3DEQUALIZER 4

VL - Python Vector Library

Vectors, matrices and quaternions in Python
Science-D-Visions, 2014-12-05

Contents

1 *Introduction*

The vector library VL is used in many 3DE4 scripts. It provides basic functions for dealing with vectors, matrices and affine transforms in one to four dimensions, and quaternions. Additionally, there are classes for handling rotations in two and three dimensions. VL should run in late 2.x python versions, but it is not supposed to run in python 3 without modifications (we didn't try this).

2 *About this document*

The document is targeted to developers of python scripts meant to run within 3DE4, although the vector library itself could be used in other environments as well (other software or standalone). The document is not complete. Although VL has been in use for years now the documentation was not widely spread. Versions are:

- version 1.1 - bugfixes, functions - 2014-01-28
- version 1.0 - initial - 2013-12-18

3 *Overview*

VL was created in order to perform linear operations in low-dimensional vector spaces. All vectors, matrices and transformations in VL are defined with respect to the field of real numbers, represented by double-precision floating point variables in Python. For the code examples in this document we import the module as

```
from vl_sdv import *
```

VL implements the following classes, ordered by category and dimension:

Category	Class			
Vectors	**vec1d**	**vec2d**	**vec3d**	**vec4d**
Matrices	**mat1d**	**mat2d**	**mat3d**	**mat4d**
Transformations	**igl1d**	**igl2d**	**igl3d**	**igl4d**
Quaternions				**quatd**
Rotations		**rot2d**	**rot3d**	

By "Transformations" we refer to affine transformations, which include translations, rotations and scalings. The rotation classes **rot2d** and **rot3d** represent a convenient and unambigious way to deal with rotations in two- and three-dimensional space.

4 *Classes*

In this section we will describe the API of all classes in detail.

4.1 Vectors

The vector classes are **vec1d**, **vec2d**, **vec3d** and **vec4d**. All vector classes are derived from the base class **vec**.

4.1.1 *Constructor*

All vector types can be instantiated without arguments (default constructor). This generates the zero-vector of the respective vector space:

```
>>> vec1d();vec2d();vec3d();vec4d();
vec1d(0.0)
vec2d(0.0,0.0)
vec3d(0.0,0.0,0.0)
vec4d(0.0,0.0,0.0,0.0)
```

Vector types can also be instantiated by passing the correct number of components.

```
>>> vec1d(2);vec2d(2,3);vec3d(2,3,5);vec4d(2,3,5,7);
vec1d(2.0)
vec2d(2.0,3.0)
vec3d(2.0,3.0,5.0)
vec4d(2.0,3.0,5.0,7.0)
```

Instead of passing components directly, vectors can be created by objects having the following properties:

- Any object *x* passed to the constructor has components which are accessed by means of the operator [ ] (i.e. the object has a method __getitem__)
- The number of components of *x* is **len**(x).

This means, vectors can be initialized by lists

```
>>> vec1d([2]);vec2d([2,3]);vec3d([2,3,5]);vec4d([2,3,5,7]);
vec1d(2.0)
vec2d(2.0,3.0)
vec3d(2.0,3.0,5.0)
vec4d(2.0,3.0,5.0,7.0)
```

or by tuples:

```
>>> vec1d((2));vec2d((2,3));vec3d((2,3,5));vec4d((2,3,5,7));
vec1d(2.0)
vec2d(2.0,3.0)
vec3d(2.0,3.0,5.0)
vec4d(2.0,3.0,5.0,7.0)
```

Vectors themselves behave like lists, which means they can be copied:

```
>>> a = vec1d(2);b = vec2d(2,3);c = vec3d(2,3,5);d = vec4d(2,3,5,7);
>>> vec1d(a);vec2d(b);vec3d(c);vec4d(d)
vec1d(2.0)
vec2d(2.0,3.0)
vec3d(2.0,3.0,5.0)
vec4d(2.0,3.0,5.0,7.0)
```

4.1.2 *Convenience functions*

Vector instances can easily be created by using the convenience function vec(). The number of arguments ranges from 1 to 4. This determines what type of vector is created, vec1d, vec2d, vec3d or vec4d.

```
>>> vec(3);vec(3,4);vec(3,4,5);vec(3,4,5,6)
vec1d(3.0)
vec2d(3.0,4.0)
vec3d(3.0,4.0,5.0)
```

```
vec4d(3.0,4.0,5.0,6.0)
```

4.1.3 *String representation*

For vectors the methods **__repr__** and **__str__** are implemented. The output of **__repr__** has already been demonstrated in the previous section. The print representation (given by the method **__str__**) is:

```
>>> print vec1d(2),vec2d(2,3),vec3d(2,3,5),vec4d(2,3,5,7)
[2.0] [2.0,3.0] [2.0,3.0,5.0] [2.0,3.0,5.0,7.0]
```

4.1.4 *Container properties*

Vectors behave like tuples or lists in the following sense: the number of components is obtained by the function **len** and the components can be read or written using the operator [ ]:

```
>>> print len(vec1d()),len(vec2d()),len(vec3d()),len(vec4d())
1 2 3 4

>>> a = vec4d(1,2,4,8)
>>> a[3] = 10
>>> print a
[1.0,2.0,4.0,10]
```

Iterators can be used to access the vector components:

```
print [10 * i for i in vec4d(2,3,5,7)]
[20.0, 30.0, 50.0, 70.0]
```

