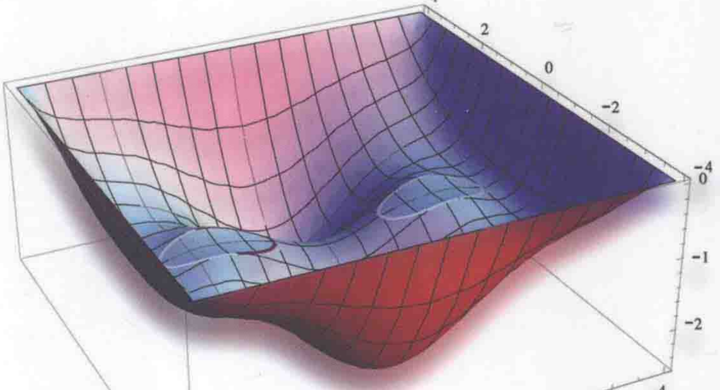
SEIICHI NOMURA

# Micromechanics with *Mathematica*



with website

WILEY

# MICROMECHANICS WITH *MATHEMATICA*

**Seiichi Nomura**

*Department of Mechanical and Aerospace Engineering*
*The University of Texas at Arlington*
*Arlington, TX*
*USA*

WILEY

*Library of Congress Cataloging-in-Publication Data applied for*

A catalogue record for this book is available from the British Library.

ISBN: 9781119945031

Set in 10/12pt, TimesLTStd by SPi Global, Chennai, India.
Printed and bound in Singapore by Markono Print Media Pte Ltd

1 2016

# Preface

Micromechanics is a branch of applied mechanics that began with the celebrated paper of Eshelby published in 1957. It refers to analytical methods for solid mechanics that can describe deformations as functions of such microstructures as voids, cracks, inclusions, and dislocations. Micromechanics is an essential tool for obtaining mechanical fields analytically in modern materials including composite and nanomaterials that did not exist 50 years ago.

There exist a number of well-written books with a similar subject title to this book (micromechanics, continuum mechanics with computer algebra, etc.). However, many of them are written by mathematicians or theoretical physicists that follow the strict style of rigorous formality (theorem, corollary, etc.), which may easily discourage aspiring students without formal background in mathematics and physics yet who want to learn what micromechanics has to offer.

The threshold of micromechanics seems high because many formulas and derivations are based on tensor algebra and analysis that calls for a substantial amount of algebra. Although it is a routine type of work, evaluation of tensorial equations requires tedious manual calculations. This scheme all changed in the 1980s with the emergence of computer algebra systems that made it possible to crunch symbols instead of numbers. It is no longer necessary to spend endless time on algebra manually as symbolically capable software such as Maple and *Mathematica* can handle complex tensor equations

The aim of this book is to introduce the concept of micromechanics in plain terms without rigorousness yet still maintaining consistency with a target audience of those who want to actually use the result of micromechanics for multiphase/heterogeneous materials, taking advantage of a computer algebra system, *Mathematica*, rather than those who need formal and rigorous derivations of the equations in micromechanics. The author has been a fan of *Mathematica* since the 1990s and believes that it is the best tool for handling subjects in micromechanics that require both analytical and numerical computations. Unlike numerically oriented computer languages such as C and Fortran, *Mathematica* can process both symbols and numerics seamlessly, thus being capable of handling lengthy tensorial manipulations that can release mundane and tedious jobs by human beings. There have been intense debates in user communities about the difference and preference among *Mathematica* and other numerical software such as MATLAB, all of which are widely used in engineering and scientific communities. The major difference is that software such as MATLAB offers only a limited support for symbolic variables through licensing Maple and is not integrated in the system seamlessly, whereas in *Mathematica*, there is no distinction between symbolic and numerical variables; more importantly, it is not possible to derive and manipulate formulas employed in this book with MATLAB alone.

One of the unique features in this book is to introduce many examples in micromechanics that can be solved only through computer algebra systems. This includes stress analysis for multiinclusions and the use of the Airy stress function for inclusion problems.

Many of the subjects presented in this book may be classical that may have existed for the past 200 years. Nevertheless, those problems presented in this book would not have been possibly solved analytically had it not been for *Mathematica* or, for that matter, any computer algebra system, which, the author believes, is the raison d'être of this book.

