transforms3d Documentation

Release 0.4.2

Matthew Brett

Sep 19, 2025

CONTENTS

This package is a collection of Python functions and classes to create and convert 3-dimensional transformations such as rotations, zooms, shears and reflections.

It is based on transformations.py by Christoph Gohlke, with some restructuring by Matthew Brett.

Install the latest version with:

```
pip install transforms3d
```

CHAPTER

ONE

CONVENTIONS FOR TRANSFORMS

For the transforms we have coded here, we have followed some conventions that are not universal, but seem common enough.

Transformation matrices are matrices for application on the left, applied to coodinates on the right, stored as column vectors.

Let's say you are transforming a set of points. The points are vectors $v^1, v^2 \dots v^N$, and the vectors are composed of x, y and z coordinates, so that $v^1 = \left(x^1, y^1, z^1\right)$, then your points matrix P would look like:

$$P = \begin{pmatrix} x^1, x^2, \dots, x^N \\ y^1, y^2, \dots, y^N \\ z^1, z^2, \dots, z^N \end{pmatrix}$$

If we are applying a 3x3 transformation matrix M, to transform points P, then the transformed points v' are given by:

$$v' = M \cdot P$$

CHAPTER

TWO

NAMING CONVENTIONS

In the code, we try to abbreviate common concepts in a standard way.

- *aff* - 4 x 4 *affine matrix* for operating on homogenous coordinates of shape (4,) or (4, N);
- *mat* - 3 x 3 transformation matrix for operating on non-homogenous coordinate vectors of shape (3,) or (3, N). A *rotation matrix* is an example of a transformation matrix;
- *euler* - *euler angles* - sequence of three scalars giving rotations about defined axes;
- *axangle* - *axis angle* - axis (vector) and angle (scalar) giving axis around which to rotate and angle of rotation;
- *quat* - *quaternion* - shape (4,);
- *rfnorm* : reflection in plane defined by normal (vector) and optional point (vector);
- *zfdir* : zooms encoded by factor (scalar) and direction (vector);
- *striu* : shears encoded by vector giving triangular portion above diagonal of N x N array (for ND transformation);
- *sadn* : shears encoded by angle scalar, direction vector, normal vector (with optional point vector).

CHAPTER

THREE

GIMBAL LOCK

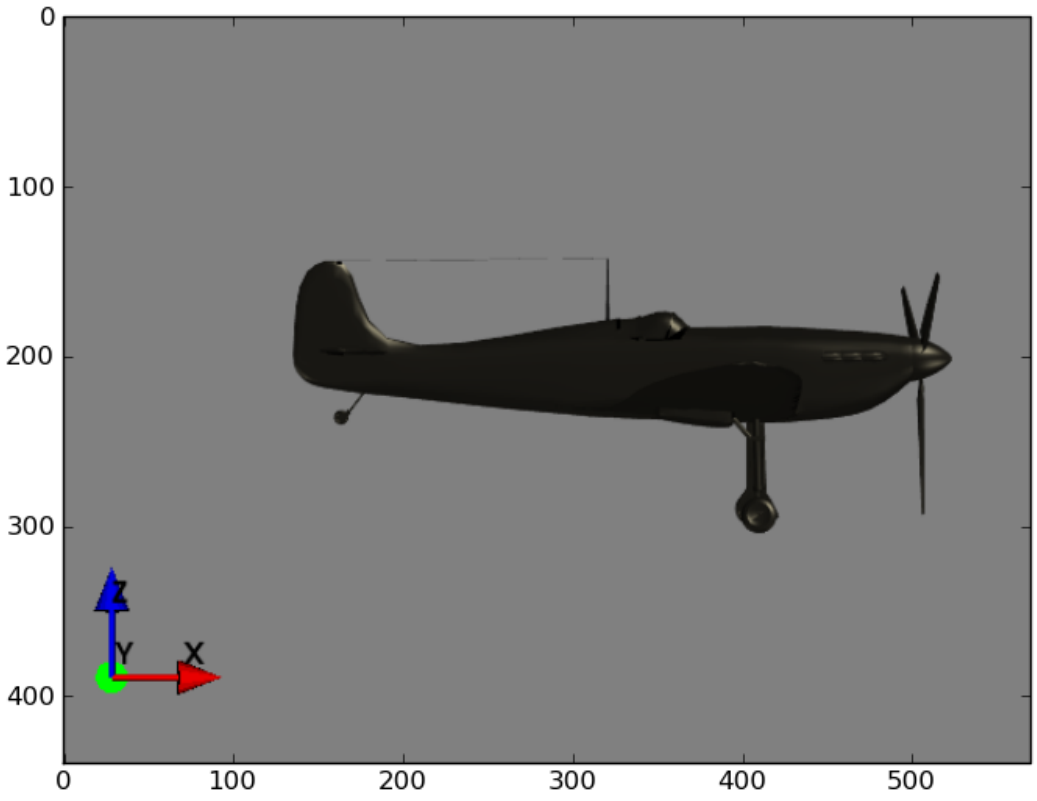
See also: http://en.wikipedia.org/wiki/Gimbal_lock

Euler angles have a major deficiency, and that is, that it is possible, in some rotation sequences, to reach a situation where two of the three Euler angles cause rotation around the same axis of the object. In the case below, rotation around the x axis becomes indistinguishable in its effect from rotation around the z axis, so the z and x axis angles collapse into one transformation, and the rotation reduces from three degrees of freedom to two.

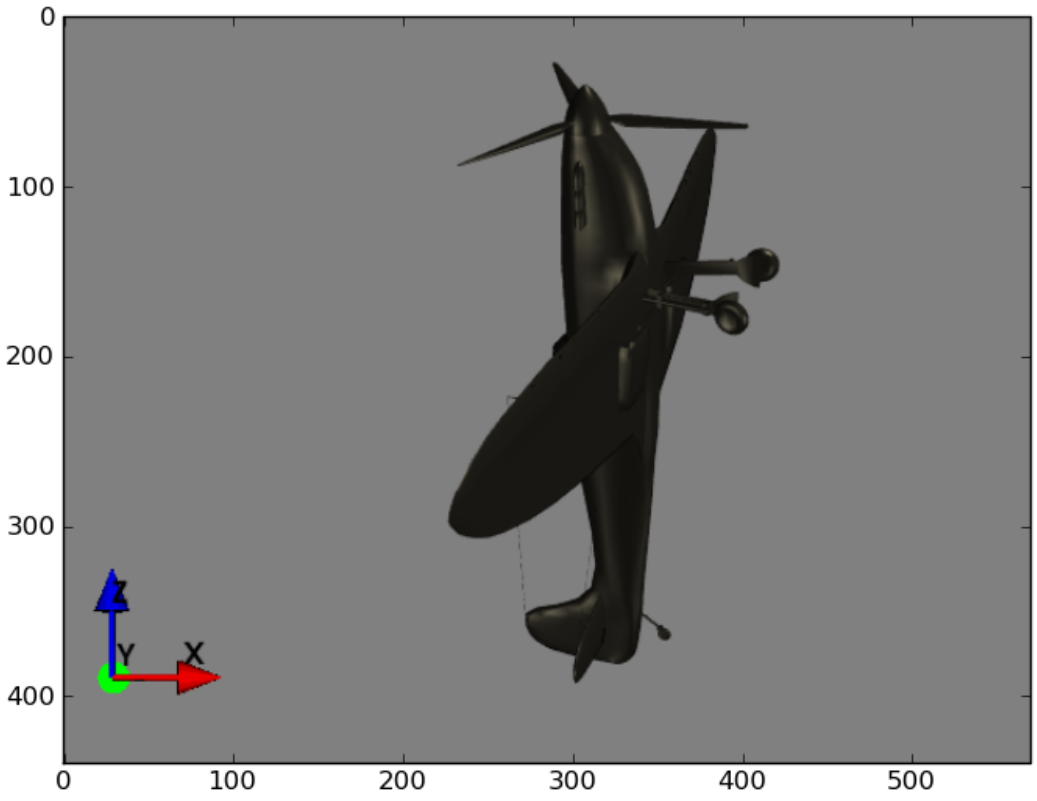
3.1 Example

Imagine that we are using the Euler angle convention of starting with a rotation around the x axis, followed by the y axis, followed by the z axis.

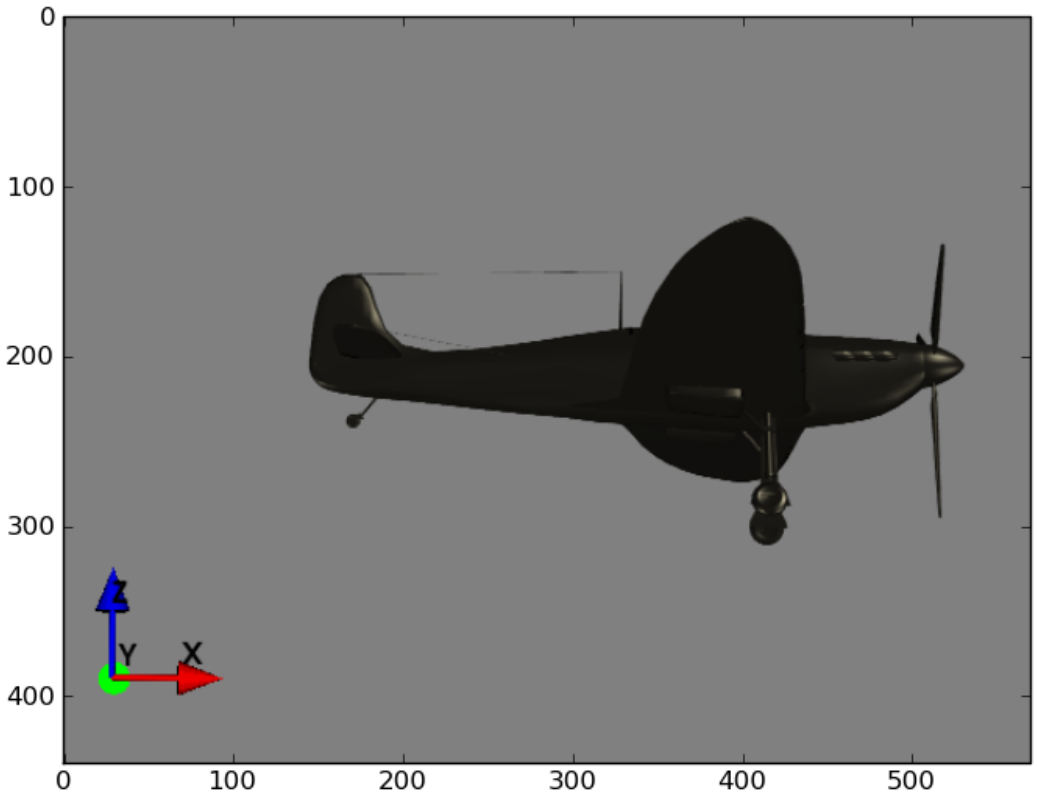
Here we see a Spitfire aircraft, flying across the screen. The x axis is left to right (tail to nose), the y axis is from the left wing tip to the right wing tip (going away from the screen), and the z axis is from bottom to top:



We want to rotate the aircraft to look something like this:

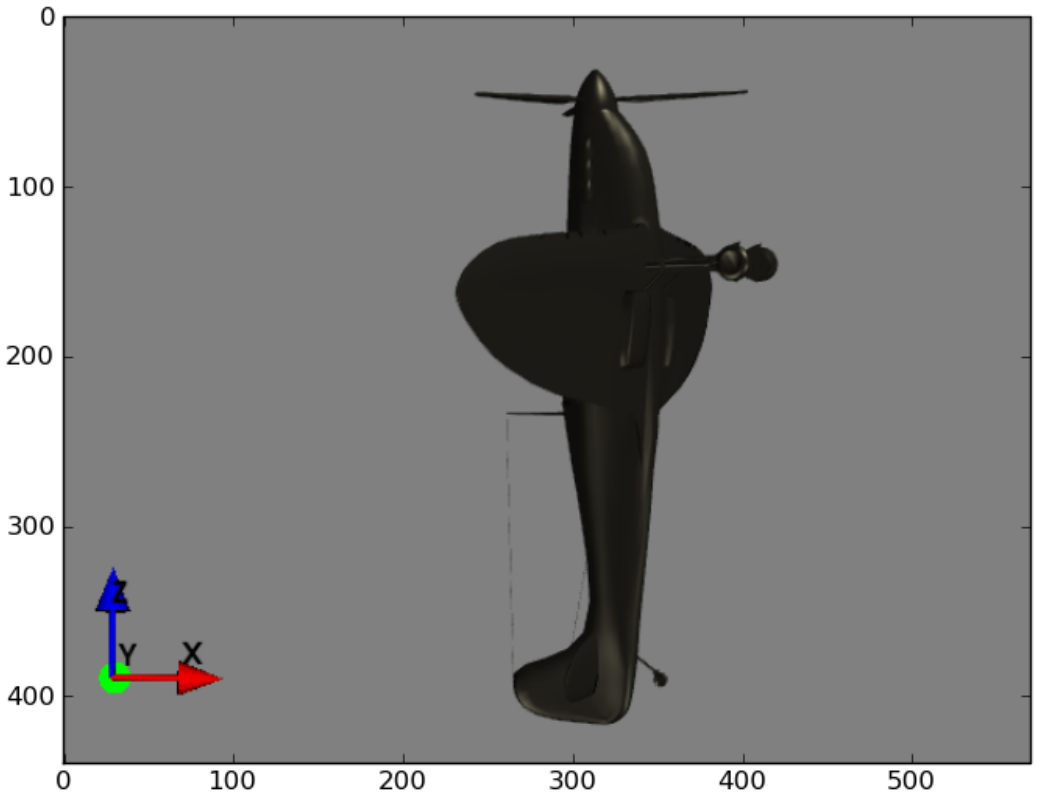


We might start by doing an x rotation, making a slight roll with the left wing tilting down (rotation about x) like this:



Let's say that the x rotation is -0.2 radians.

