SAND96-2775
Unlimited Release
Printed February 1998

Distribution
Category UC-706

VECTORS
A Fortran 90 Module for 3-Dimensional Vector and Dyadic Arithmetic

Billy C. Brock
Radar/Antenna Department
Sandia National Laboratories
Albuquerque, New Mexico 87185-0533

Abstract

A major advance contained in the new Fortran 90 language standard is the ability to define new data types and the operators associated with them. Writing computer code to implement computations with real and complex three-dimensional vectors and dyadics is greatly simplified if the equations can be implemented directly, without the need to code the vector arithmetic explicitly. The Fortran 90 module described here defines new data types for real and complex 3-dimensional vectors and dyadics, along with the common operations needed to work with these objects. Routines to allow convenient initialization and output of the new types are also included. In keeping with the philosophy of data abstraction, the details of the implementation of the data types are maintained private, and the functions and operators are made generic to simplify the combining of real, complex, single- and double-precision vectors and dyadics.

This Page Intentionally Blank

DISCLAIMER

DISCLAIMER

Portions of this document may be illegible electronic image products. Images are produced from the best available original document.

Contents

Introduction

A major advance contained in the new Fortran 90 language standard is the ability to define new data types and the operators associated with them. Writing computer code to implement computations with real and complex 3-dimensional vectors and dyadics is greatly simplified if the equations can be implemented directly. In the past, it has been necessary to break these into separate equations for each of the individual components in order to code the vector arithmetic explicitly. The Fortran 90 module described here defines new data types for real and complex 3-dimensional vectors and dyadics, along with the common operations needed to work with these objects. Routines to allow convenient initialization and output of the new types are also included. In keeping with the philosophy of data abstraction, the details of the implementation of the data types are maintained private, and the functions and operators are made generic to simplify the combining of real, complex, single- and double-precision vectors and dyadics.

Module `vectors` has been used successfully over the past year to simplify and accelerate the development and maintenance of vector-intensive computational codes. Because these data types and associated operators are becoming embedded in numerous computer codes, some of which may have an extended lifetime, it has become necessary to document the module. By publishing this documentation, I am not claiming that the implementation of this module is by any means optimal, but continued use of the module without appropriate documentation will certainly be suboptimal. Other methods of implementation are certainly possible and valid. I have made choices in the design of this module that suited my personal programming taste, which continues to evolve.

In order to handle the many possible combinations of real, complex, single precision, double precision, scalar, vector, and dyadic operands, the module has grown to a rather large size. The large size of the source code is a result of the necessity to include separate versions of each routine to handle the many different combinations of arguments. For example, a function procedure is included which takes three scalar arguments `x`, `y`, and `z`, and returns a vector of `type (real_vector)` with components `x`, `y`, and `z` along the principal axes. This function is necessary, because the default constructor is not available outside the module since the internal details of the new types are private. Naturally, a version exists where `x`, `y`, and `z` are all real, but in addition, versions exist where `x` is real but `y` and `z` are integers, and where `y` is real but `x` and `z` are integers, and so on. Also similar functions exist for double precision, complex, and double-precision complex vectors. All of these routines are mapped to one generic function `vector(x, y, z)`. With all of the possible combinations of arguments, there are 125 functions which are mapped to the same generic function, `vector(x, y, z)`. All of these routines are essentially identical, except for type statements for the arguments and return variable, which must change from routine to routine. The necessity of explicitly defining routines for each case, rather than defining a template function as in C++, causes the source code to become very large. However, once the routines are written and tested, the module greatly simplifies the writing and maintenance of any programs which use vector and dyadic arithmetic.

Rather than include a listing of the source code, the source code is contained on the enclosed high density 3.5" diskette in the MS-DOS format. The source code, `vectors.f90`, consists of two modules, `Vectors`,and `Roots`, which is a module required by `Vectors`. This source code has been compiled successfully on an Intel Pentium-based computer with the Lahey Fortran 90 compiler, version 2.01i, with Microsoft PowerStation, version 4.0, and with Digital Visual Fortran, version 5.0. It has also been successfully compiled on a Sun Sparc 20 system with SunSoft Fortran 90, version 1.1. Although the code has been extensively used and tested, it is not practical to test all possible circumstances, and no guarantee of accuracy or correctness of the code can be offered.

Overview of the Vector and Dyadic-Tensor Arithmetic Module

The Fortran 90 module `Vectors` contains type definitions and operators to facilitate three-dimensional vector and dyadic-tensor arithmetic. The module (version 1.00) consists of 561 function and 84 subroutine procedures. However, these procedures are maintained private, and are available to the user only through eight generic operators, fourteen generic functions, and two generic subroutines.

Data types

The module implements real and complex vectors in both single- and double-precision numerical kinds. The following eight new types are defined:

`type (real_vector)`	-- single-precision real vector,
`type (complex_vector)`	-- single-precision complex vector,
`type (dbl_real_vector)`	-- double-precision real vector,
`type (dbl_complex_vector)`	-- double-precision complex vector,
`type (real_dyadic)`	-- single-precision real dyadic tensor,
`type (complex_dyadic)`	-- single-precision complex dyadic tensor,
`type (dbl_real_dyadic)`	-- double-precision real dyadic tensor,
`type (dbl_complex_dyadic)`	-- double-precision complex dyadic tensor.

Operators

All procedures and operators are generic, accept any of the newly defined types, and return the appropriate result. The following generic operators are defined or extended to accept vector and dyadic operands, as appropriate:

=	assignment operator,
*	multiply vector or dyadic by scalar,
	dot product between two vectors,
	dot product between a vector and a dyadic,
	dot product between two dyadics,
.X.	cross product between two vectors,
	cross product between a vector and a dyadic,
/	divide a vector or a dyadic by a scalar,
+	add two vectors or two dyadics,
-	subtract two vectors or two dyadics,
	negation of vector or dyadic
.neg.	negation of vector or dyadic,

`.dyad.`	dyadic vector product (product of two vectors without dot or cross symbol, resulting in a dyad).

Operator precedence for the extended operators is the same as for the original Fortran 90 operators, while the new operators (`.X.`, `.neg.`, `.dyad.`) have the lowest precedence.

Functions

The following generic functions are supported:

`vector(x,y,z)`	creates a vector from three scalar components,
`dyadic(X,Y,Z)`	creates a dyadic from three vector components (the vectors `X,Y,Z` are the left-hand factors of the dyads and form the columns when the dyadic is represented as a matrix; the dyadic created is $\bar{\mathbf{X}}\hat{\mathbf{x}} + \bar{\mathbf{Y}}\hat{\mathbf{y}} + \bar{\mathbf{Z}}\hat{\mathbf{z}}$),
`abs()`	returns the magnitude of a vector,
`real()`	returns the real part of a complex vector or dyadic,
`aimag()`	returns the imaginary part of a complex vector or dyadic,
`conjg()`	returns the complex conjugate of a complex vector or dyadic,
`unit_vector()`	creates a unit vector in the direction of the vector argument,
`det()`	returns the determinant of a dyadic,
`inv()`	returns the inverse of a dyadic,
`trans()`	returns the transpose a dyadic ,
`trace()`	computes the trace of a dyadic (sum of diagonal elements)
`get(A,i)`	returns the i^{th} Cartesian element (`i`=1,2,3) of a vector or a dyadic (returns a scalar when `A` is a vector and a vector when `A` is a dyadic),
`eigen_values(A)`	returns an array of 3 complex eigen values of the dyadic `A`,
`eigen_vector(B)`	returns the eigen vector, $\hat{\mathbf{e}}$, of dyadic `B`, such that $\bar{\bar{\mathbf{B}}} \cdot \hat{\mathbf{e}} = 0$.

Subroutines

The following subroutines are included:

`set(A,b,i)`	sets element `i` of `A` to value of `b`, where `b` is a scalar when `A` is a vector and a vector when `A` is a dyadic,
`write([lun,] string,[edt,] A)`	writes a string followed by a vector or dyadic, `A`, to the default output device, or to the optional logical device (`lun`, an integer). The optional edit descriptor (`edt`) is any valid edit descriptor string for a Fortran 90 real type, for example 'f6.2'.