This book consists of four chapters that cover a variety of topics in micromechanics. Each example problem is accompanied with corresponding *Mathematica* code. Chapter 1 introduces the basic concept of the coordinate transformations and the properties of Cartesian tensors that are needed to derive equations in continuum mechanics. In Chapter 2, based on the concepts introduced in Chapter 1, the field equations in continuum mechanics are derived. Coordinate transformations in general curvilinear coordinate systems are discussed. Chapter 3 presents a new paradigm for inclusion problems embedded in an infinite matrix. After a brief introduction of the Eshelby method, new analytical approaches to derive the stress fields for an inclusion and concentrically placed inclusions in an infinite matrix are discussed along with their implementations in *Mathematica*. Chapter 4 is devoted to the inclusion problems where the matrix is finite-sized. The classical Galerkin method is combined with *Mathematica* to derive the physical and mechanical fields semi-analytically. The Appendix is an introduction to *Mathematica* that provides sufficient background information in order to understand the *Mathematica* code presented in this book.

Seiichi Nomura
Arlington, Texas

# About the Companion Website

This book is accompanied by a companion website:

www.wiley.com/go/nomura0615

This website includes:

- Listing of Selected Programs
- A Solutions Manual
- An Exercise Section

# Contents

# 1

# Coordinate Transformation and Tensors

To describe the state of the deformation for a deformable body, the coordinate transformation plays an important rule, and the most appropriate way to represent the coordinate transformation is to use tensors. In this chapter, the concept of coordinate transformations and the introduction to tensor algebra in the Cartesian coordinate system are presented along with their implementation in *Mathematica*. As this book is not meant to be a textbook on continuum mechanics, the readers are referred to some good reference books including Romano et al. (2006) and Fung (1965), among others. Manipulation involving indices requires a considerable amount of algebra work when the expressions become lengthy and complicated. It is not practical to properly handle and evaluate quantities that involve tensor manipulations by conventional scientific/engineering software such as FORTRAN, C, and MATLAB. Software packages capable of handling symbolic manipulations include *Mathematica* (Wolfram 1999), Maple (Garvan 2001), and others. In this book, *Mathematica* is exclusively used for implementation and evaluation of derived formulas. A brief introduction to the basic commands in *Mathematica* is found in the appendix, which should be appropriate to understand and execute the *Mathematica* code used in this book.

## 1.1 Index Notation

If one wants to properly express the deformation state of deformable bodies regardless of whether they are solids or fluids, the use of tensor equations is essential. There are several different ways to denote notations of tensors, one of which uses indices and others without using indices at all. In this book, the index notation is exclusively used throughout to avert unnecessary abstraction at the expense of mathematical sophistication.

The following are the main compelling reasons to mandate the use of tensor notations in order to describe the deformation state of bodies correctly.

1. The principle of physics stipulates that a physically meaningful object must be described independent of the frame of references.[1] If the equation for a physically meaningful object changes depending on the coordinate system used, that equation is no longer a correct equation.
2. Tensor equations can be shown to be invariant under the coordinate transformation. Tensor equations are thus defined as those equations that are unchanged from one coordinate system to another.

Hence, by combining the two aforementioned statements, it can be concluded that only tensor equations can describe the physical objects properly. In other words, if an equation is not in tensorial format, the equation does not represent the object physically.

The index notation, also known as the Einstein notation (Einstein et al. 1916)[2] or the summation convention, is the most widely used notation to represent tensor quantities, which will be used in this book. The index notation in the Cartesian coordinate system is summarized as follows:

1. For mathematical symbols that are referred to quantities in the $x$, $y$, and $z$ directions, use subscripts, 1, 2, 3, as in $x_1, x_2, x_3$ or $a_1, a_2, a_3$, instead of $x, y, z$ or $a, b, c$. The subscripted numbers 1, 2, and 3, refer to the $x$, $y$, and $z$ directions, respectively. Obviously, the upper limit of the number is 2 for 2-D and 3 for 3-D.
2. If there are twice repeated indices in a term of products such as $a_ib_i$, the summation with respect to that index ($i$) is always assumed. For example,

$$a_ib_i \equiv \sum_{i=1}^{3} a_ib_i = a_1b_1 + a_2b_2 + a_3b_3 \quad \text{(3-D)}.$$

   There is no exception to this rule. An expression such as $a_ib_ic_i$ is not allowed as the number of repetitions is 3 instead of 2.

A repeated index is called the *dummy index* as it does not matter what letter is used, and an unrepeated index is called the *free index*.[3] For example,

$$x_ix_i = x_jx_j = x_\alpha x_\alpha,$$

all of which represent a summation ($= \sum_i x_ix_i$). An unrepeated index such as $x_i$ (or $x_j$ or $x_\alpha$) stands for one of $x_1$, $x_2$, or $x_3$.

It should be noted that the notations and conventions introduced are valid for the Cartesian coordinate system only. In a curvilinear coordinate system such as the spherical coordinate system, the length of base vectors is not necessarily unity, and this mandates the aforementioned index notation to be modified to reflect the difference between the contravariant components and the covariant components, which will be discussed in Chapter 2.

