User Documentation for *vecmat3* Vector and Matrix classes

Ramses van Zon*

Chemical Physics Theory Group, Department of Chemistry, University of Toronto,

80 St. George Street, Toronto, Ontario M5S 3H6, Canada

May 7, 2007

Abstract

This document describes how to use the c++ Vector and Matrix classes defined in the header file vecmat3.h. These classes offer a very convenient notation for three dimensional vector and matrix algebra by using overloaded operators. Furthermore, they are constructed to be numerically efficient through the internal use of template expressions, which is not visible at the user level. As a result, one can convert mathematical matrix-vector expressions straightforwardly into the corresponding c++ expression without having to worry about incurring an efficiency penalty.

*rzon@chem.utoronto.ca

Contents

1 Introduction **3**

2 Using the Vector and Matrix classes **4**

2.1 Header file 4
2.2 Initialization methods 4
2.2.1 Initialization through constructor parameters 4
2.2.2 Initialization through assignment 5
2.2.3 Initialization through a comma separated list 5
2.2.4 Initialization through member functions 6
2.2.5 Arrays 6
2.3 Accessing the elements 7
2.4 Operators 7
2.5 Member functions 7
2.5.1 DOUBLE nrm2() 8
2.5.2 DOUBLE nrm() 9
2.5.3 DOUBLE tr() 9
2.5.4 DOUBLE det() 9
2.5.5 Vector row(int i) 9
2.5.6 Vector column(int j) 9
2.6 Non-member functions 9
2.6.1 Matrix Transpose(const Matrix & M) 9
2.6.2 Matrix Inverse(const Matrix & M) 10
2.6.3 Matrix Rodrigues(const Vector & v) 10
2.6.4 Matrix Dyadic(const Vector & a, const Vector & b) 10
2.6.5 Vector MTVmult(const Matrix & M, const Vector & v) 10
2.6.6 DOUBLE dist(const Vector & a, const Vector & b) 10
2.6.7 DOUBLE dist2(const Vector & a, const Vector & b) 10
2.6.8 DOUBLE distwithshift(const Vector&a, const Vector&b, const Vector&s) 10
2.6.9 DOUBLE dotProduct(const Vector & a, const Vector & b) 11
2.6.10 DOUBLE crossProduct(const Vector & a, const Vector & b) 11

3 Expressions **11**

Background references **11**

1 Introduction

Vectors and matrices are used frequently in scientific computation (as well as in modeling, games and movie rendering). Unfortunately, no built-in support for matrices and vectors exists c++. Now in principle, expressions involving matrices and vectors, such as

$$\vec{a} = \vec{b} + \mathsf{M} \cdot \vec{c}$$

(with $\vec{a}$, $\vec{b}$ and $\vec{c}$ vectors and M a matrix) can be implemented in c++ such that they strongly resemble their mathematical notations, e.g.

```
Vector a,b;

Matrix M;

Vector c = a + M*b;
```

The technique used to accomplish such notational convenience is operator overloading, whose straightforward implementation comes with a high computational cost due to the creation of temporary objects.

A more efficient implementation is possible by using *expression templates*. Efficient matrix-vector implementations are somewhat of a by-product of c++ templates, and this shows in the awkward and complicated notation needed for general matrix-vector manipulations. In addition, the c++ standard is somewhat quirky on what is and is not allowed when using templates.

These notational issues probably explain why there are far fewer template-based implementations of matrices and vectors available. This is especially problematic for small vectors and matrices of fixed size, for instance three-dimensional ones. These allow additional efficiency gains over general-size vectors and matrices (because loops over indices can be replaced by explicit sums in the implementation). Two known implementations are the TinyVector and TinyMatrix classes of Blitz++ and the ones by the same name of tvmet. The former is not very developed, i.e., many operations that one would like to have are not present, and indeed, the latter is aimed at fixing that. Still, tvmet lacks some functionality that built-in types do have, for instance, TinyVector¡3,double¿ v = a+b; is not possible.

This is where *vecmat3* comes in. It defines very efficient three dimensional vector and matrix manipulations. The aim is to be able to use these vectors and matrices as if they were built-in types, with which the same kind of expressions can be formed as can with built-in types without worrying about template techniques, but also without substantial losses compared to hard-coded element-by-element techniques.

vecmat3 uses expression templates and operator overloading. The restrictions of vecmat3 at present are that the elements of the vectors and matrices have to be of a single type, which is double by default, and that only three-dimensional quantities are supported (as the name suggests).

