRVM topological transformation for the four cases corresponding to equation 2-4 with $0 \leq \xi \leq 1$. The total number of grains is 12,000. The number plotted on the vertical axis represents the value for $x \pm 3 \mu\text{m}^3$.

Figure 3.3 illustrates two-dimensional sections which cut by $x = 57 \mu\text{m}$ plane through the defined 12,000 grains. Figure 3.3(a) is the section cutting through uniform tetrakaidecahedron grains. Two different size squares form the grain boundary, and they are designated as even and odd grains in this study. Figure 3.3(b) illustrates the same section after RVM (equation 2-2) with the $\omega = 0.5$. It can be seen that the grain boundaries are distorted away from squares in Figure 3.3(a) and much differences of grain size occur. Figure 3.3(c) gives the section after RRVM deformation by case 2. The geometry of grain boundary as well as grain size is completely different from those in Figure 3.3(a). Both of Figure 3.3(b) and Figure 3.3(c) show that non-uniform but space-filling grains have successfully been generated.

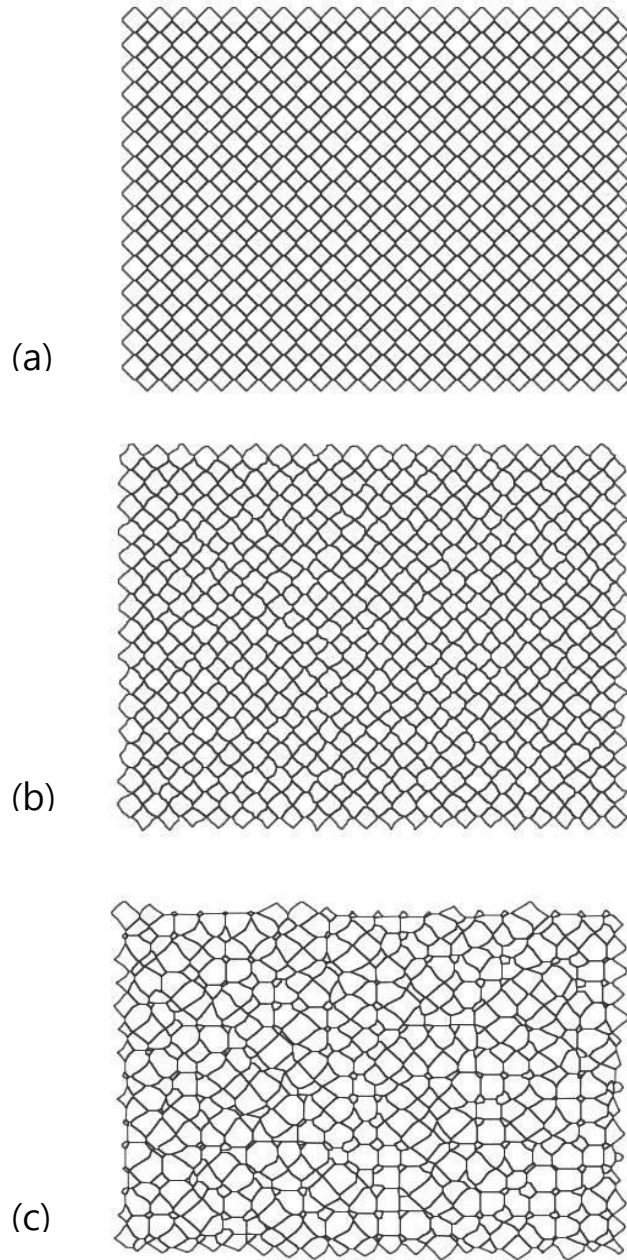


Figure 3.3 : (a) Section through a regular stack of uniform tetrakaidecahedra with $L = 5 \mu\text{m}$. (b) Same sectional after RVM with $\omega = 0.5$, (c) after RRVM Case 2 transformation added.

3.2 Applying various deformations

Once non-uniform grain structures are successfully defined, various deformation effects are applied to them. Then the changes in grain surface area and edge length are compared with the results for uniform grain [Zhu *et al.*, 2007]. Deformations are applied in four cases, which are plane strain compression, axisymmetric compression, axisymmetric tension and simple shear.

Figure 3.4 shows the effects of plane strain compression. S_V/S_{V0} represents the ratio of surface area per unit volume normalized by the quantity of undeformed grain, where L_V/L_{V0} is the corresponding ratio for edge length including both primary and secondary edges. The curve in each figure represents the analytical outcome for a uniform grain structure, while the other small points are the outcomes for eight non-uniform grain structures. The individual cases are not distinguished, because they produced similar results. It is evident that the different brought about by introducing a non-uniform grain structure on the grain surface and edge length ratio is not significant.

Figures 3.5 and 3.6 show the effects of axisymmetric compression and axisymmetric tension respectively. They also do not make much difference between the analytical solution for a uniform grain structure and the results for eight non-uniform grains. Only in the L_V/L_{V0} of axisymmetric compression deformation, the outcomes of non-uniform grains are less than expected from uniform grains. This is

because the non-uniform grains are less isotropic. Edges can contract as well as expand during compression depending on their orientation relative to the principal axes of the deformation. The effect of orientation is smaller for a regular tetrakaidecahedron than for an object with large anisotropy. Figure 3.7 is the result for simple shear deformation, with the horizontal axes both plotted in terms of shear strain instead of equivalent strain.

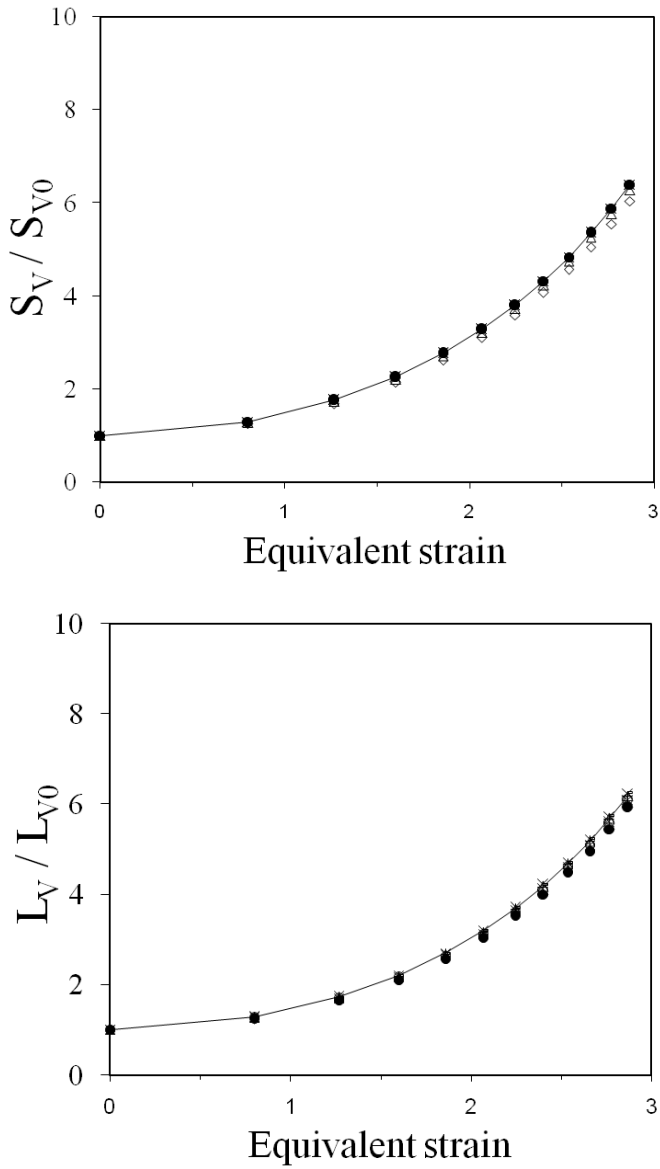


Figure 3.4 : Calculation results for plane strain compression deformation, S_V/S_{V0} and L_V/L_{V0} respectively. The curve in each graph is the analytical solution for a uniform set of tetrakaidehedra, where the other points are from the calculations from the non-uniform grain structures.

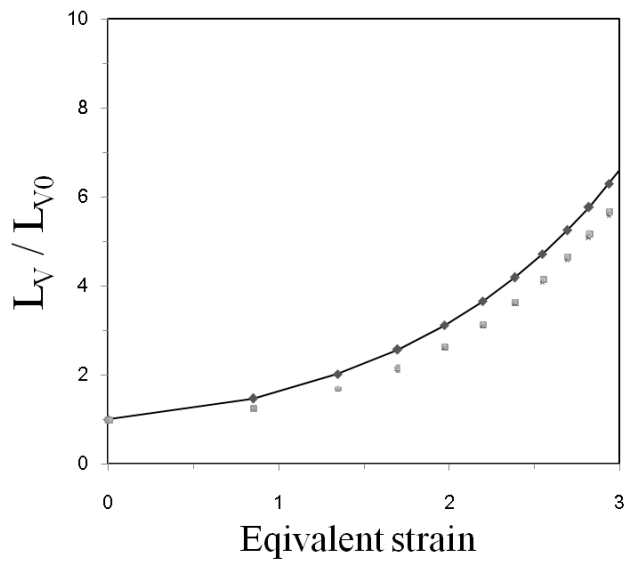
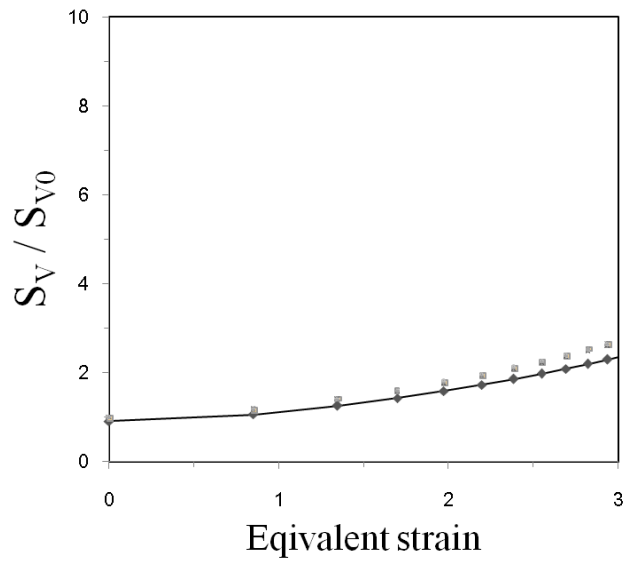


Figure 3.5 : Results from axisymmetric compression deformation. It can be seen that the ratio L_V / L_{V0} is smaller than the analytical solution from uniform grain.

