

SAND91-1752  
Unlimited Release  
Printed 2/15/99

Distribution  
UC-705

**PHYSLIB:  
A C++ Tensor Class Library  
(Version 2)**

Kent G. Budge  
1431  
Sandia National Laboratories  
Albuquerque, NM 87185

**Abstract**

PHYSLIB is a C++ class library for general use in computational physics applications. It defines vector and tensor classes and the corresponding operations. A simple change in the header file allows the user to compile either 2-D or 3-D versions of the library.

## **Acknowledgment**

The author acknowledges the assistance of J.S. Peery for reviewing this library and for much discussion of general C++ programming issues.

# Contents

Acknowledgment .....	4
Contents .....	5
Preface.....	7
Summary.....	9
1. Introduction.....	11
1.1 Vector and Tensor Operations and Notation.....	11
1.1.1 Vectors.....	11
1.1.2 Tensors.....	12
1.1.3 Symmetric and Antisymmetric Tensors.....	14
1.1.4 Vector and Tensor Components; Indicial Notation.....	14
1.1.5 Einstein Summation Convention .....	15
1.1.6 Dimensionality.....	16
1.2 The C++ Programming Language .....	17
1.2.1 Data Abstraction .....	17
1.2.2 Special Member Functions and Dynamic Memory Management .....	17
1.2.3 Function and Operator Overloading .....	18
2. The PHYSLIB Library.....	21
2.1 class Vector.....	21
2.1.1 Private Data Members.....	21
2.1.2 Special Member Functions .....	21
2.1.3 Utility Functions .....	24
2.2 class Tensor.....	25
2.2.1 Private Data Members.....	25
2.2.2 Special Member Functions .....	25
2.2.3 Utility Functions .....	31
2.3 class SymTensor .....	32
2.3.1 Private Data Members.....	32
2.3.2 Special Member Functions .....	32
2.3.3 Utility Functions .....	36
2.4 class AntiTensor.....	37

2.4.1 Private Data Members.....	37
2.4.2 Special Member Functions .....	37
2.4.3 Utility Functions .....	40
2.5 Operator Overload Functions .....	41
2.6 Methods .....	52
2.7 Predefined Constants .....	58
3. Using the PHYSLIB classes .....	61
3.1 Useless Operations.....	62
References.....	63
Index of Operators and Functions.....	65

## Preface

C++ is the first object-oriented programming language which produces sufficiently efficient code for consideration in computation-intensive physics and engineering applications. In addition, the increasing availability of massively parallel architectures requires novel programming techniques which have proven to be relatively easy to implement in C++. For these reasons, Division 9231 at Sandia National Laboratories is devoting considerable resources to the development of C++ libraries.

This document describes the first of these libraries to be released, PHYSLIB, which defines classes representing Cartesian vectors and (second-order) tensors. This library consists of the header file `physlib.h` and the source file `physlib.C`. The library is applicable to one-dimensional, two-dimensional, and three-dimensional problems; the user selects the dimensionality of the library by defining the appropriate preprocessor symbol (`ONE_D`, `TWO_D`, or `THREE_D`) when compiling `physlib.C` and his own code.

This code was produced under the auspices of Sandia National Laboratories, a federally-funded research center. This code is available to U.S. citizens and institutions under research, government use and/or commercial license agreements.

The PHYSLIB library is © 1991, 1994 Sandia Corporation.

(Intentionally Left Blank)

## Summary

PHYSLIB defines the following classes:

class Vector	Cartesian vectors
class Tensor	Cartesian 2nd-order tensors
class SymTensor	Cartesian 2nd-order symmetric tensors
class AntiTensor	Cartesian 2nd-order antisymmetric tensors

Methods that are defined for these classes include the following:

- Dot and outer products
- Cross products for vectors
- Other arithmetic operations
- Duals (dot or double dot product with the permutation symbol)
- Trace of tensors
- Transpose of tensors
- Determinants and inverses of tensors
- Symmetric and antisymmetric part of tensors
- Scalar invariants of tensors
- Norms
- Colon operator (scalar product of tensors)
- Deviatoric part of tensors
- Equality comparisons of vectors and tensors
- Standardized stream I/O for vectors

I

(Intentionally Left Blank)

## 1. Introduction

Almost every branch of theoretical physics makes use of the concepts of *vectors* and *tensors*. Vectors are conceptually simple; they are quantities having both magnitude and direction, such as the velocity of a particle. Tensors are conceptually more difficult. They represent rules that relate one set of vectors to another, and they appear in many physical formulae.

This document briefly reviews the mathematics of vectors and tensors; discusses the basic difficulties in translating vector and tensor equations into computer code; and describes how the C++ programming language has been used to alleviate these difficulties, thereby producing reliable, reusable, and transparent computer code at a much reduced cost in programmer effort.

### 1.1 Vector and Tensor Operations and Notation

We briefly review the basic concepts and language of vectors and tensors. A more complete discussion can be found in [2].

#### 1.1.1 Vectors

A *vector* is a physical quantity such as velocity that has both a magnitude (“five hundred km/sec”) and a direction (“towards the northeast”). It may be written as a lowercase symbol with an arrow over it, such as  $\vec{v}$ . Quantities such as temperature or mass that have magnitude but no direction are called *scalars* and are represented by lowercase symbols without an arrow, such as  $a$ .

The magnitude or *norm* of a vector  $\vec{a}$  is written as  $|\vec{a}|$  and is a scalar, while its direction may be written as  $\hat{a}$ . The direction of a vector is itself a vector with magnitude 1 (called a *unit vector*).

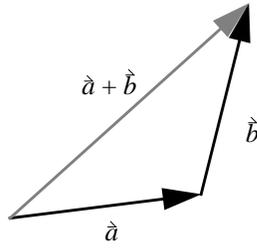
A vector may be multiplied by a scalar. The result is a vector with the same direction as the original vector and with a magnitude equal to the product of the scalar and the magnitude of the original vector. That is,

$$\text{if } \vec{b} = c\vec{a} \text{ then } |\vec{b}| = |c||\vec{a}| \text{ and } \hat{b} = \pm\hat{a} \quad (1)$$

If  $c < 0$ , the resulting vector has the opposite direction from the original vector.

Vectors may be added to or subtracted from each other; they obey the same algebraic rules as real numbers under addition and subtraction. Vector addition may be visualized by picturing each vector as an arrow with a length equal to its magnitude, as illustrated below:

Figure 1. Addition of Vectors



The opposite of a vector is a vector with the same length but in the opposite direction.

Vectors may not be multiplied in the same sense as real numbers. However, several operations exist which are distributive and which are therefore spoken of as “products”. The *inner product* (or dot product) of two vectors is a scalar and is written

$$\vec{a} \cdot \vec{b} \tag{2}$$

It is defined as the product of the magnitudes of the two vectors and the cosine of the angle between them, that is,

$$\vec{a} \cdot \vec{b} = |\vec{a}| |\vec{b}| \cos \theta_{ab} . \tag{3}$$

The diagram shows two vectors,  $\vec{a}$  and  $\vec{b}$ , originating from the same point. Vector  $\vec{a}$  points to the right, and vector  $\vec{b}$  points upwards and to the left. An arc between the two vectors indicates the angle  $\theta_{ab}$ .

Thus, the dot product is zero if the vectors are perpendicular. The dot product is *distributive* and *commutative*, that is,

$$\vec{a} \cdot (\vec{b} + \vec{c}) = \vec{a} \cdot \vec{b} + \vec{a} \cdot \vec{c} \quad \text{(Distributive law)} \tag{4}$$

$$\vec{a} \cdot \vec{b} = \vec{b} \cdot \vec{a} \quad \text{(Commutative law)} \tag{5}$$

The *outer product* of two vectors is a tensor; it is discussed below.

### 1.1.2 Tensors

A tensor is a rule that turns a vector into another vector, and it is represented symbolically by a boldface capital letter, such as  $\mathbf{A}$ . We write

$$\vec{a} = \mathbf{A} \vec{b} \tag{6}$$

to indicate that when the tensor  $\mathbf{A}$  is applied to the vector  $\vec{b}$ , it returns the vector  $\vec{a}$ . Not all rules that turn vectors into other vectors are tensors; a tensor must be linear, that is, it must be true for all  $\vec{a}$ ,  $\vec{b}$ , and  $c$  that

$$\mathbf{A}(\vec{a} + \vec{b}) = \mathbf{A} \vec{a} + \mathbf{A} \vec{b} \tag{7}$$

and

$$\mathbf{A}(c\hat{a}) = c\mathbf{A}\hat{a} . \quad (8)$$

It is customary to regard the vector  $\hat{a}$  in Equations (6) as the product of the tensor  $\mathbf{A}$  and the vector  $\hat{b}$ . We say that the vector  $\hat{b}$  is *left-multiplied* by the tensor  $\mathbf{A}$ . It is also possible to write expressions of the form

$$\hat{c} = \hat{b}\mathbf{A} \quad (9)$$

in which the vector  $\hat{b}$  is *right-multiplied* by the tensor  $\mathbf{A}$ . If

$$\mathbf{A}\hat{a} = \hat{a}\mathbf{B} \quad (10)$$

for all vectors  $\hat{a}$ , we say that  $\mathbf{A}$  is the *transpose* of  $\mathbf{B}$  and write

$$\mathbf{A} = \mathbf{B}^T . \quad (11)$$

Tensors may be added and subtracted according to the usual algebraic rules. Addition is defined such that

$$\mathbf{A} = \mathbf{B} + \mathbf{C} \text{ iff } \mathbf{A}\hat{a} = \mathbf{B}\hat{a} + \mathbf{C}\hat{a} \text{ for all } \hat{a} \quad (12)$$

The product of two tensors is defined such that

$$\mathbf{A} = \mathbf{B}\mathbf{C} \text{ iff } \mathbf{A}\hat{a} = \mathbf{B}(\mathbf{C}\hat{a}) \text{ for all } \hat{a} \quad (13)$$

The outer product of two vectors is a tensor and may be written

$$\mathbf{A} = \hat{a} \otimes \hat{b} \quad (14)$$

It is defined by

$$\mathbf{A} = \hat{a} \otimes \hat{b} \text{ iff } \mathbf{A}\hat{c} = (\hat{b} \cdot \hat{c})\hat{a} \text{ for all } \hat{c} \quad (15)$$

Note that the outer product is not commutative, unlike the inner product, since

$$\hat{a} \otimes \hat{b} = (\hat{b} \otimes \hat{a})^T \quad (16)$$

Many derived quantities in physics are expressed as tensors. For example, we observe in the laboratory that a reflective surface exposed to a set of light sources feels a force which depends on the orientation and area of the surface. If we form a vector  $\hat{s}$  whose magnitude is equal to the surface area and whose direction is perpendicular to the surface, we find that the force experienced by the surface is given by

$$\hat{f} = \mathbf{P}\hat{s} \quad (17)$$

where  $\mathbf{P}$  is a tensor (the radiation pressure tensor) which depends only on the intensity and location of the light sources relative to the location of the reflective surface.

