PHYSLIB:
A C++ Tensor Class Library

Kent G. Budge

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550
for the United States Department of Energy
under Contract DE-AC04-76DP00789
PHYSLIB:
A C++ Tensor Class Library

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 Object-Oriented Programming and the C++ Language ............................................... 17
       1.2.1 Data Abstraction ............................................................................................... 18
       1.2.2 Special Member Functions and Dynamic Memory Management .................. 18
       1.2.3 Function and Operator Overloading .................................................................. 19
2. The PHYSLIB Library ......................................................................................................... 21
   2.1 class Vector ................................................................................................................. 21
       2.1.1 Private Data Members ....................................................................................... 21
       2.1.2 Special Member Functions ............................................................................... 22
       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
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 may prove to be relatively easy to implement in C++. For these reasons, Division 1541 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, the inline code file physlib.inl, and the source file physlib.c. The library is applicable to both three-dimensional and two-dimensional problems; the user selects the 2-D version of the library by defining the symbol TWO_D in the header file physlib.h and recompiling physlib.c and his own code. Alternately, system managers may wish to provide duplicate header and object modules of each dimensionality.

This code was produced under the auspices of Sandia National Laboratories, a federally-funded research center administered for the United States Department of Energy on a non-profit basis by AT&T. This code is available to U.S. citizens and institutions under research, government use and/or commercial license agreements.

Federal agencies, universities, and other U.S. institutions who wish to support further development of this code and its sister codes are encouraged to contact Division 1541, Sandia National Laboratories. Division 1541 welcomes collaborative efforts with qualified research institutions.

The PHYSLIB library is © 1991 Sandia Corporation.
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
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.

Division 1541 at Sandia National Laboratories recently began work on a new computer code, RHALE++, which calculates the behavior of materials subjected to strong shock waves. The equations describing the physics of strong shocks are vector and tensor equations. In the past, great effort has been required to correctly translate these equations into computer code.

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 a new and very promising computer language, C++, 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 \( u \).

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,

\[
\text{if } \vec{b} = c\vec{a} \text{ then } |\vec{b}| = |c||\vec{a}| \quad \text{and} \quad \vec{b} = \pm\hat{a}
\]

(1)

If \( c < 0 \), the resulting vector has the opposite direction from the original vector.
Introduction

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

\[ \mathbf{a} \cdot \mathbf{b} \]

(2)

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

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta_{ab} \]

(3)

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

\[ \mathbf{a} \cdot (\mathbf{b} + \mathbf{c}) = \mathbf{a} \cdot \mathbf{b} + \mathbf{a} \cdot \mathbf{c} \] (Distributive law)  

(4)

\[ \mathbf{a} \cdot \mathbf{b} = \mathbf{b} \cdot \mathbf{a} \] (Commutative law)  

(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

\[ \mathbf{a} = \mathbf{A} \mathbf{b} \]

(6)
Introduction

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

$$A (\hat{a} + \hat{b}) = A \hat{a} + A \hat{b}$$

(7)

and

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

(8)

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

$$\hat{c} = \hat{b} A$$

(9)

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

$$A \hat{a} = \hat{a} B$$

(10)

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

$$A = B^T.$$  

(11)

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

$$A = B + C \text{ iff } A \hat{a} = B \hat{a} + C \hat{a} \text{ for all } \hat{a}$$

(12)

The product of two tensors is defined such that

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

(13)

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

$$A = \hat{a} \otimes \hat{b}$$

(14)

It is defined by

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

(15)

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

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

(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

\[
\vec{F} = \mathbf{P}\hat{s}
\]

(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

\[
\vec{u}' = \mathbf{J}\vec{u}
\]

(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,

\[
\mathbf{A}^T = \mathbf{A}
\]

(19)

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

\[
\mathbf{A}^T = -\mathbf{A}.
\]

(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

\[
\text{Sym}(\mathbf{A}) = \frac{1}{2} (\mathbf{A} + \mathbf{A}^T)
\]

(21)

\[
\text{Anti}(\mathbf{A}) = \frac{1}{2} (\mathbf{A} - \mathbf{A}^T)
\]

(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 mutual-
Introduction

ly 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

\[
\mathbf{a} = a_1 \hat{x} + a_2 \hat{y} + a_3 \hat{z}
\]  

(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

\[
\mathbf{A} = A_{11} (\hat{x} \otimes \hat{x}) + A_{12} (\hat{x} \otimes \hat{y}) + A_{13} (\hat{x} \otimes \hat{z})
\]

\[
+ A_{21} (\hat{y} \otimes \hat{x}) + A_{22} (\hat{y} \otimes \hat{y}) + A_{23} (\hat{y} \otimes \hat{z})
\]

\[
+ A_{31} (\hat{z} \otimes \hat{x}) + A_{32} (\hat{z} \otimes \hat{y}) + A_{33} (\hat{z} \otimes \hat{z})
\]

(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

\[
r = \mathbf{a} \cdot \mathbf{b}.
\]

(25)

It can be written in indicial notation as

\[
r = \sum_{i=1}^{3} a_i b_i.
\]

(26)

where \( a_i \) and \( b_i \) are the components of the vectors \( \mathbf{a} \) and \( \mathbf{b} \). 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
Introduction

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, and the foregoing discussion reflects this fact. However, there are many physical situations where a high degree of spatial symmetry permits a simplified treatment of vector and tensor calculations. RHALE++ therefore has been written in 2-D and 3-D versions. In the 2-D version, one assumes either plane symmetry or axisymmetry.

Plane symmetry represents the case in which there is perfect translational and reflective symmetry along the $z$ direction. Axisymmetry is the case in which rotational and reflective symmetry exists around an axis in the $z$ direction. In either case, certain components of tensors are guaranteed to be zero in the calculations performed by RHALE++ and similar programs.

To take advantage of this, the PHYSLIB library can be set up for either normal 3-D calculations or 2-D calculations. To set up PHYSLIB for 2-D calculations, one defines the macro TWO_D at the start of the file physlib.h; to set up for 3-D calculations, this macro is left undefined.

The library code contains compiler directives that test this macros and compiles different portions of the code depending on whether the macro is defined. Thus, when a 2-D program is being compiled, the tensor components that are guaranteed to be zero can be omitted, saving memory and computation time.

In addition, an integer constant, DIMENSION, is set to the number of dimensions (2 or 3).
1.2 Object-Oriented Programming and the C++ 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. To alleviate this problem, one has to rely on a coherent, well-organized programming style. Programming style includes techniques that do not change the basic calculations performed by a program and which might not even alter the machine language translation.

The most obvious element of style is the incorporation of comments and indentation. Comments are sections of text that the compiler is instructed to ignore, but which convey clarifications and explanations to a human reader. Good programmers make extensive use of commenting, especially in older languages; it is not uncommon for a well-written FORTRAN program to consist of 50% comment lines. Indention is the intelligent use of white space (blanks, tabs, and empty lines), which are ignored by the compiler, to indicate program structure. It is also an important feature of good FORTRAN coding, where indentation helps delineate the structure of DO loops and IF-THEN constructs.