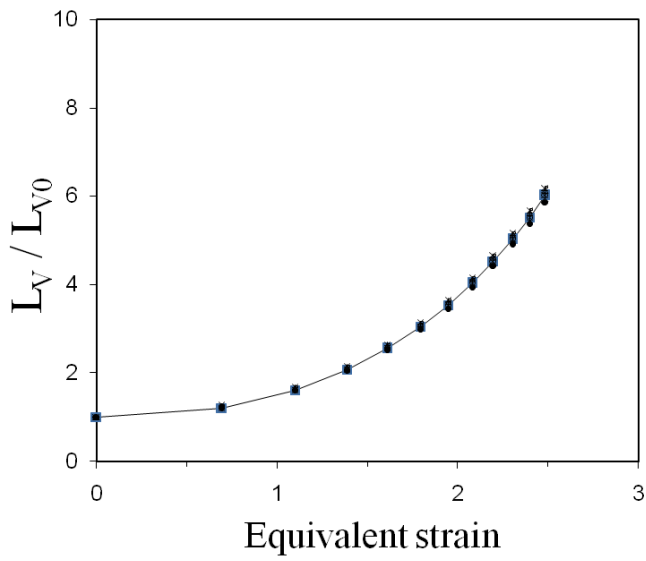
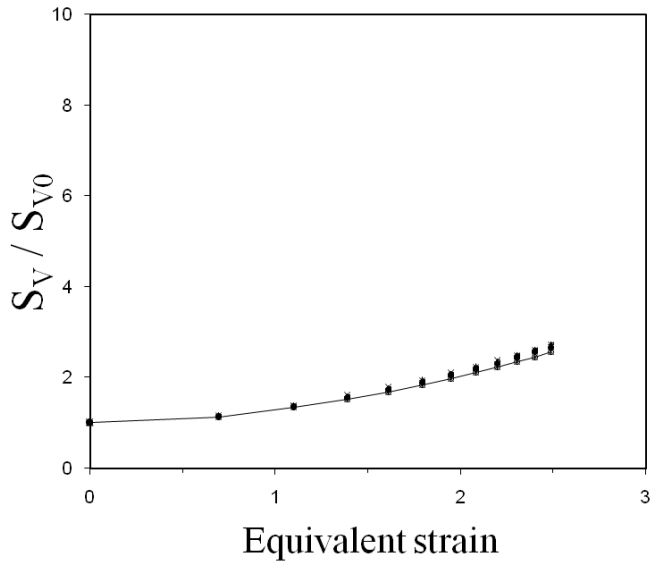


Figure 3.6 : Results from axisymmetric tension deformation. Not much difference with the result from uniform grains.

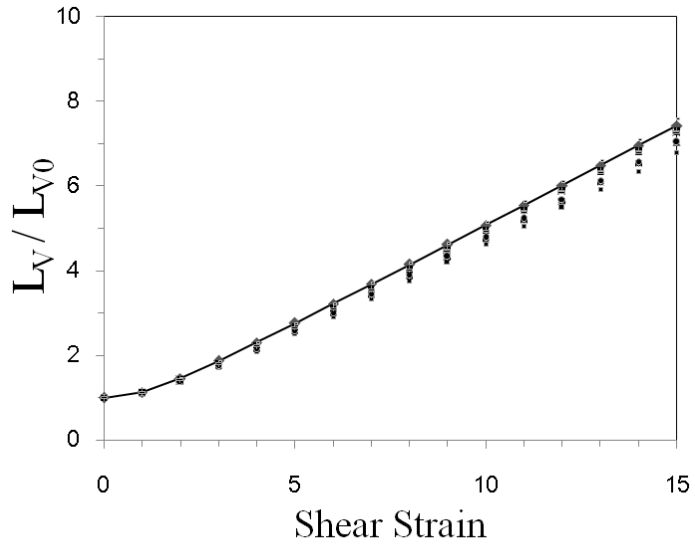
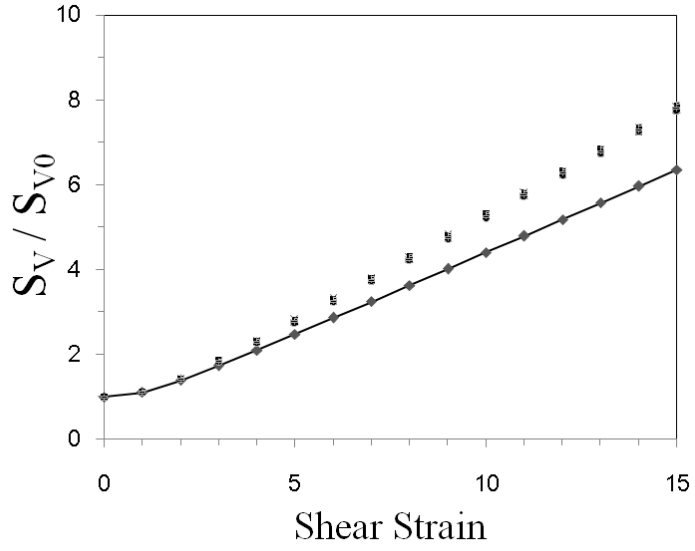


Figure 3.7 : Calculations for simple shear deformation. The results are illustrated with the x -axis in terms of shear stress, while other deformations are equivalent strain.

3.3 Analysis of the results

The reason why the results do not change much when the grain structure becomes non-uniform is that the surface area per unit volume (S_{V0}) and primary edge length per unit volume (L_{V0}^P) do not change dramatically when there is a distribution of grain sizes and shapes within the reasonable limits considered in this study.

It is worth discussing the physical significance of the secondary edge length per unit volume (L_{V0}^S). In this model, the faces of grain are geometrically constrained to be flat, for both uniform and non-uniform grain structures. In order to accommodate non-uniformity whilst at the same time fill space, it is necessary during the topological transformations to introduce these secondary edges. As pointed out earlier, these can be real, but many of them could be regarded simply as introducing curvature in the grain faces of non-uniform structures. This would drive grain growth.

IV. Summary and future work

The effects of various plastic deformations on the grain structure are important factors in the mathematical modeling of phase transformations and recrystallisation processes for the design of a particular metallic material. The effects can be quantitatively expressed as the change in the amount of grain boundary area per unit volume (S_V) and grain edge length per unit volume (L_V). To calculate those parameters for non-uniform grains was the purpose of this study, which is the extension of a previous study for uniform grains [Zhu *et al.*, 2007].

The method for generating non-uniform grain structure and how to handle the generated grains has been demonstrated here. All the vertices of uniform tetrakaidecahedra are perturbed in random directions and degrees to be converted into the non-uniform structure; some mathematical equations are considered for this process. Then several deformations, such as plane strain compression, axisymmetric compression, axisymmetric tension and simple shear, are applied to the converted non-uniform grains. S_V and L_V for each deformation are calculated by the same method with the previous study [Zhu *et al.*, 2007] but the results do not show big differences with those for uniform grains. That means that the non-uniformity in the grain structure does not sufficiently affect the change in grain size as a function of

deformation. Although no significant difference was obtained in present work, the procedures to define and to handle the non-uniform objects can be used practically in related works.

Non-uniform grain structure has been generated successfully from uniform grain structure. However, the distributions of grain size and shape are totally randomized and difficult to control using the method developed. In future work, the size and shape of grain will be determined intentionally instead of a randomized procedure. Any grain structure will be able to be defined and applied by deformations for particular purpose. For example, plate grain structure in martensite can be defined and its changes would be observed as deformations are applied. Advances in this kind of model might reduce the resources needed to design particular materials.

References

Annelie, E. : Application of mathematical modeling to hot-rolling and controlled cooling of wire rod and bars, *ISIJ International*, Vol. 32, pp. 440-449, 1992

Bate, P. and Hutchinson, W. B. : Grain boundary area and deformation, *Scripta Materialia*, Vol. 52, pp. 199-203, 2005

Bhadeshia, H. K. D. H. : 'Worked Examples in the Geometry of Crystals', *The Institute of Materials*, London, Second edition, 2001

Bhadeshia, H. K. D. H. and Honeycombe R. W. K. : Steels - Microstructure and Properties, *Butterworth-Heinemann*, Third edition, 2006

Bhadeshia, H. K. D. H. and Chae, J. Y. : MAP_STEEL_TOPOLOGY,
<http://www.msm.cam.ac.uk/map/steel/programs/topology.html>.

Black, M. P., Higginson, R. L. and Sellars, C. M. : Effect of strain path on recrystallisation kinetics during hot rolling of Al-Mn, *Materials Science and Technology*, Vol. 17, pp. 1055-1060, 1970

Callister, W. D. : Materials science and engineering - An introduction, *John Wiley & Sons*, Seventh edition, 2006

Cahn, J. W. : The kinetics of grain boundary nucleated reactions. *Acta Metallurgica*, Vol. 4, pp. 449-459, 1956

Czinege, I. and Reti, T. : Determination of local deformation in cold formed products by a measurement of the geometric characteristics of the crystallites, *18th*

International Machine Tool Design and Research Conference, Vol. 1, pp 159-163, 1977

Denis, S., Farias, D. and Simon, A. : Mathematical model coupling phase transformations and temperature evolutions in steels, *ISIJ International*, Vol. 32, pp. 316-325, 1992

Gil-Sevillano, J., van Houtte, P. and Aernoudt, E. : *Progress in Material Science*, Vol. 27, pp. 69-314, 1980

Jones, S. and Bhadeshia, H. K. D. H. : Kinetics of the simultaneous decomposition of austenite into several transformation products, *Acta Materialia*, Vol. 45, pp. 2911-2920, 1997

Knustad, O., McQueen, H. J., Ryum, N. and Solberg, J. K. : Polarized light observation of grain extension and subgrain formation in aluminium deformed at 400 °C to very high strains, *Practical Metallography*, Vol. 22, pp. 215, 1985

Kumar, A., McCulloch, C., Hawbolt, T. B. and Samarasekera, I. V. : Modelling thermal and microstructural evolution on runout table of hot strip mill, *Materials Science and Technology*, Vol. 7, pp. 360-368, 1991

Lee, K. J., Lee, J. K., Kang, K. B., and Kwon, O. : Mathematical modeling of transformation in Nb microalloyed steels, *ISIJ International*, Vol. 32, pp. 326-334, 1992

Leeuwen, Y., Vooijs, S., Sietsma, J. and Van der Zwaag, S. : Effect of geometrical assumptions in modeling solid state kinetics, *Metallurgical & Materials Transactions. A*, 29A, pp. 2925-2931, 1998

Martin, G. and Tsang, S : *Jour. Engng. Indust.*, Vol. 88, pp. 273, 1966

Matsuura, K. and Itoh, Y. : Estimation of three-dimensional grain size distribution in polycrystalline material, *Materials Transactions JIM*, Vol. 32, pp. 1042-1047, 1991