[1] This "frame of references" refers to the Galilean transformation in classical mechanics, the Lorentz transformation in special relativity, and the general curvilinear transformation in general relativity.
[2] Albert Einstein introduced this notation in 1916.
[3] This is similar to definite integrals. The variable used in a definite integral does not matter as

$$\int_a^b f(x)dx = \int_a^b f(y)dy = \int_a^b f(z)dz.$$

The variables $x$, $y$, and $z$ are called *dummy variables*.

### *1.1.1 Some Examples of Index Notation in 3-D*

1. $x_i x_i$
   As the index $i$ is repeated, the summation symbol, $\sum$, must be added in front, i.e.,

$$\begin{aligned} x_i x_i &= \sum_{i=1}^{3} x_i x_i \\ &= x_1 x_1 + x_2 x_2 + x_3 x_3 \\ &= x^2 + y^2 + z^2. \end{aligned}$$

   Note that $x_i x_i$ is different from $(x_i)^2$. While $x_i x_i$ represents a single expression with three terms, $(x_i)^2$ represents one of the three expressions $((x_1)^2, (x_2)^2$, or $(x_3)^2)$.
2. $x_i x_i x_j$
   Note that the index $i$ is a dummy index (repeated twice) while the index $j$ is a free index (no repeat). Therefore,

$$\begin{aligned} x_i x_i x_j &= x_j \sum_{i=1}^{3} x_i x_i \\ &= x_j((x_1)^2 + (x_2)^2 + (x_3)^2) \\ &= \begin{cases} x(x^2 + y^2 + z^2) \\ \quad\text{or} \\ y(x^2 + y^2 + z^2) \\ \quad\text{or} \\ z(x^2 + y^2 + z^2). \end{cases} \end{aligned}$$

3. $x_i x_i x_i$
   This is not a valid tensor expression as the number of repeated indices must be 2.

### *1.1.2 Mathematica Implementation*

As *Mathematica* itself does not support tensor manipulation natively, it is necessary to devise a way to handle index notation and tensor manipulation. In this book, a list or a list of lists (a nested list) is used to represent tensor quantities. Using a nested list to define a tensor of any rank is straightforward but at the same time limited to the Cartesian tensors. For tensors defined in a curvilinear coordinate system, a slightly different approach is needed.

When running *Mathematica* first time, a default directory should be selected so that all the notebook files can be saved and accessed in this directory. By default, *Mathematica* looks for all the files stored in `c:\users\<user>\` where `<user>` is the user's home directory.[4] The `SetDirectory` command can change this location. For example, if you want to change the default directory to `c:\tmp`, the `SetDirectory` command can specify the default directory as

[4] The Windows operating system uses "\" (backslash) as the directory delimiter while the Unix system uses "/" (forward slash) as the directory delimiter. However, the "/" in the SetDirectory command in *Mathematica* works for both.

```
In[1]:= SetDirectory["c:/tmp"]

Out[1]= c:\tmp
```

It is noted that the directory delimiter needs to be entered as "/" (forward slash) even though the Windows delimiter character is "\" (backslash).

To enter a three-dimensional vector, $\mathbf{v} = (x^2, y^2, z^2)$, the following *Mathematica* command can be entered to create a list with braces (curly brackets) as

```
In[2]:= v = {x^2, y^2, z^2}

Out[2]= {x², y², z²}
```

An individual component of $\mathbf{v}$ can be referenced using double square brackets ( `[[...]]` ) as

```
In[3]:= v[[1]]

Out[3]= x²
```

The partial derivative of $\mathbf{v}$ with respect to $x$ can be entered as

```
In[4]:= D[v, x]

Out[4]= {2 x, 0, 0}
```

You can also differentiate an individual component as

```
In[5]:= D[v[[2]], y]

Out[5]= 2 y
```

Implementation of the coordinate component, $x_i$, into *Mathematica* can be done by using the `Table` function. To define a position vector, $\mathbf{r}$, whose components are $(x_1, x_2, x_3)$, enter

```
In[1]:= r = Table[x[i], {i, 1, 3}]

Out[1]= {x[1], x[2], x[3]}
```

The given `Table` function generates a list of elements. For example, the following command generates a sequence of $i^2$ for $i = 1, \dots, 10$.

```
In[2]:= Table[ i^2, {i, 1, 10}]

Out[2]= {1, 4, 9, 16, 25, 36, 49, 64, 81, 100}
```

In the definition of the position vector, $\mathbf{r}$, it is noted that the coordinate components, $(x_1, x_2, x_3)$, are entered as `x[1], x[2], x[3]` instead of `x[[1]], x[[2]], x[[3]]`. It is important to distinguish a single square bracket ( `[...]` ) and a double square bracket ( `[[...]]` ). The single square bracket ( `[...]` ) is for a parameter used in a function. The quantity, `x[1]`, stands for a function, `x`, with the argument of `1`. By using a single square bracket, the quantities such as `x[1]`, `x[2]` can stand for themselves, meaning that initial values do not have to be preassigned. On the other hand, if `x[[1]]` were used instead of `x[1]`, `0` would be returned unless a list `x` is previously defined.