Likewise, consider a body subjected to deformation. Let the displacement between two nearby particles in the undeformed body be represented by the vector  $\vec{u}$  and the displacement between the same two particles after deformation be represented by the vector  $\vec{u}'$ . The two vectors are related by the expression

$$\vec{u}' = \mathbf{J}\vec{u} \quad (18)$$

where  $\mathbf{J}$  is called the Jacobian tensor. We note that  $\mathbf{J}$  may be different at different points in the body.

### 1.1.3 Symmetric and Antisymmetric Tensors

Many tensors important in physics are *symmetric*; that is,

$$\mathbf{A}^T = \mathbf{A} \quad (19)$$

Likewise, there are important tensors which are *antisymmetric*, having the property

$$\mathbf{A}^T = -\mathbf{A}. \quad (20)$$

If a tensor is known to have one of these symmetry properties, calculations involving that tensor can usually be simplified. In addition, it is sometimes useful to split a full tensor into symmetric and antisymmetric parts via the formulae

$$\text{Sym}(\mathbf{A}) = \frac{1}{2}(\mathbf{A} + \mathbf{A}^T) \quad (21)$$

$$\text{Anti}(\mathbf{A}) = \frac{1}{2}(\mathbf{A} - \mathbf{A}^T) \quad (22)$$

It is easily verified that these two tensors have the indicated symmetry properties and that  $\mathbf{A} = \text{Sym}(\mathbf{A}) + \text{Anti}(\mathbf{A})$ .

### 1.1.4 Vector and Tensor Components; Indicial Notation

Computers are unable to handle vectors and tensors directly. Their hardware is designed to add, subtract, multiply, and divide representations of real numbers.

Fortunately we can represent vectors and tensors as sets of real numbers. However, to do so, we must establish an arbitrary *frame of reference*. We do this by selecting three mutually orthogonal directions  $\hat{x}$ ,  $\hat{y}$ , and  $\hat{z}$ . These correspond to the x, y, and z axes of a Cartesian coordinate system. We can then express any vector in the form

$$\vec{a} = a_1\hat{x} + a_2\hat{y} + a_3\hat{z} \quad (23)$$

The three numbers  $a_1$ ,  $a_2$ , and  $a_3$  (the *components* of the vector) are real numbers and can be processed by a computer. Using Equation (23), we can represent any vector operation as a sequence of operations on sets of real numbers. We use the symbol  $a_i$  to represent the set of real numbers  $a_1$ ,  $a_2$ , and  $a_3$ .

Some computers are optimized to perform calculations on sets of real numbers; computer scientists refer to these as vector computers, but the word “vector” is not being used in the sense understood by physicists.

We can write any tensor in the form

$$\begin{aligned} \mathbf{A} = & A_{11}(\hat{x} \otimes \hat{x}) + A_{12}(\hat{x} \otimes \hat{y}) + A_{13}(\hat{x} \otimes \hat{z}) \\ & + A_{21}(\hat{y} \otimes \hat{x}) + A_{22}(\hat{y} \otimes \hat{y}) + A_{23}(\hat{y} \otimes \hat{z}) \\ & + A_{31}(\hat{z} \otimes \hat{x}) + A_{32}(\hat{z} \otimes \hat{y}) + A_{33}(\hat{z} \otimes \hat{z}) \end{aligned} \quad (24)$$

Thus, a computer can treat a tensor as if it was an array of nine real numbers. These real numbers are spoken of as the *components* of the tensor. We represent this set of numbers by the symbol  $A_{ij}$ .

We thus have a way to handle vectors and tensors on computers, but at a price: we must replace each vector and tensor by a set of real numbers and each vector or tensor operation by a (possibly extensive) sequence of operations on sets of real numbers. This sequence of operations is written using *indicial notation*. For example, the inner or dot product of two vectors is written in symbolic notation as

$$r = \vec{a} \bullet \vec{b}. \quad (25)$$

It can be written in indicial notation as

$$r = \sum_{i=1}^3 a_i b_i. \quad (26)$$

where  $a_i$  and  $b_i$  are the components of the vectors  $\vec{a}$  and  $\vec{b}$ . Proofs of the equivalence of the symbolic and indicial representations of vector operations will not be presented in this report.

### 1.1.5 Einstein Summation Convention

Sums over all values of an index, such as Equation (26), are so common that it is customary to adopt the Einstein summation convention. Under this convention, any term in which an index is repeated, such as  $a_i b_i$ , is interpreted to mean a sum over all values of the index  $i$ . That is,

$$a_i b_i \text{ (Einstein convention)} \Leftrightarrow \sum_{i=1}^3 a_i b_i \text{ (ordinary usage)} \quad (27)$$

If more than one index is repeated, we have a multiple sum, e.g.,

$$a_i B_{ij} c_j \text{ (Einstein convention)} \Leftrightarrow \sum_{i=1}^3 \sum_{j=1}^3 a_i B_{ij} c_j \text{ (ordinary usage)}. \quad (28)$$

We use the Einstein summation convention throughout this report.

### 1.1.6 Dimensionality

Physical space is three-dimensional. However, in many situations there are translational or rotational symmetries that reduce the effective dimensionality. **PHYSLIB** has therefore been written to accommodate 1-D, 2-D and 3-D calculations. The programmer selects the dimensionality by defining either `ONE_D`, `TWO_D`, or `THREE_D` as a preprocessor symbol in the command line to the C++ compiler.

By selecting `ONE_D` or `TWO_D`, the programmer eliminates certain components from the vector and tensor representations that are always zero in these cases. For example, under `ONE_D`, only the  $x$  component of the vector can be nonzero. The elimination of unused vector and tensor components reduces memory usage and increases run-time performance.

An integer constant, `DIMENSION`, is set to the number of dimensions (1, 2 or 3) selected by the programmer.

## 1.2 The C++ Programming Language

One of the characteristics of computational physics programs is their growing complexity. It is not now uncommon for a production code to exceed one hundred thousand lines in length when written in traditional programming languages such as FORTRAN. Such huge codes are also found in the areas of advanced graphics and operating systems.

Large codes are extremely difficult to manage. However, experience shows that proper use of *hierarchies* can reduce the complexity of large codes by orders of magnitude. C++ is an excellent language for large codes because it fully supports procedure hierarchies, nesting hierarchies, and inheritance hierarchies [1].

C++ is also the first high-level language with object-oriented capability to become widely popular. Because well-written C++ code approaches the efficiency of conventional C coding, C++ may prove to be the language of choice for large scientific computing projects. A description of the C++ language is beyond the scope of this report. However, we briefly describe the advantages of C++ below.

The definitive feature of C++ is the *class*. This is essentially a programmer-defined data type that supplements the standard data types (such as `int`, `float`, or `double`) that are part of the language. A class is *declared*, usually in a header file, at which time the compiler knows its characteristics; individual variables or *instances* of the class may then be declared by the programmer.

### 1.2.1 Data Abstraction

A class declaration typically includes *data members* and specifies member access rules. The data members are a set of floating numbers, integers, pointers, or instances of simpler classes. For example, a class representing complex numbers would probably contain two floating variables as data members: one for the real and one for the imaginary part of the complex number. Each time a variable of a given class is declared, enough memory is set aside to hold its data members.

Classes enforce data abstraction. Generally speaking, the data members of a class are directly accessible only to a set of functions enumerated within the class definition. These functions are the only place where an instance of a class is not viewed as a coherent object. The PHYSLIB library is built around the concept of data abstraction.

### 1.2.2 Special Member Functions and Dynamic Memory Management

The special member functions of a class are utility functions that create, destroy, or assign values to an instance of a class. Thus, whenever a class variable is declared, a constructor function is called to initialize the object. Likewise, when a class variable goes out of scope and is no longer needed, a destructor is called to do any necessary cleanup before its memory is freed. This makes it possible to carry out sophisticated dynamic memory management in a transparent manner. For example, a large array of floating numbers can be represented by a class with constructor and destructor functions. The constructor func-

tions, which are automatically called when a variable of the array class is declared, can allocate the appropriate amount of memory. The destructor, which is automatically called when the variable goes out of scope, can return the memory to the system. The programmer sees none of this; he only writes a constructor and destructor function, and the compiler sees to it that they are called at the appropriate times.

PHYSLIB does not make use of such memory management mechanisms, but future reports will discuss how memory management is carried out in more sophisticated classes used in RHALE++.