Offerman, S. E., van Dijk, N. H., Sietsma, J., Grigull, S., Lauridsen, E. M., Margulies, L., Poulsen, H. F., Rekveldt, M. Th. and van der Zwaag, S. : Grain nucleation and growth during phase transformations, *SCIENCE*, Vol. 298, 2002

Orsetti, P. L. and Sellars, C. M. : *Acta Metallurgica*, Vol. 45, pp. 137-148, 1997

Proksa, F. : *Stahlbau*, Vol. 28, pp. 29, 1959

Reed, R. C. and Bhadeshia, H. K. D. H. : Kinetics of reconstructive austenite to ferrite transformation in low-alloy steels. *Materials Science and Technology*, Vol. 8, pp. 421-435, 1992

Saito, K., Igaki, H. and Sugimoto, M. : A study on the equivalent stress and the equivalent plastic strain rate, *The Japan Society of Mechanical Engineers*, Vol. 15, pp. 33-39, 1972

Sellars, C. M. : Modelling microstructural development during hot-rolling. *Materials Science and Technology*, Vol. 6, pp. 1072-1081, 1990

Sellars, C. M. : Modelling distribution of micro structure during hot-rolling of stainless steel, *Materials Science and Technology*, Vol. 8, pp. 1090-1094, 1992

Sellars, C. M. : Basics of modeling for control of microstructure in

thermomechanical control processing, *Ironmaking and Steelmaking*, Vol. 22, pp. 459-464, 1995

Singh, S. B. , Bhadeshia, H. K. D. H. :Topology of grain deformation, *Materials Science and Technol.*, Vol. 15, pp. 832-834, 1998

Soto, Juan Ignacio : A general deformation matrix for three-dimensions, *Mathematical Geology*, Vol. 29, pp. 93-130, 1997

Takayama, Y., Furushiro, N., Tozawa, T., Kato, H. and Hori, S. : A significant method for estimation of the grain size of polycrystalline materials, *Materials Transactions JIM*, Vol. 32, pp. 214-221, 1991

Umemoto, M., Ohtsuka, H. and Tamura, I. :Transformation to pearlite from work hardened austenite, *Trans. ISIJ*, Vol. 23, pp. 775-784, 1983

Umemoto, M., Guo, Z. H. and Tamura, I. : Effect of cooling rate on grain size of ferrite in a carbon steel, *Materials Science and Technology*, Vol. 3, pp. 249-255, 1987

Underwood, E.E.: 'Quantitative stereology', Addison-Wesley Publication Company, 1970

Vatne, H. E., Furu, T., Orsund. R. and Nes, E. : Modelling recrystallization after hot deformation of aluminium, *Acta Metallurgica.*, Vol. 44, pp. 4473-4473, 1980

Zhu, Q. , Sellars, C. M. and Bhadeshia, H. K. D. H. : Quantitative metallography of deformed grains, *Material Science and Technology*, Vol. 23, pp. 757-766, 2007

Appendix A

This is the documentation for the program in this work.

Program

MAP_TOPOLOGY_NONUNIFORMGRAIN_DEFORMATION

1. Provenance
2. Purpose
3. Specification
4. Description
5. References
6. Parameter
7. Error indicators
8. Accuracy estimate
9. Further comments
10. Example
11. Auxiliary routines
12. Keywords

1. Provenance of Source code

Jae Yong Chae, Rongshan Qin and H. K. D. H. Bhadeshia

Graduate Institute of Ferrous Technology (GIFT)

Pohang University of Science and Technology

Pohang, Kyungbuk, Republic of Korea

exfeel@postech.ac.kr

2. Purpose

To generate the non-uniform grain structure and to calculate the change in grain surface area per unit volume and grain edge length per unit volume as a function of various deformations.

3. Specification

Language: C++

Product form: Executables and complete source codes

4. Description

Non-uniform grain structures are generated in the form of vertex coordinates. The degree of distribution in grain sizes can be regulated at eight different levels. The generated grains are then deformed using one of the five deformation methods. The

five deformation types that this program can support are as follows:

- Plane strain deformation
- Axisymmetric tension
- Axisymmetric compression
- Simple shear
- Any user-inputted deformation which is homogeneous

The degree of distribution and the deformation type are selected in compile level, and then the program runs a corresponding routine. The final to initial values of grain surface area per unit volume and grain edge length per unit volume are calculated and printed out for each case.

The program runs best on Microsoft Visual C++ compiler.

All the files are compressed into a file called
Visual_metal.tar

The **.tar** file contains the following files:

Random.h	Header files for variables
Tetrakaidecahedra.h	
Main.cpp	Main module. Degree of the distribution in grain sizes, deformation type, and other values needed are selected

	here.
Random.cpp	Module to generate a random value.
Tetrakaidecahedra.cpp	Generate, deform and print out all data.
*.dat	Output files which contain the calculation results. File names are determined by the deformation type and grain generating degree.
Tetra100.dat	Output file which contains the vertex coordinates of non-uniform grains
Visualmetal.exe	Executive file

5. Reference

Zhu, Q., Sellars, C. M. and Bhadeshia, H. K. D. H. : Quantitative metallography of deformed grains, *Material Science and Technology*, Vol. 23 (2007) pp. 757-766

Jae-Yong, C., Rongshan, Q. and Bhadeshia, H. K. D. H. : Topology of the deformation of a non-uniform grain structure, submitted to *Material Science and Technology*.

6. Parameters

Input parameters

The input variables are determined before compile.

caseNumber : weight in RRVM

deformCase : deformation mode

Output parameters

Vertex coordinates are listed in **Tetra100.dat**. Array of three numbers which represent *x*, *y* and *z* respectively are repeated.

The calculation results are printed in other ***.dat** files. The format of each file is:

strain theoryEdge edgeRatio 2ndaryEdge primaryEdge
theorySurface areaRatio

7. Error Indicators

None.

8. Accuracy

No information.

9. Further Comments

User interface will be improved later.

10. Example

10.1 Set *caseNumber* and *deformCase* in **main.cpp**. Case 1 in RRVM and plane strain deformation are selected here.

```
int caseNumber= 1;
int deformCase = 1;
/* DeformCase:
    1. Plane deformation
    2. Axisymmetric tension
    3. Axisymmetric Compression
    4. Simple shear
    5. Complex deformation
*/
```

10.2 Compile

10.3 Run “*VisualMetal.exe*”

10.4 “**tetra100.dat**” and “**plainDeformCase1.dat**” are generated.

Tetra100.dat																								
12.5	4	4	3	1200	5	2.5	3.75	0	1.25	2.5	0	2.5	1.25	0	3.75	2.5	0	2.5	0	1.25	1.25	0	2.5	2.5
0	3.75	3.75	0	2.5	0	3.75	2.5	0	2.5	3.75	0	1.25	2.5	0	2.5	1.25	2.5	3.75	5	1.25	2.5	5	2.5	
1.25	5	3.75	2.5	5	2.5	0	6.25	1.25	0	7.5	2.5	0	8.75	3.75	0	7.5	0	3.75	7.5	0	2.5	8.75	0	
1.25	7.5	0	2.5	6.25	2.5	3.75	10	1.25	2.5	10	2.5	1.25	10	3.75	2.5	10	2.5	0	11.25	1.25	0			
12.5	2.5	0	13.75	3.75	0	12.5	0	3.75	12.5	0	2.5	13.75	0	1.25	12.5	0	2.5	11.25	2.5	3.75				
15	1.25	2.5	15	2.5	1.25	15	3.75	2.5	...															

PlainDeformCase1.dat

```

strain theoryEdge edgeRatio 2ndaryEdge primaryEdge theorySurface areaRatio
0 1 1 582866 264741 1 1
0.800377 1.28174 1.24217 727004 325869 1.29131 1.25315
1.26857 1.73241 1.65835 974892 430739 1.75855 1.68068
1.60075 2.21275 2.11114 1.24504e+006 544380 2.25745 2.14383
1.85842 2.70229 2.57736 1.52339e+006 661202 2.76686 2.61987
2.06895 3.19578 3.0501 1.80571e+006 779569 3.28105 3.10206
2.24694 3.69132 3.52656 2.09032e+006 898826 3.79782 3.58769
2.40113 4.18807 4.0054 2.37636e+006 1.01865e+006 4.31615 4.07546
2.53714 4.68558 4.48586 2.66338e+006 1.13887e+006 4.83549 4.56464
2.6588 5.18362 4.96749 2.9511e+006 1.25938e+006 5.35552 5.05483
2.76885 5.68204 5.44999 3.23934e+006 1.38011e+006 5.87605 5.54576
2.86932 6.18072 5.93316 3.52798e+006 1.50101e+006 6.39696 6.03724
fluct=0 Lvo= 847607 Svo= 508057 V= 701889
0 1 1 585359 264900 1 1
0.800377 1.28174 1.24093 729040 326076 1.29131 1.25307
1.26857 1.73241 1.6561 977031 431081 1.75855 1.68045
1.60075 2.21275 2.108 1.24743e+006 544910 2.25745 2.14352
1.85842 2.70229 2.57339 1.5261e+006 661953 2.76686 2.61953
2.06895 3.19578 3.04535 1.80877e+006 780566 3.28105 3.10171
2.24694 3.69132 3.52107 2.09374e+006 900086 3.79782 3.58736
2.40113 4.18807 3.99919 2.38016e+006 1.02019e+006 4.31615 4.07515
2.53714 4.68558 4.47895 2.66756e+006 1.1407e+006 4.83549 4.56438
2.6588 5.18362 4.95988 2.95568e+006 1.26151e+006 5.35552 5.05462
2.76885 5.68204 5.4417 3.24431e+006 1.38255e+006 5.87605 5.54561
2.86932 6.18072 5.92421 3.53335e+006 1.50376e+006 6.39696 6.03715
fluct=0.5 Lvo= 850260 Svo= 508406 V= 701862

```

11. Auxiliary Routines

None.

12. Keywords

topology, metallography, deformation, non-uniform grain

Appendix B

Several source codes are presented.

