User Documentation for the *vecmat3* Vector and Matrix classes (version 2.3.1)

Ramses van Zon[*]

May 15, 2013

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 expression templates, without interfering 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.

[*]vanzonr@gmail.com

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++. 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).

Change history

Changes in version 2

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 `T` at one's disposal. To be more precise:

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

  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::Vector<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.

Changes between version 2 and version 2.3

(Versions 2.1 and 2.2 were internal development stages.)

- Most functionality remained the same as in version 2, except that square bracket support has been added. See Section 3.4.
- The header file now also enforces inlining the template functions for the GNU (tested version 4.4.0) and the Intel compilers (tested versions 11 and 12), even when no optimization is used.
- A major improvement of version 2.3 is that the library is now compatible with IBM's xlC compiler, which had trouble with some template constructions in versions 1 and 2. While I'm on the subject, one should compile with `-O4` when using the IBM compilers with vecmat3 in order to get all inlining done properly (in the latest version, xlC 11, the options `-O2 -qinline=level=6` suffice).

Changes between version 2.3 and version 2.3.1

Version 2.3.1 comes with an open-source license (MIT). Here is the text of the license:

2 Installation

To use this header-only library, one merely needs to copy the header file vecmat3.h to the directory of the source files that include it, or to a default directory where the compiler will look for header files (e.g. /usr/local/include).

The *vecmat3* library has been tested with the GNU g++ compiler version 4.4 and up, the Intel C++ compiler version 11 and up and IBM's XL C++ compiler version 10 and up.

3 Using the Vector and Matrix classes

3.1 Classes

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

```
    template<typename T> vecmat3::Vector;
    template<typename T> vecmat3::Matrix;
```

The template parameter T 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 a and 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`.

The definition of DOUBLE and the type definition of Vector and Matrix in the global namespace does pollute the global namespace, and in many cases is not wanted. These definitions are omitted if `NOVECMAT3DEF` is defined.

3.2 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;
```

- 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;
  ```

- Alternatively, one can have no default global Vector and Matrix class defined, and use only the namespace vecmat3, e.g.

  ```
  #define NOVECMAT3DEF
  #include "vecmat3.h"
  vecmat3::Vector<double> a;
  vecmat3::Matrix<double> R;
  ```

- 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`).

3.3 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<T>` and `vecmat3::Matrix<T>` have the same functionality.

3.3.1 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);
```

3.3.2 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 3.5 and 4.

3.3.3 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 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 the method in 3.2.1.

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

3.3.4 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.

3.3.5 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) } ;
```

3.4 Accessing the elements

There are three ways to assess the elements of Vectors and Matrices:

1. 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)`, `...R(2,2)`.

2. Bracket notation can also be used, 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]`. Note the double brackets for matrix element access. This way, Vector and Matrix objects act as if they are of type `T[3]` and `T[3][3]`, respectively. Accessing the matrix elements in this way may be moderately slower than the parenthesis method (depending on the compiler).

3. Another 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(1);

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

3.5 Operators

The available algebraic operators for the Vector and Matrix classes are summarized in table 1, in which 'Vector<T>' stands for '`const Vector<T>&`' or a Vector-valued expression, and 'Matrix<T>' stands for '`const Matrix<T>&`' or a Matrix<T>-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<T> 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.

3.6 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. Note that below, `T` stands for the typename of the template, while `Vector<T>` and `Matrix<T>` stand for `vecmat3::Vector<T>` and `vecmat3::Matrix<T>`, respectively. The examples all use `T=DOUBLE`.

3.6.1 T nrm2()

This returns the sum of the squares of the elements, which is its norm squared. E.g., with `T=DOUBLE`,

```
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
```

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

3.6.2 T nrm()

This returns the norm of a Vector or Matrix, e.g., with `T=DOUBLE`,

```
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:

3.6.3 T tr()

This returns the trace of a Matrix, i.e., the sum of its diagonal elements. E.g., with `T=DOUBLE`,

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

3.6.4 T det()

This returns the determinant of a Matrix, e.g., with `T=DOUBLE`,

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

3.6.5 Vector<T> row(int i)

This returns the ith row of a Matrix.

3.6.6 Vector<T> column(int j)

This returns the jth column of a Matrix.

3.7 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 Matrix-Expression 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.

3.7.1 Matrix<T> Transpose(const Matrix<T> & M)

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

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

3.7.2 Matrix<T> Inverse(const Matrix<T> & M)

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

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

3.7.3 Matrix<T> Rodrigues(const Vector<T> & 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 S = Rodrigues(a);
```

3.7.4 Matrix<T> Dyadic(const Vector<T> & a, const Vector<T> & b)

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

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

3.7.5 Vector<T> MTVmult(const Matrix<T> & M, const Vector<T> & v)

This simply returns Transpose(Matrix)*Vector.

3.7.6 T dist(const Vector<T> & a, const Vector<T> & b)

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

3.7.7 T dist2(const Vector<T> & a, const Vector<T> & 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()`.

3.7.8 T distwithshift(const Vector<T>&a,const Vector<T>&b,const Vector<T>&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` and `a^b` for dot and cross product may be confusing, the following equivalent alternatives are defined:

3.7.9 T dotProduct(const Vector<T> & a, const Vector<T> & 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, with `T=DOUBLE`:

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

3.7.10 Vector<T> crossProduct(const Vector<T> & a, const Vector<T> & 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);
```

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

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

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