veclib.tcl

This document contains a list of all commands of the Tcl vector library with examples and mathematical explanations, where necessary. This is the first released version (Feb. 2000) of the library and this document. The latest version can be downloaded from `www.sci-d-vis.com` or we will send it to you on demand (send a request to `info@sci-d-vis.com`). At the end of the document we have added a few notes about how to write an export filter.

Command: vget

Synopsis:

```
<real> vget $v $i
```

Parameter:

v	- vector
i	- integer from 0 to $\dim(v) - 1$

Description:

Returns component i of vector v

Example:

```
wish>set v {2 3 5}
wish>get $v 0
2
```

Command: vset

Synopsis:

```
<vec> vset v $i $s
```

Parameter:

v	- vector
i	- integer from 0 to $\dim(v) - 1$
s	- real

Description:

Sets component i of vector v to s. Returns v.

Example:

```
wish>set a {5 7 11}
wish>vset a 2 9
5 7 9
```

Command: mgetrow

Synopsis:

```
<vec> mgetrow $m $i
```

Parameter:

m	- matrix
i	- integer from 0 to number-of-rows(m) - 1

Description:

Returns row vector i of matrix m.

Example:

```
wish>set a {{5 7 11} {13 17 19} {23 29 31}}
wish>mgetrow $a 2
23 29 31
```

Command: msetrow

Synopsis:

```
<mat> msetrow m $i $v
```

Parameter:

m	- matrix
i	- integer from 0 to number-of-rows(m) - 1
v	- vector

Description:

Set the row vector i of matrix m to v. m is modified. Returns the new m.

Example:

```
wish>set a {{5 7 11} {13 17 19} {23 29 31}}
wish>msetrow a 0 {2 4 6}
{2 4 6} {13 17 19} {23 29 31}
```

Command: mget

Synopsis:

```
<real> mget $m $i $j
```

Parameter:

m	- matrix
i	- integer from 0 to number-of-rows(m) - 1
j	- integer from 0 to number-of-coloums(m) - 1

Description:

Returns the component (i, j) of matrix m.

Example:

```
wish>set a {{5 7 11} {13 17 19} {23 29 31}}
wish>mget $a 1 2
19
```

Command: mset

Synopsis:

```
<mat> mset m $i $j $c
```

Parameter:

m	- matrix
i	- integer from 0 to number-of-rows(m) - 1
j	- integer from 0 to number-of-coloums(m) - 1
c	- number

Description:

Sets component (i, j) of m to c. m is modified. Returns the new m.

Example:

```
wish>set a {{5 7 11} {13 17 19} {23 29 31}}
wish>mset a 2 0 8
{5 7 11} {13 17 19} {8 29 31}
```

Command: qget

Synopsis:

```
<real> qget $q $i
```

Parameter:

q	- quaternion
i	- integer from 0 to 3

Description:

Returns component i of quaternion q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qget $q 0
7
```

Command: qset

Synopsis:

```
<quat> qset q $i $s
```

Parameter:

q	- quaternion
i	- integer from 0 to 3
s	- number

Description:

Sets component i of quaternion q to s. q is modified. Returns the new q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qset q 1 5
7 {5 13 17}
```

Command: qgets

Synopsis:

```
<real> qgets $q
```

Parameter:

q	- quaternion

Description:

Returns the scalar part of quaternion q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qgets $q
7
```

Command: qsets

Synopsis:

```
<quat> qsets q $s
```

Parameter:

q	- quaternion
s	- number

Description:

Sets the scalar part of quaternion q to s. q is modified. Returns the new q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qsets q 55
55 {11 13 17}
```

Command: qgetvec

Synopsis:

```
<vec3> qgetvec $q
```

Parameter:

q	- quaternion

Description:

Returns the vector part of quaternion q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qgetvec $q
11 13 17
```

Command: qsetvec

Synopsis:

```
<quat> qsetvec q $v
```

Parameter:

q	- quaternion
v	- 3-dim vector

Description:

Sets the vector part of quaternion q to v. q is modified. Returns the new q.

Example:

```
wish>set q {7 {11 13 17}}
wish>qsetvec q {5 15 25}
7 {5 15 25}
```

Command: srtgetvec

Synopsis:

```
<vec> srtgetvec $l
```

Parameter:

l	- transformation

Description:

Returns the vector part of transformation l.

Example:

```
wish>set l [srt=vecmat {1 2} {{1 0 0} {0 1 0} {0 0 1}}]
wish>srtgetvec $l
1 2
```

Command: srtsetvec

Synopsis:

```
<srt> srtsetvec l $v
```

Parameter:

l	- transformation
v	- vector

Description:

Sets the vector part of transformation l to v. l is modified. Returns the new l.

Example:

```
wish>set l [srt=vecmat {1 2} {{1 0 0} {0 1 0} {0 0 1}}]
wish>srtsetvec l {3 5}
{3 5} {{1 0 0} {0 1 0} {0 0 1}}
```

Command: srtgetmat

Synopsis:

```
<mat> srtgetmat $l
```

Parameter:

l	- transformation

Description:

Returns the matrix part of transformation l.

Example:

```
wish>set l [srt=vecmat {1 2} {{1 0 0} {0 1 0} {0 0 1}}]
wish>srtgetmat $l
{1 0 0} {0 1 0} {0 0 1}
```

Command: srtsetmat

Synopsis:

```
<srt> srtsetmat l $m
```

Parameter:

l	- transformation
m	- matrix

Description:

Sets the matrix part of transformation l to m. l is modified. Returns the new l.

Example:

```
wish>set l [srt=vecmat {1 2} {{1 0 0} {0 1 0} {0 0 1}}]
wish>srtsetmat l {{11 12 13} {21 22 23} {31 32 33}}
{1 2} {{11 12 13} {21 22 23} {31 32 33}}
```

Command: vec2+vec2,vec2-vec2,vec3+vec3,vec3+vec3,quat+quat quat-quat,mat2+mat2,mat2-mat2,mat3+mat3,mat3-mat3

Synopsis:

```
<vec${n}> vec${n}${op}vec${n} $a $b
<quat> quat${op}quat $a $b
<mat${n}> mat${n}${op}mat${n} $a $b
```

Instantiation:

n is 2 or 3, *op* is + or -

Parameter:

a, b - n-dim vectors, quaternions or $n \times n$-matrices

Description:

Returns the sum or difference of a and b.

Example:

```
wish>vec2+vec2 {2 3} {5 7}
7 10
wish>vec3-vec3 {17 19 23} {2 3 5}
15 16 18
wish>quat+quat {1 {2 3 4}} {5 {6 7 8}}
6 {8 10 12}
wish>mat2-mat2 {{1 2} {3 4}} {{7 9} {11 13}}
{-6 -7} {-8 -9}
wish>mat3+mat3 {{1 2 3} {4 5 6} {7 8 9}} \
=>{{10 20 30} {40 50 60} {70 80 90}}
{11 22 33} {44 55 66} {77 88 99}
```

Command: -vec2,-vec3,-quat,-mat2,-mat3

Synopsis:

```
<vec${n}> -vec${n}$ a
<quat> -quat $a
<mat${n}> -mat${n} $a
```

Instantiation:

n is 2 or 3

Parameter:

a - n-dim vector, quaternion or $n \times n$-matrix

Description:

Returns the negative of a.

Example:

```
wish>-vec3 {5 6 7}
-5 -6 -7
wish>-quat {1 {2 3 4}}
-1 {-2 -3 -4}
```

Command: vec2*vec2,vec3*vec3,quat*quat

Synopsis:

```
<real> vec${n}*vec${n} $a $b
<real> quat*quat $a $b
```

Instantiation:

n is 2 or 3

Parameter:

a, b	- n-dim vectors or quaternions

Description:

Returns the scalar product of a and b. In case a and b are quaternions, they are interpreted as 4-dimensional real-valued vectors, the scalar product of which is a real number.

Example:

```
wish>vec2*vec2 {3 5} {7 9}
66
```

Command: vec2*s,vec3*s,quat*s,mat2*s,mat3*s

Synopsis:

```
<vec${n}> vec${n}*s $c $a
```

Instantiation:

n is 2 or 3

Parameter:

c	- number
a	- n-dim vector, quaterion or $n \times n$-matrix

Description:

Returns the object that results from scaling a by c

Example:

```
wish>s*vec3 10 {1 2 3}
10 20 30
wish>s*quat 10 {1 {2 3 4}}
10 {20 30 40}
wish>s*mat3 2 {{1 2 3} {4 5 6} {7 8 9}}
{2 4 6} {8 10 12} {14 16 18}
```

Command: vec2*s,vec3*s,quat*s,mat2*s,mat3*s

Synopsis:

```
<vec${n}> vec${n}*s $a $c
```

Instantiation:

n is 2 or 3

Parameter:

a	- n-dim vector, quaterion or $n \times n$-matrix
c	- number

Description:

Returns the object that results from scaling a by c.

Example:

```
wish>vec3*s {1 2 3} 10
10 20 30
```

Command: vec2/s,vec3/s,quat/s,mat2/s,mat3/s

Synopsis:

```
<vec${n}> vec${n}/s $a $c
```

Instantiation:

n is 2 or 3

Parameter:

a	- n-dim vector, quaterion or $n \times n$-matrix
c	- number

Description:

Returns the object that results from scaling a by $1/c$.

Example:

```
wish>vec3/s {1 2 3} 10
0.1 0.2 0.3
wish>quat/s {1 {2 3 4}} 10
0.1 {0.2 0.3 0.4}
wish>mat3/s {{1 2 3} {4 5 6} {7 8 9}} 2
{0.5 1.0 1.5} {2.0 2.5 3.0} {3.5 4.0 4.5}
```

Command: vec2^vec2

Synopsis:

```
<real> vec2^vec2 $a $b
```

Parameter:

a, b	- 2-dim vectors

Description:

Returns the skew-symmetric spinor product of a and b. This is defined as $a_0 b_1 - a_1 b_0$.

Example:

```
wish>vec2^vec2 {2 3} {7 5}
-11
```

Command: vec2&vec2,vec3&vec3

Synopsis:

```
<mat${n}> vec${n}&vec${n} $a $b
```

Instantiation:

n is 2 or 3

Parameter:

a, b - n-dim vectors

Description:

Returns the tensor product $a \otimes b$ of a and b.

Example:

```
wish>vec2&vec2 {2 3} {7 5}
{14 10} {21 15}
```

Command: vec2normquad,vec3normquad,quatnormquad

Synopsis:

```
<real> vec${n}normquad $a
<real> quatnormquad $a
```

Instantiation:

n is 2 or 3

Parameter:

a - n-dim vector or quaternion

Description:

Returns the square $|a|^2$ of the euclidian norm of a.

Example:

```
wish>vec3normquad {2 3 5}
38
```

Command: vec2norm,vec3norm,quatnorm

Synopsis:

```
<real> vec${n}norm $a
<real> quatnorm $a
```

Instantiation:

n is 2 or 3

Parameter:

a - n-dim vector or quaternion

Description:

Returns the euclidian norm $|a|$ of a.

Example:

```
wish>vec3norm {2 3 5}
6.16441
```

Command: vec2unit,vec3unit,quatunit

Synopsis:

```
<vec${n}> vec${n}unit $a
<quatunit> quatunit $a
```

Instantiation:

n is 2 or 3

Parameter:

a - n-dim non-zero vector or quaternion

Description:

Returns the normalized version of a.

Example:

```
wish>vec2unit {3 4}
0.6 0.8
```

Command: isvec2,isvec3

Synopsis:

```
<boole> isvec${n} $a
```

Instantiation:

n is 2 or 3

Parameter:

a - any type

Description:

Returns 1 if a consists of two components and 0 otherwise.

Example:

```
wish>isvec3 {1 2}
0
wish>isvec3 {1 2 3}
1
```

Command: vec3^vec3

Synopsis:

```
<vec3> vec3^vec3 $a $b
```

Parameter:

a, b - 3-dim vectors

Description:

Returns the skew-symmetric vector product $a \times b$ of a and b.

Example:

```
wish>vec3^vec3 {1 0 0} {0 1 0}
0 0 1
```

Command: vec3dual

Synopsis:

```
<mat3> vec3dual $v
```

Parameter:

v - 3-dim vector

Description:

Returns the dual matrix of vector v. This is defined as

$$({}^{*}v)_{ij} = \epsilon_{ijk} v_k$$

where ϵ_{ijk}, $i, j, k \in \{0, 1, 2\}$ is the totally skew-symmetric tensor with $\epsilon_{012} = 1$.

Example:

```
wish>set v {1 2 3}
wish>vec3dual $v
{0 3 -2} {-3 0 1} {2 -1 0}
```

Command: quat=s

Synopsis:

```
<quat> quat=s $c
```

Parameter:

c - number

Description:

Generates a scalar-type quaternion $(c, (0, 0, 0))$.

Example:

```
wish>quat=s 5
5 {0 0 0}
```

Command: quat=vec3

Synopsis:

```
<quat> quat=vec3 $a
```

Parameter:

a - 3-dim vector

Description:

Generates a vector-type quaternion $(0, a)$.

Example:

```
wish>quat=vec3 {5 7 9}
0 {5 7 9}
```

Command: quat^quat

Synopsis:

```
<quat> quat^quat $q $r
```

Parameter:

q, r - quaternions

Description:

Returns the non-commutative product of quaternions q and r. For each two quaternions $q = (s_q, v_q)$ and $r = (s_r, v_r)$ this is defined as

$$qr = (s_q s_r - v_q \cdot v_r, s_q v_r + s_r c_q + v_q \times v_r).$$

Command: quatconj

Synopsis:

```
<quat> quatconj $a
```

Parameter:

a - quaternion

Description:

Returns the conjugate quaternion $\bar{a}$ of a.

Example:

```
wish>quatconj {1 {2 3 4}}
1 {-2 -3 -4}
```

Command: quatinvert

Synopsis:

```
<quat> quatinvert $a
```

Parameter:

a - quaternion

Description:

Returns the inverse quaternion a^{-1} of a.

Example:

```
wish>puts $b
1 {2 3 4}
wish>set a [quatinvert {1 {2 3 4}}]
wish>puts $a
0.0333333 {-0.0666667 -0.1 -0.133333}
wish>quat^quat $a {1 {2 3 4}}
0.999998 {-1.1e-06 7e-07 -1e-07}
```

Command: quatmaptomat3

Synopsis:

```
<mat3> quatmaptomat3 $q
```

Parameter:

q - quaternion

Description:

Maps quaternion q into a 3×3-matrix. If q is a unit quaternion, the resulting matrix is a pure rotation matrix. If q has the length $|q|$, the resulting matrix is a rotation matrix multiplied by a scaling factor of $|q|^2$. The quaternions q and $-q$ are mapped to the same matrix. Technically speaking, the procedure represents the 2-on-1 homomorphism from the unit quaternions $\hat{H}$ to the rotation group SO(3).

Example:

```
wish>quatmaptomat3 {2 {0 0 0}}; # Pure scaling matrix
{4.0 0.0 0.0} {0.0 4.0 0.0} {0.0 0.0 4.0}
wish>quatmaptomat3 {0 {1 0 0}}; # Rotation by 180 deg. around x-axis
{1.0 0.0 0.0} {0.0 -1.0 0.0} {0.0 0.0 -1.0}
wish>quatmaptomat3 {0 {0 1 0}}; # Rotation by 180 deg. around y-axis
{-1.0 0.0 0.0} {0.0 1.0 0.0} {0.0 0.0 -1.0}
wish>quatmaptomat3 {0 {0 0 1}}; # Rotation by 180 deg. around z-axis
{-1.0 0.0 0.0} {0.0 -1.0 0.0} {0.0 0.0 1.0}
```

Command: mat2*mat2,mat3*mat3,mat4*mat4,srt2*srt2,srt3*srt3

Synopsis:

```
<mat${n}> mat${n}*mat${n} $a $b
```

Instantiation:

n is 2, 3 or 4

Parameter:

a, b - $n \times n$-matrix or 2-dim or 3-dim transformation

Description:

Returns the product of a and b.

Example:

```
wish>mat2*mat2 {{2 3} {5 7}} {{1 4} {6 8}}
{20 32} {47 76}
```

Command: mat2*vec2,mat3*vec3

Synopsis:

```
<vec${n}> mat${n}*vec${n} $a $b
```

Instantiation:

n is 2 or 3

Parameter:

a - $n \times n$-matrix
b - n-dim vector

Description:

Returns the product of a and b.

Example:

```
wish>mat2*vec2 {{2 3} {5 7}} {4 6}
26 62
```

Command: mat2=s,mat3=s

Synopsis:

```
<mat${n}> mat${n}=s $c
```

Instantiation:

n is 2 or 3

Parameter:

c - number

Description:

Returns a scalar-type matrix $\mathbf{1} \cdot c$.

Example:

```
wish>mat2=s 7
{7 0} {0 7}
wish>mat3=s 5
{5 0 0} {0 5 0} {0 0 5}
```

Command: mat2=diag,mat3=diag

Synopsis:

```
<mat2> mat2=diag $c $d
<mat3> mat3=diag $c $d $e
```

Parameter:

c, d, e - numbers

Description:

Returns the diagonal matrix $\mathrm{diag}(c, d)$ or $\mathrm{diag}(c, d, e)$, respectively.

Example:

```
wish>mat2=diag 2 4
{2 0} {0 4}
wish>mat3=diag 1 3 5
{1 0 0} {0 3 0} {0 0 5}
```

Command: mat2trace,mat3trace

Synopsis:

```
<real> mat${n}trace $a
```

Instantiation:

n is 2 or 3

Parameter:

a - $n \times n$-matrix

Description:

Returns the trace $\mathrm{tr}(a)$ of a.

Example:

```
wish>mat3trace {{1 2 3} {4 5 6} {7 8 9}}
15
```

Command: mat2trans,mat3trans

Synopsis:

```
<mat${n}> mat${n}trans $a
```

Instantiation:

n is 2 or 3

Parameter:

a	- $n \times n$-matrix

Description:

Returns the transposed a^{T} of a.

Example:

```
wish>mat2trans {{1 2} {3 4}}
{1 3} {2 4}
```

Command: mat2det,mat3det

Synopsis:

```
<real> mat${n}det $a
```

Instantiation:

n is 2 or 3

Parameter:

a	- $n \times n$-matrix

Description:

Returns the determinant of a.

Example:

```
wish>mat2det {{3 5} {7 9}}
-8
```

Command: mat2subcyc,mat3subcyc

Synopsis:

```
<real> mat2subcyc $a $i $j
```

Instantiation:

n is 2 or 3

Parameter:

a	- $n \times n$-matrix
i	- integer from 0 to number-of-rows(a) - 1
j	- integer from 0 to number-of-coloums(a) - 1

Description:

Returns the adjoint (cyclic) sub-matrix of a with respect to the component (i, j)

Example:

```
wish>mat2subcyc {{11 12} {21 22}} 0 0
22
wish>mat2subcyc {{11 12} {21 22}} 0 1
21
wish>mat2subcyc {{11 12} {21 22}} 1 0
12
wish>mat2subcyc {{11 12} {21 22}} 1 1
11
wish>mat3subcyc {{11 12 13} {21 22 23} {31 32 33}} 0 1
{23 21} {33 31}
wish>mat3subcyc {{11 12 13} {21 22 23} {31 32 33}} 2 1
{13 11} {23 21}
```

Command: mat2invert,mat3invert,srt2invert,srt3invert

Synopsis:

```
<mat${n}> mat${n}invert $a
```

Instantiation:

n is 2 or 3

Parameter:

a - $n \times n$-matrix or n-dim transformation

Description:

Returns the inverse matrix or transformation a^{-1} of a.

Example:

```
wish>set b {{1 2} {-2 -1}}
wish>puts $b
{1 2} {-2 -1}
wish>set a [mat2invert $b]
wish>puts $a
{-0.333333 -0.666667} {0.666667 0.333333}
wish>mat2*mat2 $a $b
{1.0 1e-06} {1e-06 1.0}
```

Command: mat2rotangle

Synopsis:

```
<real> mat2rotangle $m
```

Parameter:

m - 2×2 rotation matrix

Description:

Calculates the rotation angle (in radiant) out of the rotation atrix m.

Example:

```
wish>mat2rotangle { {0 1} {-1 0} }; # 90 degree clockwise = -pi/2
-1.5708
```

Command: mat2rotmatrix

Synopsis:

```
<mat2> mat2rotmatrix $w
```

Parameter:

w - angle in radiant

Description:

Calculates the 2×2 rotation matrix out of the angle (in radiant) w.

Example:

```
wish>mat2rotmatrix -1.570796327
{-2.05103e-10 1.0} {-1.0 -2.05103e-10}
```

Command: ismat2,ismat3

Synopsis:

```
<boole> ismat${n} $a
```

Parameter:

a - any type

Description:

Returns 1 if a consists of two components and each component is an n-dim vector and 0 otherwise.

Example:

```
wish>ismat2 {{1 0 0} {0 1 0} {0 0 1}}
0
wish>ismat2 {{1 0} {0 1}}
1
```

Command: mat3rotmatrix

Synopsis:

```
<mat3> mat3rotmatrix $i $j $k $wi $wj $wk
```

Parameter:

(i, j, k) - a permutation of $(0, 1, 2)$

w_i, w_j, w_k - rotation angles in radiant

Description:

In order to describe this procedure we first define the three coordinate axis x, y, z as e_0, e_1, e_2. For any $i \in \{0, 1, 2\}$, w_i is the rotation angle around the axis e_i. Let $R_i(w)$ be the rotation matrix around the axis e_i by an angle of w:

$$R_0(w) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(w) & -\sin(w) \\ 0 & \sin(w) & \cos(w) \end{pmatrix}$$

$$R_1(w) = \begin{pmatrix} \cos(w) & 0 & \sin(w) \\ 0 & 1 & 0 \\ -\sin(w) & 0 & \cos(w) \end{pmatrix}$$

$$R_2(w) = \begin{pmatrix} \cos(w) & -\sin(w) & 0 \\ \sin(w) & \cos(w) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Then the matrix generated by this procedure is

$$m = R_i(w_i)R_j(w_j)R_k(w_k)$$

Example:

```
wish>set pi/2 1.570796327; # that is 90 degree clockwise
wish>set -pi/2 -1.570796327; # that is 90 degree counterclockwise
wish>mat3rotmatrix 0 1 2 ${pi/2} 0 0
{1.0 0.0 0.0} {0.0 -2.05103e-10 -1.0} {0.0 1.0 -2.05103e-10}
```

$$m = R_0(\frac{\pi}{2}) \cdot \mathbf{1} \cdot \mathbf{1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}$$

```
wish>mat3rotmatrix 0 1 2 0 ${pi/2} 0
{-2.05103e-10 0.0 1.0} {0.0 1.0 0.0} {-1.0 0.0 -2.05103e-10}
```

$$m = \mathbf{1} \cdot R_1(\frac{\pi}{2}) \cdot \mathbf{1} = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 1 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

```
wish>mat3rotmatrix 0 1 2 0 0 ${pi/2}
{-2.05103e-10 -1.0 0.0} {1.0 -2.05103e-10 0.0} {0.0 0.0 1.0}
```

$$m = \mathbf{1} \cdot \mathbf{1} \cdot R_2(\frac{\pi}{2}) = \begin{pmatrix} 0 & -1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
wish>mat3rotmatrix 1 2 0 ${pi/2} ${-pi/2} 0
{4.20672e-20 -2.05103e-10 1.0} {-1.0 -2.05103e-10 0.0} \
{2.05103e-10 -1.0 -2.05103e-10}
```

$$m = R_1(\frac{\pi}{2})R_2(-\frac{\pi}{2}) \cdot \mathbf{1} = \begin{pmatrix} 0 & 0 & 1 \\ -1 & 0 & 0 \\ 0 & -1 & 0 \end{pmatrix}$$

Command: mat3rotangles

Synopsis:

```
<list of three numbers> mat3rotangles $m $i $j $k
```

Parameter:

m	- 3x3-matrix
(i, j, k)	- a permutation of $(0, 1, 2)$

Description:

Calculates three rotation angles (in radiant) out of a rotation matrix, which correspond to rotations around x-, y- and z-axis. The parameters i, j and k determine, which component of the returned tupel corresponds to which axis. This is the inverse function of `mat3rotmatrix` (except for the common ambiguities in angle space, which form a set of measure zero).
In order to describe this procedure we define the three coordinate axis x, y, z as e_0, e_1, e_2. For any $i \in \{0, 1, 2\}$, w_i is the rotation angle around the axis e_i. Let $R_i(w)$ be the rotation matrix around the axis e_i by an angle of w as defined under `mat3rotmatrix`. The procedure generates a partition of m in terms of $R_0(w_0)$, $R_1(w_1)$ and $R_2(w_2)$, so that the order of multiplying these matrices is reflected by the permutation (i, j, k). The returned tupel is (w_i, w_j, w_k). This procedure is useful for converting the matrix representation of rotations into six other, right-handed angle representations. Six, because there are six possibilities to permutate the multiplication order of the three axis rotations R_0, R_1 and R_2.

Example:

```
wish>set m [mat3rotmatrix 1 2 0 0.7 0.6 0.5]
wish>mat3rotangles $m 1 2 0
0.7 0.599999 0.500001
```

$$m = R_1(0.7)R_2(0.6)R_0(0.5)$$

```
wish># Nota bene:
wish>
wish>set m [mat3rotmatrix 0 1 2 8 0 0]
wish>mat3rotangles $m 0 1 2
1.71681 0.0 0.0
wish>
wish># That is (8 modulo 2 * pi) 0 0
```

Command: mat3dual

Synopsis:

```
<vec3> mat3dual $m
```

Parameter:

m	- matrix

Description:

Contracts matrix m with the totally skew-symmetric tensor ϵ_{ijk}, $i,j,k \in \{0,1,2\}$, $\epsilon_{012} = 1$. This is the inverse function of `vec3dual`.

$$({}^*m)_i = \epsilon_{ijk} m_{jk}$$

Note, that `vec3dual` is not inverse to `mat3dual`, because `vec3dual` is injective but not surjective, and `mat3dual` is surjective but not injective.

```
wish>set m [vec3dual {2 3 1}]
wish>puts $m
{0 1 -3} {-1 0 2} {3 -2 0}
wish>mat3dual $m
2.0 3.0 1.0
```

Command: mat3maptoquat

Synopsis:

```
<quat> mat3maptoquat $m
```

Parameter:

m	- 3×3 rotation/scaling matrix

Description:

Maps the rotation and uniform scaling matrix m onto the quaternions. A uniform scaling factor f of m leads to a quaternion of length $\sqrt{f}$. Only the positive hemisphere of the quaternions is covered, i.e. the resulting quaternion has a non-negative scalar part. This is the inverse function of the 2-on-1 homomorphism `quatmaptomat3`, and it is a homomorphism as well. This procedure can be used to convert the matrix representations of rotations into the quaternion representation.

```
wish>mat3maptoquat {{1 0 0} {0 0 1} {0 -1 0}}
0.707107 {-0.707107 0.0 0.0}
wish>quatmaptomat3 { 0.707107 {-0.707107 0.0 0.0}}
{1.0 0.0 0.0} {0.0 0.0 0.999998} {0.0 -0.999998 0.0}
```

Command: srt2*vec2,srt3*vec3

Synopsis:

```
<vec${n}> srt${n}*vec${n} $l $v
```

Instantiation:

n is 2 or 3

Parameter:

l	- n-dim transformation
v	- n-dim vector

Description:

Applies transformation l to vector v. For a transformation $l = (v_l, m_l)$ this is defined as

$$lv = v_l + m_l v$$

When m_l is a rotation and scaling matrix and v is interpreted as a point, this means that v is rotated and scaled with respect to the origin, and translated by v_l afterwards.

```
wish>set l {{1 2 3} {{1 0 0} {0 0 1} {0 -1 0}}}
wish>set v {0 50 0}
wish>srt3*vec3 $l $v
1 2 -47
```

Command: srt2*mat2,srt3*mat3

Synopsis:

```
<mat${n}> srt${n}*mat${n} $l $m
```

Instantiation:

n is 2 or 3

Parameter:

l	- n-dim transformation
m	- $n \times n$ matrix

Description:

Applies transformation l to matrix m. For a transformation $l = (v_l, m_l)$ this is defined as

$$lm = m_l m$$

The vector part of l is ignored.

```
wish>set l {{1 2 3} {{1 0 0} {0 0 1} {0 -1 0}}}
wish>set m {{1 2 3} {4 5 6} {7 8 9}}
wish>srt3*mat3 $l $m
{1 2 3} {7 8 9} {-4 -5 -6}
```

Command: mat3=srt2,mat4=srt3

Synopsis:

```
<mat3> mat3=<mat2> $l mat4=srt3 $l
```

Instantiation:

n is 2 or 3

Parameter:

l	- n-dim transformation

Description:

Generates an $(n+1) \times (n+1)$-matrix representation out of transformation l. For the case of $n = 3$ this is the 4×4 representation commonly used in computer graphics. This conversion is a homomorphism from `srt3` to `mat4` with respect to their multiplications.

```
wish>set l {{40 50 60} {{44 45 46} {54 55 56} {64 65 66}}}
wish>mat4=srt3 $l
{44 45 46 40} {54 55 56 50} {64 65 66 60} { 0 0 0 1 }
wish>set r {{1 2 3} {{11 12 13} {21 22 23} {31 32 33}}}
wish>mat4=srt3 [srt3*srt3 $l $r]
{2855 2990 3125 312} {3485 3650 3815 382} {4115 4310 4505 452} { 0 0 0 1 }
wish>mat4*mat4 [mat4=srt3 $l] [mat4=srt3 $r]
{2855 2990 3125 312} {3485 3650 3815 382} {4115 4310 4505 452} {0 0 0 1}
```