2 Using the Vector and Matrix classes

2.1 Header file

To use *vecmat3*, the following general procedure should be followed:

- If the elements of the vectors and matrices are to have a different type than double, first #define their type as DOUBLE, e.g.

  ```
  #define DOUBLE float
  ```

- Include the header file vecmat3.h:

  ```
  #include "vecmat3.h"
  ```

- The class Vector and the class Matrix are now defined and instances these classes can be declared as follows:

  ```
  Vector a;
  Matrix R;
  ```

  The elements are still unspecified, and likely contain garbage. Next it will be explained how to initialize these classes.

2.2 Initialization methods

There are four ways to initialize a Vector or Matrix, which we will discuss by example:

2.2.1 Initialization through constructor parameters

Example:

```
Vector a(1.1, 3.0, -4.3);
Matrix R(1, 2, 3,
         4, 5, 6,
         7, 8, 9);
```

defines a Vector a and Matrix R with specified elements. Note that the first set of three elements given to R comprise the top row of R, the second set of three the middle row and the last set of three the bottom row.

If fewer than three or nine (for Vector and Matrix, respectively) number are given, the unspecified elements are set to zero. Thus, one can define a zero Vector and Matrix simply by

```
Vector a(0);
Matrix R(0);
```

2.2.2 Initialization through assignment

Example:

```
Vector b = a;
Matrix S = R;
```

defines a Vector b with the same elements as a, and a Matrix S with the elements as R.

The right hand sides may also be an expression involving Vector's and Matrices. The allowed expressions are explained in sections 2.4 and 3.

2.2.3 Initialization through a comma separated list

Example:

```
Vector a;
a = 1.1, 3.0, -4.3;
Matrix R;
R = 1, 2, 3,
    4, 5, 6,
    7, 8, 9;
```

Note: this is the standard construction for Blitz++ and tvmet, and is achieved though an overloaded comma operator. Not everybody likes overloading the comma operator, because it may confuse the user (more than the above methods), and it is somewhat less efficient than method 1.

Furthermore, it is not possible to use this method in the declaration, i.e., one cannot write Vector a=1.1,3.0,-4.3; since c++ would consider this a declaration of 3.0 and -4.3 as being of type Vector.

If not enough elements are given in the list, the remaining elements are set to zero. Thus, one can write

```
b = 1;
```

to get the vector (1,0,0), and

```
R = 2;
```

to get the matrix $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$.

2.2.4 Initialization through member functions

Example:

```
Vector a;
a.zero();
Matrix R;
R.zero();
```

also define a Vector and Matrix, respectively, with zero elements.
For a Matrix, there also is a member function `one()` to turn it into an identity matrix:

```
Matrix S;
S.one();
```

Furthermore, one can initialize a Matrix also per row or column, e.g.

```
R.setRow(0,a);
R.setRow(1,Vector(0,2,0));
R.setRow(2,Vector(0,2,-1));
S.setColumn(0,a);
S.setColumn(1,Vector(0,2,0));
S.setColumn(2,Vector(0,2,-1));
```

Note that rows and columns are numbered from 0 to 2.

2.2.5 Arrays

Example:

```
Vector a[3];
Matrix S[3];
```

Defines arrays of three Vectors and Matrices which are non-initialized. One can initialize these arrays as follows

```
Vector a[3] = { Vector(1,2,3), Vector(3,4,5), Vector(5,6,7) };
Matrix S[3] = { Matrix(0), Matrix(1,2,3,4,5,6,7,8,9), Matrix(2) } ;
```

2.3 Accessing the elements

Basic elements of Vectors and Matrices are generally access using the parenthesis notation, i.e., the elements of a Vector a are a(0), a(1) and a(2), while those of a Matrix R are R(0,0), R(0,1), ... R(2,2).

An alternative way to access the elements is though the class members themselves, i.e., x, y and z for Vector, and xx, xy, xz, yx, yy, yz, zx, zy and zz for Matrix. This is potentially more efficient, but cannot be used for expressions, i.e., (A+B).xx is not possible, unless one writes Matrix(A+B).xx.

Furthermore, the rows and columns of a Matrix can be used as if they were vectors as follows

```
Vector v = R.row(0);

Vector w = R.column(2);
```

2.4 Operators

The available algebraic operators for the Vector and Matrix classes are summarized in table 1, in which 'Vector' stands for 'const Vector &' or a Vector-valued expression, and 'Matrix' stands for 'const Matrix &' or a Matrix-valued expression.