Unfortunately, commenting and indenting alone are not sufficient to render a code transparent to the human reader. Modern programming languages therefore include grammar that facilitates block-structured programming. Block-structured programs are broken down into logical units, each of which is relatively easy to understand. For example, iterative loops are written nowadays using a specific grammar that indicates that the loop is a logical unit. GOTO statements are generally avoided, since they tend to blur the boundaries of logical units. An important part of block-structured programming is the care with which the programmer breaks the code down into relatively small subroutines, each of which is easy to understand, and builds a tree of subroutine calls to implement his algorithm.

Block-structured programs may be written either in a top-down or a bottom-up fashion. In top-down programming, one writes a program at the top level first, using calls to as-yet nonexistent subroutines to represent major parts of the calculation; the first level of subroutines is then written the same way, writing each subroutine as a sequence of calls to lower-level subroutines, and so on. In bottom-up programming, one builds the lowest-level subroutines first, then combines these into somewhat higher-level subroutines, and so on. Both approaches have their merits.

The most recent trend in programming style is towards object-oriented programming. Conventional computers are sequential; a single processor steps through a program, carrying out one task at a time. Programs written in traditional programming languages therefore support the model of a program as a sequence of tasks. This is known as procedural programming, because a sequence of procedures is being carried out.
Introduction

Modern supercomputers are not purely sequential. In particular, vector processors such as
Cray or Convex supercomputers process entire blocks of data in an assembly-line fashion.
Massively parallel computers such as MIMD machines have many processors which can
operate independently. For such computers, the sequential model is not ideal. Instead, one
uses an object-oriented approach in which the program is thought of as a set of interacting
data objects. This approach has proven to be fruitful even on traditional sequential com-
puters. It seems to mesh well with the concept of block-structure programming; not only is
code divided into logical units, but so is data. Closely related to the concept of object-ori-
ented programming is the concept of data abstraction. This is the notion that a data struc-
ture should be treated as a coherent unit wherever possible, with only a few routines
accessing its individual components.

C++ is the first efficient 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 [1]. 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 di-
rectly 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 mem-
ory is freed. This makes it possible to carry out sophisticated dynamic memory manage-
ment 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:

```c
#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

```c
#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 three files: a header file, physlib.h; an inline function file, physlib.inl; 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 header file in turn includes the inline function file, which contains additional C++ code to define the various operator overloads and methods that are defined for the PHYSLIB 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: \( \mathbf{a} \)
Indicial Notation: \( a_i \)

2.1.1 Private Data Members

double \( x \); \hspace{1cm} \text{X component of vector (} a_i \text{)}

double \( y \); \hspace{1cm} \text{Y component of vector (} a_i \text{)}

double \( z \); \hspace{1cm} \text{Z component of vector (} a_i \text{)}

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 en-
hance 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:
```c
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:
```c
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:
```c
Vector a;
printf("The Z component of a is %f\n", a.Z());
```

void X(const double);

Symbolic notation: None  
Indicial notation: \( a_1 \leftarrow s \)

Sample code:
```c
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:
```c
Vector a;
```
The PHYSLIB Library

```c
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

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

Sample code:

```c
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: $A$

Indicial notation: $A_{ij}$

2.2.1 Private Data Members

double xx; \hspace{1cm} \text{xx component of tensor} (A_{11})
double xy; \hspace{1cm} \text{xy component of tensor} (A_{12})
double xz; \hspace{1cm} \text{xz component of tensor} (A_{13})
double yx; \hspace{1cm} \text{yx component of tensor} (A_{21})
double yy; \hspace{1cm} \text{yy component of tensor} (A_{22})
double yz; \hspace{1cm} \text{yz component of tensor} (A_{23})
double zx; \hspace{1cm} \text{zx component of tensor} (A_{31})
double zy; \hspace{1cm} \text{zy component of tensor} (A_{32})
double zz; \hspace{1cm} \text{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, const double);  

Sample code:
The PHYSLIB Library

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:*

```cpp
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:*

```cpp
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:*

```cpp
symtensor a;
anti tensor 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(AntiTens or) 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: \( \mathbf{i} \mathbf{A} \mathbf{i} \) Indicial notation: \( A_{ii} \)

Sample code:

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

double XY(void) const;
Symbolic notation: \( \mathbf{i} \mathbf{A} \mathbf{y} \) Indicial notation: \( A_{12} \)

Sample code:

Tensor A;
The PHYSLIB Library

printf("The XY component of A is \$f", A.XY());

double XZ(void) const;

Symbolic notation: A_{13} Indicial notation: A_{13}

Sample code:

Tensor A;
printf("The XZ component of A is \$f", A.XZ());

double YX(void) const;

Symbolic notation: A_{21} Indicial notation: A_{21}

Sample code:

Tensor A;
printf("The YX component of A is \$f", A.YX());

double YY(void) const;

Symbolic notation: A_{22} Indicial notation: A_{22}

Sample code:

Tensor A;
printf("The YY component of A is \$f", A.YY());

double YZ(void) const;

Symbolic notation: A_{23} Indicial notation: A_{23}

Sample code:

Tensor A;
printf("The YZ component of A is \$f", A.YZ());

double ZX(void) const;
Symbolic notation: $A_{31}$  
Indicial notation: $A_{31}$

Sample code:

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

double ZY(void) const;

Symbolic notation: $A_{32}$  
Indicial notation: $A_{32}$

Sample code:

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

double ZZ(void) const;

Symbolic notation: $A_{33}$  
Indicial notation: $A_{33}$

Sample code:

```c
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:

```c
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:

```c
Tensor A;
```
The PHYSLIB Library

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

void XZ(const double);
Symbolic notation: None Indicial notation: \( A_{13} \rightarrow s \)
Sample code:
Tensor A;
A.XZ(3.); // Set XZ component of A to 3.

void YX(const double);
Symbolic notation: None Indicial notation: \( A_{21} \rightarrow s \)
Sample code:
Tensor A;
A.YX(3.); // Set YX component of A to 3.

void YY(const double);
Symbolic notation: None Indicial notation: \( A_{22} \rightarrow s \)
Sample code:
Tensor A;
A.YY(3.); // Set YY component of A to 3.

void YZ(const double);
Symbolic notation: None Indicial notation: \( A_{23} \rightarrow s \)
Sample code:
Tensor A;
A.YZ(3.); // Set YZ component of A to 3.

void ZX(const double);
Symbolic notation: None Indicial notation: \( A_{31} \rightarrow s \)
Sample code:

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

void ZY(const double);

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

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);
The PHYSLIB Library

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

2.3.1 Private Data Members

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

```cpp
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:*

```cpp
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:*

```cpp
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:* \( \hat{A} \hat{x} \)  
*Indicial notation:* \( A_{11} \)

*Sample code:*