[Main.cpp]

```
#include "Tetrakaidecahedra.h"
#include <fstream>
#include <math.h>
#include <iostream>
using namespace std;

int main()
{
    int caseNumber=1;
    int deformCase = 1;
    /* DeformCase:
       1. Plain deformation
       2. Axisymmetric tension
       3. Axisymmetric Compression
       4. Simple shear
       5. Complex deformation
    */
    ofstream file;

    file.open("plainDeformCase1.dat");
    //file.open("axisymmetricTensionDeformCase1.dat");
    //file.open("axisymmetricCompressionDeformCase1.dat");
    //file.open("simpleShearCase4.dat");
    //file.open("omplexDeformation-ps13-16.dat");
    double s13=16.0;
    int ix=2, iy=2, iz=2;
    double aL, aL0, aSv, aSv0, aLP, aLS, aV0;
    double epsplain, edgtheory, surftheory, dbi;
    Tetrakaidecahedra tetra(4, 4, 3, 5);
    //printf("para Sv/Sv0 L/L0
    aSv/aSv0 aL/aL0");
    //printf("para aSv/aSv0 aL/aL0");
    file<<"strain"<<" "<<"theoryEdge"<<"
"<<"edgeRatio"
    <<" "<<"2ndaryEdge"<<"
"<<"primaryEdge"
    <<" "<<"theorySurface"<<"
"<<"areaRatio"<<"\n";

    for(double fl=0.; fl<0.51; fl+=0.5)
    {
        printf("\n\nfluctuation = %f\n", fl);
        //for(int i=1; i<2; i++){
        for(int i=0; i<16; i++){
            tetra.calcVtxCoord();
            tetra.randomizeVtx(fl);
            tetra.zoomAllRandomTetra(caseNumber);
            //tetra.outputVertex("tetra100.dat");
            //exit(1);
            dbi = i*1.0;

            if(deformCase == 1)
                tetra.setPlainDeformMatrix(dbi);

            if(deformCase == 2)

            tetra.setAxisymmetricTensionDeformMatrix(dbi);
            if(deformCase == 3)

            tetra.setAxisymmetricCompressionDeformMatrix(dbi);
            if(deformCase == 4)
                tetra.setSimpleShearDeformMatrix(dbi);

            if(deformCase == 5)

            tetra.setDeformationMatrix(dbi,0,s13/dbi,0,1,0,0,0,dbi);
            tetra.deformVtx();

            if((i == 1 && deformCase !=4) || (i == 0 && deformCase == 4)){
                aL0 = tetra.getEdgeOfAllRandomTetra();
                aSv0 =
            tetra.getSurfaceAreaOfAllRandomTetra();
                aV0 = tetra.getVolumeOfAllRandomTetra();
            }

            if(deformCase == 1){ // plain strain
                epsplain = 2.0*log(dbi)/sqrt(3.0);

                edgtheory=(1+dbi+2*sqrt(1.0+dbi*dbi+2/dbi/dbi))/6.0;

                surftheory=(dbi+3*(dbi*sqrt(1.0+2.0/dbi/dbi)+sqrt(dbi*dbi+2.0/dbi/dbi))+sqrt(2.0*(1+dbi*dbi))/dbi)/(3*(2.0*sqrt(3.0)+1.0));
            }
            if(deformCase == 2){ // axisymmetric tension
                epsplain = log(dbi);

                edgtheory=(dbi+1.0/dbi+2.0*sqrt(dbi*dbi+3.0/dbi))/6.0;

                surftheory=(sqrt(3*dbi)+sqrt(dbi+2.0/dbi/dbi)+1.0/3.0+sqrt(2*dbi+2/dbi/dbi)/3.0)/(2*sqrt(3.0)+1.0);
            }
            if(deformCase == 3){ // axisymmetric compression
                epsplain = sqrt(1.5)*log(dbi);

                edgtheory=(sqrt(dbi)+sqrt(2*dbi*dbi+2/dbi))/3.0;

                surftheory=(sqrt(8.0*dbi+4.0/dbi/dbi)+(2*sqrt(dbi)+1.0/dbi)/3.0)/(2*sqrt(3.0)+1.0);
            }
            if(deformCase == 4){ // simple shear
```

```

    epsplain = dbi;
    edgtheory=(1+sqrt(1+dbi*dbi/2.0-
dbi/sqrt(2.0))+sqrt(1+dbi*dbi/2.0+dbi/sqrt(2.0)
))/3.0;

    surftheory=(2+4*sqrt(1+dbi*dbi/4)+12*sqrt(3.
0/4.0)
    +6*sqrt(3.0/4.0+dbi*dbi/2+dbi/sqrt(2.))
    +2*(sqrt(3+2*dbi*dbi-
4*dbi/sqrt(2.0))+sqrt(3.0/4.0+dbi*dbi/2
-dbi/sqrt(2.0))))/(2*sqrt(3.0)+1.0)/6.0;
    }
    if(deformCase == 5)
        epsplain = 2.0*log(dbi)/sqrt(3.0);
        aL = tetra.getEdgeOfAllRandomTetra();
        aLP =
tetra.getPrimaryEdgeOfAllRandomTetra();
        aLS =
tetra.getSecondaryEdgeOfAllRandomTetra();
        aSv =
tetra.getSurfaceAreaOfAllRandomTetra();
        //tetra.sortVolumeCategory();

        file<<(float)epsplain<<"
"<<(float)edgtheory<<" "<<(float)aL/aL0
<<" "<<(float)aLS<<" "<<(float)aLP
<<" "<<(float)surftheory<<"
"<<(float)aSv/aSv0<<"\n";
        //file<<(float)epsplain<<"
"<<(float)aL/aL0<<" "<<(float)aSv/aSv0<<"\n";
        //file<<i<<"\n";
    }
    file<<"fluct="<<fl<<" "<<"Lvo="<<"
"<<(float)aL0<<" "<<"Svo="<<"
"<<(float)aSv0
<<" "<<"V="<<" "<<(float)aV0
<<"\n";
    }
    file.close();
    return 1;
}

```

[Random.cpp]

```

#include "Random.h"

void Random::setSeed(int seed)
{
    int i, ii, j, jj, k, l, m, me;
    double s, t;
    me = seed;
    i=(me + 12)%178 + 1;
    j=(me + 34)%178 + 1;
    k=(me + 56)%178 + 1;

    l=(me + 78)%168 + 1;
    ir=97;
    jr=33;
    for(ii=1; ii<=97; ii++){
        s=0.0;
        t=0.5;
        for(jj=1; jj<=24; jj++){
            m=((i*j)%179)*k%179;
            i=j;
            j=k;
            k=m;
            l=(53*1+1)%169;
            if(l*m)%64 >= 32)
                s=s+t;
            t=0.5*t;
        }
        uni[ii-1]=s;
    }
    c = 362436./16777216.;
    cd= 7654321./16777216.;
    cm=16777213./16777216.;
    uni[97]=cd;
    uni[98]=cm;
    uni[99]=c;
    uni[100]=double(ir);
    uni[101]=double(jr);
};
double Random::getValue()
{
    cd=uni[97];
    cm=uni[98];
    c=uni[99];
    ir=int(uni[100]);
    jr=int(uni[101]);
    duni=uni[jr]-uni[ir];
    if(duni < 0.)
        duni=duni+1.;
    uni[ir]=duni;
    ir=ir-1;
    if(ir == -1)
        ir=96;
    jr=jr-1;
    if(jr == -1)
        jr=96;
    c=c-cd;
    if(c < 0.)
        c=c+cm;
    duni=duni-c;
    if(duni < 0.)
        duni=duni+1.;
    uni[97]=cd;
    uni[98]=cm;
    uni[99]=c;
    uni[100]=double(ir);
    uni[101]=double(jr);
    return duni;
};
#endif //RANDOM_CPP

```


[Tetrakaidecahedra.cpp]

```
#ifndef TETRAKAIDECAHEDRA_CPP
#define TETRAKAIDECAHEDRA_CPP

#include "Tetrakaidecahedra.h"

Tetrakaidecahedra::Tetrakaidecahedra(int
numberOfGrainX, int numberOfGrainY, int
numberOfGrainZ, double grainSize)
:ngx(numberOfGrainX), ngy(numberOfGrainY),
ngz(numberOfGrainZ), gs(grainSize)
{
    if(gs < 1.0E-10){
        printf("grain size is too small, mission
aborted\n");
        exit(1);
    }
    hgs = gs/2.0;
    qgs = gs/4.0;
    ngt = 2*ngx*ngy*ngz;
    //ngt =
(((ngx+1)/2)*(ngy+1)/2)*((ngz+1)/2)+(ngx/2
)*(ngy/2)*(ngz/2));
    dbacurate = 1.0E-9;
    if(ngt == 0){
        printf("Not available for ordering zero
number of tetrakaidecahedra, mission aborted\n");
        exit(1);
    }
    xtg = ngx+1; ytg = ngy+1; ztg = ngz+1;
    //xtg = (ngx+1)/2+1; ytg = (ngy+1)/2+1; ztg =
(ngz+1)/2+1;
    nvt = xtg * ytg * ztg *12;

    vcx = new double[nvt];
    vcy = new double[nvt];
    vcz = new double[nvt];
    vtx = new double[72];
    sij = new double[9];
    if(vcx == NULL || vcy == NULL || vcz == NULL){
        printf("low memory, aborted\n");
        exit(1);
    }
    rdm.setSeed(4407);
    volfile.open("thisVolumeDistri0505.dat");
}

Tetrakaidecahedra::~Tetrakaidecahedra()
{
    if(vcx != NULL)
        delete [] vcx;
    if(vcy != NULL)
        delete [] vcy;
    if(vcz != NULL)
        delete [] vcz;
    delete [] vtx;

    delete [] sij;
}

void Tetrakaidecahedra::outputVertex(char*
filename){
    file.open(filename);
    double xm, ym, zm, tm;
    xm = hgs*(1 + ngx);
    ym = hgs*(1 + ngy);
    zm = hgs*(1 + ngz);
    if(xm>ym)
        tm=xm;
    else
        tm=ym;
    if(tm<zm)
        tm=zm;
    maxdim = (float)tm;
    //file<<maxdim<<" "<<ngx<<" "<<ngy<<"
"<<ngz<<" "<<nvt<<" "<<gs<<" ";
    for(int i=0; i<nvt; i++)
        file<<float(vcx[i])<<" "<<float(vcy[i])<<"
"<<float(vcz[i])<<"\n";
    file.close();
}

void
Tetrakaidecahedra::setDeformationMatrix(double
s11, double s12, double s13,
double s21, double s22, double s23,
double s31, double s32, double s33)
{
    sij[0]=s11; sij[1]=s12, sij[2]=s13;
    sij[3]=s21; sij[4]=s22, sij[5]=s23;
    sij[6]=s31; sij[7]=s32, sij[8]=s33;
}

void
Tetrakaidecahedra::setPlainDeformMatrix(double
s11)
{
    if((float)s11 == 0.0){
        printf("s11=0 is not allowed\n"); exit(0);
    }
    setDeformationMatrix(s11, 0, 0, 0, 1, 0, 0, 0,
1.0/s11);
}

void
Tetrakaidecahedra::setAxisymmetricTensionDefor
mMatrix(double s11)
{
    if((float)s11 == 0.0){
        printf("s11=0 is not allowed\n"); exit(0);
    }
    setDeformationMatrix(s11, 0, 0, 0, 1.0/sqrt(s11),
0, 0, 0, 1.0/sqrt(s11));
}

void
```

```

Tetrakaidecahedra::setAxisymmetricCompression
DeformMatrix(double s33)
{
    if((float)s33 == 0.0){
        printf("s33=0 is not allowed\n"); exit(0);
    }
    setDeformationMatrix(1.0/sqrt(s33), 0, 0, 0,
1.0/sqrt(s33), 0, 0, 0, s33);
}