In addition, << operators are defined for output of Vectors and Matrices to 'ostream's, such that

```
Vector a(1,2,3);

std::cout << a << endl;
```

would print the numbers 1, 2 and 3 with just a space in between.

```
Matrix M(1,2,3,4,5,6,7,8,9);

std::cout << M << endl;
```

would print a newline, the numbers 1, 2 and 3, another newline, the numbers 4, 5 and 6, another newline, the numbers 7, 8 and 9 and finally another newline.

2.5 Member functions

For any Vector, or Vector-valued expression, or for any Matrix, or Matrix-valued expression, the following properties are available as member functions:

form	description	example	mathematically
- Vector	negative	`c = -a;`	$\vec{c} = -\vec{a}$
Vector + Vector	add	`c = a + b;`	$\vec{c} = \vec{a}+\vec{b}$
Vector - Vector	subtract	`c = a - b;`	$\vec{c} = \vec{a}-\vec{b}$
DOUBLE * Vector	multiply with scalar	`c = d * a;`	$\vec{c} = d\vec{a}$
Vector * DOUBLE	multiply with scalar	`c = a * d;`	$\vec{c} = \vec{a}d$
Vector / DOUBLE	divide by scalar	`c = a / d;`	$\vec{c} = \vec{a}/d$
Vector ^ Vector	cross/outer product[†]	`c = a ^ b;`	$\vec{c} = \vec{a}\times\vec{b}$
Vector * Vector	dot/inner product[‡]	`d = a * b;`	$d = \vec{a}\cdot\vec{b}$
(Vector | Vector)	dot/inner product[‡]	`d = (a|b);`	$d = \vec{a}\cdot\vec{b}$
- Matrix	negative	`T = -S;`	$\mathsf{T} = -\mathsf{S}$
Matrix + Matrix	add	`T = S + R;`	$\mathsf{T} = \mathsf{S}+\mathsf{R}$
Matrix - Matrix	subtract	`T = S - R;`	$\mathsf{T} = \mathsf{S}-\mathsf{R}$
DOUBLE * Matrix	multiply with scalar	`T = d * S;`	$\mathsf{T} = d\mathsf{S}$
Matrix * DOUBLE	multiply with scalar	`T = S * d;`	$\mathsf{T} = \mathsf{S}d$
Matrix / DOUBLE	divide by scalar	`T = S / d;`	$\mathsf{T} = \mathsf{S}/d$
Matrix * Matrix	matrix-matrix product	`T = S * R;`	$\mathsf{T} = \mathsf{S}\mathsf{R}$
Matrix * Vector	matrix-vector product	`c = S * a;`	$\vec{c} = \mathsf{S}\vec{a}$
Vector += Vector	add	`c += b;`	$\vec{c} = \vec{c}+\vec{b}$
Vector -= Vector	subtract	`c -= b;`	$\vec{c} = \vec{c}-\vec{b}$
Vector *= DOUBLE	multiply by scalar	`c *= d;`	$\vec{c} = d\vec{c}$
Vector /= DOUBLE	divide by scalar	`c /= d;`	$\vec{c} = \vec{c}/d$
Matrix += Matrix	add	`T += R;`	$\mathsf{T} = \mathsf{T}+\mathsf{R}$
Matrix -= Matrix	subtract	`T -= R;`	$\mathsf{T} = \mathsf{T}-\mathsf{R}$
Matrix *= DOUBLE	multiply by scalar	`T *= d;`	$\mathsf{T} = d\mathsf{T}$
Matrix /= DOUBLE	divide by scalar	`T /= d;`	$\mathsf{T} = \mathsf{T}/d$

[†] The ^ operator has rather low precedence, so often one has to write (a^b).

[‡] Two operators are provided for the dot product, which do the exact same thing.

Table 1: Operators available for Vectors and Matrices

2.5.1 DOUBLE nrm2()

This returns the sum of the squares of the elements, which is its norm squared. E.g.

```
Vector a(1,2,3.316625);
Matrix R(1,2,0,
         2,0,2
         1,1,1);
DOUBLE d1 = a.nrm2(); // will be equal to 16.0000014
DOUBLE d2 = R.nrm2(); // will be equal to 16
```

2.5.2 DOUBLE nrm()

This returns the norm of a Vector or Matrix, e.g.

```
DOUBLE d3 = a.nrm(); // will be equal to 4.00000017
DOUBLE d4 = R.nrm(); // will be equal to 4
```

The following properties are for Matrices only:

2.5.3 DOUBLE tr()

This returns the trace of a Matrix, i.e., the sum of its diagonal elements. E.g.

```
DOUBLE d5 = R.tr(); // will be equal to 2
```

2.5.4 DOUBLE det()

This returns the determinant of a Matrix, e.g.

```
DOUBLE d6 = R.det(); // will be equal to -2
```

2.5.5 Vector row(int i)

This returns the ith row of a Matrix.