4.1.5 *Operators*

4.1.5.1 Unary operators

The following unary operators are implemented for all vector classes:

Operator	Return type	Argument type	Semantics
+	Vector	Vector	Identity
–	Vector	Vector	Negative of a vector (reversed direction)

Examples:

```
>>> -vec(3);-vec(3,4);-vec(3,4,5);-vec(3,4,5,6)
vec1d(-3.0)
vec2d(-3.0,-4.0)
vec3d(-3.0,-4.0,-5.0)
vec4d(-3.0,-4.0,-5.0,-6.0)

>>> +vec(3);+vec(3,4);+vec(3,4,5);+vec(3,4,5,6)
vec1d(3.0)
vec2d(3.0,4.0)
vec3d(3.0,4.0,5.0)
vec4d(3.0,4.0,5.0,6.0)
```

Note, that in order to avoid a common pitfall for C/C++-programmers the identity operator creates a new object, it does NOT return a reference to the original object:

```
>>> a = vec(3,4);b = +a;b[1] = 5
>>> print "a:",a,"\nb:",b
a: [3.0,4.0]
b: [3.0,5.0]
```

4.1.5.2 Binary operators

The following binary operators are implemented for all vector classes:

Operator	Return type	1st Arg type	2nd Arg type	Semantics
+	Vector	Vector	Vector	Sum of two vectors
−	Vector	Vector	Vector	Difference of two vectors
*	Vector	Number	Vector	Product of number and vector
*	Vector	Vector	Number	Product of number and vector
*	Vector	Vector	Matrix	Multiplication by a matrix from the right $b_i = \sum_j a_j\, m_{ji}$
*	Number	Vector	Vector	Inner product (dot product) of two vectors $a \cdot b = \sum_i a_i\, b_i$
/	Vector	Vector	Number	Quotient of vector and number
&	Matrix	Vector	Vector	Tensor product of two vectors $(a \otimes b)_{ij} = a_i\, b_j$

Due to Python's capability of operator overloading, these operations become quite readable in VL. For adding and subtracting vectors we have:

```
>>> print vec(2,3) + vec(5,7)
[7.0,10.0]

>>> print vec(5,7) - vec(2,3)
[3.0,4.0]
```

Multiplication and division by numbers is done by:

```
>>> print 3 * vec(5,6,7)
[15.0,18.0,21.0]

>>> print vec(5,6,7) * 3
[15.0,18.0,21.0]

>>> print vec(50,60,70) / 10
[5.0,6.0,7.0]
```

The inner product (also called dot product) is done by the asterisk operator as well.

```
>>> vec(2) * vec(3)
6.0

>>> vec(2,3) * vec(5,7)
31.0

>>> vec(2,3,5) * vec(7,11,13)
112.0

>>> vec(2,3,5,7) * vec(11,13,17,19)
279.0
```

For the tensor product we use the operator &. More details are described in section 4.2.5

```
>>> print vec(3,4) & vec(1,10)
[[3.0,30.0],[4.0,40.0]]
```

The skew-symmetric product is implemented for two- and three-dimensional vectors. Both arguments are vectors.

Dimension	Return type	Semantics	Math symbol
2	Number	e.g. "Spinor product"	various, e.g. a ∧ b (on 2-forms)

Dimension	Return type	Semantics	Math symbol
3	Vector	Vector product	a ∧ b (on 3-forms), a × b

The skew-symmetric product is dependent on the dimension. In two-dimensional space it's sometimes called "spinor product", in three dimensions it is called "vector product" or "cross product".

```
>>> vec(2,3) ^ vec(5,7)
-1.0

>>> vec(1,0,0) ^ vec(0,1,0)
vec3d(0.0,0.0,1.0)
```

Note, that Python's operator precedence rules make it necessary to use brackets when the tensor product or wedge product are combined with other operations.

4.1.5.3 Augmenting operators

The following non-constant ("augmenting") operators are implemented for all vector classes. The left hand side argument is always a vector type. We refer to this argument as "**self**" in the following table:

Operator	Argument type	Semantics
+=	Vector	Add a vector to **self**
-=	Vector	Subtract a vector from **self**
***=**	Matrix	Multiply **self** by a matrix from the right
***=**	Number	Multiply **self** by a number
/=	Number	Divide **self** by a number

Examples:

```
>>> a = vec(2,3,5,7);a += vec(8,4,2,1);print a
[10.0,7.0,7.0,8.0]

>>> a = vec(2,3,5);a -= vec(4,2,1);print a
[-2.0,1.0,4.0]

>>> a = vec(3,5);a *= 10;print a;
[30.0,50.0]

>>> a = vec(10);a /= 2;print a;
[5.0]
```

For the matrix multiplication from the right the vector is interpreted as a "row vector". Mathematically speaking, this is defined as $v^T m := (m^T v)^T$. Since we can write $v^T(m_a m_b) = (v^T m_a)m_b$ the augmenting operator is well-defined.

```
>>> a = vec(2,3);a *= mat(1,10,100,1000)
>>> print a
[302.0,3020.0]
```

4.1.6 *Methods*

The following methods are implemented for all vector classes.