```

```

void
Tetrakaidecahedra::setSimpleShearDeformMatrix(
double s13)
{
    setDeformationMatrix(1, 0, s13, 0, 1, 0, 0, 0, 1);
}

```

```

void Tetrakaidecahedra::deformVtx()
{
    double temx, temy, temz;
    for(int i=0; i<nvt; i++){
        temx = sij[0] * vcx[i] + sij[1] * vcy[i] + sij[2] *
vcz[i];
        temy = sij[3] * vcx[i] + sij[4] * vcy[i] + sij[5] *
vcz[i];
        temz = sij[6] * vcx[i] + sij[7] * vcy[i] + sij[8] *
vcz[i];
        vcx[i] = temx;
        vcy[i] = temy;
        vcz[i] = temz;
    }
}

```

```

void Tetrakaidecahedra::calcVtxCoord()
{
    int vertt = 0; // just a parameter
    double x0, x1, x2, x3, y0, y1,y2,y3, z0, z1, z2, z3; //
just parameters

    for(int i=0; i<xtg; i++){
        x0 = i*gs; x1=x0+qgs; x2=x1+qgs; x3=x2+qgs;
        for(int j=0; j<ytg; j++){
            y0 = j*gs; y1=y0+qgs; y2=y1+qgs;
y3=y2+qgs;
            for(int k=0; k<ztg; k++){
                z0 = k*gs; z1=z0+qgs; z2=z1+qgs; z3
=z2+qgs;
                // z plane back
                // 0
                vcx[vertt] = x2; vcy[vertt] = y3; vcz[vertt] =
z0; vertt++;
                // 1
                vcx[vertt] = x1; vcy[vertt] = y2; vcz[vertt] =
z0; vertt++;
                // 2
                vcx[vertt] = x2; vcy[vertt] = y1; vcz[vertt] =
z0; vertt++;
                // 3
                vcx[vertt] = x3; vcy[vertt] = y2; vcz[vertt] =

```

```

z0; vertt++;

                // y plane bottom
                // 4
                vcx[vertt] = x2; vcy[vertt] = y0; vcz[vertt] =
z1; vertt++;
                // 5
                vcx[vertt] = x1; vcy[vertt] = y0; vcz[vertt] =
z2; vertt++;
                // 6
                vcx[vertt] = x2; vcy[vertt] = y0; vcz[vertt] =
z3; vertt++;
                // 7
                vcx[vertt] = x3; vcy[vertt] = y0; vcz[vertt] =
z2; vertt++;

                // x plane left
                // 8
                vcx[vertt] = x0; vcy[vertt] = y3; vcz[vertt] =
z2; vertt++;
                // 9
                vcx[vertt] = x0; vcy[vertt] = y2; vcz[vertt] =
z3; vertt++;
                // 10
                vcx[vertt] = x0; vcy[vertt] = y1; vcz[vertt] =
z2; vertt++;
                // 11
                vcx[vertt] = x0; vcy[vertt] = y2; vcz[vertt] =
z1; vertt++;
            }
        }
    }
}

```

```

// put it back here
void Tetrakaidecahedra::setOneTetrakaideVtx(int
&xpos, int &ypos, int &zpos)
{
    int vertt;
    int xp, yp, zp;
    if(xpos%2==1 && ypos%2==1 &&
zpos%2==1){
        xp = (xpos-1)/2; yp = (ypos-1)/2; zp = (zpos-
1)/2;
        vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
        vcx[vertt+5] = vtx[0];
        vcy[vertt+5] = vtx[1];
        vcz[vertt+5] = vtx[2];

        vcx[vertt+10] = vtx[3];
        vcy[vertt+10] = vtx[4];
        vcz[vertt+10] = vtx[5];

        vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
        vcx[vertt+7] = vtx[6];
        vcy[vertt+7] = vtx[7];
        vcz[vertt+7] = vtx[8];

        vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;

```

```

vcx[vertt+8] = vtx[9];
vcy[vertt+8] = vtx[10];
vcz[vertt+8] = vtx[11];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+5] = vtx[12];
vcy[vertt+5] = vtx[13];
vcz[vertt+5] = vtx[14];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vcx[vertt+8] = vtx[15];
vcy[vertt+8] = vtx[16];
vcz[vertt+8] = vtx[17];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+7] = vtx[18];
vcy[vertt+7] = vtx[19];
vcz[vertt+7] = vtx[20];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+10] = vtx[21];
vcy[vertt+10] = vtx[22];
vcz[vertt+10] = vtx[23];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+4] = vtx[24];
vcy[vertt+4] = vtx[25];
vcz[vertt+4] = vtx[26];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+2] = vtx[27];
vcy[vertt+2] = vtx[28];
vcz[vertt+2] = vtx[29];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+6] = vtx[30];
vcy[vertt+6] = vtx[31];
vcz[vertt+6] = vtx[32];

vertt = ((xp*ytg+yp)*ztg+zp+1)*12;
vcx[vertt] = vtx[33];
vcy[vertt] = vtx[34];
vcz[vertt] = vtx[35];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+6] = vtx[36];
vcy[vertt+6] = vtx[37];
vcz[vertt+6] = vtx[38];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+2] = vtx[39];
vcy[vertt+2] = vtx[40];
vcz[vertt+2] = vtx[41];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+4] = vtx[42];
vcy[vertt+4] = vtx[43];
vcz[vertt+4] = vtx[44];

```

```

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vcx[vertt] = vtx[45];
vcy[vertt] = vtx[46];
vcz[vertt] = vtx[47];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+1] = vtx[48];
vcy[vertt+1] = vtx[49];
vcz[vertt+1] = vtx[50];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+9] = vtx[51];
vcy[vertt+9] = vtx[52];
vcz[vertt+9] = vtx[53];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+3] = vtx[54];
vcy[vertt+3] = vtx[55];
vcz[vertt+3] = vtx[56];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vcx[vertt+11] = vtx[57];
vcy[vertt+11] = vtx[58];
vcz[vertt+11] = vtx[59];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vcx[vertt+1] = vtx[60];
vcy[vertt+1] = vtx[61];
vcz[vertt+1] = vtx[62];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vcx[vertt+11] = vtx[63];
vcy[vertt+11] = vtx[64];
vcz[vertt+11] = vtx[65];

vertt = ((xp*ytg+yp)*ztg+zp+1)*12;
vcx[vertt+3] = vtx[66];
vcy[vertt+3] = vtx[67];
vcz[vertt+3] = vtx[68];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vcx[vertt+9] = vtx[69];
vcy[vertt+9] = vtx[70];
vcz[vertt+9] = vtx[71];
}else if(xpos%2==0 && ypos%2==0 &&
zpos%2==0){
xp = xpos/2; yp = ypos/2; zp = zpos/2;
vertt = ((xp*ytg+yp)*ztg+zp+1)*12;

vcx[vertt+3] = vtx[0];
vcy[vertt+3] = vtx[1];
vcz[vertt+3] = vtx[2];

vcx[vertt] = vtx[3];
vcy[vertt] = vtx[4];
vcz[vertt] = vtx[5];

vcx[vertt+1] = vtx[6];
vcy[vertt+1] = vtx[7];

```

```

vcz[vertt+1] = vtx[8];

vcx[vertt+2] = vtx[9];
vcy[vertt+2] = vtx[10];
vcz[vertt+2] = vtx[11];

vertt = ((xp*ytg+yp)*ztg+zp)*12;
vcx[vertt+3] = vtx[12];
vcy[vertt+3] = vtx[13];
vcz[vertt+3] = vtx[14];

vcx[vertt+2] = vtx[15];
vcy[vertt+2] = vtx[16];
vcz[vertt+2] = vtx[17];

vcx[vertt+1] = vtx[18];
vcy[vertt+1] = vtx[19];
vcz[vertt+1] = vtx[20];

vcx[vertt+0] = vtx[21];
vcy[vertt+0] = vtx[22];
vcz[vertt+0] = vtx[23];

vcx[vertt+9] = vtx[24];
vcy[vertt+9] = vtx[25];
vcz[vertt+9] = vtx[26];

vcx[vertt+8] = vtx[27];
vcy[vertt+8] = vtx[28];
vcz[vertt+8] = vtx[29];

vcx[vertt+11] = vtx[30];
vcy[vertt+11] = vtx[31];
vcz[vertt+11] = vtx[32];

vcx[vertt+10] = vtx[33];
vcy[vertt+10] = vtx[34];
vcz[vertt+10] = vtx[35];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vcx[vertt+11] = vtx[36];
vcy[vertt+11] = vtx[37];
vcz[vertt+11] = vtx[38];

vcx[vertt+8] = vtx[39];
vcy[vertt+8] = vtx[40];
vcz[vertt+8] = vtx[41];

vcx[vertt+9] = vtx[42];
vcy[vertt+9] = vtx[43];
vcz[vertt+9] = vtx[44];

vcx[vertt+10] = vtx[45];
vcy[vertt+10] = vtx[46];
vcz[vertt+10] = vtx[47];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vcx[vertt+7] = vtx[48];
vcy[vertt+7] = vtx[49];

vcz[vertt+7] = vtx[50];

vcx[vertt+4] = vtx[51];
vcy[vertt+4] = vtx[52];
vcz[vertt+4] = vtx[53];

vcx[vertt+5] = vtx[54];
vcy[vertt+5] = vtx[55];
vcz[vertt+5] = vtx[56];

vcx[vertt+6] = vtx[57];
vcy[vertt+6] = vtx[58];
vcz[vertt+6] = vtx[59];

vertt = ((xp*ytg+yp)*ztg+zp)*12;
vcx[vertt+7] = vtx[60];
vcy[vertt+7] = vtx[61];
vcz[vertt+7] = vtx[62];

vcx[vertt+6] = vtx[63];
vcy[vertt+6] = vtx[64];
vcz[vertt+6] = vtx[65];

vcx[vertt+5] = vtx[66];
vcy[vertt+5] = vtx[67];
vcz[vertt+5] = vtx[68];

vcx[vertt+4] = vtx[69];
vcy[vertt+4] = vtx[70];
vcz[vertt+4] = vtx[71];
}else{
printf("the Tetra that you requested is neither
odd nor even, aborted\n");
exit(1);
}
}

void Tetrakaidecahedra::getOneTetrakaideVtx(int
&xpos, int &ypos, int &zpos){
int vertt;
int xp, yp, zp;
if(xpos >= ngx || xpos < 0 || ypos >= ngy || ypos
< 0 || zpos >= ngz || zpos < 0){
printf("the Tetra that you requested is out of
board, aborted\n"); exit(1);}
else if(xpos%2==1 && ypos%2==1 &&
zpos%2==1){
xp = (xpos-1)/2; yp = (ypos-1)/2; zp = (zpos-
1)/2;
vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vtx[0] = vcx[vertt+5];
vtx[1] = vcy[vertt+5];
vtx[2] = vcz[vertt+5];

vtx[3] = vcx[vertt+10];
vtx[4] = vcy[vertt+10];
vtx[5] = vcz[vertt+10];
}
}