```cpp
33
```
SymTensor A;
printf("The XX component of A is %f", A.XX());

double XY(void) const;

Symbolic notation: $A_y$
Indicial notation: $A_{12}$

Sample code:

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

double XZ(void) const;

Symbolic notation: $A_z$
Indicial notation: $A_{13}$

Sample code:

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

double YY(void) const;

Symbolic notation: $A_y$
Indicial notation: $A_{22}$

Sample code:

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

double YZ(void) const;

Symbolic notation: $A_z$
Indicial notation: $A_{23}$

Sample code:

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

double ZZ(void) const;
Symbolic notation: \( ?^A ? \)  
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: \( A_{11} \leftrightarrow s \)

Sample code:

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

void XY(const double);
Symbolic notation: None  
Indicial notation: \( A_{12} \leftrightarrow s \)

Sample code:

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

void XZ(const double);
Symbolic notation: None  
Indicial notation: \( A_{13} \leftrightarrow s \)

Sample code:

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

void YY(const double);
Symbolic notation: None  
Indicial notation: \( A_{22} \leftrightarrow s \)

Sample code:
The PHYSLIB Library

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 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; \hspace{1cm} \text{xy component of the tensor} \( (A_{12} = -A_{21}) \)
double xz; \hspace{1cm} \text{xz component of the tensor} \( (A_{13} = -A_{31}) \)
double yz; \hspace{1cm} \text{yz component of the tensor} \( (A_{23} = -A_{32}) \)

2.4.2 Special Member Functions

AntiTensor(void);

Sample code:

\[
\text{AntiTensor } A; \hspace{1cm} \text{// Construct an uninitialized} \\
\text{// AntiTensor}
\]

Default constructor for instances of the class AntiTensor.

AntiTensor(const double, 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: \( A_{12} \)  
Indicial notation: \( A_{12} \)

Sample code:

AntiTensor A;

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

double XZ(void) const;

Symbolic notation: \( A_{13} \)  
Indicial notation: \( A_{13} \)

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

double YZ(void) const;
Symbolic notation: \( A_{ij} \)  \quad 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 \quad Indicial notation: \( A_{12} \leftarrow s \)
Sample code:
    AntiTensor A;
    A.XY(3.); \quad // Set XY component of A to 3.

void XZ(const double);
Symbolic notation: None \quad Indicial notation: \( A_{13} \leftarrow s \)
Sample code:
    SymTensor A;
    A.XZ(3.); \quad // Set XZ component of A to 3.

void YZ(const double);
Symbolic notation: None \quad Indicial notation: \( A_{23} \leftarrow s \)
Sample code:
    AntiTensor A;
    A.YZ(3.); \quad // Set YZ component of A to 3.
The PHYSLIB Library

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

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

**Sample code:**

```c
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: \(-\mathbf{a}\) *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: \(\mathbf{a}c\) *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 \(a \ast b \ast c\) means \((a \ast b) \ast c\) or \((\hat{a} \cdot \hat{b}) \cdot \hat{c}\)).

\[
\text{Vector\& operator}\ast\!(\text{const double});
\]
\text{Symbolic notation: } \hat{a} \leftarrow \hat{a}c \quad \text{Indicial notation: } a_i \leftarrow a_ic

\textbf{Sample code:}

```cpp
Vector a;
double c;
a = c;
```

Replace a vector by its product with a scalar.

\[
\text{Vector operator} / (\text{const Vector, const double});
\]
\text{Symbolic notation: } \hat{a} / c \quad \text{Indicial notation: } a_i / c

\textbf{Sample code:}

```cpp
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.

\[
\text{Vector\& operator} /= (\text{const double});
\]
\text{Symbolic notation: } \hat{a} \leftarrow \hat{a} / c \quad \text{Indicial notation: } a_i \leftarrow a_i / c

\textbf{Sample code:}

```cpp
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: \( \mathbf{a} \cdot \mathbf{b} \)  
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: \( \mathbf{a} \otimes \mathbf{b} \)  
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: \( \mathbf{a} + \mathbf{b} \)  
Indicial notation: \( a_i + b_i \)

Sample code:

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

Return the sum of two vectors.
The PHYSLIB Library

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.

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:

```c
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:

```c
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:

```c
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.
The PHYSLIB Library

Tensor operator/=(const double);
SymTensor& operator/=(const double);
AntiTensor& operator/=(const double);

*Symbolic notation: A ← A / c  Indicial notation: A_{ij} ← 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: A 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: a B  Indicial notation: a_{ji} B_j

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: \( A \times B \)  \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);
The PHYSLIB Library

Symbolic notation: $A + B$    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: $A \leftarrow A + B$    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: $A - B$    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: \( A \leftarrow A - B \)  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.
2.6 Methods

Vector Cross(const Vector, const Vector);

*Symbolic notation:* \( \mathbf{a} \times \mathbf{b} \)  
*Indicial notation:* \( \epsilon_{ijk}a_ib_k \)

*Sample code:*

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

Vector or cross product of two vectors. The symbol \( \epsilon_{ijk} \) 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, \( \epsilon_{122} = 0; \epsilon_{123} = 1; \) and \( \epsilon_{213} = -1 \). The cross product is distributive and associative but not commutative.

Vector Dual(const Tensor);

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

*Sample code:*

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

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

AntiTensor Dual(const Vector);

*Symbolic notation:* \( \text{Dual}(\mathbf{a}) \)  
*Indicial notation:* \( \epsilon_{ijk}\mathbf{a}_k \)

*Sample code:*

```c
Vector a;
AntiTensor B;
B = Dual(a);
```
Dual of a vector. It can be proved that \( \text{Dual} (\text{Dual}(\vec{a})) = 2\vec{a} \). The concept of the dual is closely related to the cross product, since \( \vec{b} \text{Dual}(\vec{a}) = \vec{a} \times \vec{b} \).

\[
\text{double Norm(const Vector);}
\]

*Symbolic notation: |a|*  
*Indicial notation: \(\sqrt{a_i a_i}\)*

*Sample code:*

```cpp
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.

\[
\text{double Norm(const Tensor);} \\
\text{double Norm(const SymTensor);} \\
\text{double Norm(const AntiTensor);}
\]

*Symbolic notation: |A|*  
*Indicial notation: \(\sqrt{A_{ij} A_{ij}}\)*

*Sample code:*

```cpp
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.

\[
\text{double Det(const Tensor);} \\
\text{double Det(const SymTensor);} \\
\]

*Symbolic notation: \(\det [A]\)*  
*Indicial notation: \(\frac{1}{6} \epsilon_{ijk} \epsilon_{lmn} A_{ij} A_{jm} A_{kn}\)*

*Sample code:*

```cpp
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: 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: TrA 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: 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 ten-
sor.