Method	Result type	Semantics
dotsq	Number	Inner product of a vector with itself
tensq	Matrix	Tensor product of a vector with itself
norm1	Number	Sum of absolute values of the components

Method	Result type	Semantics
norm2	Number	Euclidian norm (i.e. length of a vector)
norminf	Number	Maximum of absolute values of the components
unit	Vector	Unit vector, normalized version of a vector
para	Matrix	Projector onto the vector
ortho	Matrix	Projector into the space orthogonal to the vector

dotsq is a short hand notation for the inner product of a vector with itself. It is particularly useful when the argument is some lengthy expression.

```
>>> (vec(1,2) + vec(2,2)).dotsq()
25.0
```

A similar notation is avilable for the tensor product of a vector with itself.

```
>>> vec(3,4).tensq()
mat2d(vec2d(9.0,12.0),vec2d(12.0,16.0))
```

Vectors offer three different norms, the most important of which is of course the euclidian norm (i.e. the length of a vector):

```
>>> vec(3,-4).norm2()
5.0
```

norm1 yields the sum of absolute values of the components, **norminf** is the absolute value of the largest component.

```
>>> vec(3,-4).norm1()
7.0

>>> vec(3,-4).norminf()
4.0
```

The Notation refers to the mathematical notation of norms:

$$|a|_1, |a|_2, |a|_\infty$$

The method **unit** returns the normalized version of its argument, the vector with length 1 and same direction:

```
>>> print vec(3,4).unit()
[0.6,0.8]

>>> print vec(3,4).unit().norm2()
1.0
```

The methods **para** and **ortho** provide projection matrices into the space collinear and orthogonal to the argument vector, respectively. These projectors have the following properties ("idempotence"):

$$P_{||} = P_{||}P_{||}$$
$$P_{\perp} = P_{\perp}P_{\perp}$$

They sum up to a unit matrix:

$$P_{||} + P_{\perp} = \mathbf{1}$$

In the one-dimensional case, **ortho** leads to a zero-matrix:

```
>>> print vec(1).para()
[[1.0]]

>>> print vec(1).ortho()
[[0.0]]
```

Here is an example in three dimensions:

```
>>> print vec(0,0,1).para()
[[0.0,0.0,0.0],[0.0,0.0,0.0],[0.0,0.0,1.0]]
```

```
>>> print vec(0,0,1).ortho()
[[1.0,0.0,0.0],[0.0,1.0,0.0],[0.0,0.0,0.0]]
```

The method **dual** is only implemented in some of the vector classes. It is closely related to the skew-symmetric product and is similar to the "Hodge-star" (∗a) in differential geometry, but with a different sign-convention.

Dimension	Return type
2	Vector
3	Matrix

Our version of **dual** is defined as $(*a)\ b := a \wedge b$, for two- and three-dimensional space. The reason for us to mention this method here is, that especially in three-dimensions it has proven to be a useful tool in dealing with rotation objects.

4.2 Matrices

The matrix classes are **mat1d**, **mat2d**, **mat3d** and **mat4d**. All matrix classes are derived from the base class **mat**. A matrix in VL is an array of **row vectors**. This is reflected in the way matrices are constructed or printed. The advantage is that components can be accessed as in mathematical notation:

$$m_{ij} = \texttt{m[i][j]}$$

4.2.1 *Constructor*

All matrix types can be instantiated without arguments (default constructor). This generates a zero-matrix in the respective space.

```
>>> mat1d();mat2d();mat3d();mat4d()
mat1d(vec1d(0.0))
mat2d(vec2d(0.0,0.0),vec2d(0.0,0.0))
mat3d(vec3d(0.0,0.0,0.0),vec3d(0.0,0.0,0.0),
      vec3d(0.0,0.0,0.0))
mat4d(vec4d(0.0,0.0,0.0,0.0),vec4d(0.0,0.0,0.0,0.0),
      vec4d(0.0,0.0,0.0,0.0),vec4d(0.0,0.0,0.0,0.0))
```

4.2.2 *String representation*

For matrices the methods **__repr__** and **__str__** are implemented. The output of **__repr__** has already been demonstrated in the previous section. The print representation (given by the method **__str__**) is:

```
>>> print mat1d(2)
[[2.0]]
>>> print mat2d(3)
[[3.0,0.0],[0.0,3.0]]
>>> print mat3d(5)
[[5.0,0.0,0.0],[0.0,5.0,0.0],[0.0,0.0,5.0]]
>>> print mat4d(7)
[[7.0,0.0,0.0,0.0],[0.0,7.0,0.0,0.0],[0.0,0.0,7.0,0.0],[0.0,0.0,0.0,7.0]]
```

4.2.3 *Container properties*

Matrices are lists of row vectors. The number of row vectors can be retrieved by the function **len**

```
>>> print len(mat1d()),len(mat2d()),len(mat3d()),len(mat4d())
1 2 3 4
```

In matrix objects row vectors can be accessed by the operator [ ].

```
>>> a = mat2d(1,2,4,8)
>>> a[0] = vec2d(3,5)
>>> a[1] = [7,9]
>>> print a
[[3.0,5.0],[7.0,9.0]]
```

Since the operator [ ] is implemented for vectors, components of matrices are accessed by applying [ ] twice:

```
>>> a = mat2d(1,2,4,8)
>>> a[1][1] = 7
>>> print a
[[1.0,2.0],[4.0,7.0]]
```