Description of the Vector and Dyadic Operators and Procedures

Operators

Assignment (=)

The assignment operator expects to find the name of a vector or dyadic on the left side, and a vector or dyadic expression on the right. The left-side variable is assigned the

value of the right-side expression. Type conversion between vector types or between dyadic types is performed. However, the left and right operands must be either both vectors or dyadics. When the right side is a complex or double-precision complex vector or dyadic expression, and the left-hand variable is real or double precision, it is assigned the value of the real part of the right-hand side; the imaginary part is simply lost. The complete functionality is implemented with 24 private subroutine procedures interfaced to the assignment operator, =.

Multiplication and Dot Products (*)

The multiplication and dot product operator can return a dyadic, vector, or scalar result, depending on the types of the operands. A vector or dyadic can be multiplied by a scalar, two vectors can be dot multiplied, a vector and a dyadic can be dot multiplied, and two dyadics can be dot multiplied.

Scalar multiplication allows the scalar (which can be of any numerical type, including integer) to be on the left or the right side of the operator. The other operand can be any of the four vector types or four dyadic types. The result will be a vector or a dyadic, as appropriate, of the highest numerical kind contained in the operands. For example, multiplying a real vector by a complex scalar will return a complex vector.

When neither operand is a scalar, the operator implements the dot product between the operands. The dot product of two vectors returns a scalar of the highest numerical kind contained in the operands. The dot product between a vector (on the left or the right) and a dyadic is a vector. A dot product between two dyadics returns a dyadic of the highest numerical kind contained in the operands. The meaning of vector and dyadic dot products is described in the Summary of Vector and Dyadic Arithmetic, below.

The complete functionality requires 144 private function procedures interfaced to the multiplication and dot product operator, *.

Cross Products (.X.)

The cross-product operator, .X., returns either a vector or a dyadic, depending on the operands. If both operands are vectors, the result is a vector of the highest numerical kind contained in the operands. A cross product between a vector and a dyadic (with the dyadic on either the left or the right side) produces a dyadic of the highest numerical kind of the operands. The meaning of the cross product between a dyadic and a vector is discussed in the Summary of Vector and Dyadic Arithmetic, below. The cross product between two dyadics is not allowed, as it results in a tensor of rank 3, and this type is not implemented in the module. The complete functionality of the cross-product operator, .X. requires an interface to 48 private function procedures.

Scalar Division (/)

Scalar division is the only division operator defined. A vector or a dyadic can be divided by a scalar of any numerical kind. The scalar must be on the right. If a vector or dyadic appears immediately to the right of the division operator, an error occurs since no operation is defined for this case. The result of the scalar division is a vector

or dyadic, as appropriate, of the highest numerical kind appearing in the operands. The complete functionality requires interfacing the division operator, /, to 40 private function procedures.

Addition (+)

The addition operator implements element-by-element addition between two vectors or two dyadics. Both operands must be either vectors or dyadics, but they can have different numerical kinds. The result is a vector or dyadic, as appropriate, of the highest numerical kind contained in the operands. The complete functionality requires interfacing the addition operator, +, to 32 private function procedures.

Subtraction and Negation(-)

The subtraction and negation operator is special, in that it can be either a binary or unary operator. When there is only one operand (unary operator), the operand must be on the right of the operator. The unary negation operator returns either a vector or dyadic, as appropriate, which is the operand multiplied by the scalar `-1`. When there are two operands (binary operator), the left and right operands must both be either vectors or dyadics. The operator returns a vector or dyadic, as appropriate, which is the element-by-element difference between the left operand and the right operand. The result is of the highest numerical kind contained in the operands. The complete functionality requires 40 private function procedures interfaced to the subtraction and negation operator.

Negation (`.neg.`)

This is an alternate to the unary version of the - operator. As a unary operator, it takes a single operand, which must be on the right. It returns either a vector or dyadic, as appropriate, which is the operand multiplied by the scalar `-1`. The complete functionality requires 8 private function procedures interfaced to the `.neg.` operator. These functions are also interfaced to the - operator.

Dyadic Product (`.dyad.`)

The dyadic-product operator is a binary operator which requires two vector operands. The result is a dyadic which is the dyad composed of the left and right vectors. The numerical kind of the result is the highest numerical kind contained in the operands. The complete functionality requires 16 private function procedures interfaced to the `.dyad.` operator.

Functions

`vector(x, y, z)`

The function `vector` requires three scalar arguments, each of which can be any numerical kind. The result is a vector of the highest numerical kind contained among the three arguments, with the exception that if all arguments are integers, a vector of `type(real_vector)` is returned. The scalar arguments, which are not required to be of the same numerical kind, represent the vector components along each of the three Cartesian axes. This function provides the primary method of initializing a vector. The full functionality requires interfacing 125 function procedures to the generic function, `vector`.

dyadic(X, Y, Z)

The function `dyadic` requires three vector arguments, and returns a dyadic of the highest numerical kind contained in the three arguments. The arguments must be vectors, but are not required to be of the same numerical kind. The vectors `X, Y, Z` are the left-hand factors of the dyads and form the columns when the dyadic is represented as a matrix. The dyadic created is $\vec{X}\hat{x} + \vec{Y}\hat{y} + \vec{Z}\hat{z}$, where $\hat{x}$, $\hat{y}$, and $\hat{z}$ are unit vectors along the Cartesian axes. This function provides the primary method of initializing a dyadic. The full functionality requires interfacing of 64 private function procedures to the generic function, `dyadic`.

abs()

This function extends the intrinsic function of the same name, and returns the magnitude of a vector argument. The argument can be any of the four kinds of vector, and the numerical kind of the result is single or double precision, depending on the numerical kind of the argument. The result is real and always greater than or equal to zero. The full functionality requires interfacing 4 private function procedures to the generic function, `abs`.

real()

This function extends the intrinsic function of the same name, and returns the real part of a complex vector or dyadic argument. The argument can be either `type(complex_vector)`, `type(dbl_complex_vector`, `type(complex_dyadic)`, or `type(dbl_complex_dyadic)`, and the numerical kind of the result is single or double precision, depending on the numerical kind of the argument. The full functionality requires interfacing 4 private function procedures to the generic function, `real`.

aimag()

This function extends the intrinsic function of the same name, and returns the imaginary part of a complex vector or dyadic argument. The argument can be either `type(complex_vector)`, `type(dbl_complex_vector`, `type(complex_dyadic)`, or `type(dbl_complex_dyadic)`, and the numerical kind of the result is single or double precision, depending on the numerical kind of the argument. The full functionality requires interfacing 4 private function procedures to the generic function, `aimag`.

conjg()

This function extends the intrinsic function of the same name, and returns the complex conjugate of a complex vector or dyadic argument. The argument can be either `type(complex_vector)`, `type(dbl_complex_vector`, `type(complex_dyadic)`, or `type(dbl_complex_dyadic)`, and the type of the result will match the type of the argument. The full functionality requires interfacing 4 private function procedures to the generic function, `conjg`.

unit_vector()

This function accepts a vector argument of any of the four kinds of vectors, and returns a unit vector in the same direction. The unit vector is obtained by dividing the argument by its absolute value (magnitude). The result is a vector of the same type as the argument, and is real if the argument is a real vector and complex if the argument

is a complex vector. The full functionality is obtained with 4 private function procedures interfaced to the generic function, `unit_vector`.

`det()`

This function accepts a dyadic argument, which can be any one of the four kinds of dyadics, and returns the scalar value of the determinant of the argument. The result is single or double precision, depending on the numerical kind of the argument, and will be complex when the dyadic is a complex dyadic. The full functionality is obtained with 4 private function procedures interfaced to the generic function `det`.

`inv()`

This function accepts a dyadic argument, which can be any one of the four kinds of dyadics, and returns the inverse dyadic, which, when dot multiplied with the argument, will result in the unit dyadic. The result will have the same type as the argument. The full functionality is obtained with 4 private function procedures interfaced to the generic function `inv`.

`trans()`

This function accepts a dyadic argument, which can be any one of the four kinds of dyadics, and returns the transpose of the dyadic. The result will have the same type as the argument dyadic. The full functionality is obtained with 4 private function procedures interfaced to the generic function `trans`.