SymTensor Sym(const Tensor);

Symbolic notation: \( \frac{1}{2}(A + A^T) \)  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: \( \frac{1}{2}(A - A^T) \)  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: \( A:B \)  Indicial notation: \( A_{ij}B_{ij} \)

Sample code:
The PHYSLIB Library

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

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

Tensor Deviator(const Tensor);
SymTensor Deviator(const SymTensor);

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

Sample code:

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

Deviatoric part of a tensor. The tensor $I$ 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 $I$(const Tensor&);
double $I$(const SymTensor&);
double $I$(const AntiTensor&);

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

Sample code:

Tensor A;
double c;
c = $I$(A);
double $II$ (const Tensor&);
double $II$ (const SymTensor&);
double $II$ (const AntiTensor&);
The PHYSLIB Library

Symbolic notation:  \( \mathbf{\Pi} = \frac{1}{2} (|A|^2 - (\text{Tr} A)^2) \)

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:  \( \mathbf{\Pi} = \det A \)

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

\[ \lambda^3 - \mathbf{\Pi} \lambda^2 - \mathbf{\Pi} \lambda - \mathbf{\Pi} = 0 \]  \hspace{1cm} (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

\[ A = \text{Eigen}(B, c) \]  \hspace{1cm} (30)
The PHYSLIB Library

then

\[ D = A^T B A \]  

is a diagonal tensor whose elements are given by the vector \( e_i \).

2.7 Predefined Constants

```cpp
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.
The PHYSLIB Library
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:

```c
#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 +
```

59
Using the PHYSLIB classes

\[ 0.5 \cdot \text{delT} \cdot \Omega \cdot \text{Rotation}; \]
\[ \text{Stress} += \text{Sym}(\text{delT} \cdot (\text{GradVel} \cdot \text{Stretch} - \text{Stretch} \cdot \Omega)); \]

/* Calculate unrotated deformation and determine rotated stress */

defformation = \text{Sym}(\text{Trans}(\text{Rotation}) \cdot \text{Deformation} \cdot \text{Rotation});

\text{Stress} = \text{Sym}(\text{Rotation} \cdot \text{ComputeStress}(\text{deformation}, \text{delT}) \cdot \text{Trans}(\text{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

\begin{verbatim}
Antitensor a;
double b;
/* ... */
b = Tr(a);
\end{verbatim}

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. Some users may wish to comment out the standard conversions responsible for permitting useless code.
Conclusion

PHYSLIB defines vector and tensor classes that are fundamental to the RHALE++ programming effort, but which are general and should be useful in many scientific applications.

These classes are fundamental components of field classes that represent vector and tensor fields of various types relevant to finite element calculations. These are essentially smart arrays of vectors or tensors with corresponding operations and methods. The arrays are defined on a domain represented by a mesh class. Calculus operations such as divergence or gradient are defined in these libraries.

These field classes which utilize the PHYSLIB classes are the subject of a future document.
References


## Index of Operators and Functions

### A
- AntiTensor Anti(const Tensor) 53
- AntiTensor Dual(const Vector) 50
- AntiTensor operator-(const AntiTensor, const AntiTensor) 48
- AntiTensor operator-(void) 41
- AntiTensor operator*(const AntiTensor, const double) 44
- AntiTensor operator*(const double, const AntiTensor) 44
- AntiTensor operator+(const AntiTensor, const AntiTensor) 47
- AntiTensor operator/(const AntiTensor, const double) 45
- AntiTensor& operator*=(const double) 45
- AntiTensor& operator+=(const AntiTensor) 48
- AntiTensor& operator/=(const double) 46
- AntiTensor& operator+=(const AntiTensor&) 38
- AntiTensor& operator/=(const AntiTensor&) 49
- AntiTensor(const AntiTensor&) 38
- AntiTensor(const double, const double, const double) 37
- AntiTensor(void) 37

### D
- double Colon(const AntiTensor, const AntiTensor) 53
- double Colon(const SymTensor, const SymTensor) 53
- double Colon(const Tensor, const Tensor) 53
- double Det(const SymTensor) 51
- double Det(const Tensor) 51
- double Illt(const AntiTensor&) 55
- double Illt(const SymTensor&) 55
- double Illt(const Tensor&) 55
- double It(const AntiTensor&) 54
- double It(const SymTensor&) 54
- double It(const Tensor&) 54
- double It(const AntiTensor&) 54
- double It(const SymTensor&) 54
- double It(const Tensor&) 54
- double Norm(const Vector) 51
- double operator*(const Vector, const Vector) 43
- double Tr(const SymTensor) 52
- double Tr(const Tensor) 52
- 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 fread(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

S
SymTensor Deviator(const SymTensor) 54
SymTensor Inverse(const SymTensor) 52
SymTensor operator-(const SymTensor, const SymTensor) 48
SymTensor operator-(void) 41
SymTensor operator-=(const double, const SymTensor) 44
SymTensor operator-=(const SymTensor, const SymTensor) 44
SymTensor operator+ (const SymTensor, const SymTensor) 47
SymTensor operator+=(const SymTensor) 48
SymTensor Sym(const Tensor) 53
SymTensor& operator+=(const double) 45
SymTensor& operator+=(const SymTensor&) 33
SymTensor& operator-=(const double) 46
SymTensor& operator-=(const SymTensor&) 33
SymTensor& operator-=(const SymTensor) 49
SymTensor(const double, const double, ..., const double) 33
SymTensor(const SymTensor&) 33
SymTensor(void) 32

T
Tensor Deviator(const Tensor) 54
Tensor Eigen(const SymTensor, Vector&) 55
Tensor Inverse(const Tensor) 52
Tensor operator%(const Vector, const Vector) 43
Tensor operator-(const AntiTensor, const SymTensor) 48
Tensor operator-(const AntiTensor, const Tensor) 48
Tensor operator-(const SymTensor, const AntiTensor) 48
Tensor operator-(const SymTensor, const Tensor) 48
Tensor operator-(const Tensor, const AntiTensor) 48
Tensor operator-(const Tensor, const SymTensor) 48
Tensor operator-(const Tensor, const Tensor) 48
Tensor operator-(void) 41
Tensor operator*(const AntiTensor, const SymTensor) 47
Tensor operator*(const AntiTensor, const Tensor) 47
Tensor operator*(const double, const Tensor) 44
Tensor operator*(const SymTensor, const AntiTensor) 47
Tensor operator*(const SymTensor, const SymTensor) 47
Tensor operator*(const SymTensor, const Tensor) 47
Tensor operator*(const Tensor, const AntiTensor) 47
Tensor operator*(const Tensor, const double) 44
Tensor operator*(const Tensor, const SymTensor) 47
Tensor operator*(const Tensor, const Tensor) 47
Tensor operator+(const AntiTensor, const SymTensor) 47
Tensor operator+(const AntiTensor, const Tensor) 47
Tensor operator+(const SymTensor, const AntiTensor) 47
Tensor operator+(const SymTensor, const Tensor) 47
Tensor operator+(const Tensor, const AntiTensor) 47
Tensor operator+(const Tensor, const SymTensor) 47
Tensor operator+(const Tensor, const Tensor) 47
Tensor operator/=(const double) 46
Tensor Trans(const Tensor) 52
Tensor& operator*=(const double) 45
Tensor& operator+=(const AntiTensor) 48
Tensor& operator+=(const SymTensor) 48
Tensor& operator+=(const Tensor) 48
Tensor& operator+=(const Tensor) 49
Tensor& operator*=(const AntiTensor) 27
Tensor& operator*=(const SymTensor) 49
Tensor& operator*=(const SymTensor) 27
Tensor& operator*=(const Tensor&) 26
Tensor& operator*=(const Tensor) 49
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) 50
Vector Dual(const Tensor) 50
Vector operator-(const Vector, const Vector) 44
Vector operator-(void) 41
Vector operator*(const AntiTensor, const Vector) 46
Vector operator*(const double, const Vector) 41
Vector operator*(const SymTensor, const Vector) 46
Vector operator*(const Tensor, const Vector) 46
Vector operator*(const Vector, const AntiTensor) 46
Vector operator*(const Vector, const double) 41
Vector operator*(const Vector, const SymTensor) 46
Vector operator*(const Vector, const Tensor) 46
