

User Documentation for the *vecmat3* Vector and Matrix classes (version 2)

Ramses van Zon*

*Chemical Physics Theory Group, Department of Chemistry, University of Toronto
80 St. George Street, Toronto, Ontario M5S 3H6, Canada*

May 22, 2009

Abstract

This document describes how to use the c++ Vector and Matrix class 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 expression, 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 | Classes | 4 |
| 2.2 | Header file | 5 |
| 2.3 | Initialization methods | 6 |
| 2.3.1 | Initialization through constructor parameters | 6 |
| 2.3.2 | Initialization through assignment | 7 |
| 2.3.3 | Initialization through a comma separated list | 7 |
| 2.3.4 | Initialization through member functions | 8 |
| 2.3.5 | Arrays | 8 |
| 2.4 | Accessing the elements | 9 |
| 2.5 | Operators | 9 |
| 2.6 | Member functions | 9 |
| 2.6.1 | DOUBLE nrm2() | 10 |
| 2.6.2 | DOUBLE nrm() | 11 |
| 2.6.3 | DOUBLE tr() | 11 |
| 2.6.4 | DOUBLE det() | 11 |
| 2.6.5 | Vector row(int i) | 11 |
| 2.6.6 | Vector column(int j) | 11 |
| 2.7 | Non-member functions | 11 |
| 2.7.1 | Matrix Transpose(const Matrix & M) | 11 |
| 2.7.2 | Matrix Inverse(const Matrix & M) | 12 |
| 2.7.3 | Matrix Rodrigues(const Vector & v) | 12 |
| 2.7.4 | Matrix Dyadic(const Vector & a, const Vector & b) | 12 |
| 2.7.5 | Vector MTVmult(const Matrix & M, const Vector & v) | 12 |
| 2.7.6 | DOUBLE dist(const Vector & a, const Vector & b) | 12 |
| 2.7.7 | DOUBLE dist2(const Vector & a, const Vector & b) | 12 |
| 2.7.8 | DOUBLE distwithshift(const Vector&a,const Vector&b,const Vector&s) | 12 |
| 2.7.9 | DOUBLE dotProduct(const Vector & a, const Vector & b) | 13 |
| 2.7.10 | DOUBLE crossProduct(const Vector & a, const Vector & b) | 13 |
| 3 | Expressions | 13 |
| | Background references | 14 |

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 in C++. Now in principle, expressions involving matrices and vectors, such as

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

(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).

Changes compared to the first version of `vecmat3`

The main difference between the first version of `vecmat3` and the second is that the matrices and vectors no longer need to be all of one type in a single application. In addition to the standard `Vector` and `Matrix` classes of type `DOUBLE` (which defaults to `double`) as in version 1, in version 2 one also has three-dimensional structures of any type `VALTYPE` at one's disposal. To be more precise:

- `vecmat3::Vector<VALTYPE>` and `vecmat3::Matrix<VALTYPE>` are three-dimensional vector and matrix classes whose elements are of type `VALTYPE`.

For example, one can define a 3x3 matrix of integers as `vecmat3::Matrix<int> m;`

- This notation reflects two changes in the code:
 - Almost all of the `vecmat3` code is now contained in its own namespace called `vecmat3`.
 - The type is a template argument.
- In version 2 the standard `Vector` and `Matrix` classes are simply typedef'ed as equivalent to `vecmat3::Vector<DOUBLE>` and `vecmat3::Matrix<DOUBLE>`, whereas in version 1 they were the only vectors and matrices available.
- The typedef's of `Vector` and `Matrix` will be omitted if the compiler flag `NOVECMAT3DEF` is defined.

2 Using the Vector and Matrix classes

2.1 Classes

`vecmat3` provides two general templated classes within the namespace `vecmat3`:

```
template<class VALTYPE> vecmat3::Vector;
template<class VALTYPE> vecmat3::Matrix;
```

The template parameter `VALTYPE` determines the type of the vector and matrix elements. Thus, for a vector of integers one uses the type `vecmat3::Vector<int>`, while for a matrix of doubles one would use `vecmat3::Matrix<double>`.

Since applications often need only one type of vector, a default vector type and a default matrix type are defined outside the `vecmat3` namespace, as follows

```
typedef vecmat3::Vector<DOUBLE> Vector;
typedef vecmat3::Matrix<DOUBLE> Matrix;
```

Here, `DOUBLE` is a predefined macro that should contain the type of the elements of the default vectors and matrices. Thus, `Vector v;` defines a vector with elements of type `DOUBLE`.

The type `DOUBLE` can be defined in three ways:

1. One can write an `#define DOUBLE <something>` before including the `vecmat3.h` header, with `<something>` replaced by the desired type (e.g. `float` or `double`);
2. One can give a command line argument to the compiler to define `DOUBLE` to be `<something>` (e.g. `-DDOUBLE=float` for `g++`);
3. One can do nothing, which makes `DOUBLE` default to `double`.