Writing an export filter

When you develop (and publish) an export filter in Tcl, there are some points you should take care of. In the following we present a very simple export filter, which exports the camera / object motion curve of all frame objects for the selected point group. For simplicity, we do not consider the points and the position part of the motion curve here, but merely rotation part of the the motion paths. Please consider the following Tcl source file. The aim was to extract the rotational part of the motion curve in terms of quaternions.

```
(1)     source /usr/3dequalizer/user_data/tcl_archive/veclib.tcl
        set pgroup_id [getSelectedPGroup]
        set pgroup_name [getPGroupName $pgroup_id]
        puts "Name of the selected Pointgroup: $pgroup_name"
(5)     for {set fobj_id [getFirstFobj]} {$fobj_id != 0} {set fobj_id [getNextFobj $fobj_id]} {
                set fobj_name [getFobjName $fobj_id]
                puts "Name of the frame object: $fobj_name"

                set fobj_num_frames [getFobjNoFrames $fobj_id]
(10)            puts "Number of frames: $fobj_num_frames"

                for {set f 1} {$f <= $fobj_num_frames} {incr f} {
                        set rot_matrix [getPGroupRotation3D $pgroup_id $fobj_id $f]
                        set rot_quat [mat3maptoquat $rot_matrix]
(15)
                        if {$f > 1} {
                                if {[quat*quat $rot_quat $rot_quat_prev] < 0} {
                                set rot_quat [-quat $rot_quat]
                                set neg 1
(20)                            } \
                        else {
                                set neg 0
                                }
                        }
(25)            puts -nonewline "Frame $f , Quaternion $rot_quat"
                if {$f > 1} {
                        if {$neg == 1} {
                                puts "   #negated"
                                } \
(30)                    else {
                                puts ""
                                }
                        } \
                else {
(35)                    puts ""
                        }
                set rot_quat_prev $rot_quat
```

```
            }
        }
```

The output produced by this segment has the form:

```
Name of the selected Pointgroup: Fussboden
Name of the frame object: two_pgroups
Number of frames: 96
Frame 1 , Quaternion 0.788143 {0.0 0.615491 -0.001}
Frame 2 , Quaternion 0.808067 {0.000707107 0.58909 -0.00122474}
Frame 3 , Quaternion 0.827086 {0.00122474 0.562074 -0.00141421}
...
Frame 67 , Quaternion 0.0503984 {-0.000707107 -0.998729 0.0}
Frame 68 , Quaternion 0.0173061 {-0.000707107 -0.999849 0.000707107}
Frame 69 , Quaternion -0.0158509 {-0.000866025 -0.999874 0.0005}   #negated
Frame 70 , Quaternion -0.0489336 {-0.000707107 -0.998802 0.000707107}   #negated
...
Frame 94 , Quaternion -0.74579 {-0.000707107 -0.666181 0.001}   #negated
Frame 95 , Quaternion -0.767379 {-0.000707107 -0.641193 0.000707107}   #negated
Frame 96 , Quaternion -0.788159 {-0.000707107 -0.615471 0.001}   #negated
```

Most of the code lines are quite straight forward: in line (1) the vector library is parsed. Line (2-3): get the current point group. Line (5): a loop over the frame objects. Line (9): get the number of frames of the current frame object. Line (12): a loop over the frames of the current frame object. Line (13): get the rotation matrix for the current frame. Line (14): Convert into a quaternion.

Now, the interesting part is in lines (16-24). Normally, one might think that line (25) would be sufficient for generating the array of quaternions. However, if you do this, there can be a hard flip-around in the space of quaternions, when the motion path exceeds the positive hemisphere. Remember, that the procedure `mat3maptoquat` by definition produces quaternions with non-negative scalar part. Let us now assume, that your rendering software interpolates the camera motion path with sub-frame precision. This can be necessary when in interlaced video sequences the even fields are to be calculated, but within 3DE the odd fields were tracked. At that moment, the motion path leaves the positive hemisphere, the quaternion value flips, and the sub-frame interpolation of the renderer produces a wrong result. The solution to this problem is, that we *detect* such a flip, and remove it. The criterion for a flip is very simple. When two neighbouring quaternions have a negtive scalar product, this indicates that the rotational change from one frame to the other was about 360 degree, which is of course wrong for smooth camera movements! Therefore, when the scalar product is negative, we multiply the quaternion by -1 and project it into the negative hemisphere. Remember that a quaternion and its negative produce the same rotation, hence we are not making a mistake by doing this. However, the motion path in quaternions is now able to leave the positive hemisphere, and remains smooth, i.e. the renderer will not have any difficulty interpolating the motion path with sub-frame precision.