Then we do a pitch so we are pointing straight up (rotation around y axis). This is a rotation by $-\pi/2$ radians.



To get to our desired position from here, we need to do a turn of something like 0.2 radians of the nose towards the viewer (and the tail away from the viewer). All we have left is our z rotation (rotation around the z axis. Unfortunately, the current result of a rotation around the z axis has now become the same as a previous rotation around the x axis. To see this, look at the result of the rotation around the y axis. Notice that the x axis, as was, is now aligned with the z axis, as it is now. Rotating around the z axis will have exactly the same effect as adding an extra rotation around the x axis at the beginning. That means that, when there is a y axis rotation that rotates the x axis onto the z axis (a rotation of $\pm\pi/2$ around the y axis) - the x and y axes are "locked" together.

This does not mean that we cannot do the rotations we need, only that we can't do them by starting with the most obvious x and y rotations. In fact what we will have to do is first rotate around x by $\pi/2 - 0.2$ radians, then do a y rotation of $-\pi/2 + 0.2$ radians, and finally a z rotation of $-\pi/2$ radians. See the code below for the details.

3.2 Mathematics of gimbal lock

See *transforms3d.derivations.eulerangles*.

We see gimbal lock for this type of Euler axis convention, when $\cos(\beta) = 0$, where β is the angle of rotation around the y axis. By "this type of convention" we mean using rotation around all 3 of the x, y and z axes, rather than using the same axis twice - e.g. the physics convention of z followed by x followed by z axis rotation (the physics convention has different properties to its gimbal lock).

We can show how gimbal lock works by creating a rotation matrix for the three component rotations. Recall that, for a rotation of α radians around x, followed by a rotation β around y, followed by rotation γ around z, the rotation matrix

R is:

$$R = \begin{bmatrix} \cos(\beta)\cos(\gamma) & \sin(\alpha)\sin(\beta)\cos(\gamma) - \sin(\gamma)\cos(\alpha) & \sin(\alpha)\sin(\gamma) + \sin(\beta)\cos(\alpha)\cos(\gamma) \\ \sin(\gamma)\cos(\beta) & \sin(\alpha)\sin(\beta)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\gamma) + \sin(\beta)\sin(\gamma)\cos(\alpha) \\ -\sin(\beta) & \sin(\alpha)\cos(\beta) & \cos(\alpha)\cos(\beta) \end{bmatrix}$$

In our case the y rotation $\beta = -\pi/2, \cos(\beta) = 0, \sin(\beta) = -1$:

$$R = \begin{bmatrix} 0 & -\sin(\alpha)\cos(\gamma) - \sin(\gamma)\cos(\alpha) & \sin(\alpha)\sin(\gamma) - \cos(\alpha)\cos(\gamma) \\ 0 & -\sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma) & -\sin(\alpha)\cos(\gamma) - \sin(\gamma)\cos(\alpha) \\ 1 & 0 & 0 \end{bmatrix}$$

From the angle sum and difference identities (see also geometric proof, Mathworld treatment) we remind ourselves that, for any two angles α and β:

$$\sin(\alpha \pm \beta) = \sin\alpha\cos\beta \pm \cos\alpha\sin\beta$$
$$\cos(\alpha \pm \beta) = \cos\alpha\cos\beta \mp \sin\alpha\sin\beta$$

We can rewrite R as:

$$R = \begin{bmatrix} 0 & -W_1 & -W_2 \\ 0 & W_2 & -W_1 \\ 1 & 0 & 0 \end{bmatrix}$$

where:

$$W_1 = \sin(\alpha)\cos(\gamma) + \sin(\gamma)\cos(\alpha) = \sin(\alpha + \gamma)$$
$$W_2 = -\sin(\alpha)\sin(\gamma) + \cos(\alpha)\cos(\gamma) = \cos(\alpha + \gamma)$$

We immediately see that α and γ are going to lead to the same transformations - the mathematical expression of the observation on the spitfire above, that rotation around the x axis is equivalent to rotation about the z axis.

It's easy to do the same set of reductions for the case where $\sin(\beta) = 1$; see http://www.gregslabaugh.name/publications/euler.pdf.

3.3 The example in code

Here is what our gimbal lock looks like in code:

```
>>> import numpy as np
>>> np.set_printoptions(precision=3, suppress=True)  # neat printing
>>> from transforms3d.euler import euler2mat, mat2euler
>>> x_angle = -0.2
>>> y_angle = -np.pi / 2
>>> z_angle = -0.2
>>> R = euler2mat(x_angle, y_angle, z_angle, 'sxyz')
>>> R
array([[ 0.   ,  0.389, -0.921],
       [-0.   ,  0.921,  0.389],
       [ 1.   , -0.   ,  0.   ]])
```

This isn't the transformation we actually want because of the gimbal lock. The gimbal lock means that `x_angle` and `z_angle` result in rotations about the same axis of the object. So, we can add something to the `x_angle` and subtract the same value from `z_angle` to get the same result:

```
>>> R = euler2mat(x_angle + 0.1, y_angle, z_angle - 0.1, 'sxyz')
>>> R
array([[ 0.   ,  0.389, -0.921],
       [-0.   ,  0.921,  0.389],
       [ 1.   , -0.   ,  0.   ]])
```

In fact, we could omit the z rotation entirely and put all the rotation into the original x axis rotation and still get the same rotation matrix:

```
>>> R_dash = euler2mat(x_angle + z_angle, y_angle, 0, 'sxyz')
>>> np.allclose(R, R_dash)
True
```

So, there is no future in doing our transformations starting with this x and y rotation, if we are rotating with this axis order. We can get the transformation we actually want by doing the rotations in the order x, then z then y, like this:

```
>>> R = euler2mat(x_angle, z_angle, y_angle, 'sxzy')
>>> R
array([[ 0.   ,  0.199, -0.98 ],
       [-0.199,  0.961,  0.195],
       [ 0.98 ,  0.195,  0.039]])
```

We can get this same transformation using our original x, y, z rotation order, but using different rotation angles:

```
>>> x_dash, y_dash, z_dash = mat2euler(R, 'sxyz')
>>> np.array((x_dash, y_dash, z_dash))  # np.array for print neatness
array([ 1.371, -1.371, -1.571])
>>> R = euler2mat(x_dash, y_dash, z_dash, 'sxyz')
>>> R
array([[ 0.   ,  0.199, -0.98 ],
       [-0.199,  0.961,  0.195],
       [ 0.98 ,  0.195,  0.039]])
```

CHAPTER

FOUR

API REFERENCE

4.1 transforms3d

transforms3d package

Based on, largely using transformations.py by Christoph Gohlke

Additional code monkey work by Matthew Brett

4.2 _gohlketransforms

Homogeneous Transformation Matrices and Quaternions.

A library for calculating 4x4 matrices for translating, rotating, reflecting, scaling, shearing, projecting, orthogonalizing, and superimposing arrays of 3D homogeneous coordinates as well as for converting between rotation matrices, Euler angles, and quaternions. Also includes an Arcball control object and functions to decompose transformation matrices.

Author
Christoph Gohlke

Organization
Laboratory for Fluorescence Dynamics, University of California, Irvine

Version
2017.02.17

4.2.1 Requirements

- CPython 2.7 or 3.5
- Numpy 1.11
- Transformations.c 2017.02.17 (recommended for speedup of some functions)

4.2.2 Notes

The API is not stable yet and is expected to change between revisions.

This Python code is not optimized for speed. Refer to the transformations.c module for a faster implementation of some functions.

Documentation in HTML format can be generated with epydoc.

Matrices (M) can be inverted using numpy.linalg.inv(M), be concatenated using numpy.dot(M0, M1), or transform homogeneous coordinate arrays (v) using numpy.dot(M, v) for shape (4, *) column vectors, respectively numpy.dot(v, M.T) for shape (*, 4) row vectors ("array of points").

This module follows the "column vectors on the right" and "row major storage" (C contiguous) conventions. The translation components are in the right column of the transformation matrix, i.e. M[:3, 3]. The transpose of the transformation matrices may have to be used to interface with other graphics systems, e.g. with OpenGL's glMultMatrixd(). See also [16].

Calculations are carried out with numpy.float64 precision.

Vector, point, quaternion, and matrix function arguments are expected to be "array like", i.e. tuple, list, or numpy arrays.

Return types are numpy arrays unless specified otherwise.

Angles are in radians unless specified otherwise.

Quaternions w+ix+jy+kz are represented as [w, x, y, z].

A triple of Euler angles can be applied/interpreted in 24 ways, which can be specified using a 4 character string or encoded 4-tuple:

Axes 4-string: e.g. 'sxyz' or 'ryxy'

- first character : rotations are applied to 's'tatic or 'r'otating frame
- remaining characters : successive rotation axis 'x', 'y', or 'z'

Axes 4-tuple: e.g. (0, 0, 0, 0) or (1, 1, 1, 1)

- inner axis: code of axis ('x':0, 'y':1, 'z':2) of rightmost matrix.
- parity : even (0) if inner axis 'x' is followed by 'y', 'y' is followed by 'z', or 'z' is followed by 'x'. Otherwise odd (1).
- repetition : first and last axis are same (1) or different (0).
- frame : rotations are applied to static (0) or rotating (1) frame.

Other Python packages and modules for 3D transformations and quaternions:

- **Transforms3d**
  includes most code of this module.
- Blender.mathutils
- numpy-dtypes

4.2.3 References

(1) Matrices and transformations. Ronald Goldman. In "Graphics Gems I", pp 472-475. Morgan Kaufmann, 1990.

(2) More matrices and transformations: shear and pseudo-perspective. Ronald Goldman. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(3) Decomposing a matrix into simple transformations. Spencer Thomas. In "Graphics Gems II", pp 320-323. Morgan Kaufmann, 1991.

(4) Recovering the data from the transformation matrix. Ronald Goldman. In "Graphics Gems II", pp 324-331. Morgan Kaufmann, 1991.

(5) Euler angle conversion. Ken Shoemake. In "Graphics Gems IV", pp 222-229. Morgan Kaufmann, 1994.

(6) Arcball rotation control. Ken Shoemake. In "Graphics Gems IV", pp 175-192. Morgan Kaufmann, 1994.

(7) Representing attitude: Euler angles, unit quaternions, and rotation vectors. James Diebel. 2006.

(8) A discussion of the solution for the best rotation to relate two sets of vectors. W Kabsch. Acta Cryst. 1978. A34, 827-828.

(9) Closed-form solution of absolute orientation using unit quaternions. BKP Horn. J Opt Soc Am A. 1987. 4(4):629-642.