2.2 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
```

The type of the elements of a vector or matrix will be referred to as the “value type” in this documentation.

- 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;
```

- Alternatively, one can explicitly use any other value type than `DOUBLE` type, e.g.

```
vecmat3::Vector<int> a;
vecmat3::Matrix<int> R;
```

If vectors and matrices of a specific value type are used a lot in an application, it may be useful to `typedef` them to a shorter notation, e.g.

```
typedef vecmat3::Vector<int> intVector;
typedef vecmat3::Matrix<int> intMatrix;
intVector b;
intMatrix S;
```

- In the above examples, the elements of the vectors and matrices are unspecified, and likely contain garbage. In the next section, it will be explained how to initialize these elements of these classes.
- Note that currently, operations between matrices and vectors of different value types are not supported, even when mathematically this would make sense (such as for `int` and `double`).
- However, it is possible to assign any kind of number to an element of any value type, as long as a (standard) conversion to that type is known to the `c++` compiler. Thus, one may, for instance, assign an integer to an element of a vector, or one may multiply a vector by 2 (i.e., one may write `2*v` instead of being forced to write `2.0*v` or `2.0f*v`).

2.3 Initialization methods

There are four ways to initialize a `Vector` or `Matrix`, which we will discuss by example. In describing the initialization methods, the default `Vector` and `Matrix` types will be used; the arbitrary type versions `vecmat3::Vector<VALTYPE>` and `vecmat3::Matrix<VALTYPE>` have the same functionality.

2.3.1 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);
```

2.3.2 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 [2.5](#) and [3](#).

2.3.3 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 through 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 $\begin{pmatrix} 2 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$.

2.3.4 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.

2.3.5 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) } ;
```

2.4 Accessing the elements

Basic elements of Vectors and Matrices are generally accessible 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 $R(0,0)$, $R(0,1)$, \dots $R(2,2)$.

An alternative way to access the elements is through 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(1);  
Vector w = R.column(2);
```

2.5 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 ostreams, 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.

2.6 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 | $c = a - b;$ | $\vec{c} = \vec{a} - \vec{b}$ |
| VALTYPE * Vector | multiply with scalar | $c = d * a;$ | $\vec{c} = d\vec{a}$ |
| Vector * VALTYPE | multiply with scalar | $c = a * d;$ | $\vec{c} = \vec{a}d$ |
| Vector / VALTYPE | divide by scalar | $c = a / d;$ | $\vec{c} = \vec{a}/d$ |
| Vector ^ Vector | cross/outer product [†] | $c = a \wedge b;$ | $\vec{c} = \vec{a} \times \vec{b}$ |
| Vector * Vector | dot/inner product [‡] | $d = a * b;$ | $d = \vec{a} \cdot \vec{b}$ |
| (Vector Vector) | dot/innter product [‡] | $d = (a b);$ | $d = \vec{a} \cdot \vec{b}$ |
| - Matrix | negative | $T = -S;$ | $T = -S$ |
| Matrix + Matrix | add | $T = S + R;$ | $T = S + R$ |
| Matrix - Matrix | subtract | $T = S - R;$ | $T = S - R$ |
| VALTYPE * Matrix | multiply with scalar | $T = d * S;$ | $T = dS$ |
| Matrix * VALTYPE | multiply with scalar | $T = S * d;$ | $T = Sd$ |
| Matrix / VALTYPE | divide by scalar | $T = S / d;$ | $T = S/d$ |
| Matrix * Matrix | matrix-matrix product | $T = S * R;$ | $T = SR$ |
| Matrix * Vector | matrix-vector product | $c = S * a;$ | $\vec{c} = S\vec{a}$ |
| 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 | $T += R;$ | $T = T + R$ |
| Matrix -= Matrix | subtract | $T -= R;$ | $T = T - R$ |
| Matrix *= DOUBLE | multiply by scalar | $T *= d;$ | $T = dT$ |
| Matrix /= DOUBLE | divide by scalar | $T /= d;$ | $T = T/d$ |

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

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

Table 1: Operators available for matrices and vectors with elements of type VALTYPE.

2.6.1 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
```

2.6.2 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:

2.6.3 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
```

2.6.4 DOUBLE det()

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

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

2.6.5 Vector row(int i)

This returns the *i*th row of a Matrix.

2.6.6 Vector column(int j)

This returns the *j*th column of a Matrix.

2.7 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.

2.7.1 Matrix Transpose(const Matrix & M)

Returns the transpose of the argument, which is a Matrix, e.g.

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

2.7.2 Matrix Inverse(const Matrix & M)

Returns the inverse of the argument, which is a Matrix, e.g.

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

2.7.3 Matrix Rodrigues(const Vector & v)

Returns the Matrix-valued rotation matrix for a rotation along the axis given by the direction of the Vector argument, with the angle equal to the norm of that Vector, e.g.

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

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

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

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

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

This simply returns Transpose(Matrix)*Vector.

2.7.6 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).norm()`.

2.7.7 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).norm2()`.

2.7.8 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).norm()`.

Finally, because the notation $a*b$ and $a\hat{b}$ for dot and cross product may be confusing, the following equivalent alternatives are defined:

2.7.9 `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 equal to `(Vector|Vector)`. Example:

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

2.7.10 `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 `(Vector^Vector)`. 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. Never mind the implementation though, things work as expected.

Background references

- [1] 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).
- [2] D. Vandevorde and N. M. Josuttis, *C++ Templates: The Complete Guide* (Addison-Wesley, Boston, 2002).
- [3] <http://www.oonumerics.org/blitz>
- [4] <http://tvmets.sourceforge.net>