You can iterate over the rows of a matrix:

```
>>> a = mat4d(11,12,13,14,21,22,23,24,31,32,33,34,41,42,43,44)
>>> for i in a: print i;
...
[11.0,12.0,13.0,14.0]
[21.0,22.0,23.0,24.0]
[31.0,32.0,33.0,34.0]
[41.0,42.0,43.0,44.0]
```

4.2.4 *Convenience functions*

Matrix instances can easily created by using the convenience function **mat**(). The number of arguments is 1,4,9 or 16. By this number the function determines what type of matrix is created.

```
>>> print mat(2)
[[2.0]]
>>> print mat(2,3,5,7)
[[2.0,3.0],[5.0,7.0]]
>>> print mat(1,0,0,0,2,0,0,0,3)
[[1.0,0.0,0.0],[0.0,2.0,0.0],[0.0,0.0,3.0]]
>>> print mat(1,1,1,1,0,1,1,1,0,0,1,1,0,0,0,1)
[[1.0,1.0,1.0,1.0],[0.0,1.0,1.0,1.0],[0.0,0.0,1.0,1.0],[0.0,0.0,0.0,1.0]]
```

4.2.5 *Operators*

4.2.5.1 Unary operators

The following unary operators are implemented for all matrix classes:

Operator	Return type	Argument type	Semantics
+	Matrix	Matrix	Identity
–	Matrix	Matrix	Negative of a matrix

Examples:

```
>>> -mat1d(3);-mat2d(4);-mat3d(5);-mat4d(6)
mat1d(vec1d(-3.0))
mat2d(vec2d(-4.0,-0.0),vec2d(-0.0,-4.0))
mat3d(vec3d(-5.0,-0.0,-0.0),vec3d(-0.0,-5.0,-0.0),
      vec3d(-0.0,-0.0,-5.0))
mat4d(vec4d(-6.0,-0.0,-0.0,-0.0),vec4d(-0.0,-6.0,-0.0,-0.0),
      vec4d(-0.0,-0.0,-6.0,-0.0),vec4d(-0.0,-0.0,-0.0,-6.0))

>>> +mat1d(3);+mat2d(4);+mat3d(5);+mat4d(6)
mat1d(vec1d(3.0))
mat2d(vec2d(4.0,0.0),vec2d(0.0,4.0))
mat3d(vec3d(5.0,0.0,0.0),vec3d(0.0,5.0,0.0),
      vec3d(0.0,0.0,5.0))
mat4d(vec4d(6.0,0.0,0.0,0.0),vec4d(0.0,6.0,0.0,0.0),
      vec4d(0.0,0.0,6.0,0.0),vec4d(0.0,0.0,0.0,6.0))
```

As in the case of vectors the identity operator creates a new object, it does NOT return a reference to the original object:

```
>>> a = mat2d(3);b = +a;b[1][0] = 5
>>> print "a:",a,"\nb:",b
a: [[3.0,0.0],[0.0,3.0]]
```

```
b: [[3.0,0.0],[5.0,3.0]]
```

4.2.5.2 Binary operators

The following binary operators are implemented for all matrix classes:

Operator	Return type	1st Arg type	2nd Arg type	Semantics
+	Matrix	Matrix	Matrix	Sum of two matrices $C_{ij} = A_{ij} + B_{ij}$
-	Matrix	Matrix	Matrix	Difference of two matrices $C_{ij} = A_{ij} - B_{ij}$
*****	Matrix	Number	Matrix	Product of number and matrix $C_{ij} = A_{ij}$ x
*****	Matrix	Matrix	Number	Product of number and matrix $C_{ij} = A_{ij}$ x
*****	Vector	Matrix	Vector	Product of matrix and vector: $w_i = \sum_j A_{ij} v_j$
*****	Matrix	Matrix	Matrix	Matrix product: $C_{ik} = \sum_j A_{ij} B_{jk}$
/	Matrix	Matrix	Number	Quotient of Matrix and number $C_{ij} = A_{ij}$ / x

4.2.5.3 Augmenting operators

The following augmenting operators are defined for all matrix classes. The left hand side argument is always a matrix. We refer to this argument as "**self**" in the following table:

Operator	Argument type	Semantics
+=	Matrix	Add a matrix to **self** $S_{ij} \rightarrow S_{ij} + M_{ij}$
-=	Matrix	Subtract a matrix from **self** $S_{ij} \rightarrow S_{ij} - M_{ij}$
***=**	Matrix	Multiply **self** by a matrix from the right $S_{ij} \rightarrow \sum_k S_{ik} M_{kj}$
***=**	Number	Multiply **self** by a number $S_{ij} \rightarrow S_{ij}$ x
/=	Number	Divide **self** by a number $S_{ij} \rightarrow S_{ij}$ / x

Examples:

```
>>> a = mat(2,3,5,7);a += mat(8,4,2,1);print a
[[10.0,7.0],[7.0,8.0]]

>>> a = mat(2,3,5,7);a -= mat(8,4,2,1);print a
[[-6.0,-1.0],[3.0,6.0]]

>>> a = mat(2,3,5,7);a *= mat(8,4,2,1);print a
[[22.0,11.0],[54.0,27.0]]

>>> a = mat(2,3,5,7);a *= 10;print a
[[20.0,30.0],[50.0,70.0]]
```

```
>>> a = mat(2,3,5,7);a /= 10;print a
[[0.2,0.3],[0.5,0.7]]
```

4.2.6 *Methods*

The following methods are implemented for (almost) all matrix classes.