`trace()`

This function accepts a dyadic argument, which can be any one of the four kinds of dyadics, and returns the scalar value of the trace (sum of diagonal elements) of the dyadic. The result will have the same numerical kind as the argument dyadic. The full functionality is obtained with 4 private function procedures interfaced to the generic function `trace`.

`get(A, i)`

This function accepts two arguments. The first is a vector or dyadic, which can be any one of the four kinds of vectors or dyadics, and the second is an integer between 1 and 3. The integer indicates which Cartesian component to return. When the first argument is a vector, the function returns the scalar which is the i^{th} Cartesian component. When the first argument is a dyadic, the function returns the vector which is the i^{th} Cartesian vector component when the dyadic is written with the Cartesian unit vectors on the right side. This is the i^{th} column when the dyadic is written as a matrix. The value (scalar or vector) returned by `get` is identical to the value obtained by taking the dot product of the argument with the i^{th} unit vector on the right side. For example, `get(A, 2)` returns the scalar $\vec{\mathbf{A}} \cdot \hat{\mathbf{y}}$ when `A` is a vector, and returns the vector $\bar{\bar{\mathbf{A}}} \cdot \hat{\mathbf{y}}$ when `A` is a dyadic. The numerical kind of the result matches that of the argument. The full functionality is obtained with 8 private function procedures interfaced to `get`.

eigen_values(A)

This is an array-valued function which accepts a single dyadic argument of any of the four kinds of dyadics. The function evaluates to a complex array of three eigen values. The eigen values associated with a dyadic $\bar{\bar{\mathbf{A}}}$ are given by λ_i, such that

$$\left|\bar{\bar{\mathbf{A}}} - \lambda_i \bar{\bar{\mathbf{I}}}\right| \equiv 0, \tag{1}$$

where $\bar{\bar{\mathbf{I}}} = \hat{\mathbf{x}}\hat{\mathbf{x}} + \hat{\mathbf{y}}\hat{\mathbf{y}} + \hat{\mathbf{z}}\hat{\mathbf{z}}$ is the identity dyadic. The result array is complex, because, in general, the eigen values will be complex, even when the dyadic is real. In Fortran 90, the type of the return value must be determined at the time of the invocation of the function, before the argument is evaluated. The return type is not known when the function is invoked, since it is not known if the eigen values will all be real or complex. Thus, it is necessary to define the return type as a complex array. This function calls subroutine `cubic()`, contained in module `roots`. Module `roots`, described in Appendix A, contains routines to accurately determine the roots of cubic and quartic polynomials. Full functionality of function `eigen_values` requires an interface to 4 private function procedures.

eigen_vector(B)

This function accepts a single argument, which can be any of the four kinds of dyadics, and evaluates to a unit vector of the same numerical kind as the argument. The result vector $\hat{\mathbf{e}}$ satisfies $\bar{\bar{\mathbf{B}}} \cdot \hat{\mathbf{e}} = 0$. The argument `B` represents a singular dyadic $\bar{\bar{\mathbf{B}}}$ such that

$$\begin{aligned} \bar{\bar{\mathbf{B}}} &= \bar{\bar{\mathbf{A}}} - \lambda \bar{\bar{\mathbf{I}}} \\ \left|\bar{\bar{\mathbf{B}}}\right| &= 0 \end{aligned} \tag{2}$$

The function assumes that `B` is singular, and it is the programmer's responsibility to ensure that this is the case; if `B` is *not singular*, `eigen_vector(B)` will return an *incorrect* result. In the interest of efficiency, `eigen_vector` does *not* evaluate the determinant to ensure that `B` is singular. The full functionality requires an interface to 4 private function procedures.

Subroutines

set(A,b,i)

Subroutine `set` is complementary to function `get`, and allows a single element of a vector or dyadic to be changed. The subroutine requires three arguments. The first argument, `A`, is any of the four kinds of vector or four kinds of dyadic. The second argument, `b`, is either a scalar when `A` is a vector, or a vector when `A` is a dyadic. The third argument, `i`, is an integer between 1 and 3, which indicates which Cartesian component of `A` to replace with `b`. The numerical kind of `b` can be different from that of `A`. The type of `A` cannot be changed, so the rules of the assignment operator must apply. For example, if `A` is of `type(real_vector)` and `b` is double-precision complex, then the i^{th} element of `A` is replaced with the real part of `b`, converted to

single precision. The imaginary part is lost since A is a real type. Full functionality requires 28 private subroutine procedures interfaced to the generic subroutine set.

write([lun,] string,[edt,] A)

Subroutine write is the only output routine supplied by the module. It will accept up to four arguments; the arguments enclosed between [] are optional. If the first argument is an integer, then it refers to the logical unit to which the output is to be written. In this case, the logical unit must have been opened previously. If this argument is omitted, then the output is written to the standard output device (usually the terminal). The second argument is required, and is a string used to label the output. The string can have any length, and can even be an empty string, ''. The third argument is optional, and must be a valid Fortran 90 edit descriptor for a real number (for example, 'f10.4', '1pg15.7', '1pe23.15', etc.). If this argument is omitted, then the output is written using g13.7 for single precision quantities and g21.15 for double precision quantities. The last argument, A, is required and is the quantity to be written. It can be any of the four kinds of vector or any of the four kinds of dyadic. The subroutine formats A into a form which is easy to read. For example, the code fragment

```
use Vectors
type (real_vector)  :: u, v
type (real_dyadic)  :: A
u = vector( 1.0, 2.0, 3.0 )
v = vector( -3.0, 2.0, 1.0 )
call write('vector u = ', u)
call write('vector u = ', 'f5.2', u)
A = dyadic(u, v .X. u, v)
call write('dyadic A = ', A)
call write('dyadic A = ', 'f6.2', A)
```

will produce the following output on the default display:

```
vector u = [ 1.000000
             2.000000
             3.000000     ]
vector u = [ 1.00
             2.00
             3.00 ]
dyadic A = [   1.000000       4.000000      -3.000000
               2.000000       10.00000       2.000000
               3.000000      -8.000000       1.000000     ]
dyadic A = [  1.00  4.00 -3.00
              2.00 10.00  2.00
              3.00 -8.00  1.00 ]
```

Summary of Vector and Dyadic Arithmetic

Vectors

The Cartesian coordinate system illustrated in Fig. 1 will be used to describe the vectors and dyadic tensors (usually referred to simply as dyadics). A vector $\bar{\mathbf{a}}$ is defined, as illustrated, by its components along each coordinate axis

$$\bar{\mathbf{a}} = a_x\hat{\mathbf{x}} + a_y\hat{\mathbf{y}} + a_z\hat{\mathbf{z}}. \tag{3}$$

Dot Product Between Two Vectors

The dot product defined between two vectors, $\bar{\mathbf{a}}$ and $\bar{\mathbf{b}}$, is called the scalar dot product, and yields a scalar given by

$$\bar{\mathbf{a}} \cdot \bar{\mathbf{b}} = a_x b_x + a_y b_y + a_z b_z. \tag{4}$$

It can be seen that the scalar dot product is commutative; that is, the commutator is zero

$$\bar{\mathbf{a}} \cdot \bar{\mathbf{b}} - \bar{\mathbf{b}} \cdot \bar{\mathbf{a}} = 0. \tag{5}$$