```

```

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vtx[6] = vcx[vertt+7];
vtx[7] = vcy[vertt+7];
vtx[8] = vcz[vertt+7];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vtx[9] = vcx[vertt+8];
vtx[10] = vcy[vertt+8];
vtx[11] = vcz[vertt+8];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vtx[12] = vcx[vertt+5];
vtx[13] = vcy[vertt+5];
vtx[14] = vcz[vertt+5];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vtx[15] = vcx[vertt+8];
vtx[16] = vcy[vertt+8];
vtx[17] = vcz[vertt+8];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vtx[18] = vcx[vertt+7];
vtx[19] = vcy[vertt+7];
vtx[20] = vcz[vertt+7];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vtx[21] = vcx[vertt+10];
vtx[22] = vcy[vertt+10];
vtx[23] = vcz[vertt+10];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vtx[24] = vcx[vertt+4];
vtx[25] = vcy[vertt+4];
vtx[26] = vcz[vertt+4];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vtx[27] = vcx[vertt+2];
vtx[28] = vcy[vertt+2];
vtx[29] = vcz[vertt+2];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vtx[30] = vcx[vertt+6];
vtx[31] = vcy[vertt+6];
vtx[32] = vcz[vertt+6];

vertt = ((xp*ytg+yp)*ztg+zp+1)*12;
vtx[33] = vcx[vertt];
vtx[34] = vcy[vertt];
vtx[35] = vcz[vertt];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vtx[36] = vcx[vertt+6];
vtx[37] = vcy[vertt+6];
vtx[38] = vcz[vertt+6];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vtx[39] = vcx[vertt+2];
vtx[40] = vcy[vertt+2];

```

```

vtx[41] = vcz[vertt+2];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vtx[42] = vcx[vertt+4];
vtx[43] = vcy[vertt+4];
vtx[44] = vcz[vertt+4];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vtx[45] = vcx[vertt];
vtx[46] = vcy[vertt];
vtx[47] = vcz[vertt];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vtx[48] = vcx[vertt+1];
vtx[49] = vcy[vertt+1];
vtx[50] = vcz[vertt+1];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp)*12;
vtx[51] = vcx[vertt+9];
vtx[52] = vcy[vertt+9];
vtx[53] = vcz[vertt+9];

vertt = ((xp*ytg+yp+1)*ztg+zp+1)*12;
vtx[54] = vcx[vertt+3];
vtx[55] = vcy[vertt+3];
vtx[56] = vcz[vertt+3];

vertt = (((xp+1)*ytg+yp+1)*ztg+zp+1)*12;
vtx[57] = vcx[vertt+11];
vtx[58] = vcy[vertt+11];
vtx[59] = vcz[vertt+11];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vtx[60] = vcx[vertt+1];
vtx[61] = vcy[vertt+1];
vtx[62] = vcz[vertt+1];

vertt = (((xp+1)*ytg+yp)*ztg+zp+1)*12;
vtx[63] = vcx[vertt+11];
vtx[64] = vcy[vertt+11];
vtx[65] = vcz[vertt+11];

vertt = ((xp*ytg+yp)*ztg+zp+1)*12;
vtx[66] = vcx[vertt+3];
vtx[67] = vcy[vertt+3];
vtx[68] = vcz[vertt+3];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vtx[69] = vcx[vertt+9];
vtx[70] = vcy[vertt+9];
vtx[71] = vcz[vertt+9];
}else if(xpos%2==0 && ypos%2==0 &&
zpos%2==0){
xp = xpos/2; yp = ypos/2; zp = zpos/2;
vertt = ((xp*ytg+yp)*ztg+zp+1)*12;
vtx[0] = vcx[vertt+3];
vtx[1] = vcy[vertt+3];
vtx[2] = vcz[vertt+3];

```

```

vtx[3] = vcx[vertt];
vtx[4] = vcy[vertt];
vtx[5] = vcz[vertt];

vtx[6] = vcx[vertt+1];
vtx[7] = vcy[vertt+1];
vtx[8] = vcz[vertt+1];

vtx[9] = vcx[vertt+2];
vtx[10] = vcy[vertt+2];
vtx[11] = vcz[vertt+2];

vertt = ((xp*ytg+yp)*ztg+zp)*12;
vtx[12] = vcx[vertt+3];
vtx[13] = vcy[vertt+3];
vtx[14] = vcz[vertt+3];

vtx[15] = vcx[vertt+2];
vtx[16] = vcy[vertt+2];
vtx[17] = vcz[vertt+2];

vtx[18] = vcx[vertt+1];
vtx[19] = vcy[vertt+1];
vtx[20] = vcz[vertt+1];

vtx[21] = vcx[vertt+0];
vtx[22] = vcy[vertt+0];
vtx[23] = vcz[vertt+0];

vtx[24] = vcx[vertt+9];
vtx[25] = vcy[vertt+9];
vtx[26] = vcz[vertt+9];

vtx[27] = vcx[vertt+8];
vtx[28] = vcy[vertt+8];
vtx[29] = vcz[vertt+8];

vtx[30] = vcx[vertt+11];
vtx[31] = vcy[vertt+11];
vtx[32] = vcz[vertt+11];

vtx[33] = vcx[vertt+10];
vtx[34] = vcy[vertt+10];
vtx[35] = vcz[vertt+10];

vertt = (((xp+1)*ytg+yp)*ztg+zp)*12;
vtx[36] = vcx[vertt+11];
vtx[37] = vcy[vertt+11];
vtx[38] = vcz[vertt+11];

vtx[39] = vcx[vertt+8];
vtx[40] = vcy[vertt+8];
vtx[41] = vcz[vertt+8];

vtx[42] = vcx[vertt+9];
vtx[43] = vcy[vertt+9];
vtx[44] = vcz[vertt+9];

vtx[45] = vcx[vertt+10];

vtx[46] = vcy[vertt+10];
vtx[47] = vcz[vertt+10];

vertt = ((xp*ytg+yp+1)*ztg+zp)*12;
vtx[48] = vcx[vertt+7];
vtx[49] = vcy[vertt+7];
vtx[50] = vcz[vertt+7];

vtx[51] = vcx[vertt+4];
vtx[52] = vcy[vertt+4];
vtx[53] = vcz[vertt+4];

vtx[54] = vcx[vertt+5];
vtx[55] = vcy[vertt+5];
vtx[56] = vcz[vertt+5];

vtx[57] = vcx[vertt+6];
vtx[58] = vcy[vertt+6];
vtx[59] = vcz[vertt+6];

vertt = ((xp*ytg+yp)*ztg+zp)*12;
vtx[60] = vcx[vertt+7];
vtx[61] = vcy[vertt+7];
vtx[62] = vcz[vertt+7];

vtx[63] = vcx[vertt+6];
vtx[64] = vcy[vertt+6];
vtx[65] = vcz[vertt+6];

vtx[66] = vcx[vertt+5];
vtx[67] = vcy[vertt+5];
vtx[68] = vcz[vertt+5];

vtx[69] = vcx[vertt+4];
vtx[70] = vcy[vertt+4];
vtx[71] = vcz[vertt+4];
} else {
    printf("the Tetra that you requested is neither
odd nor even, aborted\n");
    exit(1);
}
}

double Tetrakaidehedra::distance(int point1, int
point2)
{
    double *pt1=vtx+3*point1;
    double *pt2=vtx+3*point2;
    double dist;
    dist = sqrt((*pt1-*pt2)*(*pt1-*pt2)+(*(pt1+1)-
*(pt2+1))*(*(pt1+1)-*(pt2+1))
+(*(pt1+2)-*(pt2+2))*(*(pt1+2)-*(pt2+2)));

    pt1=NULL;
    pt2=NULL;
    return dist;
}

```

```

double Tetrakaidecahedra::trangleArea(int point1,
int point2, int point3)
{
    double a, b, c, s;
    a=distance(point1, point2);
    b=distance(point1, point3);
    c=distance(point2, point3);
    s=0.5*(a+b+c);
    return sqrt(s*(s-a)*(s-b)*(s-c));
}

double Tetrakaidecahedra::quadArea(int point1,
int point2, int point3, int point4)
{
    double area = trangleArea(point1, point2,
point3);
    area += trangleArea(point1, point3, point4);
    return area;
}

double Tetrakaidecahedra::hexagonArea(int
point1, int point2, int point3, int point4, int point5,
int point6)
{
    double area = trangleArea(point1, point2,
point3);
    area += trangleArea(point1, point3, point4);
    area += trangleArea(point1, point4, point5);
    area += trangleArea(point1, point5, point6);
    return area;
}

double
Tetrakaidecahedra::getEdgeOfOneUniformTetra()
{
    // For one uniform tetrakaidecahedra only
    return 36.0*distance(0, 1);
}

double
Tetrakaidecahedra::getSurfaceAreaOfOneUniform
Tetra()
{
    // For one uniform tetrakaidecahedra only
    return
6.0*quadArea(0,1,2,3)+8.0*hexagonArea(0,14,13,
16,19,1);
}

bool Tetrakaidecahedra::atSamePlane(int point1,
int point2, int point3, int point4)
{
    double *pt1=vtx+3*point1;
    double *pt2=vtx+3*point2;
    double *pt3=vtx+3*point3;
    double *pt4=vtx+3*point4;
    double nx, ny, nz, nk, dif;
    nx = *(pt2+1)-*(pt1+1))*(*(pt3+2)-
*(pt1+2))-*(pt2+2)-*(pt1+2))*(*(pt3+1)-
*(pt1+1));
    ny = *(pt2+2)-*(pt1+2))*(*(pt3-*(pt1)-
*(pt1))*(*(pt3+2)-*(pt1+2));
    nz = *(pt2-*(pt1))*(*(pt3+1)-*(pt1+1))-
*(pt2+1)-*(pt1+1))*(*(pt3-*(pt1);
    nk = nx*(*(pt1)+ny*(*(pt1+1))+nz*(*(pt1+2));
    dif = nx*(*(pt4)+ny*(*(pt4+1))+nz*(*(pt4+2))-
nk;
    pt1=NULL;
    pt2=NULL;
    pt3=NULL;
    pt4=NULL;
    if(dif>dbacurate || dif<-dbacurate)
        return false;
    else
        return true;
}

double
Tetrakaidecahedra::getSurfaceAreaOfOneRandom
Tetra()
{
    // For one uniform tetrakaidecahedra only
    double surfarea=0;
    surfarea += quadArea(3,0,1,2);
    surfarea += quadArea(7,4,5,6);
    surfarea += quadArea(8,9,10,11);
    surfarea += quadArea(12,13,14,15);
    surfarea += quadArea(19,16,17,18);
    surfarea += quadArea(23,20,21,22);
    surfarea += hexagonArea(1,0,14,13,16,19);
    surfarea += hexagonArea(7,17,16,13,12,4);
    surfarea += hexagonArea(10,9,18,17,7,6);
    surfarea += hexagonArea(8,2,1,19,18,9);
    surfarea += hexagonArea(21,20,15,14,0,3);
    surfarea += hexagonArea(23,5,4,12,15,20);
    surfarea += hexagonArea(11,10,6,5,23,22);
    surfarea += hexagonArea(11,22,21,3,2,8);
    return surfarea;
}

double
Tetrakaidecahedra::getSurfaceAreaOfAllRandomTe
tra()
{
    double surfarea=0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++){
                if((i%2==0 && j%2==0 &&
k%2==0)||i%2==1 && j%2==1 && k%2==1){
                    getOneTetrakaideVtx(i,j,k);
                    surfarea +=
getSurfaceAreaOfOneRandomTetra();
                }
            }
    return 0.5*surfarea;
}