Method	Result type	Semantics
trans	Matrix	Transposed of a matrix $(M^{\mathrm{T}})_{ij} = M_{ji}$
trace	Number	Trace of a matrix $\mathrm{tr}M = \sum_{\mathrm{i}} M_{\mathrm{ii}}$
sub(i,j)	Matrix	Submatrix by ommiting row i and column j
adj	Matrix	Adjugate of a matrix
det	Number	Determinant of a matrix
invert	Matrix	Inverse of a matrix

We demonstrate the use of these methods for a 2x2-matrix. The program

```
a2 = mat2d(2,3,5,7)
print "original:", a2
print "   trans:", a2.trans()
print "   trace:", a2.trace()
print "sub(1,0):", a2.sub(1,0)
print "     adj:", a2.adj()
print "     det:", a2.det()
print "  invert:", a2.invert()
```

produces the following result:

```
original: [[2.0,3.0],[5.0,7.0]]
   trans: [[2.0,5.0],[3.0,7.0]]
   trace: 9.0
sub(1,0): [[3.0]]
     adj: [[7.0,-3.0],[-5.0,2.0]]
     det: -1.0
  invert: [[-7.0,3.0],[5.0,-2.0]]
```

4.2.7 *Type conversion*

4.3 Transformations

By a transformation we understand a tuple of a vector and a matrix. Transformations are useful for expressing scale, rotation and translation in one object. The transformation classes are **igl1d**, **igl2d**, **igl3d**, **igl4d** ("igl" for "inhomogenious general linear (group)").

4.3.1 *Constructor*

Transformations can be constructed as described in the following. There is a default constructor, which creates a unit element (i.e. the neutral element with respect to the multiplication):

```
>>> print igl2d()
[[[1.0,0.0],[0.0,1.0]],[0.0,0.0]]
```

Transformations can be specified by either providing a matrix or a vector or both:

```
>>> print igl2d(vec2d(2,3))
```

```
[[[0.0,0.0],[0.0,0.0]],[2.0,3.0]]
>>> print igl2d(mat2d(2,3,5,7))
[[[2.0,3.0],[5.0,7.0]],[0.0,0.0]]
>>> print igl2d(mat2d(2,0,0,3),vec2d(5,7))
[[[2.0,0.0],[0.0,3.0]],[5.0,7.0]]
```

4.3.2 *Accessing components*

The components of a transformation can be accessed for reading and writing as follows:

```
>>> a = igl2d(mat2d(2,0,0,3),vec2d(5,7))
>>> print a.m, a.v
[[2.0,0.0],[0.0,3.0]] [5.0,7.0]
>>> a.m = mat2d(1,2,3,4)
>>> a.v = vec(5,6)
>>> print a.m, a.v
[[1.0,2.0],[3.0,4.0]] [5.0,6.0]
```

4.3.3 *Convenience functions*

4.3.4 *Operators*

4.3.4.1 Unary operators

In the current version, no unary operators are defined for transformations.

4.3.4.2 Binary operators

The following binary operator/s is/are implemented for all transformation classes:

Operator	Return type	1st Arg type	2nd Arg type	Semantics
*	Transformation	Transformation	Transformation	Semi-direct product

4.3.5 *Methods*

For transformations the following methods are implemented:

Method	Result type	Semantics
invert	Transformation	Inverse
mat	Matrix	Embedding into higher dimensional matrix space

4.3.6 *Type conversion*

The space of transformations in n dimensions is a subspace of the space of n+1 dimensional matrices. The embedding is a homomorphism, i.e. the multiplication of transformation maps to the multiplication of matrices. The embedding is represented by the following cast. Given a transformation

```
>>> a = igl2d(mat2d(2,0,0,3),vec2d(5,7))
```

The three-dimensional matrix representation is:

```
>>> print mat3d(a)
[[2.0,0.0,5.0],[0.0,3.0,7.0],[0.0,0.0,1.0]]
```

4.4 Quaternions

Quaternions are very important in three-dimensional geometry, because they represent a compact way of handling rotations. In VL they are implemented as the class **quatd**. Representing the group of rotations on one hand and a vector space on the other hand makes them feasible for doing interpolation and statistics in rotational space, for which matrix or angle representations are inappropriate.

4.4.1 *Constructor*

Quaternions can be instantiated by passing no argument:

```
>>> print quatd()
[0.0,[0.0,0.0,0.0]]
```

Note that in this case a zero-quaternion is generated, although one might also argue that a unit quaternion is a reasonable default-value as well. This is due to the fact, that on one hand quaternions form a vector space, while on the other hand - without the zero quaternion - they form a multiplicative group. In this case we decided to emphasize the vector space aspect.