2.5.6 Vector column(int j)

This returns the jth column of a Matrix.

2.6 Non-member functions

In the definition of the following non-member functions, the specified return type are effective ones. E.g. a return type of Matrix may return a MatrixExpression when this is more efficient. In any case, it can be treated as a Matrix in virtually all ways. Likewise, if an argument is of Matrix type, a Matrix expression is also allowed.

2.6.1 Matrix Transpose(const Matrix & M)

Returns the transpose of M, e.g.

```
Matrix T = Transpose(R);
```

2.6.2 Matrix Inverse(const Matrix & M)

Returns the inverse of the matrix M, e.g.

```
Matrix T = Inverse(R);
```

2.6.3 Matrix Rodrigues(const Vector & v)

Returns the rotation matrix for a rotation along the axis given by the direction of v, with the angle equal to the norm of v. Example:

```
Matrix T = Rodrigues(a);
```

2.6.4 Matrix Dyadic(const Vector & a, const Vector & b)

Returns the Matrix-valued dyadic product of the two arguments, e.g.

```
Matrix T = Dyadic(a,b);
```

2.6.5 Vector MTVmult(const Matrix & M, const Vector & v)

Equivalent to 'Transpose(M)*v'.

2.6.6 DOUBLE dist(const Vector & a, const Vector & b)

Returns the length of the difference vector between a and b. This is a remnant of earlier versions of the Vector and Matrix classes, and barely if at all more efficient than (a-b).nrm().

2.6.7 DOUBLE dist2(const Vector & a, const Vector & b)

Returns the square length of the difference vector between a and b. This is a remnant of earlier versions of the Vector and Matrix classes, and barely if at all more efficient than (a-b).nrm2().

2.6.8 DOUBLE distwithshift(const Vector&a, const Vector&b, const Vector&s)

Returns the length of the difference vector between a and b shifted by s. This is a remnant of earlier versions of the vector and Matrix classes, and barely if at all more efficient than (a+s-b).nrm().

Finally, because the notation a|b, a*b and a^b for dot and cross product may be confusing, the following equivalent alternatives are defined:

2.6.9 DOUBLE dotProduct(const Vector & a, const Vector & b)

Returns the DOUBLE which is the dot, or inner, product of the two Vector arguments. It is by definition equivalent to (a|b). Example:

```
DOUBLE d = dotProduct(a,b);
```

2.6.10 DOUBLE crossProduct(const Vector & a, const Vector & b)

Returns the Vector which is the cross, or outer, product of the two Vector arguments. It is by definition equal to (a^b). Example:

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

3 Expressions

Using the above elementary operations and functions, complex expressions can be constructed just as for built-in type such as double. To be more specific, for all operator expressions in table 1 on page 8, the arguments can be expressions themselves.

For example, one can write:

```
Vector r[2] = { Vector(1,4,5), Vector(2,3,4) };
Vector v[2] = { Vector(1,0,0), Vector(-1,0,0) };
Vector s(7,1,0);
Matrix A(1,0,0,0,1,0,0,-1,0);
DOUBLE t = (r[0]+2*A*(s^r[1]) ) | (v[1]-v[0]);
// alternatively:
//DOUBLE t = dotProduct(r[0]+2*A*crossProduct(s,r[1]), v[1]-v[0]);
```

Internally, the expressions are not computed directly via temporaries, but are computed only upon assignment. As a result, the definitions of these operators and functions in vecmat3.h is not as simple as e.g. Vector operator+(Vector&,Vector&). For that reason, above we used Vector and Matrix wherever a Vector/Matrix or a Vector/Matrix expression can occur. Generally one need not worry about the details of the implementation though, and things work as expected (the only exception was explained in section 2.3.

Background references

[1] `vecmat3.h` The header file contains comments to elucidate the expression template structure.

[2] T. Veldhuizen, *Expression Templates* C++ Report, Vol. 7 No. 5 June 1995, pp. 26-31. See also http://ubiety.uwaterloo.ca/~tveldhui/papers/Expression-Templates/exprtmpl.html. Reprinted in: S. B. Lippmann (ed.) *C++ Gems* (Cambridge University Press, 1998).

[3] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide* (Addison-Wesley, Boston, 2002).

[4] http://www.oonumerics.org/blitz

[5] http://tvmet.sourceforge.net