(10) Quaternions. Ken Shoemake. http://www.sfu.ca/~jwa3/cmpt461/files/quatut.pdf

(11) From quaternion to matrix and back. JMP van Waveren. 2005. http://www.intel.com/cd/ids/developer/asmo-na/eng/293748.htm

(12) Uniform random rotations. Ken Shoemake. In "Graphics Gems III", pp 124-132. Morgan Kaufmann, 1992.

(13) Quaternion in molecular modeling. CFF Karney. J Mol Graph Mod, 25(5):595-604

(14) New method for extracting the quaternion from a rotation matrix. Itzhack Y Bar-Itzhack, J Guid Contr Dynam. 2000. 23(6): 1085-1087.

(15) Multiple View Geometry in Computer Vision. Hartley and Zissermann. Cambridge University Press; 2nd Ed. 2004. Chapter 4, Algorithm 4.7, p 130.

(16) Column Vectors vs. Row Vectors. http://steve.hollasch.net/cgindex/math/matrix/column-vec.html

4.2.4 Examples

```
>>> alpha, beta, gamma = 0.123, -1.234, 2.345
>>> origin, xaxis, yaxis, zaxis = [0, 0, 0], [1, 0, 0], [0, 1, 0], [0, 0, 1]
>>> I = identity_matrix()
>>> Rx = rotation_matrix(alpha, xaxis)
>>> Ry = rotation_matrix(beta, yaxis)
>>> Rz = rotation_matrix(gamma, zaxis)
>>> R = concatenate_matrices(Rx, Ry, Rz)
>>> euler = euler_from_matrix(R, 'rxyz')
>>> numpy.allclose([alpha, beta, gamma], euler)
True
>>> Re = euler_matrix(alpha, beta, gamma, 'rxyz')
>>> is_same_transform(R, Re)
True
>>> al, be, ga = euler_from_matrix(Re, 'rxyz')
>>> is_same_transform(Re, euler_matrix(al, be, ga, 'rxyz'))
True
>>> qx = quaternion_about_axis(alpha, xaxis)
>>> qy = quaternion_about_axis(beta, yaxis)
>>> qz = quaternion_about_axis(gamma, zaxis)
>>> q = quaternion_multiply(qx, qy)
>>> q = quaternion_multiply(q, qz)
>>> Rq = quaternion_matrix(q)
>>> is_same_transform(R, Rq)
True
>>> S = scale_matrix(1.23, origin)
>>> T = translation_matrix([1, 2, 3])
>>> Z = shear_matrix(beta, xaxis, origin, zaxis)
>>> R = random_rotation_matrix(numpy.random.rand(3))
>>> M = concatenate_matrices(T, R, Z, S)
>>> scale, shear, angles, trans, persp = decompose_matrix(M)
>>> numpy.allclose(scale, 1.23)
True
```

(continues on next page)

(continued from previous page)

```
>>> numpy.allclose(trans, [1, 2, 3])
True
>>> numpy.allclose(shear, [0, math.tan(beta), 0])
True
>>> is_same_transform(R, euler_matrix(axes='sxyz', *angles))
True
>>> M1 = compose_matrix(scale, shear, angles, trans, persp)
>>> is_same_transform(M, M1)
True
>>> v0, v1 = random_vector(3), random_vector(3)
>>> M = rotation_matrix(angle_between_vectors(v0, v1), vector_product(v0, v1))
>>> v2 = numpy.dot(v0, M[:3,:3].T)
>>> numpy.allclose(unit_vector(v1), unit_vector(v2))
True
```

Arcball([initial])	Virtual Trackball Control.
affine_matrix_from_points(v0, v1[, shear, ...])	Return affine transform matrix to register two point sets.
angle_between_vectors(v0, v1[, directed, axis])	Return angle between vectors.
arcball_constrain_to_axis(point, axis)	Return sphere point perpendicular to axis.
arcball_map_to_sphere(point, center, radius)	Return unit sphere coordinates from window coordinates.
arcball_nearest_axis(point, axes)	Return axis, which arc is nearest to point.
clip_matrix(left, right, bottom, top, near, far)	Return matrix to obtain normalized device coordinates from frustum.
compose_matrix([scale, shear, angles, ...])	Return transformation matrix from sequence of transformations.
concatenate_matrices(*matrices)	Return concatenation of series of transformation matrices.
decompose_matrix(matrix)	Return sequence of transformations from transformation matrix.
euler_from_matrix(matrix[, axes])	Return Euler angles from rotation matrix for specified axis sequence.
euler_from_quaternion(quaternion[, axes])	Return Euler angles from quaternion for specified axis sequence.
euler_matrix(ai, aj, ak[, axes])	Return homogeneous rotation matrix from Euler angles and axis sequence.
identity_matrix()	Return 4x4 identity/unit matrix.
inverse_matrix(matrix)	Return inverse of square transformation matrix.
is_same_quaternion(q0, q1)	Return True if two quaternions are equal.
is_same_transform(matrix0, matrix1)	Return True if two matrices perform same transformation.
orthogonalization_matrix(lengths, angles)	Return orthogonalization matrix for crystallographic cell coordinates.
projection_from_matrix(matrix[, pseudo])	Return projection plane and perspective point from projection matrix.
projection_matrix(point, normal[, ...])	Return matrix to project onto plane defined by point and normal.
quaternion_about_axis(angle, axis)	Return quaternion for rotation about axis.
quaternion_conjugate(quaternion)	Return conjugate of quaternion.
quaternion_from_euler(ai, aj, ak[, axes])	Return quaternion from Euler angles and axis sequence.
quaternion_from_matrix(matrix[, isprecise])	Return quaternion from rotation matrix.

continues on next page

Table 2 – continued from previous page

quaternion_imag(quaternion)	Return imaginary part of quaternion.
quaternion_inverse(quaternion)	Return inverse of quaternion.
quaternion_matrix(quaternion)	Return homogeneous rotation matrix from quaternion.
quaternion_multiply(quaternion1, quaternion0)	Return multiplication of two quaternions.
quaternion_real(quaternion)	Return real part of quaternion.
quaternion_slerp(quat0, quat1, fraction[, ...])	Return spherical linear interpolation between two quaternions.
random_quaternion([rand])	Return uniform random unit quaternion.
random_rotation_matrix([rand])	Return uniform random rotation matrix.
random_vector(size)	Return array of random doubles in the half-open interval [0.0, 1.0).
reflection_from_matrix(matrix)	Return mirror plane point and normal vector from reflection matrix.
reflection_matrix(point, normal)	Return matrix to mirror at plane defined by point and normal vector.
rotation_from_matrix(matrix)	Return rotation angle and axis from rotation matrix.
rotation_matrix(angle, direction[, point])	Return matrix to rotate about axis defined by point and direction.
scale_from_matrix(matrix)	Return scaling factor, origin and direction from scaling matrix.
scale_matrix(factor[, origin, direction])	Return matrix to scale by factor around origin in direction.
shear_from_matrix(matrix)	Return shear angle, direction and plane from shear matrix.
shear_matrix(angle, direction, point, normal)	Return matrix to shear by angle along direction vector on shear plane.
superimposition_matrix(v0, v1[, scale, usesvd])	Return matrix to transform given 3D point set into second point set.
translation_from_matrix(matrix)	Return translation vector from translation matrix.
translation_matrix(direction)	Return matrix to translate by direction vector.
unit_vector(data[, axis, out])	Return ndarray normalized by length, i.e. Euclidean norm, along axis.
vector_norm(data[, axis, out])	Return length, i.e. Euclidean norm, of ndarray along axis.
vector_product(v0, v1[, axis])	Return vector perpendicular to vectors.

4.2.5 Arcball

class transforms3d._gohlketransforms.**Arcball**(*initial=None*)

Bases: object

Virtual Trackball Control.

```
>>> ball = Arcball()
>>> ball = Arcball(initial=numpy.identity(4))
>>> ball.place([320, 320], 320)
>>> ball.down([500, 250])
>>> ball.drag([475, 275])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 3.90583455)
True
>>> ball = Arcball(initial=[1, 0, 0, 0])
```

(continues on next page)

(continued from previous page)

```
>>> ball.place([320, 320], 320)
>>> ball.setaxes([1, 1, 0], [-1, 1, 0])
>>> ball.constrain = True
>>> ball.down([400, 200])
>>> ball.drag([200, 400])
>>> R = ball.matrix()
>>> numpy.allclose(numpy.sum(R), 0.2055924)
True
>>> ball.next()
```

__init__(*initial=None*)

Initialize virtual trackball control.

initial : quaternion or rotation matrix

property constrain

Return state of constrain to axis mode.

down(*point*)

Set initial cursor window coordinates and pick constrain-axis.

drag(*point*)

Update current cursor window coordinates.

matrix()

Return homogeneous rotation matrix.

next(*acceleration=0.0*)

Continue rotation in direction of last drag.

place(*center*, *radius*)

Place Arcball, e.g. when window size changes.

center
[sequence[2]] Window coordinates of trackball center.

radius
[float] Radius of trackball in window coordinates.

setaxes(**axes*)

Set axes to constrain rotations.

4.2.6 affine_matrix_from_points

transforms3d._gohlketransforms.**affine_matrix_from_points**(*v0*, *v1*, *shear=True*, *scale=True*, *usesvd=True*)

Return affine transform matrix to register two point sets.

v0 and v1 are shape (ndims, *) arrays of at least ndims non-homogeneous coordinates, where ndims is the dimensionality of the coordinate space.

If shear is False, a similarity transformation matrix is returned. If also scale is False, a rigid/Euclidean transformation matrix is returned.

By default the algorithm by Hartley and Zissermann [15] is used. If usesvd is True, similarity and Euclidean transformation matrices are calculated by minimizing the weighted sum of squared deviations (RMSD) according

to the algorithm by Kabsch [8]. Otherwise, and if ndims is 3, the quaternion based algorithm by Horn [9] is used, which is slower when using this Python implementation.

The returned matrix performs rotation, translation and uniform scaling (if specified).

```
>>> v0 = [[0, 1031, 1031, 0], [0, 0, 1600, 1600]]
>>> v1 = [[675, 826, 826, 677], [55, 52, 281, 277]]
>>> affine_matrix_from_points(v0, v1)
array([[   0.14549,    0.00062,  675.50008],
       [   0.00048,    0.14094,   53.24971],
       [   0.     ,    0.     ,    1.     ]])
>>> T = translation_matrix(numpy.random.random(3)-0.5)
>>> R = random_rotation_matrix(numpy.random.random(3))
>>> S = scale_matrix(random.random())
>>> M = concatenate_matrices(T, R, S)
>>> v0 = (numpy.random.rand(4, 100) - 0.5) * 20
>>> v0[3] = 1
>>> v1 = numpy.dot(M, v0)
>>> v0[:3] += numpy.random.normal(0, 1e-8, 300).reshape(3, -1)
>>> M = affine_matrix_from_points(v0[:3], v1[:3])
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
```

More examples in superimposition_matrix()

4.2.7 angle_between_vectors

transforms3d._gohlketransforms.**angle_between_vectors**(*v0*, *v1*, *directed=True*, *axis=0*)

Return angle between vectors.

If directed is False, the input vectors are interpreted as undirected axes, i.e. the maximum angle is pi/2.

```
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3])
>>> numpy.allclose(a, math.pi)
True
>>> a = angle_between_vectors([1, -2, 3], [-1, 2, -3], directed=False)
>>> numpy.allclose(a, 0)
True
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
>>> v1 = [[3], [0], [0]]
>>> a = angle_between_vectors(v0, v1)
>>> numpy.allclose(a, [0, 1.5708, 1.5708, 0.95532])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
>>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
>>> a = angle_between_vectors(v0, v1, axis=1)
>>> numpy.allclose(a, [1.5708, 1.5708, 1.5708, 0.95532])
True
```

4.2.8 arcball_constrain_to_axis

transforms3d._gohlketransforms.**arcball_constrain_to_axis**(*point*, *axis*)

Return sphere point perpendicular to axis.

4.2.9 arcball_map_to_sphere

transforms3d._gohlketransforms.**arcball_map_to_sphere**(*point*, *center*, *radius*)

Return unit sphere coordinates from window coordinates.