We consider quaternions as a pair of a number (the scalar part) and a three-dimensional vector (the vector part). Hence quaternions can be instantiated by passing a number and a vector:

```
>>> a = 1;b = vec(0.1,0.2,0.3);
>>> print quatd(a,b)
[1.0,[0.1,0.2,0.3]]

>>> print quatd(1,[0.1,0.2,0.3]);
[1.0,[0.1,0.2,0.3]]
```

We can also instantiate the quaternion by passing either scalar or vector part:

```
>>> print quatd(1)
[1.0,[0.0,0.0,0.0]]

>>> print quatd(vec(0.1,0.2,0.3))
[0.0,[0.1,0.2,0.3]]
```

Furthermore, we can simply pass four numbers or a four-dimensional vector. The behaviour of a copy constructor is implemented as well:

```
>>> print quatd(1,0.1,0.2,0.3)
[1.0,[0.1,0.2,0.3]]

>>> print quatd(vec4d(1,0.1,0.2,0.3))
[1.0,[0.1,0.2,0.3]]

>>> a = quatd(2,3,5,7);b = quatd(a)
>>> print b
[2.0,[3.0,5.0,7.0]]
```

4.4.2 *Accessing components*

The ways of accessing the components of a quaternion reflect the fact that it consists of a scalar and a vector part.

```
>>> a = quatd(2,3,5,7);

>>> print a[0];
2.0

>>> print a[1];
[3.0,5.0,7.0]
```

For convenience, managed attributes (a functionality of python) for accessing scalar and vector part are implemented in **quatd**:

```
>>> a = quatd(2,3,5,7);

>>> print a.s;
2.0

>>> print a.v;
[3.0,5.0,7.0]
```

4.4.3 *Operators*

4.4.3.1 Unary operators

The following unary operators are implemented for quaternions:

Operator	Return type	Semantics
+	Quaternion	Identity
−	Quaternion	Negative of a quaternion

Examples:

```
>>> a = quatd(2,3,5,7);

>>> print -a;
[-2.0,[-3.0,-5.0,-7.0]]

>>> print +a;
[2.0,[3.0,5.0,7.0]]
```

4.4.3.2 Binary operators

The following binary operators are implemented for quaternions:

Operator	Return type	1st Arg type	2nd Arg type	Semantics
+	Quaternion	Quaternion	Quaternion	Sum of two quaternions
−	Quaternion	Quaternion	Quaternion	Difference of two quaternions
*	Quaternion	Number	Quaternion	Product of number and quaternion
*	Quaternion	Quaternion	Number	Product of number and quaternion
*	Quaternion	Quaternion	Quaternion	(Non-commuting) Product of two quaternions
/	Quaternion	Quaternion	Number	Quotient of quaternion and number

Sum, difference, multiplication by a number, division by a number are the same as for four-dimensional vectors:

```
>>> print quatd(2,3,5,7) + quatd(1,2,4,8)
[3.0,[5.0,9.0,15.0]]

>>> print quatd(2,3,5,7) - quatd(1,2,4,8)
[1.0,[1.0,1.0,-1.0]]

>>> print quatd(2,3,5,7) * 10
[20,[30,50,70]]

>>> print quatd(2,3,5,7) / 10
[0.2,[0.3,0.5,0.7]]
```

The interesting property of quaternions is their non-commutative product. The product of two unit quaternions is a unit quaternion. Unit quaternions represent rotations in three-dimensional space (by some appropriate mapping), and multiplying two quaternions is like concatenating rotations. We use the operator * for this multiplication.

```
>>> print quatd(0,1,0,0) * quatd(0,0,1,0)
[0.0,[0.0,0.0,1.0]]
```

4.4.3.3 Augmenting operators

The following non-constant ("augmenting") operators are implemented for quaternions. The left hand side argument is always a quaternion. We refer to this argument as "**self**" in the following table:

Operator	Argument type	Semantics

Operator	Argument type	Semantics
+=	Quaternion	Add a quaternion to **self**
-=	Quaternion	Subtract a quaternion from **self**
***=**	Number	Multiply **self** by a number
/=	Number	Divide **self** by a number

Examples:

```
>>> a = quatd(2,3,5,7)

>>> b = quatd(a);b += quatd(1,2,4,8)
>>> print b
[3.0,[5.0,9.0,15.0]]

>>> b = quatd(a);b -= quatd(1,2,4,8)
>>> print b
[1.0,[1.0,1.0,-1.0]]

>>> b = quatd(a);b *= 10
>>> print b
[20.0,[30.0,50.0,70.0]]

>>> b = quatd(a);b /= 10
>>> print b
[0.2,[0.3,0.5,0.7]]
```

4.4.4 *Methods*

The following methods are implemented in **quatd**. The methods to not modify their object.

Method	Result type	Semantics
conjugate	Quaternion	Conjugate of the quaternion (vector part reverted)
invert	Quaternion	Returns the inverse quaternion with respect to *
dotsq	Number	Inner product of the quaternion (interpreted as four-dimensional vector) with itself
norm2	Number	Length of the quaternion (interpreted as four-dimensional vector)
unit	Quaternion	Returns the normalized quaternion

The Method **conjugate** returns the quaternional conjugate of a quaternion. This operation is analogue to **conjugate** for the built-in type **complex**:

```
>>> print quatd(2,3,5,7).conjugate()
[2.0,[-3.0,-5.0,-7.0]]

>>> print (3 + 4j).conjugate()
(3-4j)
```

invert calculates the inverse element. Note, that for unit quaternions it is faster and numerically more precise to use **conjugate** instead.

```
>>> a = quatd(2,3,5,7)
>>> print a * a.invert()
[1.0,[0.0,0.0,0.0]]
>>> print a.invert() * a
[1.0,[0.0,0.0,0.0]]
```

The methods **dotsq**, **norm2** and **unit** interpret the quaternion as a four-dimensional vector. Hence, these methods behave like the corresponding methods in **vec4d**:

```
>>> a = quatd(1,2,4,10)

>>> print a.dotsq()
121.0

>>> print a.norm2()
11.0

>>> print a.unit()
[0.0909090909091,[0.181818181818,0.363636363636,0.909090909091]]
```