To define a function in *Mathematica*, use the following syntax:

```
In[1]:=  f[x_] := x^2 + 1

In[2]:=  f[6]

Out[2]=  37

In[3]:=  f[a^2]

Out[3]=  1 + a^4
```

In the aforementioned example code, a user-defined function, `f[x]`, that returns $x^2 + 1$ is defined. The syntax is such that variables of the function must be presented to the left of the equal sign with the underscore and the definition of the function is given to the right of a colon and an equal sign (:=). It is important to note that a function in *Mathematica* returns itself if no prior definition is given.

```
In[1]:=  f[x_, y_] := x^2 + y^2

In[2]:=  f[a^3, b^2]

Out[2]=  a^6 + b^4

In[3]:=  f[1]

Out[3]=  f[1]

In[4]:=  g[a]

Out[4]=  g[a]
```

In the aforementioned example, `f[x, y]` is defined as a function that takes two variables returning $x^2 + y^2$. However, when `f` is called with only one variable, `1`, it returns itself, i.e., `f[1]`, as `f` with only one variable has not been defined. When `g[a]` is entered, it returns itself without evaluation as there is no prior definition of `g[x]` given. It is this property of a function in *Mathematica* that enables manipulating index notations.

As an example of using `x[i]` as the coordinate components, here is how to implement the summation convention. As *Mathematica* does not have support for the summation convention built in, if `x[i]x[i]` meant as $x_i x_i = x_1^2 + x_2^2 + x_3^2$ is entered as

```
In[1]:=  x[i] x[i]

Out[1]=  x[i]^2
```

*Mathematica* does not automatically expand $x_i x_i$ and reduce the result to $r^2$. Hence, it is necessary to explicitly use the `Sum[]` command as

```
In[6]:=  Sum[x[i] x[i], {i, 1, 3}]

Out[6]=  x[1]^2 + x[2]^2 + x[3]^2
```

for $x_i x_i$.

However, it is possible to implement the summation convention in *Mathematica* with the following procedure.

```
In[1]:= Unprotect[Times];
        x[i_Symbol] x[i_Symbol] := r^2;
        Protect[Times];
```

The aforementioned three-line code adds a new rule to automatically replace `x[i]x[i]` by $r^2$ through pattern matching. When *Mathematica* evaluates the product of two quantities, it calls its internal function, `Times`, which tries to simplify the result with various pattern-matching algorithms. It is not possible to modify the pattern-matching algorithms as they are part of the definition of the `Times` function and they are protected by default. Therefore, it is necessary to unprotect the multiplication operator, `Times`, in *Mathematica* with the `Unprotect` command so that a new rule for the summation convention can be added. The second line is to tell *Mathematica* a new rule that whenever a pattern of `x[i]x[i]` where `i` is any symbol but not a number is entered, it is automatically replaced by $r^2$. Finally, in the third line, the `Times` function is given back the `Protect` attribute so that any further modification is prevented. After these three lines are entered, the summation convention for `x[i] x[i]` is automatic as

```
In[4]:= x[j] x[j]

Out[4]= r^2

In[5]:= x[3] x[3]

Out[5]= x[3]^2

In[6]:= x[j] x[j] x[i]

Out[6]= r^2 x[i]

In[7]:= x[i] x[i] x[j] x[j]

Out[7]= r^4
```

In the aforementioned examples, $x_jx_j$ is reduced to $r^2$ but $x_3x_3$ remains unchanged. The expression, $x_jx_jx_i$ is simplified to $r^2x_i$ and $x_ix_ix_jx_j$ is reduced to $r^4$. With this pattern-matching capability of *Mathematica*, manipulation of tensor quantities can be greatly simplified. More detailed examples will be shown later.

### *1.1.3 Kronecker Delta*

One of the most important symbols in tensor algebra is the Kronecker delta, which is defined as

$$\delta_{ij} = \begin{cases} 1, & \text{if } (i,j) = (1,1), (2,2), \text{or } (3,3) \\ 0, & \text{otherwise.} \end{cases}$$

The Kronecker delta is what is equivalent to 1 in numbers or the identity matrix in linear algebra. It is also used to define the inverse of a tensor.