4.2.10 arcball_nearest_axis

transforms3d._gohlketransforms.**arcball_nearest_axis**(*point*, *axes*)

Return axis, which arc is nearest to point.

4.2.11 clip_matrix

transforms3d._gohlketransforms.**clip_matrix**(*left*, *right*, *bottom*, *top*, *near*, *far*, *perspective=False*)

Return matrix to obtain normalized device coordinates from frustum.

The frustum bounds are axis-aligned along x (left, right), y (bottom, top) and z (near, far).

Normalized device coordinates are in range [-1, 1] if coordinates are inside the frustum.

If perspective is True the frustum is a truncated pyramid with the perspective point at origin and direction along z axis, otherwise an orthographic canonical view volume (a box).

Homogeneous coordinates transformed by the perspective clip matrix need to be dehomogenized (divided by w coordinate).

```
>>> frustum = numpy.random.rand(6)
>>> frustum[1] += frustum[0]
>>> frustum[3] += frustum[2]
>>> frustum[5] += frustum[4]
>>> M = clip_matrix(perspective=False, *frustum)
>>> numpy.dot(M, [frustum[0], frustum[2], frustum[4], 1])
array([-1., -1., -1.,  1.])
>>> numpy.dot(M, [frustum[1], frustum[3], frustum[5], 1])
array([ 1.,  1.,  1.,  1.])
>>> M = clip_matrix(perspective=True, *frustum)
>>> v = numpy.dot(M, [frustum[0], frustum[2], frustum[4], 1])
>>> v / v[3]
array([-1., -1., -1.,  1.])
>>> v = numpy.dot(M, [frustum[1], frustum[3], frustum[4], 1])
>>> v / v[3]
array([ 1.,  1., -1.,  1.])
```

4.2.12 compose_matrix

transforms3d._gohlketransforms.**compose_matrix**(*scale=None*, *shear=None*, *angles=None*, *translate=None*, *perspective=None*)

Return transformation matrix from sequence of transformations.

This is the inverse of the decompose_matrix function.

Sequence of transformations:

scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

```
>>> scale = numpy.random.random(3) - 0.5
>>> shear = numpy.random.random(3) - 0.5
>>> angles = (numpy.random.random(3) - 0.5) * (2*math.pi)
>>> trans = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(4) - 0.5
>>> M0 = compose_matrix(scale, shear, angles, trans, persp)
>>> result = decompose_matrix(M0)
>>> M1 = compose_matrix(*result)
>>> is_same_transform(M0, M1)
True
```

4.2.13 concatenate_matrices

transforms3d._gohlketransforms.**concatenate_matrices**(**matrices*)

Return concatenation of series of transformation matrices.

```
>>> M = numpy.random.rand(16).reshape((4, 4)) - 0.5
>>> numpy.allclose(M, concatenate_matrices(M))
True
>>> numpy.allclose(numpy.dot(M, M.T), concatenate_matrices(M, M.T))
True
```

4.2.14 decompose_matrix

transforms3d._gohlketransforms.**decompose_matrix**(*matrix*)

Return sequence of transformations from transformation matrix.

matrix
: [array_like] Non-degenerative homogeneous transformation matrix

Return tuple of:
: scale : vector of 3 scaling factors shear : list of shear factors for x-y, x-z, y-z axes angles : list of Euler angles about static x, y, z axes translate : translation vector along x, y, z axes perspective : perspective partition of matrix

Raise ValueError if matrix is of wrong type or degenerative.

```
>>> T0 = translation_matrix([1, 2, 3])
>>> scale, shear, angles, trans, persp = decompose_matrix(T0)
>>> T1 = translation_matrix(trans)
>>> numpy.allclose(T0, T1)
True
>>> S = scale_matrix(0.123)
>>> scale, shear, angles, trans, persp = decompose_matrix(S)
>>> scale[0]
0.123
>>> R0 = euler_matrix(1, 2, 3)
>>> scale, shear, angles, trans, persp = decompose_matrix(R0)
>>> R1 = euler_matrix(*angles)
>>> numpy.allclose(R0, R1)
True
```

4.2.15 euler_from_matrix

transforms3d._gohlketransforms.**euler_from_matrix**(*matrix*, *axes='sxyz'*)

Return Euler angles from rotation matrix for specified axis sequence.

axes : One of 24 axis sequences as string or encoded tuple

Note that many Euler angle triplets can describe one matrix.

```
>>> R0 = euler_matrix(1, 2, 3, 'syxz')
>>> al, be, ga = euler_from_matrix(R0, 'syxz')
>>> R1 = euler_matrix(al, be, ga, 'syxz')
>>> numpy.allclose(R0, R1)
True
>>> angles = (4*math.pi) * (numpy.random.random(3) - 0.5)
>>> for axes in _AXES2TUPLE.keys():
...    R0 = euler_matrix(axes=axes, *angles)
...    R1 = euler_matrix(axes=axes, *euler_from_matrix(R0, axes))
...    if not numpy.allclose(R0, R1): print(axes, "failed")
```

4.2.16 euler_from_quaternion

transforms3d._gohlketransforms.**euler_from_quaternion**(*quaternion*, *axes='sxyz'*)

Return Euler angles from quaternion for specified axis sequence.

```
>>> angles = euler_from_quaternion([0.99810947, 0.06146124, 0, 0])
>>> numpy.allclose(angles, [0.123, 0, 0])
True
```

4.2.17 euler_matrix

transforms3d._gohlketransforms.**euler_matrix**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return homogeneous rotation matrix from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
>>> R = euler_matrix(1, 2, 3, 'syxz')
>>> numpy.allclose(numpy.sum(R[0]), -1.34786452)
True
>>> R = euler_matrix(1, 2, 3, (0, 1, 0, 1))
>>> numpy.allclose(numpy.sum(R[0]), -0.383436184)
True
>>> ai, aj, ak = (4*math.pi) * (numpy.random.random(3) - 0.5)
>>> for axes in _AXES2TUPLE.keys():
...    R = euler_matrix(ai, aj, ak, axes)
>>> for axes in _TUPLE2AXES.keys():
...    R = euler_matrix(ai, aj, ak, axes)
```

4.2.18 identity_matrix

transforms3d._gohlketransforms.**identity_matrix**()

Return 4x4 identity/unit matrix.

```
>>> I = identity_matrix()
>>> numpy.allclose(I, numpy.dot(I, I))
True
>>> numpy.sum(I), numpy.trace(I)
(4.0, 4.0)
>>> numpy.allclose(I, numpy.identity(4))
True
```

4.2.19 inverse_matrix

transforms3d._gohlketransforms.**inverse_matrix**(*matrix*)

Return inverse of square transformation matrix.

```
>>> M0 = random_rotation_matrix()
>>> M1 = inverse_matrix(M0.T)
>>> numpy.allclose(M1, numpy.linalg.inv(M0.T))
True
>>> for size in range(1, 7):
...     M0 = numpy.random.rand(size, size)
...     M1 = inverse_matrix(M0)
...     if not numpy.allclose(M1, numpy.linalg.inv(M0)): print(size)
```

4.2.20 is_same_quaternion

transforms3d._gohlketransforms.**is_same_quaternion**(*q0*, *q1*)

Return True if two quaternions are equal.

4.2.21 is_same_transform

transforms3d._gohlketransforms.**is_same_transform**(*matrix0*, *matrix1*)

Return True if two matrices perform same transformation.

```
>>> is_same_transform(numpy.identity(4), numpy.identity(4))
True
>>> is_same_transform(numpy.identity(4), random_rotation_matrix())
False
```

4.2.22 orthogonalization_matrix

transforms3d._gohlketransforms.**orthogonalization_matrix**(*lengths*, *angles*)

Return orthogonalization matrix for crystallographic cell coordinates.

Angles are expected in degrees.

The de-orthogonalization matrix is the inverse.

```
>>> O = orthogonalization_matrix([10, 10, 10], [90, 90, 90])
>>> numpy.allclose(O[:3, :3], numpy.identity(3, float) * 10)
True
>>> O = orthogonalization_matrix([9.8, 12.0, 15.5], [87.2, 80.7, 69.7])
>>> numpy.allclose(numpy.sum(O), 43.063229)
True
```

4.2.23 projection_from_matrix

transforms3d._gohlketransforms.**projection_from_matrix**(*matrix*, *pseudo=False*)

Return projection plane and perspective point from projection matrix.

Return values are same as arguments for projection_matrix function: point, normal, direction, perspective, and pseudo.

```
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, direct)
>>> result = projection_from_matrix(P0)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=False)
>>> result = projection_from_matrix(P0, pseudo=False)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
>>> P0 = projection_matrix(point, normal, perspective=persp, pseudo=True)
>>> result = projection_from_matrix(P0, pseudo=True)
>>> P1 = projection_matrix(*result)
>>> is_same_transform(P0, P1)
True
```

4.2.24 projection_matrix

transforms3d._gohlketransforms.**projection_matrix**(*point*, *normal*, *direction=None*, *perspective=None*, *pseudo=False*)

Return matrix to project onto plane defined by point and normal.

Using either perspective point, projection direction, or none of both.

If pseudo is True, perspective projections will preserve relative depth such that Perspective = dot(Orthogonal, PseudoPerspective).

```
>>> P = projection_matrix([0, 0, 0], [1, 0, 0])
>>> numpy.allclose(P[1:, 1:], numpy.identity(4)[1:, 1:])
True
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> persp = numpy.random.random(3) - 0.5
>>> P0 = projection_matrix(point, normal)
>>> P1 = projection_matrix(point, normal, direction=direct)
>>> P2 = projection_matrix(point, normal, perspective=persp)
```

(continues on next page)

(continued from previous page)

```
>>> P3 = projection_matrix(point, normal, perspective=persp, pseudo=True)
>>> is_same_transform(P2, numpy.dot(P0, P3))
True
>>> P = projection_matrix([3, 0, 0], [1, 1, 0], [1, 0, 0])
>>> v0 = (numpy.random.rand(4, 5) - 0.5) * 20
>>> v0[3] = 1
>>> v1 = numpy.dot(P, v0)
>>> numpy.allclose(v1[1], v0[1])
True
>>> numpy.allclose(v1[0], 3-v1[1])
True
```

4.2.25 quaternion_about_axis

transforms3d._gohlketransforms.**quaternion_about_axis**(*angle*, *axis*)

Return quaternion for rotation about axis.

```
>>> q = quaternion_about_axis(0.123, [1, 0, 0])
>>> numpy.allclose(q, [0.99810947, 0.06146124, 0, 0])
True
```

4.2.26 quaternion_conjugate

transforms3d._gohlketransforms.**quaternion_conjugate**(*quaternion*)

Return conjugate of quaternion.

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_conjugate(q0)
>>> q1[0] == q0[0] and all(q1[1:] == -q0[1:])
True
```

4.2.27 quaternion_from_euler

transforms3d._gohlketransforms.**quaternion_from_euler**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return quaternion from Euler angles and axis sequence.

ai, aj, ak : Euler's roll, pitch and yaw angles axes : One of 24 axis sequences as string or encoded tuple

```
>>> q = quaternion_from_euler(1, 2, 3, 'ryxz')
>>> numpy.allclose(q, [0.435953, 0.310622, -0.718287, 0.444435])
True
```

4.2.28 quaternion_from_matrix

transforms3d._gohlketransforms.**quaternion_from_matrix**(*matrix*, *isprecise=False*)

Return quaternion from rotation matrix.