If a class has no constructor functions, the compiler simply allocates memory for the data members whenever an instance of the class is declared. Likewise, if a class has no destructor function, the compiler simply frees the memory allocated for an instance of a class when it goes out of scope.

Other special member functions may be declared to assign values to an object. For example, an instance of an array class would need to free its old storage area before allocating new memory to receive a new value. If no assignment function is declared for a class, the compiler simply copies the values of all the data members when an assignment is made.

### 1.2.3 Function and Operator Overloading

When data abstraction is implemented in less sophisticated programming languages, the code tends to dissolve into many calls to a few privileged routines that manipulate individual components of the various data structures. Many of these routines implement distinct operations on the data structures that could just as well be represented by arithmetic operators. For example, if data structures representing complex numbers are used in a C program, there will be many calls to functions that implement complex addition and multiplication.

The C++ language permits programmers to *overload* the standard set of operator symbols. For example, the programmer can declare that the ‘\*’ operator represents complex multiplication when applied to complex variables. This adds a new context-dependent meaning to this symbol. The compiler can distinguish whether the ‘\*’ represents ordinary floating-point multiplication or complex multiplication by examining the type of its operands.

When an overloaded operator is used in this manner, the compiler replaces it with a call to the appropriate function defined by the programmer. Thus, the actual machine code generated is not much different than that described above for a C program. However, the code the programmer writes is much more aesthetically pleasing; and, when another programmer is trying to read and understand the code, aesthetics is everything.

The C++ language permits programmers to overload function names as well as operators. Every function declaration includes the argument list, as with ANSI C. However, more than one function with a given name can exist if they have different argument lists. When one of the functions is called, the compiler selects the correct function based on the types

of the arguments. If a function call has an argument list that does not match any function by that name, the compiler reports an error.

Consider this example of a C code:

```
#include <math.h>
#include "complex.h"
main(){
    struct Complex a = {3., 2.5}, b = {2., 0.}, c;
    c = CSqrt(CAdd(CMult(a,a), CMult(b,b)));
    fprintf("The result is %f, %f\n", c.Real, c.Imag);
}
```

This short program evaluates and prints a complicated complex expression. Note the many function calls needed to implement data abstraction.

In C++ one might have

```
#include <math.h>
#include "complex.h"
main(){
    Complex a(3., 2.5), b(2., 0.), c;
    c = sqrt(a*a + b*b);
    fprintf("The result is %f, %f\n", c.Real(), c.Imag());
}
```

This illustrates how the function calls have been replaced by more transparent operator notation. The actual machine code generated by the compiler replaces the operators with the appropriate function calls. In addition, the `sqrt()` function has been overloaded; the two versions are `double sqrt(const double)` and `Complex sqrt(const Complex)`. The first version takes and returns floating point numbers, while the second takes and returns complex numbers. In the program above, the second version has been used, which the compiler correctly recognizes from the fact that `a*a + b*b` is an expression with type `Complex`.



## 2. The PHYSLIB Library

The PHYSLIB library consists of two files: a header file, `physlib.h`, and a C++ source file, `physlib.C`.

The header file contains C++ code that defines the four classes described below. It must be included at the start of any C++ program that wishes to use these classes. The source file contains a few large functions that are not appropriate for inlining, and it is compiled and linked with the users' code.

Inlining is a way to reduce computation time at the cost of increased memory usage. An inline function is not actually called whenever it is referenced; instead, a local copy of the function body is inserted in the calling routine by the compiler. This eliminates the overhead associated with making a function call and permits global optimizations (such as vectorization) that are normally inhibited by function calls. The trade-off is that there are numerous local copies of the function in the code rather than one global copy. If the function is very simple and is called many times, as is usually the case for PHYSLIB functions, the savings in computation time are worth the increase in memory usage.

In each case, the reference frame is implied by the values used to initialize the vectors and tensors in a calculation. In addition, it is assumed that all floating numbers are represented in double precision. This is wasteful on intrinsically double-precision machines such as a Cray; the Cray version of the library will replace `double` with `float` everywhere.

### 2.1 class Vector

This class represents Cartesian vectors, which are quantities having both magnitude and direction.

*Symbolic Notation:*  $\vec{a}$                       *Indicial Notation:*  $a_i$

#### 2.1.1 Private Data Members

<code>double x;</code>	X component of vector ( $a_1$ )
<code>double y;</code>	Y component of vector ( $a_2$ )
<code>double z;</code>	Z component of vector ( $a_3$ )

The Z component is required even in the 2-D version of the library. This is because RHALE++ and some other finite element codes use a rotation algorithm that requires vectors with Z components.

#### 2.1.2 Special Member Functions

```
Vector(void);
```

*Sample code:*

```
Vector a;           // Default constructor called
                   // when a is declared
```

This is the default constructor for instances of the `Vector` class. It does nothing to initialize the vector. It is declared only to let the compiler know that initialization can be skipped.

```
Vector(const double, const double, const double);
```

*Sample code:*

```
Vector a(5., 6., 2.);
```

Construct a vector with the given components.

```
Vector(const Vector&);
```

*Sample code:*

```
Vector a;
Vector b = a;       // Construct and initialize
```

This is the copy constructor for objects of class `Vector`. It is defined mainly to enhance vectorization on CRAY computers.

```
Vector& operator=(const Vector&);
```

*Sample code:*

```
Vector a, b;
a = b;
```

This is the assignment operator for objects of class `Vector`. It is defined mainly to enhance vectorization on CRAY computers.

```
double X(void) const;
```

*Symbolic notation:*  $\hat{a} \cdot \hat{x}$       *Indicial notation:*  $a_1$

*Sample code:*

```
Vector a;
printf("The X component of a is %f\n", a.X());
```

```
double Y(void) const;
```

*Symbolic notation:*  $\hat{a} \cdot \hat{y}$       *Indicial notation:*  $a_2$

*Sample code:*

```
Vector a;
printf("The Y component of a is %f\n", a.Y());
```

```
double Z(void) const;
```

*Symbolic notation:*  $\hat{a} \cdot \hat{z}$       *Indicial notation:*  $a_3$

*Sample code:*

```
Vector a;
printf("TheZ component of a is %f\n", a.Z());
```

```
void X(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $a_1 \leftarrow s$

*Sample code:*

```
Vector a;
a.X(2.);                    // set X component of a to 2.
```

```
void Y(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $a_2 \leftarrow s$

*Sample code:*

```
Vector a;
a.Y(2.);                    // set Y component of a to 2.
```

```
void Z(const double);
```

*Symbolic notation: None*      *Indicial notation:  $a_3 \leftarrow s$*

*Sample code:*

```
Vector a;
    a.Z(2.);           // set Z component of a to 2.
```

Provide access to the components of a vector. This is required chiefly for I/O but is also a means for letting future classes work with vectors without requiring a huge list of friend functions in the vector class definition. It does not violate the idea of data abstraction, since nonprivileged functions must still access the components of a vector through a functional interface.

### 2.1.3 Utility Functions

```
int fread(Vector&, FILE*);
int fwrite(const Vector&, FILE*);
int fread(Vector*, int, FILE*);
int fwrite(const Vector*, const int, FILE*);
```

*Sample code:*

```
Vector a, b, c[2], d[5];
FILE* InFile, OutFile;
fread (a, InFile);
fread (c, 2, InFile);
fwrite (b, OutFile);
fwrite (d, 5, OutFile);
```

These overloads provide a convenient interface to the `fread()` and `fwrite()` library functions for binary input/output. The second version of each is intended for arrays of vectors (e.g., `Vector c[2];` declares an array of two vectors).

These functions were written to be as consistent as possible with the standard `fread()` and `fwrite()` functions. Thus, they are friends rather than member functions, and the integer returned is the number of objects read or written.

## 2.2 class Tensor

This class represents general Cartesian 2nd-order tensors. In the 2-D version, the off-diagonal z terms  $A_{13}$ ,  $A_{23}$ ,  $A_{31}$ , and  $A_{32}$  are omitted. The diagonal z term,  $A_{33}$ , is needed in 2-D finite element codes.

*Symbolic notation:*  $\mathbf{A}$

*Indicial notation:*  $A_{ij}$

### 2.2.1 Private Data Members

<code>double xx;</code>	xx component of tensor ( $A_{11}$ )
<code>double xy;</code>	xy component of tensor ( $A_{12}$ )
<code>double xz;</code>	xz component of tensor ( $A_{13}$ )
<code>double yx;</code>	yz component of tensor ( $A_{21}$ )
<code>double yy;</code>	yy component of tensor ( $A_{22}$ )
<code>double yz;</code>	yz component of tensor ( $A_{23}$ )
<code>double zx;</code>	zx component of tensor ( $A_{31}$ )
<code>double zy;</code>	zy component of tensor ( $A_{32}$ )
<code>double zz;</code>	zz component of tensor ( $A_{33}$ )

### 2.2.2 Special Member Functions

```
Tensor(void);
```

*Sample code:*

```
Tensor a;           // Declare an uninitialized
                   // tensor.
```

Default constructor for instances of the Tensor class.

```
Tensor(const double, const double, const double, const
double, const double, const double, const double,
const double, const double);
```

*Sample code:*

```
Tensor a(2., 3., 5.,
```

```
4., 6., 4.,
1., 9., 11.);
```

Construct a tensor with the given components. The arguments corresponding to off-diagonal z terms are omitted in the 2-D version.

```
Tensor(const Tensor&);
```

*Sample code:*

```
Tensor a;
Tensor b = a;           // Construct and initialize
```

This is the copy constructor for objects of class Tensor. It is defined mainly to enhance vectorization on CRAY computers.

```
Tensor& operator=(const Tensor&);
```

*Sample code:*

```
Tensor a, b;
a = b;
```

This is the assignment operator for objects of class Tensor. It is defined mainly to enhance vectorization on CRAY computers.

```
Tensor(const SymTensor&);
```

```
Tensor(const AntiTensor&);
```

*Sample code:*

```
SymTensor a;
AntiTensor b;
Tensor c = a, d = b;
```

Convert a symmetric or antisymmetric tensor to full tensor representation. These operators become standard conversions that the compiler invokes implicitly where needed. However, most operators are explicitly defined for mixed tensor types, since this is more efficient.