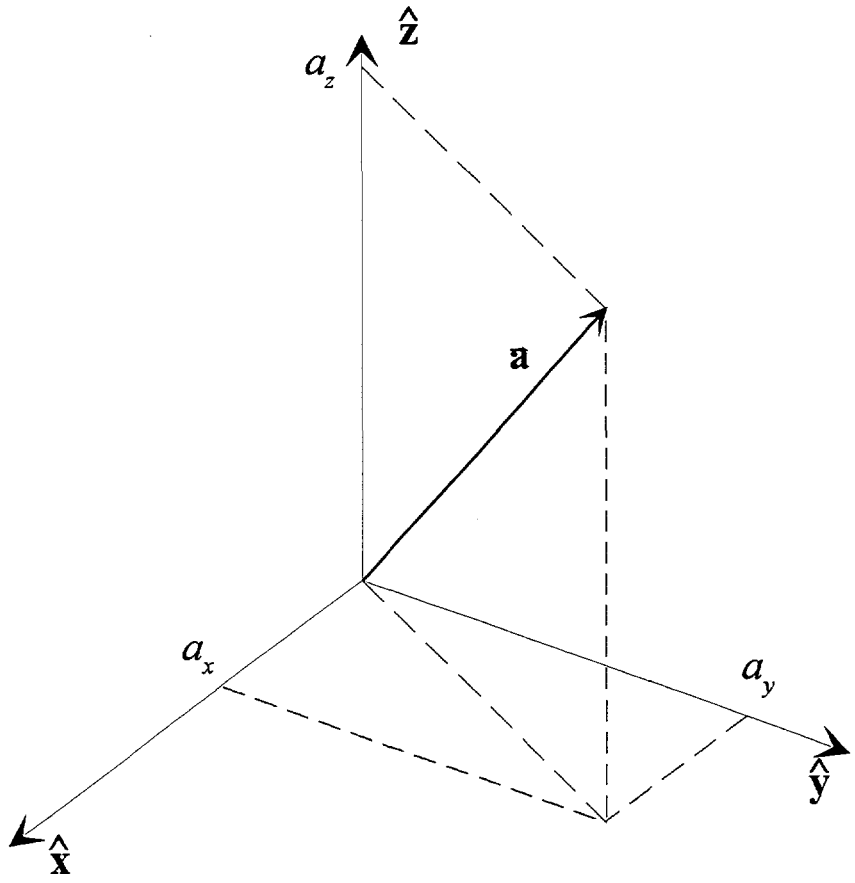

Fig. 1 Reference coordinate system used to describe the vectors and dyadics.

Cross Product Between Two Vectors

The cross product defined between two vectors satisfies the following conditions for the Cartesian unit vectors

$$\hat{\mathbf{x}} \times \hat{\mathbf{x}} = \hat{\mathbf{y}} \times \hat{\mathbf{y}} = \hat{\mathbf{z}} \times \hat{\mathbf{z}} = 0, \tag{6}$$

$$\hat{\mathbf{x}} \times \hat{\mathbf{y}} = -\hat{\mathbf{y}} \times \hat{\mathbf{x}} = \hat{\mathbf{z}}, \tag{7}$$

$$\hat{\mathbf{y}} \times \hat{\mathbf{z}} = -\hat{\mathbf{z}} \times \hat{\mathbf{y}} = \hat{\mathbf{x}}, \tag{8}$$

and

$$\hat{\mathbf{z}} \times \hat{\mathbf{x}} = -\hat{\mathbf{x}} \times \hat{\mathbf{z}} = \hat{\mathbf{y}}. \tag{9}$$

The cross product between two vectors is not commutative

$$\vec{\mathbf{a}} \times \vec{\mathbf{b}} - \vec{\mathbf{b}} \times \vec{\mathbf{a}} \neq 0, \tag{10}$$

but it is anticommutative (the anticommutator is zero)

$$\vec{\mathbf{a}} \times \vec{\mathbf{b}} + \vec{\mathbf{b}} \times \vec{\mathbf{a}} = 0. \tag{11}$$

Associativity of Vector Dot and Cross Products

Note that the scalar dot product and the cross product between two vectors do not form an associative pair of operators. The operations in the expression $\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} \times \vec{\mathbf{c}}$ can only be performed in one order because $\left(\vec{\mathbf{a}} \cdot \vec{\mathbf{b}}\right) \times \vec{\mathbf{c}}$ has no meaning; the operator $\times$ requires two vector operands, not a scalar and a vector operand. Because of the precedence rules in Fortran 90, the operator representing the cross product, `.X.`, has a lower precedence than the operator representing the dot product, `*`. Thus the expression $\vec{\mathbf{a}} \cdot \vec{\mathbf{b}} \times \vec{\mathbf{c}}$ must be coded `a*(b.X.c)`, because `a*b.X.c` will cause an error. Since, in keeping with the precedence rules, `a*b` is evaluated first in the expression `a*b.X.c`, the left operand for the `.X.` operator is of the wrong type (scalar, not vector).

Dyads and Dyadics

A dyad is the product of two vectors, written without any symbol separating them,

$$\bar{\bar{\mathbf{P}}} = \hat{\mathbf{x}}\hat{\mathbf{y}}. \tag{12}$$

A dyad has, at most, six independent scalar components. A dyadic tensor (often shortened to dyadic) is a sum of dyads,

$$\begin{aligned} \bar{\bar{\mathbf{Q}}} = {} & Q_{xx}\hat{\mathbf{x}}\hat{\mathbf{x}} + Q_{xy}\hat{\mathbf{x}}\hat{\mathbf{y}} + Q_{xz}\hat{\mathbf{x}}\hat{\mathbf{z}} \\ & + Q_{yx}\hat{\mathbf{y}}\hat{\mathbf{x}} + Q_{yy}\hat{\mathbf{y}}\hat{\mathbf{y}} + Q_{yz}\hat{\mathbf{y}}\hat{\mathbf{z}}. \\ & + Q_{zx}\hat{\mathbf{z}}\hat{\mathbf{x}} + Q_{zy}\hat{\mathbf{z}}\hat{\mathbf{y}} + Q_{zz}\hat{\mathbf{z}}\hat{\mathbf{z}} \end{aligned} \tag{13}$$

A dyadic has nine scalar components, or three vector components. The dyadic $\bar{\bar{\mathbf{Q}}}$ can be written

$$\bar{\bar{\mathbf{Q}}} = \vec{\mathbf{q}}_x\hat{\mathbf{x}} + \vec{\mathbf{q}}_y\hat{\mathbf{y}} + \vec{\mathbf{q}}_z\hat{\mathbf{z}}, \tag{14}$$

where the component vectors are

$$\begin{aligned}\vec{\mathbf{q}}_x &= Q_{xx}\hat{\mathbf{x}} + Q_{yx}\hat{\mathbf{y}} + Q_{zx}\hat{\mathbf{z}}\\ \vec{\mathbf{q}}_y &= Q_{xy}\hat{\mathbf{x}} + Q_{yy}\hat{\mathbf{y}} + Q_{zy}\hat{\mathbf{z}}.\\ \vec{\mathbf{q}}_z &= Q_{xz}\hat{\mathbf{x}} + Q_{yz}\hat{\mathbf{y}} + Q_{zz}\hat{\mathbf{z}}\end{aligned} \tag{15}$$

This is the particular decomposition of a dyadic into vector components that is implemented in the module. An alternate decomposition can also be used

$$\bar{\bar{\mathbf{Q}}} = \hat{\mathbf{x}}\vec{\mathbf{p}}_x + \hat{\mathbf{y}}\vec{\mathbf{p}}_y + \hat{\mathbf{z}}\vec{\mathbf{p}}_z, \tag{16}$$

where the component vectors are

$$\begin{aligned}\vec{\mathbf{p}}_x &= Q_{xx}\hat{\mathbf{x}} + Q_{xy}\hat{\mathbf{y}} + Q_{xz}\hat{\mathbf{z}}\\ \vec{\mathbf{p}}_y &= Q_{yx}\hat{\mathbf{x}} + Q_{yy}\hat{\mathbf{y}} + Q_{yz}\hat{\mathbf{z}}.\\ \vec{\mathbf{p}}_z &= Q_{zx}\hat{\mathbf{x}} + Q_{zy}\hat{\mathbf{y}} + Q_{zz}\hat{\mathbf{z}}\end{aligned} \tag{17}$$

For a nonsymmetric dyadic, $\vec{\mathbf{q}}_i \neq \vec{\mathbf{p}}_i$, while for a symmetric dyadic, $\vec{\mathbf{q}}_i = \vec{\mathbf{p}}_i$. The function `dyadic(qx, qy, qz)` uses the representation in (14) to create the dyadic $\bar{\bar{\mathbf{Q}}}$ where `qx`, `qy`, `qz` are the vectors defined by (15). Note that, if the $\vec{\mathbf{p}}_i$ are known instead, $\bar{\bar{\mathbf{Q}}}$ can be created with `trans(dyadic(px, py, pz))`, where `trans()` is the transpose operation.