If isprecise is True, the input matrix is assumed to be a precise rotation matrix and a faster algorithm is used.

```
>>> q = quaternion_from_matrix(numpy.identity(4), True)
>>> numpy.allclose(q, [1, 0, 0, 0])
True
>>> q = quaternion_from_matrix(numpy.diag([1, -1, -1, 1]))
>>> numpy.allclose(q, [0, 1, 0, 0]) or numpy.allclose(q, [0, -1, 0, 0])
True
>>> R = rotation_matrix(0.123, (1, 2, 3))
>>> q = quaternion_from_matrix(R, True)
>>> numpy.allclose(q, [0.9981095, 0.0164262, 0.0328524, 0.0492786])
True
>>> R = [[-0.545, 0.797, 0.260, 0], [0.733, 0.603, -0.313, 0],
...      [-0.407, 0.021, -0.913, 0], [0, 0, 0, 1]]
>>> q = quaternion_from_matrix(R)
>>> numpy.allclose(q, [0.19069, 0.43736, 0.87485, -0.083611])
True
>>> R = [[0.395, 0.362, 0.843, 0], [-0.626, 0.796, -0.056, 0],
...      [-0.677, -0.498, 0.529, 0], [0, 0, 0, 1]]
>>> q = quaternion_from_matrix(R)
>>> numpy.allclose(q, [0.82336615, -0.13610694, 0.46344705, -0.29792603])
True
>>> R = random_rotation_matrix()
>>> q = quaternion_from_matrix(R)
>>> is_same_transform(R, quaternion_matrix(q))
True
>>> is_same_quaternion(quaternion_from_matrix(R, isprecise=False),
...                    quaternion_from_matrix(R, isprecise=True))
True
>>> R = euler_matrix(0.0, 0.0, numpy.pi/2.0)
>>> is_same_quaternion(quaternion_from_matrix(R, isprecise=False),
...                    quaternion_from_matrix(R, isprecise=True))
True
```

4.2.29 quaternion_imag

transforms3d._gohlketransforms.**quaternion_imag**(*quaternion*)

Return imaginary part of quaternion.

```
>>> quaternion_imag([3, 0, 1, 2])
array([ 0.,  1.,  2.])
```

4.2.30 quaternion_inverse

transforms3d._gohlketransforms.**quaternion_inverse**(*quaternion*)

Return inverse of quaternion.

```
>>> q0 = random_quaternion()
>>> q1 = quaternion_inverse(q0)
>>> numpy.allclose(quaternion_multiply(q0, q1), [1, 0, 0, 0])
True
```

4.2.31 quaternion_matrix

transforms3d._gohlketransforms.**quaternion_matrix**(*quaternion*)

Return homogeneous rotation matrix from quaternion.

```
>>> M = quaternion_matrix([0.99810947, 0.06146124, 0, 0])
>>> numpy.allclose(M, rotation_matrix(0.123, [1, 0, 0]))
True
>>> M = quaternion_matrix([1, 0, 0, 0])
>>> numpy.allclose(M, numpy.identity(4))
True
>>> M = quaternion_matrix([0, 1, 0, 0])
>>> numpy.allclose(M, numpy.diag([1, -1, -1, 1]))
True
```

4.2.32 quaternion_multiply

transforms3d._gohlketransforms.**quaternion_multiply**(*quaternion1*, *quaternion0*)

Return multiplication of two quaternions.

```
>>> q = quaternion_multiply([4, 1, -2, 3], [8, -5, 6, 7])
>>> numpy.allclose(q, [28, -44, -14, 48])
True
```

4.2.33 quaternion_real

transforms3d._gohlketransforms.**quaternion_real**(*quaternion*)

Return real part of quaternion.

```
>>> quaternion_real([3, 0, 1, 2])
3.0
```

4.2.34 quaternion_slerp

transforms3d._gohlketransforms.**quaternion_slerp**(*quat0*, *quat1*, *fraction*, *spin=0*, *shortestpath=True*)

Return spherical linear interpolation between two quaternions.

```
>>> q0 = random_quaternion()
>>> q1 = random_quaternion()
>>> q = quaternion_slerp(q0, q1, 0)
>>> numpy.allclose(q, q0)
True
>>> q = quaternion_slerp(q0, q1, 1, 1)
>>> numpy.allclose(q, q1)
True
>>> q = quaternion_slerp(q0, q1, 0.5)
>>> angle = math.acos(numpy.dot(q0, q))
>>> numpy.allclose(2, math.acos(numpy.dot(q0, q1)) / angle) or         numpy.
↪allclose(2, math.acos(-numpy.dot(q0, q1)) / angle)
True
```

4.2.35 random_quaternion

transforms3d._gohlketransforms.**random_quaternion**(*rand=None*)

Return uniform random unit quaternion.

rand: array like or None
Three independent random variables that are uniformly distributed between 0 and 1.

```
>>> q = random_quaternion()
>>> numpy.allclose(1, vector_norm(q))
True
>>> q = random_quaternion(numpy.random.random(3))
>>> len(q.shape), q.shape[0]==4
(1, True)
```

4.2.36 random_rotation_matrix

transforms3d._gohlketransforms.**random_rotation_matrix**(*rand=None*)

Return uniform random rotation matrix.

rand: array like
Three independent random variables that are uniformly distributed between 0 and 1 for each returned quaternion.

```
>>> R = random_rotation_matrix()
>>> numpy.allclose(numpy.dot(R.T, R), numpy.identity(4))
True
```

4.2.37 random_vector

transforms3d._gohlketransforms.**random_vector**(*size*)

Return array of random doubles in the half-open interval [0.0, 1.0).

```
>>> v = random_vector(10000)
>>> numpy.all(v >= 0) and numpy.all(v < 1)
True
>>> v0 = random_vector(10)
>>> v1 = random_vector(10)
>>> numpy.any(v0 == v1)
False
```

4.2.38 reflection_from_matrix

transforms3d._gohlketransforms.**reflection_from_matrix**(*matrix*)

Return mirror plane point and normal vector from reflection matrix.

```
>>> v0 = numpy.random.random(3) - 0.5
>>> v1 = numpy.random.random(3) - 0.5
>>> M0 = reflection_matrix(v0, v1)
>>> point, normal = reflection_from_matrix(M0)
>>> M1 = reflection_matrix(point, normal)
>>> is_same_transform(M0, M1)
True
```

4.2.39 reflection_matrix

transforms3d._gohlketransforms.**reflection_matrix**(*point*, *normal*)

Return matrix to mirror at plane defined by point and normal vector.

```
>>> v0 = numpy.random.random(4) - 0.5
>>> v0[3] = 1.
>>> v1 = numpy.random.random(3) - 0.5
>>> R = reflection_matrix(v0, v1)
>>> numpy.allclose(2, numpy.trace(R))
True
>>> numpy.allclose(v0, numpy.dot(R, v0))
True
>>> v2 = v0.copy()
>>> v2[:3] += v1
>>> v3 = v0.copy()
>>> v2[:3] -= v1
>>> numpy.allclose(v2, numpy.dot(R, v3))
True
```

4.2.40 rotation_from_matrix

transforms3d._gohlketransforms.**rotation_from_matrix**(*matrix*)

Return rotation angle and axis from rotation matrix.

```
>>> angle = (random.random() - 0.5) * (2*math.pi)
>>> direc = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> R0 = rotation_matrix(angle, direc, point)
>>> angle, direc, point = rotation_from_matrix(R0)
>>> R1 = rotation_matrix(angle, direc, point)
>>> is_same_transform(R0, R1)
True
```

4.2.41 rotation_matrix

transforms3d._gohlketransforms.**rotation_matrix**(*angle*, *direction*, *point=None*)

Return matrix to rotate about axis defined by point and direction.

```
>>> R = rotation_matrix(math.pi/2, [0, 0, 1], [1, 0, 0])
>>> numpy.allclose(numpy.dot(R, [0, 0, 0, 1]), [1, -1, 0, 1])
True
>>> angle = (random.random() - 0.5) * (2*math.pi)
>>> direc = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> R0 = rotation_matrix(angle, direc, point)
>>> R1 = rotation_matrix(angle-2*math.pi, direc, point)
>>> is_same_transform(R0, R1)
True
>>> R0 = rotation_matrix(angle, direc, point)
>>> R1 = rotation_matrix(-angle, -direc, point)
>>> is_same_transform(R0, R1)
```

(continues on next page)

(continued from previous page)

```
True
>>> I = numpy.identity(4, numpy.float64)
>>> numpy.allclose(I, rotation_matrix(math.pi*2, direc))
True
>>> numpy.allclose(2, numpy.trace(rotation_matrix(math.pi/2,
...                                               direc, point)))
True
```

4.2.42 scale_from_matrix

transforms3d._gohlketransforms.**scale_from_matrix**(*matrix*)

Return scaling factor, origin and direction from scaling matrix.

```
>>> factor = random.random() * 10 - 5
>>> origin = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> S0 = scale_matrix(factor, origin)
>>> factor, origin, direction = scale_from_matrix(S0)
>>> S1 = scale_matrix(factor, origin, direction)
>>> is_same_transform(S0, S1)
True
>>> S0 = scale_matrix(factor, origin, direct)
>>> factor, origin, direction = scale_from_matrix(S0)
>>> S1 = scale_matrix(factor, origin, direction)
>>> is_same_transform(S0, S1)
True
```

4.2.43 scale_matrix

transforms3d._gohlketransforms.**scale_matrix**(*factor*, *origin=None*, *direction=None*)

Return matrix to scale by factor around origin in direction.

Use factor -1 for point symmetry.

```
>>> v = (numpy.random.rand(4, 5) - 0.5) * 20
>>> v[3] = 1
>>> S = scale_matrix(-1.234)
>>> numpy.allclose(numpy.dot(S, v)[:3], -1.234*v[:3])
True
>>> factor = random.random() * 10 - 5
>>> origin = numpy.random.random(3) - 0.5
>>> direct = numpy.random.random(3) - 0.5
>>> S = scale_matrix(factor, origin)
>>> S = scale_matrix(factor, origin, direct)
```

4.2.44 shear_from_matrix

transforms3d._gohlketransforms.**shear_from_matrix**(*matrix*)

Return shear angle, direction and plane from shear matrix.

```
>>> angle = (random.random() - 0.5) * 4*math.pi
>>> direct = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.cross(direct, numpy.random.random(3))
>>> S0 = shear_matrix(angle, direct, point, normal)
>>> angle, direct, point, normal = shear_from_matrix(S0)
>>> S1 = shear_matrix(angle, direct, point, normal)
>>> is_same_transform(S0, S1)
True
```

4.2.45 shear_matrix

transforms3d._gohlketransforms.**shear_matrix**(*angle*, *direction*, *point*, *normal*)

Return matrix to shear by angle along direction vector on shear plane.

The shear plane is defined by a point and normal vector. The direction vector must be orthogonal to the plane's normal vector.

A point P is transformed by the shear matrix into P" such that the vector P-P" is parallel to the direction vector and its extent is given by the angle of P-P'-P", where P' is the orthogonal projection of P onto the shear plane.

```
>>> angle = (random.random() - 0.5) * 4*math.pi
>>> direct = numpy.random.random(3) - 0.5
>>> point = numpy.random.random(3) - 0.5
>>> normal = numpy.cross(direct, numpy.random.random(3))
>>> S = shear_matrix(angle, direct, point, normal)
>>> numpy.allclose(1, numpy.linalg.det(S))
True
```

4.2.46 superimposition_matrix

transforms3d._gohlketransforms.**superimposition_matrix**(*v0*, *v1*, *scale=False*, *usesvd=True*)

Return matrix to transform given 3D point set into second point set.

v0 and v1 are shape (3, *) or (4, *) arrays of at least 3 points.

The parameters scale and usesvd are explained in the more general affine_matrix_from_points function.