```

```

double
Tetrakaidecahedra::getPrimaryEdgeOfAllRandomTetra()
{
    double edglength = 0.0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++){
                if((i%2==0 && j%2==0 && k%2==0)||
                    (i%2==1 && j%2==1 && k%2==1)){
                    getOneTetrakaideVtx(i,j,k);
                    edglength +=
getNaturalEdgeLengthOfOneTetra()/3.0;
                }
            }
    return edglength;
}

```

```

double
Tetrakaidecahedra::getSecondaryEdgeOfAllRandomTetra()
{
    double edglength = 0.0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++){
                if((i%2==0 && j%2==0 && k%2==0)||
                    (i%2==1 && j%2==1 && k%2==1)){
                    getOneTetrakaideVtx(i,j,k);
                    edglength +=
getAnomalousEdgeLengthOfOneTetra()/2.0;
                }
            }
    return edglength;
}

```

```

double
Tetrakaidecahedra::getEdgeOfAllRandomTetra()
{
    double edglength = 0.0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++){
                if((i%2==0 && j%2==0 && k%2==0)||
                    (i%2==1 && j%2==1 && k%2==1)){
                    getOneTetrakaideVtx(i,j,k);
                    edglength +=
getNaturalEdgeLengthOfOneTetra()/3.0;
                    edglength +=
getAnomalousEdgeLengthOfOneTetra()/2.0;
                }
            }
    return edglength;
}

```

```

void Tetrakaidecahedra::getGrainCentrePos(int &
xpos, int & ypos, int & zpos)
{
    if(xpos%2==1 && ypos%2==1 &&

```

```

zpos%2==1){
    xgc = (xpos/2+1)*gs;
    ygc = (ypos/2+1)*gs;
    zgc = (zpos/2+1)*gs;
} else if(xpos%2==0 && ypos%2==0 &&
zpos%2==0){
    xgc = (xpos/2+0.5)*gs;
    ygc = (ypos/2+0.5)*gs;
    zgc = (zpos/2+0.5)*gs;
}
}

```

```

void Tetrakaidecahedra::zoomAllRandomTetra(int
caseNumber)
{
    double volume=0;
    double manip;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++){
                if((i%2==0 && j%2==0 && k%2==0)||
                    (i%2==1 && j%2==1 && k%2==1)){
                    getOneTetrakaideVtx(i,j,k);
                    if(caseNumber == 1)
                        manip = rdm.getValue();
                    else if(caseNumber == 2)
                        manip = sqrt(rdm.getValue());
                    else if(caseNumber == 3)
                        manip = sqrt(sqrt(rdm.getValue()));
                    else if(caseNumber == 4){
                        manip = rdm.getValue();
                        manip = 1.0/exp(2.0*(manip-1)*(manip-
1));
                    }
                    else{
                        cout<<"the case has not beed defined,
aborted\n";
                        exit(1);
                    }
                }
            }
}

```

```

zoomOneGrainVtxTowardCenter(i,j,k,manip);
}
}

```

```

void
Tetrakaidecahedra::zoomOneGrainVtxTowardCenter(int & xpos,
int & ypos, int & zpos, double zoom)
{
    double volume=0.0;
    getGrainCentrePos(xpos, ypos, zpos);
    for(int i=0; i<24; i++){
        vtx[i*3]= zoom * vtx[i*3] + (1-zoom)*xgc;
        vtx[i*3+1]= zoom * vtx[i*3+1] + (1-
zoom)*ygc;
        vtx[i*3+2]= zoom * vtx[i*3+2] + (1-zoom)*zgc;
    }
    setOneTetrakaideVtx(xpos, ypos, zpos);
}

```



```

double
Tetrakaidecahedra::getEdgeOfOneRandomTetra()
{
    // For one uniform tetrakaidecahedra only
    double edgelen = 0.0;
    edgelen +=
    getNaturalEdgeLengthOfOneTetra();
    edgelen +=
    getAnomalousEdgeLengthOfOneTetra();
    return edgelen;
}

```

```

double
Tetrakaidecahedra::getPrimaryEdgeOfOneRandom
Tetra()
{
    return getNaturalEdgeLengthOfOneTetra();
}

```

```

double
Tetrakaidecahedra::getSecondaryEdgeOfOneRand
omTetra()
{
    return getAnomalousEdgeLengthOfOneTetra();
}

```

```

void Tetrakaidecahedra::sortVolumeCategory()
{
    double *thisvo = new
double[totalGrainNumber()];
    int kkk = 0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++)
                if((i%2==0 && j%2==0 &&
k%2==0)||((i%2==1 && j%2==1 && k%2==1))){
                    getOneTetrakaideVtx(i,j,k);
                    getGrainCentrePos(i,j,k);
                    thisvo[kkk] =
getVolumeOfOneRandomTetra();
                    kkk++;
                }
    double minv = thisvo[0];
    double maxv = thisvo[0];

    for(int i=1; i<kkk; i++){
        if(thisvo[i]>maxv)
            maxv = thisvo[i];
        if(thisvo[i]<minv)
            minv = thisvo[i];
    }
    minv = 0;
    maxv = 120;
    cout<<totalGrainNumber()<<" "<<kkk<<"
"<<maxv<<" "<<minv<<endl;
    double spanv = (maxv-minv)/20.0;
    int numg[20];
    for(int i=0; i<20; i++)
        numg[i] = 0;

```

```

        for(int i=0; i<kkk; i++){
            for(int j=0; j<20; j++){
                if(thisvo[i]>minv+j*spanv &&
(thisvo[i]<minv+(j+1)*spanv))
                    numg[j]++;
            }
            for(int i=0; i<20; i++)
                volfile<<minv+0.5*(2*i+1)*spanv<<"
"<<numg[i]<<endl;
        }

```

```

double
Tetrakaidecahedra::getVolumeOfAllRandomTetra()
{
    double volume=0;
    for(int i=0; i<ngx; i++)
        for(int j=0; j<ngy; j++)
            for(int k=0; k<ngz; k++)
                if((i%2==0 && j%2==0 &&
k%2==0)||((i%2==1 && j%2==1 && k%2==1))){
                    getOneTetrakaideVtx(i,j,k);
                    getGrainCentrePos(i,j,k);
                    volume +=
getVolumeOfOneRandomTetra();
                }
    return volume;
}

```

```

void Tetrakaidecahedra::deformGrainCenter()
{
    double temx, temy, temz;
    temx = sij[0] * xgc + sij[1] * ygc + sij[2] * zgc;
    temy = sij[3] * xgc + sij[4] * ygc + sij[5] * zgc;
    temz = sij[6] * xgc + sij[7] * ygc + sij[8] * zgc;
    xgc = temx;
    ygc = temy;
    zgc = temz;
}

```

```

double
Tetrakaidecahedra::getVolumeOfOneRandomTetra
()
{
    double volume=0.0;
    deformGrainCenter();

    volume += getVolumeTriangularPyramid(1,0,3);
    volume += getVolumeTriangularPyramid(1,3,2);

    volume += getVolumeTriangularPyramid(5,4,7);
    volume += getVolumeTriangularPyramid(5,7,6);

    volume +=
getVolumeTriangularPyramid(8,10,9);
    volume +=
getVolumeTriangularPyramid(8,11,10);

    volume +=

```

```

getVolumeTriangularPyramid(12,14,13);
    volume +=
getVolumeTriangularPyramid(12,15,14);

    volume +=
getVolumeTriangularPyramid(19,17,16);
    volume +=
getVolumeTriangularPyramid(19,18,17);

    volume +=
getVolumeTriangularPyramid(21,20,23);
    volume +=
getVolumeTriangularPyramid(21,23,22);

    volume +=
getVolumeTriangularPyramid(1,19,16);
    volume +=
getVolumeTriangularPyramid(1,16,13);
    volume +=
getVolumeTriangularPyramid(1,13,14);
    volume +=
getVolumeTriangularPyramid(1,14,0);

    volume +=
getVolumeTriangularPyramid(7,4,12);
    volume +=
getVolumeTriangularPyramid(7,12,13);
    volume +=
getVolumeTriangularPyramid(7,13,16);
    volume +=
getVolumeTriangularPyramid(7,16,17);

    volume +=
getVolumeTriangularPyramid(10,6,7);
    volume +=
getVolumeTriangularPyramid(10,7,17);
    volume +=
getVolumeTriangularPyramid(10,17,18);
    volume +=
getVolumeTriangularPyramid(10,18,9);

    volume +=
getVolumeTriangularPyramid(8,9,18);
    volume +=
getVolumeTriangularPyramid(8,18,19);
    volume +=
getVolumeTriangularPyramid(8,19,1);
    volume += getVolumeTriangularPyramid(8,1,2);

    volume +=
getVolumeTriangularPyramid(21,3,0);
    volume +=
getVolumeTriangularPyramid(21,0,14);
    volume +=
getVolumeTriangularPyramid(21,14,15);
    volume +=
getVolumeTriangularPyramid(21,15,20);

    volume +=