These conversions are somewhat dangerous, since useless operations such as `Trans(SymTensor)` or `Tr(AntiTensor)` will be accepted by the compiler. The worst consequence of permitting these conversions is that operations such as `Inverse(AntiTensor)` will be attempted and result in a singular matrix error. The RHALE++ development team felt that, since these conversions are so natural, they should be included in PHYSLIB in spite of the potential dangers.

```
Tensor& operator=(const SymTensor&);
```

```
Tensor& operator=(const AntiTensor&);
```

*Sample code:*

```
SymTensor a;
AntiTensor b;
Tensor c, d;
c = a;
d = b;
```

Assign a symmetric or antisymmetric tensor value to a preexisting tensor variable. If these operations were not defined, the compiler would call the conversion constructors defined above and assign the result, which is less efficient than assigning the values directly.

```
double XX(void) const;
```

*Symbolic notation:*  $\hat{x}A\hat{x}$       *Indicial notation:*  $A_{11}$

*Sample code:*

```
Tensor A;
printf("The XX component of A is %f", A.XX());
```

```
double XY(void) const;
```

*Symbolic notation:*  $\hat{x}A\hat{y}$       *Indicial notation:*  $A_{12}$

*Sample code:*

```
Tensor A;
```

```
printf("The XY component of A is %f", A.XY());
```

```
double XZ(void) const;
```

*Symbolic notation:*  $\hat{x}A\hat{z}$       *Indicial notation:*  $A_{13}$

*Sample code:*

```
Tensor A;  
printf("The XZ component of A is %f", A.XZ());
```

```
double YX(void) const;
```

*Symbolic notation:*  $\hat{y}A\hat{x}$       *Indicial notation:*  $A_{21}$

*Sample code:*

```
Tensor A;  
printf("The YX component of A is %f", A.YX());
```

```
double YY(void) const;
```

*Symbolic notation:*  $\hat{y}A\hat{y}$       *Indicial notation:*  $A_{22}$

*Sample code:*

```
Tensor A;  
printf("The YY component of A is %f", A.YY());
```

```
double YZ(void) const;
```

*Symbolic notation:*  $\hat{y}A\hat{z}$       *Indicial notation:*  $A_{23}$

*Sample code:*

```
Tensor A;  
printf("The YZ component of A is %f", A.YZ());
```

```
double ZX(void) const;
```

*Symbolic notation:*  $\hat{\mathbf{A}}\hat{\mathbf{x}}$       *Indicial notation:*  $A_{31}$

*Sample code:*

```
Tensor A;
printf("The ZX component of A is %f", A.ZX());
```

```
double ZY(void) const;
```

*Symbolic notation:*  $\hat{\mathbf{A}}\hat{\mathbf{y}}$       *Indicial notation:*  $A_{32}$

*Sample code:*

```
Tensor A;
printf("The ZY component of A is %f", A.ZY());
```

```
double ZZ(void) const;
```

*Symbolic notation:*  $\hat{\mathbf{A}}\hat{\mathbf{z}}$       *Indicial notation:*  $A_{33}$

*Sample code:*

```
Tensor A;
printf("The ZZ component of A is %f", A.ZZ());
```

```
void XX(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{11} \leftarrow s$

*Sample code:*

```
Tensor A;
A.XX(3.);                            // Set XX component of A to 3.
```

```
void XY(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{12} \leftarrow s$

*Sample code:*

```
Tensor A;
```

```
A.XY(3.); // Set XY component of A to 3.
```

```
void XZ(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{13} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.XZ(3.); // Set XZ component of A to 3.
```

```
void YX(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{21} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.YX(3.); // Set YX component of A to 3.
```

```
void YY(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{22} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.YY(3.); // Set YY component of A to 3.
```

```
void YZ(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{23} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.YZ(3.); // Set YZ component of A to 3.
```

```
void ZX(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{31} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.ZX(3.);           // Set ZX component of A to 3.
```

```
void ZY(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{32} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.ZY(3.);           // Set ZY component of A to 3.
```

```
void ZZ(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{33} \leftarrow s$*

*Sample code:*

```
Tensor A;
A.ZZ(3.);           // Set ZZ component of A to 3.
```

Provide access to components of a tensor through a functional interface. The functions corresponding to off-diagonal z terms do not exist in the 2-D version of the library, since these components always vanish in 2-D finite element codes.

### 2.2.3 Utility Functions

```
int fread(Tensor&, FILE*);
int fwrite(const Tensor&, FILE*);
int fread(Tensor*, int, FILE*);
int fwrite(const Tensor*, const int, FILE*);
```

*Sample code:*

```
Tensor a, b, c[2], d[5];
FILE* InFile, OutFile;
fread (a, InFile);
fread (c, 2, InFile);
```

```
fwrite (b, OutFile);
fwrite (d, 5, OutFile);
```

These overloads provide a convenient interface to the `fread()` and `fwrite()` library functions for binary input/output.

These functions were written to be as consistent as possible with the standard `fread()` and `fwrite()` functions. Thus, they are friends rather than member functions, and the integer returned is the number of objects read or written.

## 2.3 class SymTensor

This class represents symmetric tensors. By providing a separate representation of symmetric tensors, we save both memory and computation time, since a symmetric tensor has fewer independent components. Since symmetric tensor are simply a special case of general tensors, they share the same notation and operations.

*Symbolic notation:*  $\mathbf{A}$                       *Indicial notation:*  $A_{ij}$

### 2.3.1 Private Data Members

<code>double xx;</code>	xx component of a symmetric tensor ( $A_{11}$ )
<code>double xy;</code>	xy component of a symmetric tensor ( $A_{12} = A_{21}$ )
<code>double xz;</code>	xz component of a symmetric tensor ( $A_{13} = A_{31}$ )
<code>double yy;</code>	yy component of a symmetric tensor ( $A_{22}$ )
<code>double yz;</code>	yz component of a symmetric tensor ( $A_{23} = A_{32}$ )
<code>double zz;</code>	zz component of a symmetric tensor ( $A_{33}$ )

### 2.3.2 Special Member Functions

```
SymTensor(void);
```

*Sample code:*

```
SymTensor a;                      // Construct an uninitialized
                                 // SymTensor.
```

Default constructor for instances of the class `SymTensor`.

```
SymTensor(const double, const double, const double,
const double, const double, const double);
```

*Sample code:*

```
SymTensor a(1., 5., 3.,
            4., 6.,
            5.);
```

Construct a symmetric tensor with the given components. The arguments corresponding to off-diagonal z components are omitted in the 2-D version.

```
SymTensor(const SymTensor&);
```

*Sample code:*

```
SymTensor a;
SymTensor b = a;    // Construct and initialize
```

This is the copy constructor for objects of class `SymTensor`. It is defined mainly to enhance vectorization on CRAY computers.

```
SymTensor& operator=(const SymTensor&);
```

*Sample code:*

```
SymTensor a, b;
a = b;
```

This is the assignment operator for objects of class `SymTensor`. It is defined mainly to enhance vectorization on CRAY computers.

```
double XX(void) const;
```

*Symbolic notation:*  $\hat{\mathbf{A}}$       *Indicial notation:*  $A_{11}$

*Sample code:*

```
SymTensor A;  
printf("The XX component of A is %f", A.XX());
```

```
double XY(void) const;
```

*Symbolic notation:*  $\hat{x}\mathbf{A}\hat{y}$       *Indicial notation:*  $A_{12}$

*Sample code:*

```
SymTensor A;  
printf("The XY component of A is %f", A.XY());
```

```
double XZ(void) const;
```

*Symbolic notation:*  $\hat{x}\mathbf{A}\hat{z}$       *Indicial notation:*  $A_{13}$

*Sample code:*

```
SymTensor A;  
printf("The XZ component of A is %f", A.XZ());
```

```
double YY(void) const;
```

*Symbolic notation:*  $\hat{y}\mathbf{A}\hat{y}$       *Indicial notation:*  $A_{22}$

*Sample code:*

```
SymTensor A;  
printf("The YY component of A is %f", A.YY());
```

```
double YZ(void) const;
```

*Symbolic notation:*  $\hat{y}\mathbf{A}\hat{z}$       *Indicial notation:*  $A_{23}$

*Sample code:*

```
SymTensor A;  
printf("The YZ component of A is %f", A.YZ());
```

```
double ZZ(void) const;
```

*Symbolic notation:*  $\hat{A}$       *Indicial notation:*  $A_{33}$

*Sample code:*

```
SymTensor A;
printf("The ZZ component of A is %f", A.ZZ());
```

```
void XX(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{11} \leftarrow s$

*Sample code:*

```
SymTensor A;
A.XX(3.);                    // Set XX component of A to 3.
```

```
void XY(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{12} \leftarrow s$

*Sample code:*

```
SymTensor A;
A.XY(3.);                    // Set XY component of A to 3.
```

```
void XZ(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{13} \leftarrow s$

*Sample code:*

```
SymTensor A;
A.XZ(3.);                    // Set XZ component of A to 3.
```

```
void YY(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{22} \leftarrow s$

*Sample code:*

```
SymTensor A;
A.YY(3.);           // Set YY component of A to 3.
```

```
void YZ(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{23} \leftarrow s$*

*Sample code:*

```
SymTensor A;
A.YZ(3.);           // Set YZ component of A to 3.
```

```
void ZZ(const double);
```

*Symbolic notation: None*      *Indicial notation:  $A_{33} \leftarrow s$*

*Sample code:*

```
SymTensor A;
A.ZZ(3.);           // Set ZZ component of A to 3.
```

Provide access to components of a symmetric tensor through a functional interface. The functions corresponding to off-diagonal z terms do not exist in the 2-D version of the library, since these components always vanish in 2-D finite element codes.