The returned matrix is a similarity or Euclidean transformation matrix. This function has a fast C implementation in transformations.c.

```
>>> v0 = numpy.random.rand(3, 10)
>>> M = superimposition_matrix(v0, v0)
>>> numpy.allclose(M, numpy.identity(4))
True
>>> R = random_rotation_matrix(numpy.random.random(3))
>>> v0 = [[1,0,0], [0,1,0], [0,0,1], [1,1,1]]
>>> v1 = numpy.dot(R, v0)
>>> M = superimposition_matrix(v0, v1)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> v0 = (numpy.random.rand(4, 100) - 0.5) * 20
>>> v0[3] = 1
>>> v1 = numpy.dot(R, v0)
```

(continues on next page)

(continued from previous page)

```
>>> M = superimposition_matrix(v0, v1)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> S = scale_matrix(random.random())
>>> T = translation_matrix(numpy.random.random(3)-0.5)
>>> M = concatenate_matrices(T, R, S)
>>> v1 = numpy.dot(M, v0)
>>> v0[:3] += numpy.random.normal(0, 1e-9, 300).reshape(3, -1)
>>> M = superimposition_matrix(v0, v1, scale=True)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> M = superimposition_matrix(v0, v1, scale=True, usesvd=False)
>>> numpy.allclose(v1, numpy.dot(M, v0))
True
>>> v = numpy.empty((4, 100, 3))
>>> v[:, :, 0] = v0
>>> M = superimposition_matrix(v0, v1, scale=True, usesvd=False)
>>> numpy.allclose(v1, numpy.dot(M, v[:, :, 0]))
True
```

4.2.47 translation_from_matrix

transforms3d._gohlketransforms.**translation_from_matrix**(*matrix*)

Return translation vector from translation matrix.

```
>>> v0 = numpy.random.random(3) - 0.5
>>> v1 = translation_from_matrix(translation_matrix(v0))
>>> numpy.allclose(v0, v1)
True
```

4.2.48 translation_matrix

transforms3d._gohlketransforms.**translation_matrix**(*direction*)

Return matrix to translate by direction vector.

```
>>> v = numpy.random.random(3) - 0.5
>>> numpy.allclose(v, translation_matrix(v)[:3, 3])
True
```

4.2.49 unit_vector

transforms3d._gohlketransforms.**unit_vector**(*data*, *axis=None*, *out=None*)

Return ndarray normalized by length, i.e. Euclidean norm, along axis.

```
>>> v0 = numpy.random.random(3)
>>> v1 = unit_vector(v0)
>>> numpy.allclose(v1, v0 / numpy.linalg.norm(v0))
True
>>> v0 = numpy.random.rand(5, 4, 3)
>>> v1 = unit_vector(v0, axis=-1)
```

(continues on next page)

(continued from previous page)

```
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=2)), 2)
>>> numpy.allclose(v1, v2)
True
>>> v1 = unit_vector(v0, axis=1)
>>> v2 = v0 / numpy.expand_dims(numpy.sqrt(numpy.sum(v0*v0, axis=1)), 1)
>>> numpy.allclose(v1, v2)
True
>>> v1 = numpy.empty((5, 4, 3))
>>> unit_vector(v0, axis=1, out=v1)
>>> numpy.allclose(v1, v2)
True
>>> list(unit_vector([]))
[]
>>> list(unit_vector([1]))
[1.0]
```

4.2.50 vector_norm

transforms3d._gohlketransforms.**vector_norm**(*data*, *axis=None*, *out=None*)

Return length, i.e. Euclidean norm, of ndarray along axis.

```
>>> v = numpy.random.random(3)
>>> n = vector_norm(v)
>>> numpy.allclose(n, numpy.linalg.norm(v))
True
>>> v = numpy.random.rand(6, 5, 3)
>>> n = vector_norm(v, axis=-1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=2)))
True
>>> n = vector_norm(v, axis=1)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> v = numpy.random.rand(5, 4, 3)
>>> n = numpy.empty((5, 3))
>>> vector_norm(v, axis=1, out=n)
>>> numpy.allclose(n, numpy.sqrt(numpy.sum(v*v, axis=1)))
True
>>> vector_norm([])
0.0
>>> vector_norm([1])
1.0
```

4.2.51 vector_product

transforms3d._gohlketransforms.**vector_product**(*v0*, *v1*, *axis=0*)

Return vector perpendicular to vectors.

```
>>> v = vector_product([2, 0, 0], [0, 3, 0])
>>> numpy.allclose(v, [0, 0, 6])
True
>>> v0 = [[2, 0, 0, 2], [0, 2, 0, 2], [0, 0, 2, 2]]
```

(continues on next page)

(continued from previous page)

```
>>> v1 = [[3], [0], [0]]
>>> v = vector_product(v0, v1)
>>> numpy.allclose(v, [[0, 0, 0, 0], [0, 0, 6, 6], [0, -6, 0, -6]])
True
>>> v0 = [[2, 0, 0], [2, 0, 0], [0, 2, 0], [2, 0, 0]]
>>> v1 = [[0, 3, 0], [0, 0, 3], [0, 0, 3], [3, 3, 3]]
>>> v = vector_product(v0, v1, axis=1)
>>> numpy.allclose(v, [[0, 0, 6], [0, -6, 0], [6, 0, 0], [0, -6, 6]])
True
```

4.3 _version

get_versions()	

4.3.1 get_versions

transforms3d._version.**get_versions**()

4.4 affines

Compose and decompose homogenous affine - usually 4x4 - matrices

compose(T, R, Z[, S])	Compose translations, rotations, zooms, [shears] to affine
decompose(A)	Decompose homogenous affine transformation matrix *A* into parts.
decompose44(A44)	Decompose 4x4 homogenous affine matrix into parts.

4.4.1 compose

transforms3d.affines.**compose**(*T*, *R*, *Z*, *S=None*)

Compose translations, rotations, zooms, [shears] to affine

Parameters

T
[array-like shape (N,)] Translations, where N is usually 3 (3D case)

R
[array-like shape (N,N)] Rotation matrix where N is usually 3 (3D case)

Z
[array-like shape (N,)] Zooms, where N is usually 3 (3D case)

S
[array-like, shape (P,), optional] Shear vector, such that shears fill upper triangle above diagonal to form shear matrix. P is the (N-2)th Triangular number, which happens to be 3 for a 4x4 affine (3D case)

Returns

A

[array, shape (N+1, N+1)] Affine transformation matrix where N usually == 3 (3D case)

Examples

```
>>> T = [20, 30, 40] # translations
>>> R = [[0, -1, 0], [1, 0, 0], [0, 0, 1]] # rotation matrix
>>> Z = [2.0, 3.0, 4.0] # zooms
>>> A = compose(T, R, Z)
>>> A
array([[ 0., -3.,  0., 20.],
       [ 2.,  0.,  0., 30.],
       [ 0.,  0.,  4., 40.],
       [ 0.,  0.,  0.,  1.]])
>>> S = np.zeros(3)
>>> B = compose(T, R, Z, S)
>>> assert(np.all(A == B))  # True
```

A null set

```
>>> compose(np.zeros(3), np.eye(3), np.ones(3), np.zeros(3))
array([[1., 0., 0., 0.],
       [0., 1., 0., 0.],
       [0., 0., 1., 0.],
       [0., 0., 0., 1.]])
```

4.4.2 decompose

`transforms3d.affines.`**`decompose`**`(A)`

Decompose homogenous affine transformation matrix *A* into parts.

The parts are translations, rotations, zooms, shears.

A can be any square matrix, but is typically shape (4,4).

Decomposes A into `T, R, Z, S`, such that, if A is shape (4,4):

```
Smat = np.array([[1, S[0], S[1]],
                 [0,    1, S[2]],
                 [0,    0,    1]])
RZS = np.dot(R, np.dot(np.diag(Z), Smat))
A = np.eye(4)
A[:3,:3] = RZS
A[:-1,-1] = T
```

The order of transformations is therefore shears, followed by zooms, followed by rotations, followed by translations.

The case above (A.shape == (4,4)) is the most common, and corresponds to a 3D affine, but in fact A need only be square.

Parameters

A

[array shape (N,N)]

Returns

T
: [array, shape (N-1,)] Translation vector

R
: [array shape (N-1, N-1)] rotation matrix

Z
: [array, shape (N-1,)] Zoom vector. May have one negative zoom to prevent need for negative determinant R matrix above

S
: [array, shape (P,)] Shear vector, such that shears fill upper triangle above diagonal to form shear matrix. P is the (N-2)th Triangular number, which happens to be 3 for a 4x4 affine.

Notes

We have used a nice trick from SPM to get the shears. Let us call the starting N-1 by N-1 matrix `RZS`, because it is the composition of the rotations on the zooms on the shears. The rotation matrix `R` must have the property `np.dot(R.T, R) == np.eye(N-1)`. Thus `np.dot(RZS.T, RZS)` will, by the transpose rules, be equal to `np.dot((ZS).T, (ZS))`. Because we are doing shears with the upper right part of the matrix, that means that the Cholesky decomposition of `np.dot(RZS.T, RZS)` will give us our `ZS` matrix, from which we take the zooms from the diagonal, and the shear values from the off-diagonal elements.

Examples

```
>>> T = [20, 30, 40] # translations
>>> R = [[0, -1, 0], [1, 0, 0], [0, 0, 1]] # rotation matrix
>>> Z = [2.0, 3.0, 4.0] # zooms
>>> S = [0.2, 0.1, 0.3] # shears
>>> # Now we make an affine matrix
>>> A = np.eye(4)
>>> Smat = np.array([[1, S[0], S[1]],
...                  [0,    1, S[2]],
...                  [0,    0,    1]])
>>> RZS = np.dot(R, np.dot(np.diag(Z), Smat))
>>> A[:3,:3] = RZS
>>> A[:-1,-1] = T # set translations
>>> Tdash, Rdash, Zdash, Sdash = decompose(A)
>>> np.allclose(T, Tdash)
True
>>> np.allclose(R, Rdash)
True
>>> np.allclose(Z, Zdash)
True
>>> np.allclose(S, Sdash)
True
```

4.4.3 decompose44

`transforms3d.affines.`**`decompose44`**(*A44*)

Decompose 4x4 homogenous affine matrix into parts.

The parts are translations, rotations, zooms, shears.

This is the same as *`decompose()`* but specialized for 4x4 affines.

Decomposes *A44* into `T, R, Z, S`, such that:

```
Smat = np.array([[1, S[0], S[1]],
                 [0,    1, S[2]],
                 [0,    0,    1]])
RZS = np.dot(R, np.dot(np.diag(Z), Smat))
A44 = np.eye(4)
A44[:3,:3] = RZS
A44[:-1,-1] = T
```

The order of transformations is therefore shears, followed by zooms, followed by rotations, followed by translations.

This routine only works for shape (4,4) matrices

Parameters

A44
[array shape (4,4)]

Returns

T
[array, shape (3,)] Translation vector

R
[array shape (3,3)] rotation matrix

Z
[array, shape (3,)] Zoom vector. May have one negative zoom to prevent need for negative determinant R matrix above

S
[array, shape (3,)] Shear vector, such that shears fill upper triangle above diagonal to form shear matrix (type `striu`).

Notes

The implementation inspired by:

Decomposing a matrix into simple transformations by Spencer W. Thomas, pp 320-323 in *Graphics Gems II*, James Arvo (editor), Academic Press, 1991, ISBN: 0120644819.

The upper left 3x3 of the affine consists of a matrix we'll call RZS:

```
RZS = R * Z *S
```

where R is a rotation matrix, Z is a diagonal matrix of scalings:

```
Z = diag([sx, sy, sz])
```

and S is a shear matrix of form:

```
S = [[1, sxy, sxz],
     [0,   1, syz],
     [0,   0,   1]])
```

Running all this through sympy (see 'derivations' folder) gives `RZS` as

```
[R00*sx, R01*sy + R00*sx*sxy, R02*sz + R00*sx*sxz + R01*sy*syz]
[R10*sx, R11*sy + R10*sx*sxy, R12*sz + R10*sx*sxz + R11*sy*syz]
[R20*sx, R21*sy + R20*sx*sxy, R22*sz + R20*sx*sxz + R21*sy*syz]
```

R is defined as being a rotation matrix, so the dot products between the columns of R are zero, and the norm of each column is 1. Thus the dot product:

```
R[:,0].T * RZS[:,1]
```

that results in:

```
[R00*R01*sy + R10*R11*sy + R20*R21*sy + sx*sxy*R00**2 + sx*sxy*R10**2 +␣
↪sx*sxy*R20**2]
```

simplifies to `sy*0 + sx*sxy*1 == sx*sxy`. Therefore:

```
R[:,1] * sy = RZS[:,1] - R[:,0] * (R[:,0].T * RZS[:,1])
```

allowing us to get `sy` with the norm, and sxy with `R[:,0].T * RZS[:,1] / sx`.

Similarly `R[:,0].T * RZS[:,2]` simplifies to `sx*sxz`, and `R[:,1].T * RZS[:,2]` to `sy*syz` giving us the remaining unknowns.