Transpose of Dyadic

The transpose of a dyadic exchanges the order of the vectors in the dyads, so that

$$\begin{aligned}\bar{\bar{\mathbf{Q}}}^T &= \hat{\mathbf{x}}\vec{\mathbf{q}}_x + \hat{\mathbf{y}}\vec{\mathbf{q}}_y + \hat{\mathbf{z}}\vec{\mathbf{q}}_z\\ &= Q_{xx}\hat{\mathbf{x}}\hat{\mathbf{x}} + Q_{xy}\hat{\mathbf{y}}\hat{\mathbf{x}} + Q_{xz}\hat{\mathbf{z}}\hat{\mathbf{x}}\\ &\quad + Q_{yx}\hat{\mathbf{x}}\hat{\mathbf{y}} + Q_{yy}\hat{\mathbf{y}}\hat{\mathbf{y}} + Q_{yz}\hat{\mathbf{z}}\hat{\mathbf{y}}\\ &\quad + Q_{zx}\hat{\mathbf{x}}\hat{\mathbf{z}} + Q_{zy}\hat{\mathbf{y}}\hat{\mathbf{z}} + Q_{zz}\hat{\mathbf{z}}\hat{\mathbf{z}}\end{aligned}. \tag{18}$$

Trace of Dyadic

The trace of a dyadic is

$$\text{trace}\left(\bar{\bar{\mathbf{Q}}}\right) = \sum_{i=x,y,z} Q_{ii}, \tag{19}$$

and the trace of a dyad $\vec{\mathbf{a}}\vec{\mathbf{b}}$ is the scalar dot product between the two vectors,

$$\text{trace}\left(\vec{\mathbf{a}}\vec{\mathbf{b}}\right) = \vec{\mathbf{a}} \cdot \vec{\mathbf{b}}. \tag{20}$$

Vector Dot Product

The vector dot product is the dot product between a dyadic, $\bar{\bar{\mathbf{Q}}}$, and a vector, $\bar{\mathbf{a}}$, and is the vector given by

$$\bar{\bar{\mathbf{Q}}} \cdot \bar{\mathbf{a}} = a_x \bar{\mathbf{q}}_x + a_y \bar{\mathbf{q}}_y + a_z \bar{\mathbf{q}}_z . \tag{21}$$

Another vector dot product also exists

$$\bar{\mathbf{a}} \cdot \bar{\bar{\mathbf{Q}}} = a_x \bar{\mathbf{p}}_x + a_y \bar{\mathbf{p}}_y + a_z \bar{\mathbf{p}}_z . \tag{22}$$

From the definition of the transpose (18), it is seen that

$$\bar{\mathbf{a}} \cdot \bar{\bar{\mathbf{Q}}}^T = \bar{\bar{\mathbf{Q}}} \cdot \bar{\mathbf{a}} . \tag{23}$$

The vector dot product is not commutative, unless the dyadic is symmetric. Thus, in general, the commutator for the vector dot product is not zero

$$\bar{\mathbf{a}} \cdot \bar{\bar{\mathbf{Q}}} - \bar{\bar{\mathbf{Q}}} \cdot \bar{\mathbf{a}} \neq 0 , \tag{24}$$

but, when $\bar{\bar{\mathbf{Q}}}$ is symmetric,

$$\bar{\mathbf{a}} \cdot \bar{\bar{\mathbf{Q}}}_{symmetric} - \bar{\bar{\mathbf{Q}}}_{symmetric} \cdot \bar{\mathbf{a}} = 0 . \tag{25}$$

Note that the vector dot product is similar to the multiplication of a square matrix by a column or row matrix, except that the column matrix must be transposed to a row matrix when moved from the right side of the square matrix to the left side. Here, the similarity ends, because no transpose operation is needed for vectors in the vector-dyadic arithmetic. In fact, no transpose exists for these vectors. However, a transpose does exist for the dyadic.

Cross Product Between a Dyadic and a Vector

A cross product between a vector and a dyadic is defined

$$\begin{aligned} \bar{\bar{\mathbf{Q}}} \times \bar{\mathbf{a}} &= \hat{\mathbf{x}}(\bar{\mathbf{p}}_x \times \bar{\mathbf{a}}) + \hat{\mathbf{y}}(\bar{\mathbf{p}}_y \times \bar{\mathbf{a}}) + \hat{\mathbf{z}}(\bar{\mathbf{p}}_z \times \bar{\mathbf{a}}) \\ &= \bar{\mathbf{q}}_x(\hat{\mathbf{x}} \times \bar{\mathbf{a}}) + \bar{\mathbf{q}}_y(\hat{\mathbf{y}} \times \bar{\mathbf{a}}) + \bar{\mathbf{q}}_z(\hat{\mathbf{z}} \times \bar{\mathbf{a}}) \\ &= (\bar{\mathbf{q}}_y a_z - \bar{\mathbf{q}}_z a_y)\hat{\mathbf{x}} + (\bar{\mathbf{q}}_z a_x - \bar{\mathbf{q}}_x a_z)\hat{\mathbf{y}} + (\bar{\mathbf{q}}_x a_y - \bar{\mathbf{q}}_y a_x)\hat{\mathbf{z}} \end{aligned} , \tag{26}$$

when the vector is on the right or

$$\begin{aligned} \bar{\mathbf{a}} \times \bar{\bar{\mathbf{Q}}} &= (\bar{\mathbf{a}} \times \bar{\mathbf{q}}_x)\hat{\mathbf{x}} + (\bar{\mathbf{a}} \times \bar{\mathbf{q}}_y)\hat{\mathbf{y}} + (\bar{\mathbf{a}} \times \bar{\mathbf{q}}_z)\hat{\mathbf{z}} \\ &= (\bar{\mathbf{a}} \times \hat{\mathbf{x}})\bar{\mathbf{p}}_x + (\bar{\mathbf{a}} \times \hat{\mathbf{y}})\bar{\mathbf{p}}_y + (\bar{\mathbf{a}} \times \hat{\mathbf{z}})\bar{\mathbf{p}}_z \\ &= \hat{\mathbf{x}}(a_y \bar{\mathbf{p}}_z - a_z \bar{\mathbf{p}}_y) + \hat{\mathbf{y}}(a_z \bar{\mathbf{p}}_x - a_x \bar{\mathbf{p}}_z) + \hat{\mathbf{z}}(a_x \bar{\mathbf{p}}_y - a_y \bar{\mathbf{p}}_x) \end{aligned} , \tag{27}$$

when the vector is on the left. In general, the cross product between a vector and a dyadic is not commutative

$$\bar{\bar{\mathbf{Q}}} \times \bar{\mathbf{a}} - \bar{\mathbf{a}} \times \bar{\bar{\mathbf{Q}}} \neq 0 . \tag{28}$$

If the dyadic is symmetric, the cross product is anticommutative

$$\bar{\bar{\mathbf{Q}}}_{symmetric} \times \bar{\mathbf{a}} + \bar{\mathbf{a}} \times \bar{\bar{\mathbf{Q}}}_{symmetric} = 0, \tag{29}$$

while if the dyadic is antisymmetric ($\bar{\mathbf{q}}_i = -\bar{\mathbf{p}}_i$), the cross product is commutative

$$\bar{\bar{\mathbf{Q}}}_{antisymmetric} \times \bar{\mathbf{a}} - \bar{\mathbf{a}} \times \bar{\bar{\mathbf{Q}}}_{antisymmetric} = 0. \tag{30}$$

Note that the diagonal elements, Q_{ii}, must be zero if the dyadic is antisymmetric.