4.4.5 *Type conversion*

Quaternions can be casted to four-dimensional vectors without loss of information via the constructor of **vec4d**:

```
>>> a = quatd(2,3,5,7)
>>> print vec4d(a)
[2.0,3.0,5.0,7.0]
```

For reasons of completeness there is a cast to four-dimensional matrices (injective homomorphism with respect to **+** and ***):

```
>>> print mat4d(quatd(2,3,5,7))
[[2.0,-3.0,-5.0,-7.0],[3.0,2.0,-7.0,5.0],[5.0,7.0,2.0,-3.0],[7.0,-5.0,3.0,2.0]]
```

4.5 Rotations

Rotation classes represent rotations in euclidian space without unveiling their inner representation to the application. You could, of course, use quaternions or matrices directly. Hence, the rotation classes should be considered as "convenience types" for dealing with rotational subsets of **mat2d**, **mat3d** and **quatd**. At present, VL supports rotations in two- and in three-dimensional space, by means of the classes **rot2d** and **rot3d**. Please note that angles are **always** represented in **radian, not in degree**.

4.5.1 *Constructor*

4.5.1.1 rot2d and rot3d

Rotations can be instantiated without parameters, which leads to the identity. The output format for **rot2d** is the rotation angle. For **rot3d** the output format is a quaternion.

```
>>> print rot2d()
0.0

>>> print rot3d()
[1.0,[0.0,0.0,0.0]]
```

Both classes can be instantiated by copying another instance.

```
>>> a = rot2d(0.1);b = rot2d(a)
>>> print b
0.1

>>> a = rot3d();b = rot3d(a)
>>> print b
[1.0,[0.0,0.0,0.0]]
```

The remaining ways of constructing rotations in two- and three-dimensional space are quite different, and we shall consider them separately.

4.5.1.2 rot2d

Two-dimensional rotations are initialized by passing an angle:

```
>>> print rot2d(0.2)
0.2
```

They can alse be initialized by a 2×2-matrix. The matrix passed is interpreted as rotation matrix. If this is not the case, VL projects the matrix into rotation space, so that there is no need to worry about numerical inaccuracies.

```
>>> phi = 0.3
>>> c = math.cos(phi);s = math.sin(phi)
>>> print rot2d(mat2d(c,-s,s,c))
0.3
```

Another way of initializing **rot2d** is to pass two vectors of type **vec2d**. This creates a rotation, which maps the unit vector of a to the unit vector of b. In the example above it is a counterclockwise rotation around 90° (= π / 2).

```
>>> a = vec(3,4);b = vec(-4,3);
>>> print rot2d(a,b)
1.57079632679
```

Finally, rotations in two-dimensional space are represented by unit complex numbers. The constructor projects the complex number passed onto a unit length number. We consider the result of passing 0+0j as undefined.

```
>>> a = 1 + 2j;
>>> print complex(rot2d(a)) * abs(a)
(1+2j)
```

4.5.1.3 rot3d

Three-dimensional rotations can be initialized by a (unit) quaternion. Note, that the string representation of **rot3d** objects is the same as for quaternions, regardless of the way they are constructed.

```
>>>  print rot3d(quatd(1,.1,.2,.3))
[0.707106781187,[0.0,0.707106781187,0.0]]
```

In analogy to **rot2d** they can also be initialized by a rotation matrix:

```
>>> from math import *
>>> c = cos(radians(60));s = sin(radians(60))
>>> m = mat3d(c,-s,0,s,c,0,0,0,1)
>>> print rot3d(m)
[0.866025403784,[-0.0,-0.0,0.5]]
```

In analogy to **rot2d** we can pass two three-dimensional vectors. The resulting rotation object maps the direction of the first vector onto to the direction of the second. In the following example a rotation of -90° around the z-axis is constructed:

```
>>> a = vec(0,1,0);b = vec(1,0,0)
>>> print rot3d(a,b)
[0.707106781187,[0.0,0.0,-0.707106781187]]
```

A very important method in numerics is to initialize **rot3d** by exponentiating a so-called Lie-Algebra-element, which is a three-dimensional vector:

```
>>> print rot3d(vec(.1,.2,.3))
[0.982550982155,[0.0497088433249,0.0994176866497,0.149126529975]]
```

Another way is to pass an axis vector and a rotation angle (in radian) The axis is a non-zero three-dimensional vector.

```
>>> print rot3d(vec(10,0,0),radians(90))
[0.707106781187,[0.707106781187,0.0,0.0]]
```

Finally, one of the most important ways in practice is to pass three Euler-angles and specifying their order of application. The order of application is represented by one of the following symbols:

- **VL_APPLY_XYZ**
- **VL_APPLY_XZY**
- **VL_APPLY_YXZ**
- **VL_APPLY_YZX**

- **VL_APPLY_ZXY**
- **VL_APPLY_ZYX**

For interpreting the names of these symbols please imagine rotating an object placed in a three dimensional coordinate frame. Now rotations of the object around *x*-, *y*- and *z*-axis of the coordinate frame are performed in the order as described by the symbol. Since the resulting numbers are quite unintuitive we restrict ourselves to combining some 90°-rotations. The following two sequences are equivalent:

- first rotate(-90°,z), then rotate(90°,y)
- first rotate(-90°,x), then rotate(-90°,y)

or in VL:

```
>>> print rot3d(0,math.radians(90),math.radians(-90),VL_APPLY_ZYX)
[0.5,[-0.5,0.5,-0.5]]
>>> print rot3d(math.radians(-90),0,math.radians(-90),VL_APPLY_XYZ)
[0.5,[-0.5,0.5,-0.5]]
```

4.5.2 *Operators*

Mathematically speaking, the class **rot3d** represents the group of rotations in three-dimensional space. This means, there is an (associative) operation ***** that maps two rotations into a new rotation, a unity element (given by the default constructor), and an inverse element for each object. We will illustrate the inverse later. The group operation looks like this (we concatenate two rotations around 180°):

```
>>> a = rot3d(quatd(0,1,0,0))
>>> b = rot3d(quatd(0,0,1,0))
>>> print a * b
[0.0,[0.0,0.0,1.0]]
```

We can also apply this operation to three-dimensional vectors:

```
>>> a = rot3d(quatd(1,0,0,1))
>>> print a * vec(1,0,0)
[0.0,1.0,0.0]
```

The operator ***** is not applicable to matrices (**mat3d**), since there is no "default" transformation behaviour of matrices.

4.5.3 *Methods*

As mentioned, **rot3d** represents a mathematical group, i.e. each element has an inverse. In VL the inverse mapping is given by the method **invert**:

```
>>> a = rot3d(quatd(1,0.1,0.2,0.3))
>>> print a * a.invert()
[1.0,[0.0,0.0,0.0]]
>>> print a.invert() * a
[1.0,[0.0,0.0,0.0]]
```

There are some methods used for converting the rotation object into various representations. There is a method that converts into a three-dimensional matrix:

```
>>> a = rot3d(0,radians(45),0,VL_APPLY_XYZ)
>>> print a.mat()
[[0.707106781187,0.0,0.707106781187],
 [0.0,1.0,0.0],
 [-0.707106781187,0.0,0.707106781187]]
```

In order to convert the rotation object into a unit quaternion the method **quat** is used

```
>>> a = rot3d(radians(90),radians(90),0,VL_APPLY_XYZ)
>>> print a.quat()
[0.5,[0.5,0.5,-0.5]]
```

The mapping from rotations to quaternions is not unique. A unit quaternion and its negative lead to the same rotation. Therefore, it is possible to pass a "hint" to this method, which is used to decide wether a quaternion or its negative is returned. This technique is useful in interpolating in rotation space. If you are dealing with this type of problems, please study the following example. The rotations *a* and *b* are very close to each other: the former is a rotation around +179°, the latter around -179°, which should be equivalent to rotating around +181°. Nevertheless, the represetation in unit quaternions leads to almost diametral quaternions. By passing a hint however, the quaternion representation of *b* becomes more like the representation of *a*, which reflects the actual situation in rotation space.

```
>>> a = rot3d(radians(179),0,0,VL_APPLY_XYZ)
>>> print a.quat()
[0.00872653549837,[0.999961923064,0.0,0.0]]

>>> b = rot3d(radians(-179),0,0,VL_APPLY_XYZ)
>>> print b.quat()
[0.00872653549837,[-0.999961923064,0.0,0.0]]

>>> print b.quat(a.quat())
[-0.00872653549837,[0.999961923064,-0.0,-0.0]]
```

Rotations can be represented by an axis an a rotation angle, as mentioned before. The axis and the angle are accessed by the methods **axis** and **angle**

```
>>> a = rot3d(vec(6,8,0),radians(90))
>>> print a.axis()
[0.6,0.8,0.0]
>>> print degrees(a.angle())
90.0
```

Finally, the rotation object can be represented by three angles with a given order of application, as we discussed in section 4.5.1.3. This is done using the method **angles**.

```
>>> a = rot3d(radians(20),radians(10),radians(30),VL_APPLY_ZYX)
>>> phi_x,phi_y,phi_z = a.angles(VL_APPLY_ZYX)
>>> print degrees(phi_x),degrees(phi_y),degrees(phi_z)
20.0 10.0 30.0
```

4.5.4 *Type conversion*

Objects of class **rot3d** can be converted to 3×3-matrices and to (unit-)quaternions. The conversion to quaternions is not unique. The result depends on the internal representation of the **rot3d**-object. This is due to the fact that the mapping from unit quaternions to the rotation group is a 2:1-mapping: a unit quaternion and its negative represent the same rotation.

```
>>> print quatd(rot3d())
[1.0,[0.0,0.0,0.0]]
>>> print mat3d(rot3d())
[[1.0,0.0,0.0],[0.0,1.0,0.0],[0.0,0.0,1.0]]
```

5 *Functions*

Some of the operators are also implemented as functions. The following tables shows functions with two vector-valued arguments:

Function	Dimension	Return type	1st Arg type	2nd Arg type	Semantics
dot	1 to 4	Number	Vector	Vector	Dot product, inner product
ten	1 to 4	Matrix	Vector	Vector	Tensor product, dyadic product
wdg	2	Number	Vector	Vector	Skew-symmetric product

Function	Dimension	Return type	1st Arg type	2nd Arg type	Semantics
`wdg`	3	Vector	Vector	Vector	Skew-symmetric product, vector product

Index

V

Document created with CDML from /server/devel/sdv/privat/uwe/source/vl_sdv_py/doc/cdml/vl.xml