Examples

```
>>> T = [20, 30, 40] # translations
>>> R = [[0, -1, 0], [1, 0, 0], [0, 0, 1]] # rotation matrix
>>> Z = [2.0, 3.0, 4.0] # zooms
>>> S = [0.2, 0.1, 0.3] # shears
>>> # Now we make an affine matrix
>>> A = np.eye(4)
>>> Smat = np.array([[1, S[0], S[1]],
...                  [0,    1, S[2]],
...                  [0,    0,    1]])
>>> RZS = np.dot(R, np.dot(np.diag(Z), Smat))
>>> A[:3,:3] = RZS
>>> A[:-1,-1] = T # set translations
>>> Tdash, Rdash, Zdash, Sdash = decompose44(A)
>>> np.allclose(T, Tdash)
True
>>> np.allclose(R, Rdash)
True
>>> np.allclose(Z, Zdash)
True
>>> np.allclose(S, Sdash)
True
```

4.5 axangles

Functions for working with axis, angle rotations

See `quaternions` for conversions between axis, angle pairs and quaternions.

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *aff* : affine array shape (4, 4) (3D homogenous coordinates)
- *axangle* : rotations encoded by axis vector and angle scalar

aff2axangle(aff)	Return axis, angle and point from affine
axangle2aff(axis, angle[, point])	Return affine encoding rotation by *angle* about *axis*.
axangle2mat(axis, angle[, is_normalized])	Rotation matrix for rotation angle *angle* around *axis*
mat2axangle(mat[, unit_thresh])	Return axis, angle and point from (3, 3) matrix *mat*

4.5.1 aff2axangle

transforms3d.axangles.**aff2axangle**(*aff*)

Return axis, angle and point from affine

Parameters

aff
[array-like shape (4,4)]

Returns

axis
[array shape (3,)] vector giving axis of rotation

angle
[scalar] angle of rotation in radians.

point
[array shape (3,)] point around which rotation is performed

Notes

http://en.wikipedia.org/wiki/Rotation_matrix#Axis_of_a_rotation

Examples

```
>>> direc = np.random.random(3) - 0.5
>>> angle = (np.random.random() - 0.5) * (2*math.pi)
>>> point = np.random.random(3) - 0.5
>>> R0 = axangle2aff(direc, angle, point)
>>> direc, angle, point = aff2axangle(R0)
>>> R1 = axangle2aff(direc, angle, point)
>>> np.allclose(R0, R1)
True
```

4.5.2 axangle2aff

transforms3d.axangles.**axangle2aff**(*axis*, *angle*, *point=None*)

Return affine encoding rotation by *angle* about *axis*.

Parameters

axis
[array shape (3,)] vector giving axis of rotation

angle
[scalar] angle of rotation, in radians.

Returns

A
[array shape (4, 4)] Affine array to be multiplied on left of coordinate column vector to apply given rotation.

Notes

Applying a rotation around a point is the same as applying a translation of `-point` to move `point` to the origin, rotating, then applying a translation of `point`. If R is the rotation matrix, than the affine for the rotation about point P is:

```
[R00, R01, R02, P0 - P0*R00 - P1*R01 - P2*R02]
[R10, R11, R12, P1 - P0*R10 - P1*R11 - P2*R12]
[R20, R21, R22, P2 - P0*R20 - P1*R21 - P2*R22]
[  0,   0,   0,                             1]
```

(see derivations)

Examples

```
>>> angle = (np.random.random() - 0.5) * (2*math.pi)
>>> direc = np.random.random(3) - 0.5
>>> point = np.random.random(3) - 0.5
>>> R0 = axangle2aff(direc, angle, point)
>>> R1 = axangle2aff(direc, angle-2*math.pi, point)
>>> np.allclose(R0, R1)
True
>>> R0 = axangle2aff(direc, angle, point)
>>> R1 = axangle2aff(-direc, -angle, point)
>>> np.allclose(R0, R1)
True
>>> I = np.identity(4, np.float64)
>>> np.allclose(I, axangle2aff(direc, math.pi*2))
True
>>> np.allclose(2., np.trace(axangle2aff(direc,
...                                       math.pi/2,
...                                       point)))
True
```

4.5.3 axangle2mat

`transforms3d.axangles.`**`axangle2mat`**(*axis*, *angle*, *is_normalized=False*)

Rotation matrix for rotation angle *angle* around *axis*

Parameters

axis
[3 element sequence] vector specifying axis for rotation.

angle
[scalar] angle of rotation in radians.

is_normalized
[bool, optional] True if *axis* is already normalized (has norm of 1). Default False.

Returns

mat
[array shape (3,3)] rotation matrix for specified rotation

Notes

From: http://en.wikipedia.org/wiki/Rotation_matrix#Axis_and_angle

4.5.4 mat2axangle

`transforms3d.axangles.`**`mat2axangle`**(*mat*, *unit_thresh=1e-05*)

Return axis, angle and point from (3, 3) matrix *mat*

Parameters

mat
[array-like shape (3, 3)] Rotation matrix

unit_thresh
[float, optional] Tolerable difference from 1 when testing for unit eigenvalues to confirm *mat* is a rotation matrix.

Returns

axis
[array shape (3,)] vector giving axis of rotation

angle
[scalar] angle of rotation in radians.

Notes

http://en.wikipedia.org/wiki/Rotation_matrix#Axis_of_a_rotation

Examples

```
>>> direc = np.random.random(3) - 0.5
>>> angle = (np.random.random() - 0.5) * (2*math.pi)
>>> R0 = axangle2mat(direc, angle)
>>> direc, angle = mat2axangle(R0)
>>> R1 = axangle2mat(direc, angle)
>>> np.allclose(R0, R1)
True
```

4.6 derivations

4.6.1 Module: derivations.angle_axes

Derivations for rotations of angle around axis

angle_axis2mat(theta, vector)	Rotation matrix of angle *theta* around *vector*
angle_axis2quat(theta, vector)	Quaternion for rotation of angle *theta* around *vector* Notes ----- Formula from http://mathworld.wolfram.com/EulerParameters.html
orig_aa2mat(angle, direction)	
quat2angle_axis(quat)	Convert quaternion to rotation of angle around axis

4.6.2 Module: derivations.decompositions

Derivations for extracting rotations, zooms, shears

4.6.3 Module: derivations.eulerangles

These give the derivations for Euler angles to rotation matrix and Euler angles to quaternion. We use the rotation matrix derivation only in the tests. The quaternion derivation is in the tests, and, in more compact form, in the `euler2quat` code.

The rotation matrices operate on column vectors, thus, if `R` is the 3x3 rotation matrix, `v` is the 3 x N set of N vectors to be rotated, and `vdash` is the matrix of rotated vectors:

```
vdash = np.dot(R, v)
```

x_rotation(theta)	Rotation angle *theta* around x-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three
y_rotation(theta)	Rotation angle *theta* around y-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three
z_rotation(theta)	Rotation angle *theta* around z-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three

4.6.4 Module: derivations.quaternions

Symbolic formulae for quaternions

qmult(q1, q2)	Multiply two quaternions
quat2mat(quat)	Symbolic conversion from quaternion to rotation matrix
quat_around_axis(theta, axis)	Quaternion for rotation of angle *theta* around axis *axis*

4.6.5 Module: derivations.utils

Utilities for derivations

make_matrix(name_prefix, N, M)	
matrices_equal(M1, M2)	

continues on next page

Table 11 – continued from previous page

matrix_simplify(M)

angle_axis2mat

transforms3d.derivations.angle_axes.**angle_axis2mat**(*theta*, *vector*)

Rotation matrix of angle *theta* around *vector*

Parameters

theta
[scalar] angle of rotation

vector
[3 element sequence] vector specifying axis for rotation.

is_normalized
[bool, optional] True if vector is already normalized (has norm of 1). Default False

Returns

mat
[array shape (3,3)] rotation matrix specified rotation

Notes

From: http://en.wikipedia.org/wiki/Rotation_matrix#Axis_and_angle

angle_axis2quat

transforms3d.derivations.angle_axes.**angle_axis2quat**(*theta*, *vector*)

Quaternion for rotation of angle *theta* around *vector* Notes —– Formula from http://mathworld.wolfram.com/EulerParameters.html

orig_aa2mat

transforms3d.derivations.angle_axes.**orig_aa2mat**(*angle*, *direction*)

quat2angle_axis

transforms3d.derivations.angle_axes.**quat2angle_axis**(*quat*)

Convert quaternion to rotation of angle around axis

x_rotation

transforms3d.derivations.eulerangles.**x_rotation**(*theta*)

Rotation angle *theta* around x-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three

y_rotation

transforms3d.derivations.eulerangles.**y_rotation**(*theta*)

Rotation angle *theta* around y-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three

z_rotation

transforms3d.derivations.eulerangles.**z_rotation**(*theta*)

Rotation angle *theta* around z-axis http://en.wikipedia.org/wiki/Rotation_matrix#Dimension_three

qmult

transforms3d.derivations.quaternions.**qmult**(*q1*, *q2*)

Multiply two quaternions

Parameters

q1
[4 element sequence]

q2
[4 element sequence]

Returns

q12
[shape (4,) array]

Notes

See : http://en.wikipedia.org/wiki/Quaternions#Hamilton_product

quat2mat

transforms3d.derivations.quaternions.**quat2mat**(*quat*)

Symbolic conversion from quaternion to rotation matrix

For a unit quaternion

From: http://en.wikipedia.org/wiki/Rotation_matrix#Quaternion

quat_around_axis

transforms3d.derivations.quaternions.**quat_around_axis**(*theta*, *axis*)

Quaternion for rotation of angle *theta* around axis *axis*

Parameters

theta
[symbol] angle of rotation

axis
[3 element sequence] vector (assumed normalized) specifying axis for rotation

Returns

quat
[4 element sequence of symbols] quaternion giving specified rotation

Notes

Formula from http://mathworld.wolfram.com/EulerParameters.html

make_matrix

transforms3d.derivations.utils.**make_matrix**(*name_prefix*, *N*, *M*)

matrices_equal

transforms3d.derivations.utils.**matrices_equal**(*M1*, *M2*)

matrix_simplify

transforms3d.derivations.utils.**matrix_simplify**(*M*)

4.7 euler

Generic Euler rotations

See:

- http://en.wikipedia.org/wiki/Rotation_matrix
- http://en.wikipedia.org/wiki/Euler_angles
- http://mathworld.wolfram.com/EulerAngles.html

See also: *Representing Attitude with Euler Angles and Quaternions: A Reference* (2006) by James Diebel. A cached PDF link last found here:

http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.110.5134

4.7.1 Defining rotations

Euler's rotation theorem tells us that any rotation in 3D can be described by 3 angles. Let's call the 3 angles the *Euler angle vector* and call the angles in the vector $alpha$, $beta$ and $gamma$. The vector is [$alpha$, $beta$. $gamma$] and, in this description, the order of the parameters specifies the order in which the rotations occur (so the rotation corresponding to $alpha$ is applied first).

In order to specify the meaning of an *Euler angle vector* we need to specify the axes around which each of the rotations corresponding to $alpha$, $beta$ and $gamma$ will occur.

There are therefore three axes for the rotations $alpha$, $beta$ and $gamma$; let's call them i j, k.

Let us express the rotation $alpha$ around axis *i* as a 3 by 3 rotation matrix *A*. Similarly $beta$ around *j* becomes 3 x 3 matrix *B* and $gamma$ around *k* becomes matrix *G*. Then the whole rotation expressed by the Euler angle vector [$alpha$, $beta$. $gamma$], *R* is given by:

```
R = np.dot(G, np.dot(B, A))
```

See http://mathworld.wolfram.com/EulerAngles.html

The order GBA expresses the fact that the rotations are performed in the order of the vector ($alpha$ around axis *i* = *A* first).

To convert a given Euler angle vector to a meaningful rotation, and a rotation matrix, we need to define:

- the axes *i*, *j*, *k*;
- whether the rotations move the axes as they are applied (intrinsic rotations) - compared the situation where the axes stay fixed and the vectors move within the axis frame (extrinsic);
- whether a rotation matrix should be applied on the left of a vector to be transformed (vectors are column vectors) or on the right (vectors are row vectors);

- the handedness of the coordinate system.

See: http://en.wikipedia.org/wiki/Rotation_matrix#Ambiguities

This module implements intrinsic and extrinsic axes, with standard conventions for axes *i*, *j*, *k*. We assume that the matrix should be applied on the left of the vector, and right-handed coordinate systems. To get the matrix to apply on the right of the vector, you need the transpose of the matrix we supply here, by the matrix transpose rule: $(M.V)^T = V^T M^T$.