### 2.3.3 Utility Functions

```
int fread(SymTensor&, FILE*);
int fwrite(const SymTensor&, FILE*);
int fread(SymTensor*, int, FILE*);
int fwrite(const SymTensor*, const int, FILE*);
```

*Sample code:*

```
SymTensor a, b, c[2], d[5];
FILE* InFile, OutFile;
fread (a, InFile);
```

```
fread (c, 2, InFile);
fwrite (b, OutFile);
fwrite (d, 5, OutFile);
```

These overloads provide a convenient interface to the `fread()` and `fwrite()` library functions for binary input/output.

These functions were written to be as consistent as possible with the standard `fread()` and `fwrite()` functions. Thus, they are friends rather than member functions, and the integer returned is the number of objects read or written.

## 2.4 class `AntiTensor`

This class represents antisymmetric tensors. By providing a separate representation, we save quite a lot of memory and computation time. Since antisymmetric tensors are a special case of general tensors, the notation and operators are identical.

*Symbolic notation:*  $A$                       *Indicial notation:*  $A_{ij}$

### 2.4.1 Private Data Members

```
double xy;                      xy component of the tensor ( $A_{12} = -A_{21}$ )
double xz;                      xz component of the tensor ( $A_{13} = -A_{31}$ )
double yz;                      yz component of the tensor ( $A_{23} = -A_{32}$ )
```

### 2.4.2 Special Member Functions

```
AntiTensor(void);
```

*Sample code:*

```
AntiTensor A;                    // Construct an uninitialized
                               // AntiTensor
```

Default constructor for instances of the class `AntiTensor`.

```
AntiTensor(const double, const double, const double);
```

*Sample code:*

```
AntiTensor A(-2., -3., -1.);
```

Construct an antisymmetric tensor with the given components. The second and third arguments are omitted in 3-D.

```
AntiTensor(const AntiTensor&);
```

*Sample code:*

```
AntiTensor a;
AntiTensor b = a;    // Construct and initialize
```

This is the copy constructor for objects of class AntiTensor. It is defined mainly to enhance vectorization on CRAY computers.

```
AntiTensor& operator=(const AntiTensor&);
```

*Sample code:*

```
AntiTensor a, b;
a = b;
```

This is the assignment operator for objects of class AntiTensor. It is defined mainly to enhance vectorization on CRAY computers.

```
double XY(void) const;
```

*Symbolic notation:*  $\hat{x}\mathbf{A}\hat{y}$       *Indicial notation:*  $A_{12}$

*Sample code:*

```
AntiTensor A;
printf("The XY component of A is %f", A.XY());
```

```
double XZ(void) const;
```

*Symbolic notation:*  $\hat{x}\mathbf{A}\hat{z}$       *Indicial notation:*  $A_{13}$

*Sample code:*

```
AntiTensor A;
printf("The XZ component of A is %f", A.XZ());
```

```
double YZ(void) const;
```

*Symbolic notation:*  $\hat{y}A\hat{z}$       *Indicial notation:*  $A_{23}$

*Sample code:*

```
AntiTensor A;
printf("The YZ component of A is %f", A.YZ());
```

```
void XY(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{12} \leftarrow s$

*Sample code:*

```
AntiTensor A;
A.XY(3.);                      // Set XY component of A to 3.
```

```
void XZ(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{13} \leftarrow s$

*Sample code:*

```
SymTensor A;
A.XZ(3.);                      // Set XZ component of A to 3.
```

```
void YZ(const double);
```

*Symbolic notation:* None      *Indicial notation:*  $A_{23} \leftarrow s$

*Sample code:*

```
AntiTensor A;
A.YZ(3.);                      // Set YZ component of A to 3.
```

Provide access to components of an antisymmetric tensor through a functional interface. The functions corresponding to off-diagonal z terms do not exist in the 2-D version of the library, since these components always vanish in 2-D finite element codes.

### 2.4.3 Utility Functions

```
int fread(AntiTensor&, FILE*);
int fwrite(const AntiTensor&, FILE*);
int fread(AntiTensor*, int, FILE*);
int fwrite(const AntiTensor*, const int, FILE*);
```

*Sample code:*

```
AntiTensor a, b, c[2], d[5];
FILE* InFile, OutFile;
fread (a, InFile);
fread (c, 2, InFile);
fwrite (b, OutFile);
fwrite (d, 5, OutFile);
```

These overloads provide a convenient interface to the `fread()` and `fwrite()` library functions for binary input/output.

These functions were written to be as consistent as possible with the standard `fread()` and `fwrite()` functions. Thus, they are friends rather than member functions, and the integer returned is the number of objects read or written.

## 2.5 Operator Overload Functions

Vector operator-(void) const;

*Symbolic notation:*  $-\vec{a}$       *Indicial notation:*  $-a_i$

*Sample code:*

```
Vector a, b;
a = -b;
```

Return the opposite of a vector.

Tensor operator-(void) const;

SymTensor operator-(void) const;

AntiTensor operator-(void) const;

*Symbolic notation:*  $-\mathbf{A}$       *Indicial notation:*  $-A_{ij}$

*Sample code:*

```
Tensor A, B;
A = -B;
```

Return the opposite of a tensor.

Vector operator\*(const Vector&, const double);

Vector operator\*(const double, const Vector&);

*Symbolic notation:*  $\vec{a}c$       *Indicial notation:*  $a_i c$

*Sample code:*

```
Vector a, b;
double c;
a = b * c;
```

Return the product of a scalar and a vector. This operation commutes (as can be seen from its indicial representation) but C++ makes no assumptions about commutivity of operations; hence, both orderings must be defined. C++ *does* assume

the usual rules of associativity for overloaded operators (thus  $a*b*c$  means  $(a*b)*c$  or  $(a \bullet b)c$ ).

`Vector& operator*=(const double);`

*Symbolic notation:*  $\vec{a} \leftarrow \vec{a}c$       *Indicial notation:*  $a_i \leftarrow a_i c$

*Sample code:*

```
Vector a;
double c;
a *= c;
```

Replace a vector by its product with a scalar.

`Vector operator/(const Vector&, const double);`

*Symbolic notation:*  $\vec{a}/c$       *Indicial notation:*  $a_i/c$

*Sample code:*

```
Vector a, b;
double c;
a = b/c;
```

Return the quotient of a vector with a scalar. The case  $c = 0$  results in a divide-by-zero error, which is handled differently on different computers.

`Vector& operator/=(const double);`

*Symbolic notation:*  $\vec{a} \leftarrow \vec{a}/c$       *Indicial notation:*  $a_i \leftarrow a_i/c$

*Sample code:*

```
Vector a;
double c;
a /= c;
```

Replace a vector by its quotient with a scalar. The case  $c = 0$  results in a divide-by-zero error, which is handled differently on different computers.

```
double operator*(const Vector&, const Vector&);
```

*Symbolic notation:*  $\vec{a} \cdot \vec{b}$       *Indicial notation:*  $a_i b_i$

*Sample code:*

```
Vector a, b;
double c;
c = a * b;
```

Return the dot or inner product of two vectors.

```
Tensor operator%(const Vector&, const Vector&);
```

*Symbolic notation:*  $\vec{a} \otimes \vec{b}$       *Indicial notation:*  $a_i b_j$

*Sample code:*

```
Vector a, b;
Tensor c;
c = a % b;
```

Return the tensor or outer product of two vectors. The operator ‘%’ represents the modulo operation when applied to integers. It was selected to represent the outer product of vectors because the compiler assigns it the same precedence as multiplication.

```
Vector operator+(const Vector&, const Vector&);
```

*Symbolic notation:*  $\vec{a} + \vec{b}$       *Indicial notation:*  $a_i + b_i$

*Sample code:*

```
Vector a, b, c;
a = b + c;
```

Return the sum of two vectors.

```
Vector& operator+=(const Vector&);
```

*Symbolic notation:*  $\vec{a} \leftarrow \vec{a} + \vec{b}$     *Indicial notation:*  $a_i \leftarrow a_i + b_i$

*Sample code:*

```
Vector a, b;  
a += b;
```

Replace a vector by its sum with another vector.

```
Vector operator-(const Vector&, const Vector&);
```

*Symbolic notation:*  $\vec{a} - \vec{b}$     *Indicial notation:*  $a_i - b_i$

*Sample code:*

```
Vector a, b, c;  
a = b - c;
```

Return the difference of two vectors.

```
Vector& operator-=(const Vector&);
```

*Symbolic notation:*  $\vec{a} \leftarrow \vec{a} - \vec{b}$     *Indicial notation:*  $a_i \leftarrow a_i - b_i$

*Sample code:*

```
Vector a, b;  
a -= b;
```

Replace a vector by its difference with a vector.

```
int operator==(const Vector&, const Vector&);
```

*Symbolic notation:*  $\vec{a} == \vec{b}$     *Indicial notation:*  $a_i == b_i$

*Sample code:*

```
int is_equal;  
Vector a, b;
```

```
is_equal = (a == b);
```

Determine if two vectors are equal.

```
int operator!=(const Vector&, const Vector&);
```