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) 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) 22
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) 24
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
Distribution

1. External Distribution

R. W. Alcwinc
DARPA/RMO
1400 Wilson Blvd.
Arlington, VA 22209

R. T. Allen
Pacific Technology
P.O. Box 148
Del Mar, CA 92014

Alliant Techsystems Inc. (2)
Attn: G.R. Johnson
O. Souka
7225 Northland Dr.
Brooklyn Park, MN 55428

M. Alme
Logicor RDA
2100 Washington Blvd.
Arlington, VA 22204-5706

Anatech International Corporation (2)
Attn: R.S. Dunham
R. E. Nickell
Joe Rashid
3344 N. Torrey Pines Ct.
Suite 320
La Jolla, CA 92037

James C. Almond
Director
Center for High Performance Computing
Bancoes Research Center
10100 N. Burnet Road
Austin, TX 78758-4497

Dan Anderson
Ford Motor Co.
Suite 1100
Village Place
22400 Michigan Ave.
Dearborn, MI 48124

Andy Arenth
National Security Agency
Savage Road
Ft. Meade, MD
Attn: C6

Ali S. Argon
Department of Mechanical Engineering
Room 1-304
Massachusetts Institute of Technology
Cambridge, MA 02139

S. Atluri
Center for the Advancement of Computational Mechanics
School of Civil Engineering
Georgia Institute of Technology
Atlanta, GA 30332

D. M. Austin
University of Minnesota
1300 S. Second St.
Minneapolis, MN 55415

William E. Bachrach
Aerojet Research Propulsion Institute
P.O. Box 13502
Sacramento, CA 95853-4502

F. R. Bailey
MS200-4
Director, Astrophysics
NASA Ames Research Center
Moffett field, CA 94035

R. E. Bank
Department of Mathematics
University of California at San Diego
La Jolla, CA 92039
Richard E. Danell  
Research Officer  
Central Research Laboratories  
BHP Research & New Technology  
P.O. Box 188  
Wallsend NSW 2287  
Australia

L. Davis  
Executive Vice President  
Cray Research Inc.  
1168 Industrial Blvd.  
Chippawa Falls, WI 54729

DARPA/DSO (2)  
Attn: Lt. Col. Joseph Beno  
T. Kiehne  
J. Richardson  
1500 Wilson Boulevard  
Arlington, VA 22209-2308

Mr. Frank R. Deis  
Martin Marietta  
Falcon AFB, CO 80912-5000

R. A. DeMillo  
Director, Comp. & Comput. Resch.  
Rm. 304  
National Science Foundation  
Washington, DC 20550

L. Deng  
Applied Mathematics Department  
SUNY at Stony Brook  
Stony Brook, NY 11794-3600

A. Trent DePersia  
Program Manager  
DARPA/ASTO  
1400 Wilson Blvd.  
Arlington, VA 22209-2308

Ranji Digumarthi  
Org. 8111, Bldg. 157  
Lockheed MSD  
P.O. Box 3504  
Sunnyvale, CA 94088-3504

J. Donald Dixon  
Spokane Research Center  
U.S. Bureau of Mines  
315 Montgomery Avenue  
Spokane, WA 99207

J. J. Dongarra  
Computer Science Department  
104 Ayres Hall  
University of Tennessee  
Knoxville, TN 37996-1301

L. Dowdy  
Computer Science Department  
Vanderbilt University  
Nashville, TN 37235

I. S. Duff  
CSS Division  
Harwell Laboratory  
Oxfordshire, OX11 ORA  
United Kingdom

S. C. Eisenstat  
Computer Science Department  
Yale University  
P.O. Box 2158  
New Haven, CT 06520

H. Elman  
Computer Science Department  
University of Maryland  
College Park, MD 20842

Julius W. Enig  
Enig Associates, Inc.  
11120 New Hampshire Ave.  
Suite 500  
Silver Spring, MD 20904-2633

J. N. Entzminger  
DARPA/ITTO  
1400 Wilson Blvd.  
Arlington, VA 22209

A. M. Erisman  
MS 7L-21  
Boeing Computer Services  
P.O. Box 24346  
Seattle, WA 98124-0346
James W. Jones  
Swanson Service Corporation  
18700 Beach Blvd.  
Suite 200-210  
Huntington Beach, CA 92648

T. H. Jordan  
Department of Earth, Atmospheric and Planetary Sciences  
MIT  
Cambridge, MA 02139

M.H. Kalos, Director  
Cornell Theory Center  
514A Engineering and Theory Center  
Hoy Road, Cornell University  
Ithaca, NY 14853

Kaman Sciences Corporation (2)  
Attn: S. Diehl  
V. Smith  
1500 Garden of the Gods Road  
Colorado Springs, CO 80933

H. G. Kaper  
Mathematics and Computer Science Division  
Argonne National Laboratory  
Argonne, IL 60439

S. Karin, Director  
Supercomputing Department  
9500 Gilman Drive  
University of California at San Diego  
La Jolla, CA 92093

Dr. A.H. Kazi, Director  
Nuclear Effects Directorate  
U.S. Army Combat Systems Test Activity  
Aberdeen Proving Ground, MD 21005-5059

H. B. Keller  
217-50  
Applied Mathematics Department  
Caltech  
Pasadena, CA 91125

M. J. Kelley  
DARPA/DMO  
1400 Wilson Blvd.  
Arlington, VA 22209

K. W. Kennedy  
Computer Science Department  
Rice University  
P.O. Box 1892  
Houston, TX 77251

Gary Keiner  
Research Engineer  
Applied Mechanics and Structures  
Battelle Pacific Northwest Laboratories  
P.O. Box 999  
Richland, WA 99352

Dr. Aram K. Kevorkian  
Code 7304  
Naval Ocean Systems Center  
271 Catalina Blvd.  
San Diego, CA 92152-5000

Sam. Key  
MacNeal-Schwendler  
815 Colorado Blvd.  
Los Angeles, CA 90041

D. R. Kinkaid  
Center for Numerical Analysis  
RLM 13-150  
University of Texas at Austin  
Austin, TX 78712

T. A. Kitchens  
Office of Energy Research  
U.S. Department of Energy  
Washington, DC 20554

Max Kooonz  
DOE/OAC/DP 5.1  
Forrestal Building  
1000 Independence Ave.  
Washington, DC 20585

Dr. Peter L. Knepell  
NTBIC/GEODYNAMICS  
MS N-8930  
Falcon AFB, CO 80912-5000

Raymond D. Krieg  
Engineering Science and Mechanics  
301 Perkins Hall  
University of Tennessee  
Knoxville, TN 37996
V. Kumar  
Computer Science Department  
University of Minnesota  
Minneapolis, MN 55455

J. Lannutti  
MS B-186  
Director, Supercomputer Research Inst.  
Florida State University  
Tallahassee, FL 32306

P. D. Lax  
New York University- Courant  
251 Mercer St.  
New York, NY 10012

J.K. Lee  
Department of Engineering Mechanics  
Ohio State University  
Columbus, OH 43210

Lawrence A. Lee, Executive Director  
North Carolina Supercomputing Center  
P.O. Box 12889  
3021 Cornwallis Road  
Research Triangle Park, NC 27709

David Levine  
Mathematics and Computer Science  
Argonne National Laboratory  
9700 Cass Avenue South  
Argonne, IL 60439

Trent R. Logan  
Rockwell International Group  
Mail Code NA40  
12214 Lakewood Blvd.  
Downey, CA 90242

Mr. Louis S. Lome  
SDIO/TNI  
The Pentagon  
Washington, DC 20301-7100

G. Lyles  
CIA  
6219 Lavell Ct.  
Springfield VA 22152

S. F. McCormick  
Computer Mathematics Group  
University of Colorado at Denver  
1200 Larimar St.  
Denver, CO 80204

Frank Maestas  
Principal Engineer  