Associativity of Products of a Dyadic and a Vector

The vector dot product and the cross product between a vector and a dyadic form an associative pair of operators. Consider

$$\begin{aligned}
\bar{\mathbf{a}} \cdot \left(\bar{\bar{\mathbf{Q}}} \times \bar{\mathbf{b}}\right) &= \bar{\mathbf{a}} \cdot \left[\left(\bar{\mathbf{q}}_x \hat{\mathbf{x}} \times \bar{\mathbf{b}}\right) + \left(\bar{\mathbf{q}}_y \hat{\mathbf{y}} \times \bar{\mathbf{b}}\right) + \left(\bar{\mathbf{q}}_z \hat{\mathbf{z}} \times \bar{\mathbf{b}}\right)\right] \\
&= \left(\bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_x \hat{\mathbf{x}} \times \bar{\mathbf{b}}\right) + \left(\bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_y \hat{\mathbf{y}} \times \bar{\mathbf{b}}\right) + \left(\bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_z \hat{\mathbf{z}} \times \bar{\mathbf{b}}\right) \\
&= \left(\bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_x \hat{\mathbf{x}} + \bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_y \hat{\mathbf{y}} + \bar{\mathbf{a}} \cdot \bar{\mathbf{q}}_z \hat{\mathbf{z}}\right) \times \bar{\mathbf{b}} \\
&= \left[\bar{\mathbf{a}} \cdot \left(\bar{\mathbf{q}}_x \hat{\mathbf{x}} + \bar{\mathbf{q}}_y \hat{\mathbf{y}} + \bar{\mathbf{q}}_z \hat{\mathbf{z}}\right)\right] \times \bar{\mathbf{b}} \\
&= \left(\bar{\mathbf{a}} \cdot \bar{\bar{\mathbf{Q}}}\right) \times \bar{\mathbf{b}}
\end{aligned} \tag{31}$$

In this case, the precedence rules do not interfere with the proper evaluation of the code `a*Q.X.b`, since either order will be valid and will give the same result. Similarly,

$$\bar{\mathbf{a}} \times \left(\bar{\bar{\mathbf{Q}}} \cdot \bar{\mathbf{b}}\right) = \left(\bar{\mathbf{a}} \times \bar{\bar{\mathbf{Q}}}\right) \cdot \bar{\mathbf{b}}\ . \tag{32}$$

Dyadic Dot Product

Given two dyadics,

$$\bar{\bar{\mathbf{Q}}} = \bar{\mathbf{q}}_x \hat{\mathbf{x}} + \bar{\mathbf{q}}_y \hat{\mathbf{y}} + \bar{\mathbf{q}}_z \hat{\mathbf{z}},$$

and

$$\bar{\bar{\mathbf{S}}} = \bar{\mathbf{s}}_x \hat{\mathbf{x}} + \bar{\mathbf{s}}_y \hat{\mathbf{y}} + \bar{\mathbf{s}}_z \hat{\mathbf{z}},$$

the dyadic dot product between $\bar{\bar{\mathbf{Q}}}$ and $\bar{\bar{\mathbf{S}}}$ is a dyadic given by

$$\begin{aligned}
\bar{\bar{\mathbf{Q}}} \cdot \bar{\bar{\mathbf{S}}} &= \left(\bar{\mathbf{q}}_x \hat{\mathbf{x}} + \bar{\mathbf{q}}_y \hat{\mathbf{y}} + \bar{\mathbf{q}}_z \hat{\mathbf{z}}\right) \cdot \left(\bar{\mathbf{s}}_x \hat{\mathbf{x}} + \bar{\mathbf{s}}_y \hat{\mathbf{y}} + \bar{\mathbf{s}}_z \hat{\mathbf{z}}\right) \\
&= \bar{\mathbf{q}}_x \left[\left(\bar{\mathbf{s}}_x \cdot \hat{\mathbf{x}}\right)\hat{\mathbf{x}} + \left(\bar{\mathbf{s}}_y \cdot \hat{\mathbf{x}}\right)\hat{\mathbf{y}} + \left(\bar{\mathbf{s}}_z \cdot \hat{\mathbf{x}}\right)\hat{\mathbf{z}}\right] \\
&+ \bar{\mathbf{q}}_y \left[\left(\bar{\mathbf{s}}_x \cdot \hat{\mathbf{y}}\right)\hat{\mathbf{x}} + \left(\bar{\mathbf{s}}_y \cdot \hat{\mathbf{y}}\right)\hat{\mathbf{y}} + \left(\bar{\mathbf{s}}_z \cdot \hat{\mathbf{y}}\right)\hat{\mathbf{z}}\right] \\
&+ \bar{\mathbf{q}}_z \left[\left(\bar{\mathbf{s}}_x \cdot \hat{\mathbf{z}}\right)\hat{\mathbf{x}} + \left(\bar{\mathbf{s}}_y \cdot \hat{\mathbf{z}}\right)\hat{\mathbf{y}} + \left(\bar{\mathbf{s}}_z \cdot \hat{\mathbf{z}}\right)\hat{\mathbf{z}}\right]
\end{aligned} \tag{33}$$

If the alternate representation of $\bar{\bar{\mathbf{S}}}$ is used,

$$\bar{\bar{S}} = \hat{x}\bar{\sigma}_x + \hat{y}\bar{\sigma}_y + \hat{z}\bar{\sigma}_z ,$$

then the dot product is written simply

$$\begin{aligned} \bar{\bar{Q}} \cdot \bar{\bar{S}} &= \left(\bar{q}_x\hat{x} + \bar{q}_y\hat{y} + \bar{q}_z\hat{z}\right) \cdot \left(\hat{x}\bar{\sigma}_x + \hat{y}\bar{\sigma}_y + \hat{z}\bar{\sigma}_z\right) \\ &= \bar{q}_x\bar{\sigma}_x + \bar{q}_y\bar{\sigma}_y + \bar{q}_z\bar{\sigma}_z \end{aligned} . \tag{34}$$

Since

$$\begin{aligned} \bar{\bar{S}} \cdot \bar{\bar{Q}} &= \left(\bar{s}_x\hat{x} + \bar{s}_y\hat{y} + \bar{s}_z\hat{z}\right) \cdot \left(\bar{q}_x\hat{x} + \bar{q}_y\hat{y} + \bar{q}_z\hat{z}\right) \\ &= \bar{s}_x\left[\left(\bar{q}_x \cdot \hat{x}\right)\hat{x} + \left(\bar{q}_y \cdot \hat{x}\right)\hat{y} + \left(\bar{q}_z \cdot \hat{x}\right)\hat{z}\right] \\ &+ \bar{s}_y\left[\left(\bar{q}_x \cdot \hat{y}\right)\hat{x} + \left(\bar{q}_y \cdot \hat{y}\right)\hat{y} + \left(\bar{q}_z \cdot \hat{y}\right)\hat{z}\right] \\ &+ \bar{s}_z\left[\left(\bar{q}_x \cdot \hat{z}\right)\hat{x} + \left(\bar{q}_y \cdot \hat{z}\right)\hat{y} + \left(\bar{q}_z \cdot \hat{z}\right)\hat{z}\right] \end{aligned} , \tag{35}$$

we see that the commutator is not zero,

$$\bar{\bar{Q}} \cdot \bar{\bar{S}} - \bar{\bar{S}} \cdot \bar{\bar{Q}} \neq 0 . \tag{36}$$

Eigen Values and Eigen Vectors

The eigen values associated with a dyadic $\bar{\bar{A}}$ are given by λ_i, such that

$$\left|\bar{\bar{A}} - \lambda_i\bar{\bar{I}}\right| \equiv 0 , \tag{37}$$

where $\bar{\bar{I}} = \hat{x}\hat{x} + \hat{y}\hat{y} + \hat{z}\hat{z}$ is the identity dyadic. The eigen vectors are the unit vectors, $\hat{e}_i$, such that

$$\left(\bar{\bar{A}} - \lambda_i\bar{\bar{I}}\right) \cdot \hat{e}_i = \bar{\bar{B}} \cdot \hat{e}_i \equiv 0 . \tag{38}$$

The eigen value equation leads to a cubic equation for the eigen values, λ_i. The solutions can be real or complex. Note that if $\bar{\bar{A}}$ is a real dyadic, then at least one of the eigen values is real. However, since $\bar{\bar{A}}$ is not constrained to be real, all of the λ_i may be complex. A method of obtaining the roots of the cubic equation with complex coefficients is described in Appendix A, which documents the module `roots`.