*Symbolic notation:*  $a \neq b_i$       *Indicial notation:*  $a_i \neq b_i$

*Sample code:*

```
int is_unequal;
Vector a, b;
is_unequal = (a != b);
```

Determine if two vectors are unequal.

```
ostream& operator<<(ostream&, const Vector&);
```

*Sample code:*

```
Vector a;
ofstream str("output.dat");
str << a;
```

Write a vector to an output stream in the form (x,y,z).

```
istream& operator>>(istream&, Vector&);
```

*Sample code:*

```
Vector a;
ifstream str("input.dat");
str >> a;
```

Read a vector from an input stream in the form (x,y,z).

```
Tensor operator*(const Tensor&, const double);
```

```
SymTensor operator*(const SymTensor&, const double);
```

```
AntiTensor operator*(const AntiTensor&, const double);
Tensor operator*(const double, const Tensor&);
SymTensor operator*(const double, const SymTensor&);
AntiTensor operator*(const double, const AntiTensor&);
```

*Symbolic notation:*  $A c$       *Indicial notation:*  $A_{ij}c$

*Sample code:*

```
Tensor A, B;
double c;
B = A * c;
```

Return the product of a tensor with a scalar.

```
Tensor& operator*=(const double);
SymTensor& operator*=(const double);
AntiTensor& operator*=(const double);
```

*Symbolic notation:*  $A \leftarrow A c$       *Indicial notation:*  $A_{ij} \leftarrow A_{ij}c$

*Sample code:*

```
Tensor A;
double c;
A *= c;
```

Replace a tensor by its product with a scalar.

```
Tensor operator/(const Tensor&, const double);
SymTensor operator/(const SymTensor&, const double);
AntiTensor operator/(const AntiTensor&, const double);
```

*Symbolic notation:*  $A/c$       *Indicial notation:*  $A_{ij}/c$

*Sample code:*

```
Tensor A, B;
```

```
double c;
```

```
B = A/c;
```

Return the quotient of a tensor with a scalar. The case  $c = 0$  results in a divide-by-zero error, which is handled differently by different computers.

```
Tensor operator/=(const double);
```

```
SymTensor& operator/=(const double);
```

```
AntiTensor& operator/=(const double);
```

*Symbolic notation:*  $\mathbf{A} \leftarrow \mathbf{A}/c$     *Indicial notation:*  $A_{ij} \leftarrow A_{ij}/c$

*Sample code:*

```
Tensor A;
```

```
double c;
```

```
A /= c;
```

Replace a tensor by its quotient with a scalar. The case  $c = 0$  results in a divide-by-zero error, which is handled differently by different computers.

```
Vector operator*(const Tensor&, const Vector&);
```

```
Vector operator*(const AntiTensor&, const Vector&);
```

```
Vector operator*(const SymTensor&, const Vector&);
```

*Symbolic notation:*  $\mathbf{A}\vec{b}$     *Indicial notation:*  $A_{ij}b_j$

*Sample code:*

```
Tensor A;
```

```
Vector b, c;
```

```
c = A * b;
```

Return the result of left-multiplying a vector by a tensor. There are three cases, corresponding to the three varieties of tensor implemented in PHYSLIB; all are identical in notation and usage, however.

```
Vector operator*(const Vector&, const Tensor&);
```

```
Vector operator*(const Vector&, const AntiTensor&);
```

```
Vector operator*(const Vector&, const SymTensor&);
```

*Symbolic notation:*  $\mathbf{aB}$       *Indicial notation:*  $a_j B_{ji}$

*Sample code:*

```
Vector a;
Tensor b, c;
c = a * b;
```

Return the result of right-multiplying a vector by a tensor.

```
Tensor operator*(const Tensor&, const Tensor&);
```

```
Tensor operator*(const SymTensor&, const Tensor&);
```

```
Tensor operator*(const Tensor&, const SymTensor&);
```

```
Tensor operator*(const SymTensor&, const SymTensor&);
```

```
Tensor operator*(const AntiTensor&, const Tensor&);
```

```
Tensor operator*(const Tensor&, const AntiTensor&);
```

```
Tensor operator*(const AntiTensor&, const SymTensor&);
```

```
Tensor operator*(const SymTensor&, const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{AB}$       *Indicial notation:*  $A_{ij} B_{jk}$

*Sample code:*

```
Tensor A, B, C;
C = A * B;
```

Return the product of a tensor with a tensor.

```
Tensor operator+(const Tensor&, const Tensor&);
```

```
Tensor operator+(const SymTensor&, const Tensor&);
```

```
Tensor operator+(const Tensor&, const SymTensor&);
```

```
SymTensor operator+(const SymTensor&, const SymTensor&);
```

```
Tensor operator+(const AntiTensor&, const Tensor&);
```

```

Tensor operator+(const Tensor&, const AntiTensor&);
Tensor operator+(const AntiTensor&, const SymTensor&);
Tensor operator+(const SymTensor&, const AntiTensor&);
AntiTensor operator+(const AntiTensor&, const AntiTensor&);

```

*Symbolic notation:*  $\mathbf{A} + \mathbf{B}$       *Indicial notation:*  $A_{ij} + B_{ij}$

*Sample code:*

```

Tensor A, B, C;
C = A + B;

```

Return the sum of two tensors.

```

Tensor& operator+=(const Tensor&);
Tensor& operator+=(const SymTensor&);
SymTensor& operator+=(const SymTensor&);
Tensor& operator+=(const AntiTensor&);
AntiTensor& operator+=(const AntiTensor&);

```

*Symbolic notation:*  $\mathbf{A} \leftarrow \mathbf{A} + \mathbf{B}$       *Indicial notation:*  $A_{ij} \leftarrow A_{ij} + B_{ij}$

*Sample code:*

```

Tensor A, B;
A += B;

```

Replace a tensor by its sum with another tensor.

```

Tensor operator-(const Tensor&, const Tensor&);
Tensor operator-(const SymTensor&, const Tensor&);
Tensor operator-(const Tensor&, const SymTensor&);
SymTensor operator-(const SymTensor&, const SymTensor&);
Tensor operator-(const AntiTensor&, const Tensor&);
Tensor operator-(const Tensor&, const AntiTensor&);

```

```
Tensor operator-(const AntiTensor&, const SymTensor&);
Tensor operator-(const SymTensor&, const AntiTensor&);
AntiTensor operator-(const AntiTensor&, const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{A} - \mathbf{B}$       *Indicial notation:*  $A_{ij} - B_{ij}$

*Sample code:*

```
Tensor A, B, C;
C = A - B;
```

Return the difference of two tensors.

```
Tensor& operator-=(const Tensor&);
Tensor& operator-=(const SymTensor&);
SymTensor& operator-=(const SymTensor&);
Tensor& operator-=(const AntiTensor&);
AntiTensor& operator-=(const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{A} \leftarrow \mathbf{A} - \mathbf{B}$       *Indicial notation:*  $A_{ij} \leftarrow A_{ij} - B_{ij}$

*Sample code:*

```
Tensor A, B;
A -= B;
```

Replace a tensor by its difference with another tensor.

```
int operator==(const Tensor&, const Tensor&);
int operator==(const SymTensor&, const SymTensor&);
int operator==(const AntiTensor&, const AntiTensor&);
```

*Symbolic notation:*  $\vec{a} == \vec{b}$       *Indicial notation:*  $a_i == b_i$

*Sample code:*

```
Int is_equal;
```

```
Vector a, b;  
is_equal = (a == b);
```

Determine if two vectors are equal..

```
int operator!=(const Tensor&, const Tensor&);  
int operator!=(const SymTensor&, const SymTensor&);  
int operator!=(const AntiTensor&, const AntiTensor&);
```

*Symbolic notation:*  $a \neq b_i$       *Indicial notation:*  $a_i \neq b_i$

*Sample code:*

```
Int is_unequal;  
Vector a, b;  
is_unequal = (a != b);
```

Determine if two vectors are unequal..

## 2.6 Methods

`Vector Cross(const Vector&, const Vector&);`

*Symbolic notation:*  $\vec{a} \times \vec{b}$       *Indicial notation:*  $\varepsilon_{ijk}a_jb_k$

*Sample code:*

```
Vector a, b, c;
c = Cross(a, b);
```

Vector or cross product of two vectors. The symbol  $\varepsilon_{ijk}$  is the permutation symbol, which is 0 if any of the  $i, j$ , or  $k$  are equal, 1 if they are an even permutation of the sequence 1, 2, 3, and -1 if they are an odd permutation of the sequence 1, 2, 3. For example,  $\varepsilon_{122} = 0$ ;  $\varepsilon_{123} = 1$ ; and  $\varepsilon_{213} = -1$ . The cross product is distributive and associative but not commutative.

`Vector Dual(const Tensor&);`

*Symbolic notation:*  $\text{Dual}(\mathbf{A})$       *Indicial notation:*  $\varepsilon_{ijk}A_{jk}$

*Sample code:*

```
Tensor A;
Vector b;
b = Dual(A);
```

Any tensor  $\mathbf{A}$  can be split into a symmetric part  $\frac{1}{2}(\mathbf{A} + \mathbf{A}^T)$  and an antisymmetric part  $\frac{1}{2}(\mathbf{A} - \mathbf{A}^T)$ . The dual of a tensor is a vector which depends uniquely on its antisymmetric part.

`AntiTensor Dual(const Vector&);`

*Symbolic notation:*  $\text{Dual}(\vec{a})$       *Indicial notation:*  $\varepsilon_{ijk}\vec{a}_k$

*Sample code:*

```
Vector a;
AntiTensor B;
B = Dual(a);
```

Dual of a vector. It can be proved that  $\text{Dual}(\text{Dual}(\hat{a})) = 2\hat{a}$ . The concept of the dual is closely related to the cross product, since  $\hat{b}\text{Dual}(\hat{a}) = \hat{a} \times \hat{b}$ .