```

```

getVolumeTriangularPyramid(23,20,15);
    volume +=
getVolumeTriangularPyramid(23,15,12);
    volume +=
getVolumeTriangularPyramid(23,12,4);
    volume +=
getVolumeTriangularPyramid(23,4,5);

    volume +=
getVolumeTriangularPyramid(11,22,23);
    volume +=
getVolumeTriangularPyramid(11,23,5);
    volume +=
getVolumeTriangularPyramid(11,5,6);
    volume +=
getVolumeTriangularPyramid(11,6,10);

    volume +=
getVolumeTriangularPyramid(11,8,2);
    volume +=
getVolumeTriangularPyramid(11,2,3);
    volume +=
getVolumeTriangularPyramid(11,3,21);
    volume +=
getVolumeTriangularPyramid(11,21,22);
    return volume;
}

double
Tetrakaidecahedra::getVolumeTriangularPyramid(
int point1, int point2, int point3)
{
    double a,b,c,d,s,height, area, volume;
    double *pta=vtx+3*point1;
    double *ptb=vtx+3*point2;
    double *ptc=vtx+3*point3;
    a = (*(ptb+1)-*(pta+1))*(*(ptc+2)-*(pta+2))
        -(*(ptb+2)-*(pta+2))*(*(ptc+1)-*(pta+1));
    b = (*(ptb+2)-*(pta+2))*(*(ptc)-*(pta))
        -(*(ptb)-*(pta))*(*(ptc+2)-*(pta+2));
    c = (*(ptb)-*(pta))*(*(ptc+1)-*(pta+1))
        -(*(ptb+1)-*(pta+1))*(*(ptc)-*(pta));
    d = - a*( *pta)-b*( *pta+1)-c*( *pta+2);
    s = sqrt(a*a+b*b+c*c);
    pta=NULL;
    ptb=NULL;
    ptc=NULL;
    height = (a*xgc+b*ygc+c*zgc+d)/s;
    if(height<0){
        printf("found negative volume\n"); exit(0);
    }
    area = trangleArea(point1, point2, point3);
    volume = height * area / 3.0;
    return volume;
}

void Tetrakaidecahedra::randomizeVtx(double
randomIntensity)
{

```

```

if(randomIntensity<-1 || randomIntensity>1){
    printf("randomIntensity too large, should be
within -1 and 1, aborted\n");
    exit(1);
}
rdm.setSeed(5000);
for(int i=0; i<nvt; i++){
    vcx[i] += qgs*(rdm.getValue()-
0.5)*randomIntensity;
    vcy[i] += qgs*(rdm.getValue()-
0.5)*randomIntensity;
    vcz[i] += qgs*(rdm.getValue()-
0.5)*randomIntensity;
}
}

```

double

Tetrakaidedehedra::getNaturalEdgeLengthOfOneTetra()

```

{
    double edgelen = 0.0;
    if(!atSamePlane(14,1,3,0)) // 1
        edgelen += distance(0,1);
    if(!atSamePlane(3,1,8,2)) // 2
        edgelen += distance(1,2);
    if(!atSamePlane(3,1,2,11)) // 3
        edgelen += distance(2,3);
    if(!atSamePlane(21,0,1,3)) // 4
        edgelen += distance(3,0);
    if(!atSamePlane(23,5,7,4)) // 5
        edgelen += distance(5,4);
    if(!atSamePlane(7,5,11,6)) // 6
        edgelen += distance(5,6);
    if(!atSamePlane(7,5,6,10)) // 7
        edgelen += distance(6,7);
    if(!atSamePlane(7,12,4,5)) // 8
        edgelen += distance(4,7);
    if(!atSamePlane(8,18,9,10)) // 9
        edgelen += distance(8,9);
    if(!atSamePlane(8,18,9,10)) // 10
        edgelen += distance(9,10);
    if(!atSamePlane(8,10,6,11)) // 11
        edgelen += distance(10,11);
    if(!atSamePlane(8,10,11,2)) // 12
        edgelen += distance(8,11);
    if(!atSamePlane(12,7,13,14)) // 13
        edgelen += distance(12,13);
    if(!atSamePlane(12,13,1,14)) // 14
        edgelen += distance(13,14);
    if(!atSamePlane(12,14,21,15)) // 15
        edgelen += distance(14,15);
    if(!atSamePlane(12,14,15,23)) // 16
        edgelen += distance(12,15);
    if(!atSamePlane(16,7,17,19)) // 17
        edgelen += distance(16,17);
    if(!atSamePlane(19,17,10,18)) // 18
        edgelen += distance(17,18);
    if(!atSamePlane(19,17,18,8)) // 19

```

```

    edgelen += distance(18,19);
    if(!atSamePlane(1,16,17,19)) // 20
        edgelen += distance(16,19);
    if(!atSamePlane(15,21,23,20)) // 21
        edgelen += distance(20,21);
    if(!atSamePlane(23,21,11,22)) // 22
        edgelen += distance(21,22);
    if(!atSamePlane(23,21,22,11)) // 23
        edgelen += distance(22,23);
    if(!atSamePlane(15,20,21,23)) // 24
        edgelen += distance(20,23);
    if(!atSamePlane(21,14,1,0)) // 25
        edgelen += distance(0,14);
    if(!atSamePlane(13,7,16,1)) // 26
        edgelen += distance(13,16);
    if(!atSamePlane(1,16,19,8)) // 27
        edgelen += distance(1,19);
    if(!atSamePlane(12,23,4,7)) // 28
        edgelen += distance(4,12);
    if(!atSamePlane(7,10,17,16)) // 29
        edgelen += distance(7,17);
    if(!atSamePlane(6,11,10,7)) // 30
        edgelen += distance(6,10);
    if(!atSamePlane(9,8,18,10)) // 31
        edgelen += distance(9,18);
    if(!atSamePlane(8,11,2,1)) // 32
        edgelen += distance(8,2);
    if(!atSamePlane(21,20,23,15)) // 33
        edgelen += distance(20,15);
    if(!atSamePlane(23,11,5,4)) // 34
        edgelen += distance(23,5);
    if(!atSamePlane(21,11,23,22)) // 35
        edgelen += distance(11,22);
    if(!atSamePlane(21,0,3,11)) // 36
        edgelen += distance(3,21);
    return edgelen;
}

```

double

Tetrakaidedehedra::getAnomalousEdgeLengthOfOneTetra()

```

{
    double edgelen = 0.0;
    if(!atSamePlane(0,1,2,3)) // 37
        edgelen += distance(1,3);
    if(!atSamePlane(16,17,18,19)) // 37 -- from
wrong notice
        edgelen += distance(17,19);
    if(!atSamePlane(4,5,6,7)) // 38
        edgelen += distance(5,7);
    if(!atSamePlane(20,21,22,23)) // 39
        edgelen += distance(21,23);
    if(!atSamePlane(8,9,10,11)) // 40
        edgelen += distance(8,10);
    if(!atSamePlane(12,13,14,15)) // 41
        edgelen += distance(12,14);
    if(!atSamePlane(1,0,14,13)) // 42
        edgelen += distance(1,14);
    if(!atSamePlane(1,14,13,16)) // 43

```

```

    edgelenlength += distance(1,13);
    if(!atSamePlane(1,13,16,19)) // 44
    edgelenlength += distance(1,16);
    if(!atSamePlane(23,22,11,5)) // 45
    edgelenlength += distance(23,11);
    if(!atSamePlane(23,11,6,5)) // 46
    edgelenlength += distance(11,5);
    if(!atSamePlane(5,11,10,6)) // 47
    edgelenlength += distance(11,6);
    if(!atSamePlane(8,2,1,19)) // 48
    edgelenlength += distance(8,1);
    if(!atSamePlane(8,1,19,18)) // 49
    edgelenlength += distance(8,19);
    if(!atSamePlane(8,19,18,9)) // 50
    edgelenlength += distance(8,18);
    if(!atSamePlane(23,5,4,12)) // 51
    edgelenlength += distance(23,4);
    if(!atSamePlane(23,4,12,15)) // 52
    edgelenlength += distance(23,12);
    if(!atSamePlane(23,12,15,20)) // 53
    edgelenlength += distance(23,15);
    if(!atSamePlane(21,20,15,14)) // 54
    edgelenlength += distance(21,15);
    if(!atSamePlane(21,15,14,0)) // 55
    edgelenlength += distance(21,14);
    if(!atSamePlane(21,14,0,3)) // 56
    edgelenlength += distance(21,0);
    if(!atSamePlane(10,6,7,17)) // 57
    edgelenlength += distance(19,7);
    if(!atSamePlane(10,7,17,18)) // 58
    edgelenlength += distance(10,17);
    if(!atSamePlane(10,17,18,9)) // 59
    edgelenlength += distance(10,18);
    if(!atSamePlane(7,17,16,13)) // 60
    edgelenlength += distance(7,16);
    if(!atSamePlane(7,16,13,12)) // 61
    edgelenlength += distance(7,13);
    if(!atSamePlane(7,13,12,4)) // 62
    edgelenlength += distance(7,12);
    if(!atSamePlane(11,3,2,8)) // 63
    edgelenlength += distance(11,2);
    if(!atSamePlane(11,21,3,2)) // 64
    edgelenlength += distance(11,3);
    if(!atSamePlane(11,22,21,3)) // 65
    edgelenlength += distance(11,21);
    return edgelenlength;
}

#endif // TETRAKAIDECAHEDRA_CPP

```

ACKNOWLEDGEMENTS

I would like to express my deep thanks to my supervisor, Professor Bhadeshia, H. K. D. H. for his help and encouragement. I would also like to thank to Professor Rongshan Qin for the great support. And thanks to Professor In Gee Kim and Professor Bruno C. de Cooman for their advise, support and friendship.

I am grateful to all members in Computational Metallurgy Laboratory, for all their help and for all the memories we have. And also to all the people including office staff in the Graduate Institute of Ferrous Technology in POSTECH.

I am also grateful to Professor Hae Geon Lee for introducing and giving me a chance to study GIFT.

Finally, I am extremely grateful to all my family, especially my parents, for their devotional support, big encouragement, confidence and everlasting love.

사랑하는 우리 가족들에게 감사의 마음을 전합니다. 특히 항상 저를 바른길로 이끌어주시고 아껴주신 부모님께 보잘것없지만 이 논문을 바칩니다.

CURRICULUM VITAE

Name: Chae, Jae yong (蔡在鎔)

Date of birth: 12th July, 1981

Place of birth: Pohang, Kyungbuk, Republic of Korea

Address: Graduate Institute of Ferrous Technology, Pohang University of Science and Technology San 31, Hyoja-Dong, Nam-gu, Pohang, Kyungbuk, 790-784, Republic of Korea

Education:

M. S. 2008, POSTECH (Pohang, Korea), Graduate Institute of Ferrous Technology, Computational Metallurgy.

B. S. 2006, POSTECH (Pohang, Korea), Computer Science and Engineering.

Parts of this work have been submitted to appear in the following publication.

Jae-Yong, C., Rongshan, Q. and Bhadeshia, H. K. D. H. : Topology of the deformation of a non-uniform grain structure, submitted to *Material Science and Technology*.