**Examples**

1. $\delta_{ii}$ (3-D)

$$\begin{aligned}\delta_{ii} &= \sum_{i=1}^{3} \delta_{ii} \\ &= \delta_{11} + \delta_{22} + \delta_{33} \\ &= 3.\end{aligned}$$

2. $\delta_{ij}\delta_{ij}$ (3-D)

$$\begin{aligned}\delta_{ij}\delta_{ij} &= \sum_{i=1}^{3}\sum_{j=1}^{3} \delta_{ij}\delta_{ij} \\ &= \delta_{11}\delta_{11} + \delta_{12}\delta_{12} + \delta_{13}\delta_{13} \\ &+ \delta_{21}\delta_{21} + \delta_{22}\delta_{22} + \delta_{23}\delta_{23} \\ &+ \delta_{31}\delta_{31} + \delta_{32}\delta_{32} + \delta_{33}\delta_{33} \\ &= 3.\end{aligned}$$

3. $\delta_{ii}\delta_{jj}$ (3-D)

$$\begin{aligned}\delta_{ii}\delta_{jj} &= \sum_{i=1}^{3} \delta_{ii} \sum_{j=1}^{3} \delta_{jj} \\ &= (\delta_{11} + \delta_{22} + \delta_{33})(\delta_{11} + \delta_{22} + \delta_{33}) \\ &= 9.\end{aligned}$$

4. $\delta_{ij}x_i x_j$ (3-D)

$$\begin{aligned}\delta_{ij}x_i x_j &= \sum_{i=1}^{3}\sum_{j=1}^{3} \delta_{ij}x_i x_j \\ &= \delta_{11}x_1x_1 + \delta_{12}x_1x_2 + \delta_{13}x_1x_3 \\ &+ \delta_{21}x_2x_1 + \delta_{22}x_2x_2 + \delta_{23}x_2x_3 \\ &+ \delta_{31}x_3x_1 + \delta_{32}x_3x_2 + \delta_{33}x_3x_3 \\ &= (x_1)^2 + (x_2)^2 + (x_3)^2.\end{aligned}$$

The Kronecker delta can be implemented in *Mathematica* by several different ways. The following implementation is to use a function to define the Kronecker delta, which is valid for any number of dimensions.

```
In[1]:= delta[i_Integer, j_Integer] := If[i == j, 1, 0]

In[2]:= delta[1, 2]

Out[2]= 0

In[3]:= delta[i, j]

Out[3]= delta[i, j]

In[4]:= Sum[delta[i, i], {i, 3}]

Out[4]= 3
```

The underline "_" after the variable name is used by *Mathematica* to restrict the type of variable to be the type that follows _. In this case, `delta[i, j]` is evaluated only when both `i` and `j` are integers. The function `If[condition, t, f]` gives `t` if `condition` is true and `f` otherwise. Note that the two equal signs (==) in "`i==j`" mean equality, not an assignment as in most of the programming languages.

#### 1.1.3.1 Summation Convention for $\delta_{ij}$

Some of the properties of the Kronecker delta that involve the summation convention include:

$$\delta_{ii} = 3,\ \ a_j\delta_{ij} = a_i,\ \ a_{ij}\delta_{ik} = a_{kj},\ \ \delta_{ij}\delta_{ik} = \delta_{jk}.$$

The following *Mathematica* code implements the aforementioned rules so that summation convention that involves the Kronecker delta is always automatically executed.

```
In[10]:= SetAttribute[delta, Orderless];
         delta[i_Integer, j_Integer] := If[i == j, 1, 0];
         delta[i_Symbol, i_Symbol] := 3;

In[14]:= Unprotect[Times];
         Times[a_Symbol[j_Symbol], delta[i_, j_Symbol]] := a[i];
         Times[a_Symbol[i_Symbol, j_], delta[i_Symbol, k_]] := a[k, j];
         Times[delta[i_Symbol, j_], delta[i_Symbol, k_]] := delta[j, k];
         Protect[Times];
```

The `SetAttribute[delta, Orderless]` command sorts the variables in `delta` into standard order so that `delta[j,i]` is automatically changed to `delta[i,j]`. The part `i_Integer` specifies that the variable `i` must be an integer value, and the part `i_Symbol` specifies that the variable `i` must be a symbol, not a number. The subsequent code instructs *Mathematica* to add new rules that use the summation convention with `delta[i,j]`. After this *Mathematica* code is entered, the summation convention involving the Kronecker delta is automatically enforced.