```
double Norm(const Vector&);
```

*Symbolic notation:*  $|\hat{a}|$       *Indicial notation:*  $\sqrt{a_i a_i}$

*Sample code:*

```
Vector a;
double b;
b = Norm(a);
```

Returns the magnitude or norm of a vector. This is calculated as the square root of the dot product of the vector with itself.

```
double Norm(const Tensor&);
```

```
double Norm(const SymTensor&);
```

```
double Norm(const AntiTensor&);
```

*Symbolic notation:*  $|\mathbf{A}|$       *Indicial notation:*  $\sqrt{A_{ij} A_{ij}}$

*Sample code:*

```
Tensor A;
double c;
c = Norm(A);
```

Returns the norm of a tensor. This is calculated as the square root of the scalar product of the tensor with itself.

```
double Det(const Tensor&);
```

```
double Det(const SymTensor&);
```

*Symbolic notation:*  $\det[\mathbf{A}]$       *Indicial notation:*  $\frac{1}{6} \epsilon_{ijk} \epsilon_{lmn} A_{il} A_{jm} A_{kn}$

*Sample code:*

```
Tensor A;
```

```
double c;
c = Det(A);
```

Determinant of a tensor. It is always zero for an antisymmetric tensor.

```
Tensor Inverse(const Tensor&);
SymTensor Inverse(const SymTensor&);
```

*Symbolic notation:*  $\mathbf{A}^{-1}$

*Sample code:*

```
Tensor A, B;
B = Inverse(A);
```

Inverse of a tensor. If the tensor is singular, a divide-by-zero error will result (which may be ignored on machines using the IEEE floating point standard). Antisymmetric tensors are always singular.

```
double Tr(const Tensor&);
double Tr(const SymTensor&);
```

*Symbolic notation:*  $\text{Tr}\mathbf{A}$       *Indicial notation:*  $A_{kk}$

*Sample code:*

```
Tensor A;
double c;
c = Tr(A);
```

Trace of a tensor. The trace of an antisymmetric tensor is always zero.

```
Tensor Trans(const Tensor&);
```

*Symbolic notation:*  $\mathbf{A}^T$       *Indicial notation:*  $A_{jk}$

*Sample code:*

```
Tensor A, B;
B = Trans(A);
```

Transpose of a tensor. By definition, the transpose of a symmetric tensor is the tensor, while the transpose of an antisymmetric tensor is the opposite of the tensor.

```
SymTensor Sym(const Tensor&);
```

*Symbolic notation:*  $\frac{1}{2}(\mathbf{A} + \mathbf{A}^T)$     *Indicial notation:*  $\frac{1}{2}(A_{ij} + A_{ji})$

*Sample code:*

```
Tensor A, B;
```

```
B = Sym(A);
```

Symmetric part of a tensor.

```
AntiTensor Anti(const Tensor&);
```

*Symbolic notation:*  $\frac{1}{2}(\mathbf{A} - \mathbf{A}^T)$     *Indicial notation:*  $\frac{1}{2}(A_{ij} - A_{ji})$

*Sample code:*

```
Tensor A, B;
```

```
B = Anti(A);
```

Antisymmetric part of a tensor.

```
double Colon(const Tensor&, const Tensor&);
```

```
double Colon(const Tensor&, const SymTensor&);
```

```
double Colon(const SymTensor&, const Tensor&);
```

```
double Colon(const SymTensor&, const SymTensor&);
```

```
double Colon(const Tensor&, const AntiTensor&);
```

```
double Colon(const AntiTensor&, const Tensor&);
```

```
double Colon(const AntiTensor&, const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{A}:\mathbf{B}$     *Indicial notation:*  $A_{ij}B_{ij}$

*Sample code:*

```
Tensor A, B;
double c;
c = Colon(A, B);
```

Inner or scalar product of two tensor, also written  $\text{Tr}(\mathbf{A}^T\mathbf{B})$ . The scalar product of a symmetric and an antisymmetric tensor is always zero.

```
Tensor Deviator(const Tensor&);
SymTensor Deviator(const SymTensor&);
```

*Symbolic notation:*  $\mathbf{A} - \frac{1}{3}\text{Tr}(\mathbf{A})\mathbf{1}$     *Indicial notation:*  $A_{ij} - \frac{1}{3}A_{kk}\delta_{ij}$

*Sample code:*

```
Tensor A, B;
B = Deviator(A);
```

Deviatoric part of a tensor. The tensor  $\mathbf{1}$  is the identity tensor, which is the unique tensor that transforms any vector into itself and whose components are represented by the Kronecker delta  $\delta_{ij}$ . The deviator of an antisymmetric tensor is the tensor itself.

```
double It(const Tensor&);
double It(const SymTensor&);
double It(const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{I}_t = \text{Tr}(\mathbf{A})$     *Indicial notation:*  $A_{kk}$

*Sample code:*

```
Tensor A;
double c;
c = It(A);

double IIt(const Tensor&);
double IIt(const SymTensor&);
double IIt(const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{II}_I = \frac{1}{2}(|\mathbf{A}|^2 - (\text{Tr}\mathbf{A})^2)$     *Indicial notation:*  $\frac{1}{2}(A_{ij}A_{ij} - (A_{kk})^2)$

*Sample code:*

```
Tensor A;
double c;
c = IIIt(A);
```

```
double IIIIt(const Tensor&);
double IIIIt(const SymTensor&);
double IIIIt(const AntiTensor&);
```

*Symbolic notation:*  $\mathbf{III}_I = \text{Det}\mathbf{A}$     *Indicial notation:*  $\frac{1}{6}\varepsilon_{ijk}\varepsilon_{lmn}A_{il}A_{jm}A_{kn}$

*Sample code:*

```
Tensor A;
double c;
c = IIIIt(A);
```

Scalar invariants of a tensor. These are the coefficients appearing in the characteristic equation of a tensor. They are the only three independent scalars that can be formed in a frame-independent manner from a single tensor; all other scalars that can be formed from a tensor are functions of the scalar invariants.

The first invariant is a synonym for the trace; the third is a synonym for the determinant. Only the second invariant is nonzero for an antisymmetric tensor.

The characteristic equation itself takes the form

$$\lambda^3 - \mathbf{I}_I\lambda^2 - \mathbf{II}_I\lambda - \mathbf{III}_I = 0 \quad (29)$$

and its roots are the principal values of the tensor.

```
Tensor Eigen(const SymTensor&, Vector&);
```

This function returns the orthonormal tensor whose columns are the eigenvectors of the given symmetric matrix. The principal values are placed in the vector specified by the second argument. Thus, if

$$\mathbf{A} = \text{Eigen}(\mathbf{B}, \mathbf{e}_i) \quad (30)$$

then

$$\mathbf{D} = \mathbf{A}^T \mathbf{B} \mathbf{A} \quad (31)$$

is a diagonal tensor whose elements are given by the vector  $e_i$ .

## 2.7 Predefined Constants

```
const int DIMENSION = 3;
```

This is an integer constant giving the dimensionality of the library. It is defined to be equal to 2 if the 2-D version of the library is being used.

```
extern const Vector ZeroVector;
```

```
extern const Tensor ZeroTensor;
```

```
extern const AntiTensor ZeroAntiTensor;
```

```
extern const SymTensor ZeroSymTensor;
```

These are objects of the various classes whose components are all zero.

```
extern const Tensor IdentityTensor;
```

```
extern const SymTensor IdentitySymTensor;
```

These are objects of the given classes corresponding to the identity tensor, which is the tensor that transforms any vector into itself. The off-diagonal components are zero and the diagonal components are equal to one in any coordinate system. The identity tensor is symmetric and is given in both symmetric and full tensor representations.

(This page intentionally left blank)



### 3. Using the PHYSLIB classes

The classes defined in PHYSLIB are essentially new arithmetic types analogous to the predefined `int`, `float`, and `double` types. Their use is illustrated by the program fragment below:

```
#include "physlib.h"           // The example is 3-D

/* ... */

const Tensor One(1., 0., 0.,
                 0., 1., 0.,
                 0., 0., 1.);

Tensor GradVel;               // Velocity gradient
SymTensor Deformation, deformation, Stretch, Stress;
AntiTensor W, Omega;
Vector omega;

/* ... */

Deformation = Sym(GradVel);
W = Anti(GradVel);

/* Integrate rotation and stretch tensors */

omega = 2.*Inverse(Tr(Stretch)*One - Stretch) *
        Dual(GradVel*Stretch);
Omega = 0.5*Dual(omega);
Rotation = Inverse(One - 0.5*delT*Omega)*(One +
```

```

    0.5*delT*Omega)*Rotation;

    Stretch += Sym(delT*(GradVel*Stretch-Stretch*Omega));

/* Calculate unrotated deformation and determine rotated
   stress */

deformation = Sym(Trans(Rotation)*Deformation*
    Rotation);

Stress = Sym(Rotation *
    ComputeStress(deformation, delT) * Trans(Rotation));

```

This particular program fragment is taken from the internal forces routine in RHALE++. The velocity gradient is decomposed into its rotation and stretch rate components, the rotation and stretch are updated to the new time, and the deformation rate is rotated to the material configuration for the calculation of the new stress (which is done in the user-defined routine `SymTensor ComputeStress(SymTensor&, double)`). The new stress is then rotated back to the laboratory configuration.