Consider

$$\bar{\bar{B}} = \bar{b}_x\hat{x} + \bar{b}_y\hat{y} + \bar{b}_z\hat{z} , \tag{39}$$

for which

$$\left|\bar{\bar{B}}\right| = 0 . \tag{40}$$

The eigen-vector equation is

$$\begin{aligned}\bar{\bar{\mathbf{B}}}\cdot\hat{\mathbf{e}} = 0 &= \vec{\mathbf{b}}_x\hat{\mathbf{x}}\cdot\hat{\mathbf{e}} + \vec{\mathbf{b}}_y\hat{\mathbf{y}}\cdot\hat{\mathbf{e}} + \vec{\mathbf{b}}_z\hat{\mathbf{z}}\cdot\hat{\mathbf{e}} \\ &= \hat{\mathbf{x}}\vec{\beta}_x\cdot\hat{\mathbf{e}} + \hat{\mathbf{y}}\vec{\beta}_y\cdot\hat{\mathbf{e}} + \hat{\mathbf{z}}\vec{\beta}_z\cdot\hat{\mathbf{e}}\end{aligned} \tag{41}$$

where

$$\begin{aligned}\vec{\beta}_x &= \hat{\mathbf{x}}\cdot\bar{\bar{\mathbf{B}}} \\ \vec{\beta}_y &= \hat{\mathbf{y}}\cdot\bar{\bar{\mathbf{B}}}. \\ \vec{\beta}_z &= \hat{\mathbf{z}}\cdot\bar{\bar{\mathbf{B}}}\end{aligned} \tag{42}$$

We can expand the determinant as follows

$$\begin{aligned}\left|\bar{\bar{\mathbf{B}}}\right| &= \begin{vmatrix}\beta_{x,x} & \beta_{x,y} & \beta_{x,z} \\ \beta_{y,x} & \beta_{y,y} & \beta_{y,z} \\ \beta_{z,x} & \beta_{z,y} & \beta_{z,z}\end{vmatrix} \\ &= \beta_{x,x}\left(\beta_{y,y}\beta_{z,z} - \beta_{y,z}\beta_{z,y}\right) - \beta_{x,y}\left(\beta_{y,x}\beta_{z,z} - \beta_{y,z}\beta_{z,x}\right) + \beta_{x,z}\left(\beta_{y,x}\beta_{z,y} - \beta_{y,y}\beta_{z,x}\right) \\ &= \vec{\beta}_x\cdot\left(\vec{\beta}_y\times\vec{\beta}_z\right) \\ &= -\beta_{y,x}\left(\beta_{x,y}\beta_{z,z} - \beta_{x,z}\beta_{z,y}\right) + \beta_{y,y}\left(\beta_{x,x}\beta_{z,z} - \beta_{x,z}\beta_{z,x}\right) - \beta_{y,z}\left(\beta_{x,x}\beta_{z,y} - \beta_{x,y}\beta_{z,x}\right). \\ &= -\vec{\beta}_y\cdot\left(\vec{\beta}_x\times\vec{\beta}_z\right) \\ &= \beta_{z,x}\left(\beta_{x,y}\beta_{y,z} - \beta_{x,z}\beta_{y,y}\right) - \beta_{z,y}\left(\beta_{x,x}\beta_{y,z} - \beta_{x,z}\beta_{y,x}\right) + \beta_{z,z}\left(\beta_{x,x}\beta_{y,y} - \beta_{x,y}\beta_{y,x}\right) \\ &= \vec{\beta}_z\cdot\left(\vec{\beta}_x\times\vec{\beta}_y\right)\end{aligned} \tag{43}$$

Since $\left|\bar{\bar{\mathbf{B}}}\right| = 0$, we see immediately that the eigen vector can be represented with either

$$\vec{\mathbf{e}} = \vec{\beta}_x\times\vec{\beta}_y, \tag{44}$$

or

$$\vec{\mathbf{e}} = \vec{\beta}_x\times\vec{\beta}_z, \tag{45}$$

or

$$\vec{\mathbf{e}} = \vec{\beta}_y\times\vec{\beta}_z. \tag{46}$$

Also, since $\left|\bar{\bar{\mathbf{B}}}\right| = 0$, the three vectors $\vec{\beta}_i$ contained in $\bar{\bar{\mathbf{B}}}$ are not independent. In fact, one of the vectors, $\vec{\beta}_i$, must be a linear combination of the other two

$$\vec{\beta}_i = A\vec{\beta}_j + B\vec{\beta}_k, \tag{47}$$

where i, j, and k, represent any ordering of x, y, and z. Thus, we see that the three cross products are all parallel

$$\vec{\beta}_k \times \vec{\beta}_i = A\vec{\beta}_k \times \vec{\beta}_j, \tag{48}$$

and

$$\vec{\beta}_j \times \vec{\beta}_i = B\vec{\beta}_j \times \vec{\beta}_k. \tag{49}$$

Any of the three cross products can be used for the eigen vector, provided, of course, that A or B is not zero. This amounts to saying that none of the $\vec{\beta}_i$ are parallel to each other. If a pair of the $\vec{\beta}_i$ happen to be parallel, then a cross product containing the other $\vec{\beta}_i$ must be used. If all three of the $\vec{\beta}_i$ are parallel, then any vector perpendicular to $\vec{\beta}_i$ can be used for the eigen vector. Note that since the $\vec{\beta}_i$ are dependent, they must all lie in the same plane. Thus, since a plane has only one normal direction (to within a constant factor), all of the cross products produce the single unique solution to (41), to within a constant factor.

This Page Intentionally Blank

Appendix A: Solution of Cubic and Quartic Equations with Complex Coefficients

Module `roots` contains procedures which solve for the roots of cubic and quartic equations with complex coefficients. These routines are implemented only with double precision arithmetic, in order to preserve the most precision in the roots. Subroutine `cubic(a, b, c, z)` computes the three complex roots of

$$z^3 + az^2 + bz + c = 0,$$

while subroutine `quartic(a, b, c, d, z)` computes the four complex roots of

$$z^4 + az^3 + bz^2 + cz + d = 0.$$

The routines assume that the coefficients are complex, so the coefficients (`a, b, c, d`) must be defined as double-precision complex variables. Both subroutines return the roots in a double-precision complex array, `z`, which must have at least 3 elements for a call to `cubic()` and at least 4 elements for a call to `quartic()`. Note that subroutine `quartic()` is not used by module `vectors`. The methods implemented by the routines are described below.

Cubic Equation

The solution of the cubic equation of the form

$$x^3 + ax^2 + bx + c = 0 \tag{A-1}$$

when the coefficients *a, b, c* are real numbers has been known since the 16th century. The solution is attributed to Geronimo Cardano who published the solution in 1545. However, the solution did not originate with Cardano. He obtained it from Niccolo Tartaglia in 1539, but the solution is thought to have been first achieved by Scipione del Ferro around 1500. However, since Cardano was the first to publish it, the solution is known as Cardano's formula [1]. The solution is published in many references [2, 3, 4, 5,6], but is valid as published only for real coefficients. Here, the solution is simply extended to accommodate complex coefficients. The extension is simple and straightforward.