Applied Research Associates  
4300 San Mateo Blvd.  
Suite A220  
Albuquerque, NM 87110

H. Mair (5)  
Naval Surface Warfare Center  
10901 New Hampshire Ave.  
Silver Springs, MD 20903-5000  
Attn: H. Mair, W. Reed, W. Holt, K.B.Kim,  
P.Walters

Mark Majerus  
California Research and Technology, Inc.  
PO Box 2229  
Princeton, NJ 08543-2229

T. A. Manteuffel  
Department of Mathematics  
University of Colorado at Denver  
Denver, CO 80202

Carlos Marino  
Industry, Science, and Technology Department  
Cray Research Park  
655 E. Lone Oak Dr.  
Eagan, MN 55121

William S. Mark, Ph.D.  
Lockheed - Org. 96-01  
Building 254E  
3251 Hanover Street  
Palo Alto, CA 94303-1191

D. Matuska  
Orlando Technology, Inc.  
P. O. Box 855  
Shalimar, FL 32579

John May (2)  
Kaman Sciences Corporation  
1500 Garden of the Gods Road  
Colorado Springs, CO 80933  
Attn: John May and S. Hones

J. Mesirov  
Thinking Machines Inc.  
245 First Street  
Cambridge, MA 94550
P. C. Messina
158-79
Mathematics and Computer Science Department
Caltech
Pasadena, CA 91125

Craig Miller
Unit 973
General Electric Company
Neutron Devices Department
P.O. Box 2908
Largo, FL 34294-2908

Robert E. Millstein
TMC
245 First Street
Cambridge, MA 02142

G. Mohnkern
Naval Ocean Systems Center
Code 73
San Diego, CA 92152-5000

C. Moler
The Mathworks
325 Linfield Place
Menlo Park, CA 94025

J.J. Murphy
Vehicle Technology 59-22
Bldg 580
Lockheed Missile and Space Co.
P.O. Box 3504
Sunnyvale, CA 94088

V.D. Murty
5000 N. Willamette Blvd.
School of Engineering
University of Portland
Portland, OR 97203

Naval Underwater Systems Center (3)
Attn: Dan Bowles
G. Letiecq
S. Prashaw
Mail Code 8123
Newport, RI 02841-5047

C. E. Needham
Maxwell Laboratories, Inc.
2501 Yale S.E., Suite 300
Albuquerque, NM 87106

D. B. Nelson, Exec. Dir
Office of Energy Research
U.S. Department of Energy
Washington, DC 20545

Jim Nemes
Code 6331
Naval Research Laboratory
Washington, DC 20375-5000

William Nester
Oak Ridge National Laboratory
PO Box 2009
Oak Ridge, TN 37831-8058

Jeff Newmeyer
Org. 81-04
Building 157
1111 Lockheed Way
Sunnyvale, CA 94089-3504

Dean Norman
Waterways Experiment Station
P.O. Box 631
Vicksburg, MS 39180

D. M. Nosanchuck
Mech. and Aerospace Engineering Dept.
D302 E. Quad
Princeton University
Princeton, NJ 08544

Office of Naval Research (2)
Attn: Rembert Jones
A.S. Kushner
Structural Mechanics Division (Code 434)
800 N. Quincy Street
Arlington, VA 22217

C. E. Oliver, Director
Office of Laboratory Computing, Bldg. 4500N
Oak Ridge National Laboratory
P.O. Box 4141
Oak Ridge, TN 37831-6251

Dennis L. Orphal (3)
California Research & Technology Inc.
5117 Johnson Dr.
Pleasanton, CA 94588
Attn: D.L. Orphal, P.N. Schneider, S.P. Segan
J. M. Ortega  
Applied Mathematics Department  
University of Virginia  
Charlottesville, VA 22903

John Palmer  
TMC  
245 First Street  
Cambridge, MA 02142

Robert J. Paluck, President  
Convex Computer Corporation  
3000 Waterview Parkway  
P.O. Box 733851  
Richardson, TX 75083-3851

Robert Pardue  
Martin Marietta  
Y-12 Plant, Bldg. 9998  
Mail Stop 2  
Oak Ridge, TN 37831

Kim Parnell  
Failure Analysis Associates, Inc.  
149 Commonwealth Ave.  
PO Box 3015  
Menlo Park, CA 94025

S. V. Patter  
Department of Mathematics  
Van Vleck Hall  
University of Wisconsin  
Madison, WI 53706

Dr. Nisheeth Patel  
U.S. Army Ballistic Research Lab  
AMXBR-LFD  
Aberdeen Proving Ground, MD 21005-5066

A. T. Patena  
Mechanical Engineering Department  
77 Massachusetts Ave.  
MIT  
Cambridge, MA 02139

A. Patinos, Acting Director  
Atmos. and Climate Resch. Division  
Office of Energy Research, ER-74  
U.S. Department of Energy  
Washington, DC 20545

R. F. Peierls  
Mathematics Sciences Group, Bldg. 515  
Brookhaven National Laboratory  
Upton, NY 11973

K. Perko  
Supercomputing Solutions, Inc.  
6175 Mancy Ridge Dr.  
San Diego, CA 92121

John Perresky  
Ballistic Research Lab, Launch & Flight Div.  
SLCBR-LF-C  
Aberdeen Proving Ground, MD 21005-5006

Mitchell R. Philabaum  
Monsanto Research Corporation  
MRC-MOUND  
Miamisburg, OH 45342

Phillips Laboratory (3)  
Attn:  F. Allahadi  
D. Fulk  
J. Secary  
Nuclear Technology Branch  
Kirtland AFB, NM 87117-6008

Dr. Leslie Pierre  
SDIO/ENA  
The Pentagon  
Washington, DC 20301-7100

R. J. Plemmons  
Department of Mathematics and Computer Science  
Wake Forest University  
P.O. Box 7311  
Winston Salem, NC 27109

John Prentice  
Amparo Corporation  
3700 Rio Grande, NW  
Suite 5  
Albuquerque, NM 87107-3042

Peter P. F. Radkowski III  
P.O. Box 1121  
Los Alamos, NM 87544

J. Rattner  
Intel Scientific Computers  
15201 NW Greenbriar Pkwy.  
Beaverton, OR 97006
Mark Smith  
Aerophysics Branch  
Calspan Corporation/AEDC Operations  
MS 440  
Arnold AFB, TN 37389

William R. Somsky  
Ballistic Research Laboratory  
SLCBR-SE-A, Bldg. 394  
Aberdeen Proving Ground, MD 21005-5066

D. C. Sorenson  
Department of Mathematical Sciences  
Rice University  
P.O. Box 1892  
Houston, TX 77251

Southwest Research Institute (4)  
Attn: Charles E. Anderson  
C.J Kuhlman  
Samit Roy  
J.D. Walker  
P.O. Drawer 28510  
San Antonio, TX 78284

S. Squires  
DARPA/ISTO  
1400 Wilson Blvd.  
Arlington, VA 11109

N. Srinivasan  
AMOCO Corporation  
P.O. Box 87703  
Chicago, IL 60680-0703

D. E. Stein  
AT&T  
100 South Jefferson Rd.  
Whippany, NJ 07981

M. Steuerwalt, Program Director  
Division of Mathematical Sciences  
National Science Foundation  
Washington, DC 20550

G. W. Stewart  
Computer Science Department  
University of Maryland  
College Park, MD 20742

O. Storassli, MS-244  
NASA Langley Research Center  
Hampton, VA 23665

C. Stuart  
DARPA/TTO  
1400 Wilson Blvd.  
Arlington, VA 22209

LTC James Sweeney  
SDIO/SDA  
The Pentagon  
Washington, DC 20301-7100

D.V. Swenson  
Mechanical Engineering Department  
Durland Hall  
Kansas State University  
Manhattan, KS 66506

H.T. Tang  
Electric Power Research Institute  
3412 Hillview Avenue  
P.O. Box 10412  
Palo Alto, CA 94304

Sing C. Tang  
P.O. Box 2053  
RM 3039 Scientific Lab  
Dearborn, MI 48121-2053

Bill Tanner  
Space Science Laboratory  
Baylor University  
PO Box 7303  
Waco, TX 76798

R. A. Tapia  
Mathematical Sciences Department  
Rice University  
P.O. Box 1892  
Houston, TX 77251

Gligor A. Tashkovich  