### 3.1 Useless Operations

Certain operations are mathematically well-defined but useless. For example, the trace or the determinant of an antisymmetric tensor is well-defined but trivially zero. The transpose of a symmetric tensor is itself. These operations are not explicitly defined in PHYSLIB, but if the programmer were to write code such as

```

Antitensor a;

double b;

/* ... */

b = Tr(a);

```

the code would compile and run normally. The compiler recognizes that there is a standard conversion from `AntiTensor` to `Tensor`. This conversion is called for `a` and the result is passed to `Tr(Tensor)`, which returns the correct value of 0.

Obviously, programmers should avoid such useless constructs, since they needlessly consume time and memory. However, instantiation of *template functions* may require such constructs, and the standard conversions to `Tensor` will ensure that these compile and run successfully.

## References

- [1] M.A.Ellis and B.Stroustrup, *The Annotated C++ Reference Manual*, 1990. Reading, MA: Addison-Wesley Publishing Company.
- [2] L.E.Malvern, *Introduction to the Mechanics of a Continuous Medium*, 1969. Englewood Cliffs, NJ: Prentice-Hall, Inc.

(This page intentionally left blank)

## Index of Operators and Functions

### A

AntiTensor Anti(const Tensor) 55  
AntiTensor Dual(const Vector) 52  
AntiTensor operator-(const AntiTensor, const AntiTensor) 50  
AntiTensor operator-(void) 41  
AntiTensor operator\*(const AntiTensor, const double) 46  
AntiTensor operator\*(const double, const AntiTensor) 46  
AntiTensor operator+(const AntiTensor, const AntiTensor) 49  
AntiTensor operator/(const AntiTensor, const double) 46  
AntiTensor& operator\*=(const double) 46  
AntiTensor& operator+=(const AntiTensor) 49  
AntiTensor& operator/=(const double) 47  
AntiTensor& operator=(const AntiTensor&) 38  
AntiTensor& operator-=(const AntiTensor) 50  
AntiTensor(const AntiTensor&) 38  
AntiTensor(const double, const double, const double) 37  
AntiTensor(void) 37

### D

double Colon(const AntiTensor, const AntiTensor) 55  
double Colon(const SymTensor, const SymTensor) 55  
double Colon(const Tensor, const Tensor) 55  
double Det(const SymTensor) 53  
double Det(const Tensor) 53  
double IIIt(const AntiTensor&) 57  
double IIIt(const SymTensor&) 57  
double IIIt(const Tensor&) 57  
double IIIt(const AntiTensor&) 56  
double IIIt(const SymTensor&) 56  
double IIIt(const Tensor&) 56  
double It(const AntiTensor&) 56  
double It(const SymTensor&) 56  
double It(const Tensor&) 56  
double Norm(const Vector) 53  
double operator\*(const Vector, const Vector) 43  
double Tr(const SymTensor) 54  
double Tr(const Tensor) 54  
double X(void) 22  
double XX(void) 27, 33  
double XY(const double) 39  
double XY(void) 27, 34, 38  
double XZ(const double) 39  
double XZ(void) 28, 34, 38  
double Y(void) 23  
double YX(void) 28  
double YY(void) 28, 34

double YZ(const double) 39  
double YZ(void) 28, 34, 39  
double Z(void) 23  
double ZX(void) 28  
double ZY(void) 29  
double ZZ(void) 29, 35

## I

int fread(AntiTensor&, FILE\*) 40  
int fread(AntiTensor\*, int, FILE\*) 40  
int fread(SymTensor&, FILE\*) 36  
int fread(SymTensor\*, int, FILE\*) 36  
int fread(Tensor&, FILE\*) 31  
int fread(Tensor\*, int, FILE\*) 31  
int fread(Vector&, FILE\*) 24  
int freadI(Vector\*, int, FILE\*) 24  
int fwrite(const AntiTensor\*, const int, FILE\*) 40  
int fwrite(const AntiTensor, FILE\*) 40  
int fwrite(const SymTensor\*, const int, FILE\*) 36  
int fwrite(const SymTensor, FILE\*) 36  
int fwrite(const Tensor\*, const int, FILE\*) 31  
int fwrite(const Tensor, FILE\*) 31  
int fwrite(const Vector\*, const int, FILE\*) 24  
int fwrite(const Vector, FILE\*) 24  
int operator!=(const AntiTensor&, const AntiTensor&) 51  
int operator!=(const SymTensor&, const SymTensor&) 51  
int operator!=(const Tensor&, const Tensor&) 51  
int operator!=(const Vector&, const Vector&) 45  
int operator==(const AntiTensor&, const AntiTensor&) 50  
int operator==(const SymTensor&, const SymTensor&) 50  
int operator==(const Tensor&, const Tensor&) 50  
int operator==(const Vector&, const Vector&) 44  
istream& operator>>(istream&, Vector&) 45

## O

ostream& operator 45

## S

SymTensor Deviator(const SymTensor) 56  
SymTensor Inverse(const SymTensor) 54  
SymTensor operator-(const SymTensor, const SymTensor) 49  
SymTensor operator-(void) 41  
SymTensor operator\*(const double, const SymTensor) 46  
SymTensor operator\*(const SymTensor, const double) 45  
SymTensor operator+(const SymTensor, const SymTensor) 48  
SymTensor operator+=(const SymTensor) 49  
SymTensor operator/(const SymTensor, const double) 46  
SymTensor Sym(const Tensor) 55  
SymTensor& operator\*=(const double) 46  
SymTensor& operator/=(const double) 47

SymTensor& operator=(const SymTensor&) 33  
SymTensor& operator-=(const SymTensor) 50  
SymTensor(const double, const double, ... , const double) 33  
SymTensor(const SymTensor&) 33  
SymTensor(void) 32

## T

Tensor Deviator(const Tensor) 56  
Tensor Eigen(const SymTensor, Vector&) 57  
Tensor Inverse(const Tensor) 54  
Tensor operator%(const Vector, const Vector) 43  
Tensor operator-(const AntiTensor, const SymTensor) 50  
Tensor operator-(const AntiTensor, const Tensor) 49  
Tensor operator-(const SymTensor, const AntiTensor) 50  
Tensor operator-(const SymTensor, const Tensor) 49  
Tensor operator-(const Tensor, const AntiTensor) 49  
Tensor operator-(const Tensor, const SymTensor) 49  
Tensor operator-(const Tensor, const Tensor) 49  
Tensor operator-(void) 41  
Tensor operator\*(const AntiTensor, const SymTensor) 48  
Tensor operator\*(const AntiTensor, const Tensor) 48  
Tensor operator\*(const double, const Tensor) 46  
Tensor operator\*(const SymTensor, const AntiTensor) 48  
Tensor operator\*(const SymTensor, const SymTensor) 48  
Tensor operator\*(const SymTensor, const Tensor) 48  
Tensor operator\*(const Tensor, const AntiTensor) 48  
Tensor operator\*(const Tensor, const double) 45  
Tensor operator\*(const Tensor, const SymTensor) 48  
Tensor operator\*(const Tensor, const Tensor) 48  
Tensor operator+(const AntiTensor, const SymTensor) 49  
Tensor operator+(const AntiTensor, const Tensor) 48  
Tensor operator+(const SymTensor, const AntiTensor) 49  
Tensor operator+(const SymTensor, const Tensor) 48  
Tensor operator+(const Tensor, const AntiTensor) 49  
Tensor operator+(const Tensor, const SymTensor) 48  
Tensor operator+(const Tensor, const Tensor) 48  
Tensor operator/(const Tensor, const double) 46  
Tensor operator/=(const double) 47  
Tensor Trans(const Tensor) 54  
Tensor& operator\*=(const double) 46  
Tensor& operator+=(const AntiTensor) 49  
Tensor& operator+=(const SymTensor) 49  
Tensor& operator+=(const Tensor) 49  
Tensor& operator-=(const AntiTensor) 50  
Tensor& operator-=(const AntiTensor) 27  
Tensor& operator-=(const SymTensor) 50  
Tensor& operator-=(const SymTensor) 27  
Tensor& operator-=(const Tensor&) 26  
Tensor& operator-=(const Tensor) 50  
Tensor(const AntiTensor) 26

Tensor(const double, const double, ..., const double) 25  
Tensor(const SymTensor) 26  
Tensor(const Tensor&) 26  
Tensor(void) 25

## **V**

Vector Cross(const Vector, const Vector) 52  
Vector Dual(const Tensor) 52  
Vector operator-(const Vector, const Vector) 44  
Vector operator-(void) 41  
Vector operator\*(const AntiTensor, const Vector) 47  
Vector operator\*(const double, const Vector) 41  
Vector operator\*(const SymTensor, const Vector) 47  
Vector operator\*(const Tensor, const Vector) 47  
Vector operator\*(const Vector, const AntiTensor) 48  
Vector operator\*(const Vector, const double) 41  
Vector operator\*(const Vector, const SymTensor) 48  
Vector operator\*(const Vector, const Tensor) 47  
Vector operator+(const Vector, const Vector) 43  
Vector operator/(const Vector, const double) 42  
Vector& operator\*=(const double) 42  
Vector& operator+=(const Vector) 44  
Vector& operator/=(const double) 42  
Vector& operator=(const Vector&) 22  
Vector& operator-=(const Vector) 44  
Vector(const double, const double, const double) 22  
Vector(const Vector&) 22  
Vector(void) 21  
void XX(const double) 29, 35  
void XY(const double) 29, 35  
void XZ(const double) 30, 35  
void YX(const double) 30  
void YY(const double) 30, 35  
void YZ(const double) 30, 36  
void Z(const double) 23  
void ZX(const double) 30  
void ZY(const double) 31  
void ZZ(const double) 31, 36

## **X**

X(const double) 23

## **Y**

Y(const double) 23

## **Z**

Z(void) 22