The solution proceeds as follows. Let $y = x - a/3$ so that

$$y^3 + py + q = 0, \tag{A-2}$$

where

$$p = b - \frac{a^2}{3},$$

and

$$q = c - \frac{ab}{3} + \frac{2a^3}{27}.$$

Now, let $y = u + v$ so that

$$u^3 + v^3 + (u+v)(p+3uv) + q = 0. \tag{A-3}$$

Since there are now two unknowns, a second constraint is needed. We are free to choose the constraint in any way to expedite the solution, with the understanding that spurious roots may arise. Cardano's formula is based on the following constraints

$$u^3 + v^3 + q = 0, \tag{A-4}$$

and

$$(p+3uv) = 0. \tag{A-5}$$

Eliminating v results in

$$u^6 + qu^3 - \frac{p^3}{27} = 0. \tag{A-6}$$

Elimination of u results in the same equation with u replaced by v. Equation (A-6) is quadratic in u^3, and can be solved easily. When p is very small ($p \sim 0$), loss of precision in the computation of one of the roots of a quadratic can occur, unless care is taken. In order to preserve the most precision in the roots, we use [7]

$$s = -\frac{1}{2}\left(q + \operatorname{sgn}(q)\sqrt{q^2 + 4\frac{p^3}{27}}\right), \tag{A-7}$$

$$u_k = \sqrt[3]{s}\zeta_k;\ \ k = 0,1,2, \tag{A-8}$$

and

$$v_k = \begin{cases} \sqrt[3]{\dfrac{-p^3}{27s}}\zeta_k;\ \ k = 0,1,2 \ \text{ when } s \neq 0 \\ \sqrt[3]{-\dfrac{1}{2}\left(q - \operatorname{sgn}(q)\sqrt{q^2 + 4\dfrac{p^3}{27}}\right)}\zeta_k;\ \ k = 0,1,2 \ \text{ when } s = 0 \end{cases}, \tag{A-9}$$

where $\zeta_k = e^{jk2\pi/3}$ are the three cube roots of 1, and

$$\operatorname{sgn}(z) = \begin{cases} 1;\ |z| = 0 \\ \dfrac{z}{\sqrt{z^2}};\ |z| \neq 0 \end{cases}, \tag{A-10}$$

where $\sqrt{\ }$ represents the principal square root.

Since there are three cube roots, there are nine possible solutions for y,

$$y_{k\ell} = u_k + v_\ell;\ \ k,\ell = 0,1,2\,. \tag{A-11}$$

However, there are only three valid solutions for y, so the spurious solutions must be detected and rejected. Principal roots u, v are found so that

$$\left[\arg(u) + \arg(v)\right]_{\mathrm{mod}(2\pi)} = \left[\arg(-p)\right]_{\mathrm{mod}(2\pi)}, \tag{A-12}$$

which assures that (A-5) is satisfied. Then, noting that $\zeta_1\zeta_2 = 1$, the three roots of the original cubic equation (A-1) are

$$x_0 = u + v - \frac{a}{3},$$

$$x_1 = ue^{j2\pi/3} + ve^{j4\pi/3} - \frac{a}{3},$$

and

$$x_2 = ue^{j4\pi/3} + ve^{j2\pi/3} - \frac{a}{3}.$$

When the coefficients are real, at least one of the roots, x_0, is real and (A-12) is satisfied with $k = 0$ in (A-8) and (A-9). All that is necessary to extend Cardano's formula to the case of complex coefficients is to define the principal roots, u, v, so that (A-12) is satisfied. This requires an additional step during the computation of the roots. The principal roots are

$$u = u_0, \tag{A-13}$$

$$v = v_\ell, \tag{A-14}$$

where

$$\ell = \left\langle \frac{\left[\arg(-p) - \arg(u_0) - \arg(v_0)\right]_{\mathrm{mod}(2\pi)}}{2\pi/3} \right\rangle, \tag{A-15}$$

where $\langle x \rangle$ means the nearest integer to x.

When $\frac{4p^3}{27} \sim -q^2$, one might expect that expanding the term

$$q^2 + \frac{4p^3}{27} = c^2 - \frac{2abc}{2} + \frac{4a^3b}{27} - \frac{a^2b^2}{27} + \frac{b^3}{27}$$

would allow some improvement in numeric precision, since the a^6 and a^4b terms cancel. However, the right-hand side requires differencing terms with larger magnitude than the terms

on the left-hand side. Thus, precision will be lost if the expression on the right is used when $\frac{4p^3}{27} \sim -q^2$. In this case, the simpler expression on the left is more accurate.

Some increase in precision is obtained by refining the roots with one stage of Newton's method

$$x_{k,n+1} = x_{k,n} - \frac{c + x_{k,n}\left(b + x_{k,n}\left(a + x_{k,n}\right)\right)}{b + x_{k,n}\left(2a + 3x_{k,n}\right)}. \tag{A-16}$$

Note that the polynomials are computed by factoring x from successive terms. This is done to preserve numerical precision. Negligible improvement is obtained with a second stage of refinement.

Quartic Equation

The solution for the quartic equation,

$$x^4 + ax^3 + bx^2 + cx + d = 0, \tag{A-17}$$

was also obtained in the early part of the 16th century. It was found by L. Ferrari, a student of Cardano, and subsequently published by Cardano [1]. The roots of (A-17) are

$$x_k = -\frac{a}{4} + \alpha_1 \frac{R}{2} + \alpha_2 \frac{S(\alpha_1)}{2}, \tag{A-18}$$

where

$$R = \sqrt{\frac{a^2}{4} - b - y},$$

$$S = \begin{cases} \sqrt{\frac{3a^2}{4} - 2b + \alpha_1 \frac{4ab - 8c - a^3}{4R}}; & R \neq 0 \\ \sqrt{\frac{3a^2}{4} - 2b + \alpha_1 2\sqrt{y^2 - 4d}}; & R = 0 \end{cases},$$

$$\alpha_1 = \pm 1,$$

$$\alpha_2 = \pm 1,$$

and y is any solution of the cubic equation [5]

$$y^3 - by^2 + (ac - 4d)y - a^2 d + 4bd - c^2 = 0. \tag{A-19}$$

Note that α_1 and α_2 are uncorrelated. This solution is valid when the coefficients are complex.

Improved numerical precision is obtained with two stages of root refinement using Newton's method

$$x_{k,n+1} = x_{k,n} - \frac{d + x_{k,n}\left(c + x_{k,n}\left(b + x_{k,n}\left(a + x_{k,n}\right)\right)\right)}{c + x_{k,n}\left(2b + x_{k,n}\left(3a + 4x_{k,n}\right)\right)}. \tag{A-20}$$

References

[1] W. Gellert, H. Küster, M. Hellwich, H. Kästner, ed. **The VNR Concise Encyclopedia of Mathematics**, Van Nostrand Reinhold Company, New York, 1977.

[2] M. Abramowitz, I. A. Stegun, **Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables**, Dover Publications, Inc., New York, ninth printing, 1970.

[3] G. A. Korn, T. M. Korn, **Mathematical Handbook for Scientists and Engineers**, 2nd edition, McGraw-Hill Book Company, New York, 1968.

[4] C. E. Pearson, **Handbook of Applied Mathematics**, 2nd edition, Van Nostrand Reinhold Company, New York, 1983.

[5] W. H. Beyer, **CRC Standard Math Tables**, 25th edition, CRC Press, Inc., West Palm Beach, 1978

[6] W. H. Beyer, **CRC Handbook of Mathematical Sciences**, 5th edition, CRC Press, Inc., West Palm Beach, 1978.

[7] W. H. Press, B. P. Flannery, S. A. Teukolsky, W. T. Vetterling, **Numerical Recipes, The Art of Scientific Computing**, Cambridge University Press, New York, 1986.

Distribution:

1	Gerardo Aguirre Texas Instruments 13532 N. Central Expressway P.O. Box 655012 MS477 Dallas Texas 79265	
1	MS 0529	A. W. Doerry, 2345
1	MS 0529	B. C. Walker, 2345
1	MS-0529	M. B. Murphy, 2346
1	MS-0531	D. L. Bickel, 2344
1	MS-0531	R. M. Axline, 2344
1	MS 0531	J. T. Cordaro, 2344
1	MS-0531	W. H. Hensley, 2344
1	MS 0531	D. A. Jelinek, 2344
1	MS 0533	S. E. Allen, 2343
10	MS 0533	B. C. Brock, 2343
1	MS 0533	W. E. Patitz, 2343
1	MS 0533	W. H. Schaedla, 2343
1	MS 0533	K. W. Sorensen, 2343
1	MS-0865	W. A. Johnson, 9753
1	MS-0865	L. K. Warne, 9753
1	MS-1166	J. D. Kotulski, 9352
1	MS-1166	D. J. Riley, 9352
1	MS-1166	D. Turner, 9352
1	MS-1328	G. K. Froehlich, 6849
1	MS 9018	Central Technical Files, 8523-2
5	MS 0899	Technical Library, 4414
2	MS 0619	Review & Approval Desk, 12630 For DOE/OSTI

This Page Intentionally Blank

Software License Information

Export Control Classification Number: EAR99