210 Lake Street, Apt. 5F  
Ithaca, NY 14850-3854

William J. Tedeschi  
DNA/SPSP  
6801 Telegraph Rd.  
Alexandria, VA 22310
Attn: J. K. Meier, MS G787
Attn: R. W. Meier, MS G787
Attn: K.A. Meyer, MS F663
Attn: N.R. Morse, MS B260
Attn: D.C. Nelson, MS G787
Attn: A. T. Oyer, MS G787
Attn: R.B. Parker, MS G787
Attn: D.A. Rabern, MS G787
Attn: M. Rich, MS F669
Attn: P.R. Romero, MS G787
Attn: J.J. Rumine, MS C931
Attn: M. Sahota, MS B257
Attn: D.J. Sandstorm, MS G756
Attn: W. Sparks, MS F663
Attn: L.H. Sullivan, MS K557
Attn: D. Tonks, MS B257
Attn: H. E. Trease, MS B257
Attn: B.M. Wheat, MS G787
Attn: A.B. White, MS-265
Attn: T.F. Wimett, MS J562
Attn: L. Witt, MS C936
Attn: S. Woodruff, MS K557
Attn: Robert Young, MS K574

University of California
Lawrence Livermore National Laboratory
7000 East Ave.
P.O. Box 808
Livermore, CA 94550

Attn: R. R. Borcher, MS L-669
Attn: D. E. Burton, MS L-18
Attn: R. C. Y. Chin, MS L-321
Attn: R. B. Christensen, MS L-35
Attn: R. E. Huddleston, MS L-61
Attn: J. M. LeBlanc, MS L-35
Attn: J. R. McGraw, MS L-316
Attn: G. A. Michael, MS L-306
Attn: M. J. Murphy, MS L-368
Attn: L. R. Petzold, MS L-316
Attn: J. E. Reaugh, MS L-290
Attn: D. J. Steinberg, MS L-35
Attn: R. Stoudt, MS L-200
Attn: R. E. Tipton, MS L-35
Attn: C. E. Rhoades, MS L-298
1. Internal

<table>
<thead>
<tr>
<th>Page</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1231</td>
<td>T.W.L. Sanford</td>
</tr>
<tr>
<td>1270</td>
<td>J.K. Rice</td>
</tr>
<tr>
<td>1271</td>
<td>G.O. Allshouse</td>
</tr>
<tr>
<td>1400</td>
<td>E.H. Barsis</td>
</tr>
<tr>
<td>1420</td>
<td>W. J. Camp</td>
</tr>
<tr>
<td>1421</td>
<td>S. S. Donajh</td>
</tr>
<tr>
<td>1421</td>
<td>D. R. Gardner</td>
</tr>
<tr>
<td>1425</td>
<td>J. H. Biffl~</td>
</tr>
<tr>
<td>1425</td>
<td>S. W. Attaway</td>
</tr>
<tr>
<td>1510</td>
<td>S. T. Montgomery</td>
</tr>
<tr>
<td>1511</td>
<td>J. S. Rottler</td>
</tr>
<tr>
<td>1512</td>
<td>A. C. Raatzel</td>
</tr>
<tr>
<td>1513</td>
<td>J. C. Cummings, Acting</td>
</tr>
<tr>
<td>1514</td>
<td>H. S. Morgan</td>
</tr>
<tr>
<td>1514</td>
<td>V. L. Bergmann</td>
</tr>
<tr>
<td>1514</td>
<td>B. J. Thorne</td>
</tr>
<tr>
<td>1540</td>
<td>J. R. Asay</td>
</tr>
<tr>
<td>1541</td>
<td>J. M. McGlaun</td>
</tr>
<tr>
<td>1541</td>
<td>K. Badge (50)</td>
</tr>
<tr>
<td>1541</td>
<td>M. G. Elrick</td>
</tr>
<tr>
<td>1541</td>
<td>E. S. Hertel</td>
</tr>
<tr>
<td>1541</td>
<td>R. J. Lawrence</td>
</tr>
<tr>
<td>1541</td>
<td>J. S. Peery</td>
</tr>
<tr>
<td>1541</td>
<td>A. C. Robinson</td>
</tr>
<tr>
<td>1541</td>
<td>T. G. Trucano</td>
</tr>
<tr>
<td>1541</td>
<td>L. Yarrington</td>
</tr>
<tr>
<td>1541</td>
<td>RHALE Day File</td>
</tr>
<tr>
<td>1542</td>
<td>P. Yarrington</td>
</tr>
<tr>
<td>1542</td>
<td>R. L. Bell</td>
</tr>
<tr>
<td>1542</td>
<td>W. T. Brown</td>
</tr>
<tr>
<td>1542</td>
<td>P. J. Chen</td>
</tr>
<tr>
<td>1542</td>
<td>J. E. Dunn</td>
</tr>
<tr>
<td>1542</td>
<td>H. E. Fang</td>
</tr>
<tr>
<td>1542</td>
<td>A. V. Farnsworth</td>
</tr>
<tr>
<td>1542</td>
<td>G. H. Kerley</td>
</tr>
<tr>
<td>1542</td>
<td>M. E. Kipp</td>
</tr>
<tr>
<td>1542</td>
<td>F. R. Norwood</td>
</tr>
<tr>
<td>1542</td>
<td>S. A. Siling</td>
</tr>
<tr>
<td>1543</td>
<td>P. L. Stanton</td>
</tr>
<tr>
<td>1543</td>
<td>J. A. Ang</td>
</tr>
<tr>
<td>1543</td>
<td>L. C. Chhabildas</td>
</tr>
<tr>
<td>1543</td>
<td>M. D. Furnish</td>
</tr>
<tr>
<td>1543</td>
<td>D. E. Grady</td>
</tr>
<tr>
<td>1543</td>
<td>J. W. Swegle</td>
</tr>
<tr>
<td>1543</td>
<td>J. L. Wise</td>
</tr>
<tr>
<td>1544</td>
<td>J. R. Asay, Acting</td>
</tr>
<tr>
<td>1544</td>
<td>C. R. Adams</td>
</tr>
<tr>
<td>1544</td>
<td>K. W. Gwinn</td>
</tr>
<tr>
<td>1544</td>
<td>E. Kephart</td>
</tr>
<tr>
<td>1544</td>
<td>F. J. Mello</td>
</tr>
<tr>
<td>1544</td>
<td>K. E. Metzinger</td>
</tr>
<tr>
<td>1544</td>
<td>E. D. Reedy</td>
</tr>
<tr>
<td>1544</td>
<td>K. W. Schuler</td>
</tr>
<tr>
<td>1544</td>
<td>G. D. Sjaardema</td>
</tr>
<tr>
<td>1544</td>
<td>A. M. Slavin</td>
</tr>
<tr>
<td>1544</td>
<td>P. P. Stibis</td>
</tr>
<tr>
<td>1544</td>
<td>R. K. Thomas</td>
</tr>
<tr>
<td>1545</td>
<td>D. R. Martinez</td>
</tr>
<tr>
<td>1545</td>
<td>J. J. Allen</td>
</tr>
<tr>
<td>1545</td>
<td>L. Branstetter</td>
</tr>
<tr>
<td>1545</td>
<td>J. Dohner</td>
</tr>
<tr>
<td>1545</td>
<td>C. R. Dohrmann</td>
</tr>
<tr>
<td>1545</td>
<td>G. R. Eiser</td>
</tr>
<tr>
<td>1545</td>
<td>J. T. Foley</td>
</tr>
<tr>
<td>1545</td>
<td>D. W. Lobitz</td>
</tr>
<tr>
<td>1545</td>
<td>D. B. Longcope</td>
</tr>
<tr>
<td>1545</td>
<td>E. L. Marek</td>
</tr>
<tr>
<td>1545</td>
<td>J. Pott</td>
</tr>
<tr>
<td>1545</td>
<td>J. R. Red-Horse</td>
</tr>
<tr>
<td>1545</td>
<td>D. J. Segulman</td>
</tr>
<tr>
<td>1550</td>
<td>C. W. Peterson</td>
</tr>
<tr>
<td>1551</td>
<td>J. K. Cole</td>
</tr>
<tr>
<td>1552</td>
<td>D. D. McBride</td>
</tr>
<tr>
<td>1553</td>
<td>W. L. Hermiina</td>
</tr>
<tr>
<td>1554</td>
<td>D. P. Aeschliman</td>
</tr>
<tr>
<td>1555</td>
<td>W. P. Wolfe</td>
</tr>
<tr>
<td>1556</td>
<td>W. L. Oberkampf</td>
</tr>
<tr>
<td>1600</td>
<td>W. Herrmann</td>
</tr>
<tr>
<td>2513</td>
<td>D. E. Mitchell</td>
</tr>
<tr>
<td>2513</td>
<td>S.H. Fischer</td>
</tr>
<tr>
<td>3141</td>
<td>Technical Library (5)</td>
</tr>
<tr>
<td>3141-5</td>
<td>Document Processing for DOE/OSTI (8)</td>
</tr>
<tr>
<td>3151</td>
<td>Technical Communications (3)</td>
</tr>
<tr>
<td>6418</td>
<td>S. L. Thompson</td>
</tr>
<tr>
<td>6418</td>
<td>L. N. Kmetyk</td>
</tr>
<tr>
<td>6429</td>
<td>K. E. Washington</td>
</tr>
<tr>
<td>6429</td>
<td>R. W. Ostensen</td>
</tr>
<tr>
<td>6463</td>
<td>M. Berman</td>
</tr>
<tr>
<td>6463</td>
<td>K. Boyack</td>
</tr>
<tr>
<td>8240</td>
<td>C. W. Robinson</td>
</tr>
<tr>
<td>8241</td>
<td>G. A. Benedetti</td>
</tr>
<tr>
<td>8241</td>
<td>M. L. Chiesa</td>
</tr>
<tr>
<td>8241</td>
<td>L. E. Voelker</td>
</tr>
<tr>
<td>8242</td>
<td>M. R. Birnbaum</td>
</tr>
<tr>
<td>8242</td>
<td>J. L. Cherry</td>
</tr>
<tr>
<td>8242</td>
<td>J. J. Dike</td>
</tr>
<tr>
<td>8242</td>
<td>B. L. Kistler</td>
</tr>
<tr>
<td>8242</td>
<td>A. McDonald</td>
</tr>
<tr>
<td>8242</td>
<td>V. D. Revelli</td>
</tr>
<tr>
<td>8242</td>
<td>L. A. Rogers</td>
</tr>
<tr>
<td>8242</td>
<td>K. V. Trinh</td>
</tr>
<tr>
<td>8242</td>
<td>L. I. Weingarten</td>
</tr>
</tbody>
</table>