4.7.2 Rotation axes

Rotations given as a set of three angles can refer to any of 24 different ways of applying these rotations, or equivalently, 24 conventions for rotation angles. See http://en.wikipedia.org/wiki/Euler_angles.

The different conventions break down into two groups of 12. In the first group, the rotation axes are fixed (also, global, static), and do not move with rotations. These are called *extrinsic* axes. The axes can also move with the rotations. These are called *intrinsic*, local or rotating axes.

Each of the two groups (*intrinsic* and *extrinsic*) can further be divided into so-called Euler rotations (rotation about one axis, then a second and then the first again), and Tait-Bryan angles (rotations about all three axes). The two groups (Euler rotations and Tait-Bryan rotations) each have 6 possible choices. There are therefore 2 * 2 * 6 = 24 possible conventions that could apply to rotations about a sequence of three given angles.

This module gives an implementation of conversion between angles and rotation matrices for which you can specify any of the 24 different conventions.

4.7.3 Specifying angle conventions

You specify conventions for interpreting the sequence of angles with a four character string.

The first character is 'r' (rotating == intrinsic), or 's' (static == extrinsic).

The next three characters give the axis ('x', 'y' or 'z') about which to perform the rotation, in the order in which the rotations will be performed.

For example the string 'szyx' specifies that the angles should be interpreted relative to extrinsic (static) coordinate axes, and be performed in the order: rotation about z axis; rotation about y axis; rotation about x axis. This is a relatively common convention, with customized implementations in `taitbryan` in this package.

The string 'rzxz' specifies that the angles should be interpreted relative to intrinsic (rotating) coordinate axes, and be performed in the order: rotation about z axis; rotation about the rotated x axis; rotation about the rotated z axis. Wolfram Mathworld claim this is the most common convention : http://mathworld.wolfram.com/EulerAngles.html.

4.7.4 Direction of rotation

The direction of rotation is given by the right-hand rule (orient the thumb of the right hand along the axis around which the rotation occurs, with the end of the thumb at the positive end of the axis; curl your fingers; the direction your fingers curl is the direction of rotation). Therefore, the rotations are counterclockwise if looking along the axis of rotation from positive to negative.

4.7.5 Terms used in function names

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *euler* : (sequence of) rotation angles about the z, y, x axes (in that order)
- *axangle* : rotations encoded by axis vector and angle scalar
- *quat* : quaternion shape (4,)

EulerFuncs(axes)	Namespace for Euler angles functions with given axes specification
TBZYX()	Namespace for Tait-Bryan ZYX Euler angle convention functions
axangle2euler(vector, theta[, axes])	Convert axis, angle pair to Euler angles
euler2axangle(ai, aj, ak[, axes])	Return angle, axis corresponding to Euler angles, axis specification
euler2mat(ai, aj, ak[, axes])	Return rotation matrix from Euler angles and axis sequence.
euler2quat(ai, aj, ak[, axes])	Return *quaternion* from Euler angles and axis sequence *axes*
mat2euler(mat[, axes])	Return Euler angles from rotation matrix for specified axis sequence.
quat2euler(quaternion[, axes])	Euler angles from *quaternion* for specified axis sequence *axes*

4.7.6 EulerFuncs

class transforms3d.euler.**EulerFuncs**(*axes*)

Bases: object

Namespace for Euler angles functions with given axes specification

__init__(*axes*)

Initialize namespace for Euler angles functions

Parameters

axes
[str] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. sxyz (the default).

axangle2euler(*vector*, *theta*)

Convert axis, angle pair to Euler angles

See *axangle2euler()* for details.

euler2axangle(*ai*, *aj*, *ak*)

Angle, axis corresponding to Euler angles

See *euler2axangle()* for details.

euler2mat(*ai*, *aj*, *ak*)

Return rotation matrix from Euler angles

See *euler2mat()* for details.

euler2quat(*ai*, *aj*, *ak*)

Return *quaternion* from Euler angles

See *euler2quat()* for details.

mat2euler(*mat*)

Return Euler angles from rotation matrix *mat*

See *mat2euler()* for details.

quat2euler(*quat*)

Euler angles from *quaternion*

See *quat2euler()* for details.

4.7.7 TBZYX

class transforms3d.euler.**TBZYX**

Bases: *EulerFuncs*

Namespace for Tait-Bryan ZYX Euler angle convention functions

__init__()

Initialize Tait-Bryan ZYX namespace

euler2mat(*ai*, *aj*, *ak*)

Return rotation matrix from Euler angles

See *transforms3d.taitbryan.euler2mat()* for details.

euler2quat(*ai*, *aj*, *ak*)

Return *quaternion* from Euler angles

See *transforms3d.taitbryan.euler2quat()* for details.

mat2euler(*mat*)

Return Euler angles from rotation matrix *mat*

See *transforms3d.taitbryan.mat2euler()* for details.

4.7.8 axangle2euler

transforms3d.euler.**axangle2euler**(*vector*, *theta*, *axes='sxyz'*)

Convert axis, angle pair to Euler angles

Parameters

vector
[3 element sequence] vector specifying axis for rotation.

theta
[scalar] angle of rotation

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

Examples

```
>>> ai, aj, ak = axangle2euler([1, 0, 0], 0)
>>> np.allclose((ai, aj, ak), 0)
True
```

4.7.9 euler2axangle

transforms3d.euler.**euler2axangle**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return angle, axis corresponding to Euler angles, axis specification

Parameters

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

vector
[array shape (3,)] axis around which rotation occurs

theta
[scalar] angle of rotation

Examples

```
>>> vec, theta = euler2axangle(0, 1.5, 0, 'szyx')
>>> np.allclose(vec, [0, 1, 0])
True
>>> theta
1.5
```

4.7.10 euler2mat

transforms3d.euler.**euler2mat**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return rotation matrix from Euler angles and axis sequence.

Parameters

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

mat
[array (3, 3)] Rotation matrix or affine.

Examples

```
>>> R = euler2mat(1, 2, 3, 'syxz')
>>> np.allclose(np.sum(R[0]), -1.34786452)
True
>>> R = euler2mat(1, 2, 3, (0, 1, 0, 1))
>>> np.allclose(np.sum(R[0]), -0.383436184)
True
```

4.7.11 euler2quat

`transforms3d.euler.`**`euler2quat`**(*ai*, *aj*, *ak*, *axes='sxyz'*)

Return *quaternion* from Euler angles and axis sequence *axes*

Parameters

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

quat
[array shape (4,)] Quaternion in w, x, y z (real, then vector) format

Examples

```
>>> q = euler2quat(1, 2, 3, 'ryxz')
>>> np.allclose(q, [0.435953, 0.310622, -0.718287, 0.444435])
True
```

4.7.12 mat2euler

`transforms3d.euler.`**`mat2euler`**(*mat*, *axes='sxyz'*)

Return Euler angles from rotation matrix for specified axis sequence.

Note that many Euler angle triplets can describe one matrix.

Parameters

mat
[array-like shape (3, 3) or (4, 4)] Rotation matrix or affine.

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

Examples

```
>>> R0 = euler2mat(1, 2, 3, 'syxz')
>>> al, be, ga = mat2euler(R0, 'syxz')
>>> R1 = euler2mat(al, be, ga, 'syxz')
>>> np.allclose(R0, R1)
True
```

4.7.13 quat2euler

`transforms3d.euler.`**`quat2euler`**(*quaternion*, *axes='sxyz'*)

Euler angles from *quaternion* for specified axis sequence *axes*

Parameters

q
[4 element sequence] w, x, y, z of quaternion

axes
[str, optional] Axis specification; one of 24 axis sequences as string or encoded tuple - e.g. `sxyz` (the default).

Returns

ai
[float] First rotation angle (according to *axes*).

aj
[float] Second rotation angle (according to *axes*).

ak
[float] Third rotation angle (according to *axes*).

Examples

```
>>> angles = quat2euler([0.99810947, 0.06146124, 0, 0])
>>> np.allclose(angles, [0.123, 0, 0])
True
```

4.8 quaternions

Functions to operate on, or return, quaternions.

Quaternions here consist of 4 values `w, x, y, z`, where `w` is the real (scalar) part, and `x, y, z` are the complex (vector) part.

Note - rotation matrices here apply to column vectors, that is, they are applied on the left of the vector. For example:

```
>>> import numpy as np
>>> q = [0, 1, 0, 0] # 180 degree rotation around axis 0
>>> M = quat2mat(q) # from this module
>>> vec = np.array([1, 2, 3]).reshape((3,1)) # column vector
>>> tvec = np.dot(M, vec)
```

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *aff* : affine array shape (4, 4) (3D homogenous coordinates)
- *quat* : quaternion shape (4,)
- *axangle* : rotations encoded by axis vector and angle scalar

axangle2quat(vector, theta[, is_normalized])	Quaternion for rotation of angle *theta* around *vector*
fillpositive(xyz[, w2_thresh])	Compute unit quaternion from last 3 values
mat2quat(M)	Calculate quaternion corresponding to given rotation matrix
nearly_equivalent(q1, q2[, rtol, atol])	Returns True if *q1* and *q2* give near equivalent transforms
qconjugate(q)	Conjugate of quaternion
qexp(q)	Return exponential of quaternion
qeye([dtype])	Return identity quaternion
qinverse(q)	Return multiplicative inverse of quaternion *q*
qisunit(q)	Return True is this is very nearly a unit quaternion
qlog(q)	Return natural logarithm of quaternion
qmult(q1, q2)	Multiply two quaternions
qnorm(q)	Return norm of quaternion
qpow(q, n)	Return the *n* th power of quaternion *q*
quat2axangle(quat[, identity_thresh])	Convert quaternion to rotation of angle around axis
quat2mat(q)	Calculate rotation matrix corresponding to quaternion
rotate_vector(v, q[, is_normalized])	Apply transformation in quaternion *q* to vector *v*

4.8.1 axangle2quat

`transforms3d.quaternions.`**`axangle2quat`**(*vector*, *theta*, *is_normalized=False*)

Quaternion for rotation of angle *theta* around *vector*

Parameters

vector
[3 element sequence] vector specifying axis for rotation.

theta
[scalar] angle of rotation in radians.

is_normalized
[bool, optional] True if vector is already normalized (has norm of 1). Default False.

Returns

quat
[4 element sequence of symbols] quaternion giving specified rotation

Notes

Formula from http://mathworld.wolfram.com/EulerParameters.html

Examples

```
>>> q = axangle2quat([1, 0, 0], np.pi)
>>> np.allclose(q, [0, 1, 0,  0])
True
```

4.8.2 fillpositive

`transforms3d.quaternions.fillpositive`(*xyz*, *w2_thresh=None*)

Compute unit quaternion from last 3 values

Parameters

xyz
[iterable] iterable containing 3 values, corresponding to quaternion x, y, z

w2_thresh
[None or float, optional] threshold to determine if w squared is really negative. If None (default) then w2_thresh set equal to `-np.finfo(xyz.dtype).eps`, if possible, otherwise `-np.finfo(np.float64).eps`

Returns

wxyz
[array shape (4,)] Full 4 values of quaternion

Notes

If w, x, y, z are the values in the full quaternion, assumes w is positive.

Gives error if w*w is estimated to be negative

w = 0 corresponds to a 180 degree rotation

The unit quaternion specifies that np.dot(wxyz, wxyz) == 1.

If w is positive (assumed here), w is given by:

w = np.sqrt(1.0-(x*x+y*y+z*z))

w2 = 1.0-(x*x+y*y+z*z) can be near zero, which will lead to numerical instability in sqrt. Here we use the system maximum float type to reduce numerical instability

Examples

```
>>> import numpy as np
>>> wxyz = fillpositive([0,0,0])
>>> assert np.all(wxyz == [1, 0, 0, 0])
>>> wxyz = fillpositive([1,0,0]) # Corner case; w is 0
>>> assert np.all(wxyz == [0, 1, 0, 0])
>>> assert np.dot(wxyz, wxyz) == 1
```

4.8.3 mat2quat

transforms3d.quaternions.**mat2quat**(*M*)

Calculate quaternion corresponding to given rotation matrix

Method claimed to be robust to numerical errors in *M*.

Constructs quaternion by calculating maximum eigenvector for matrix K (constructed from input *M*). Although this is not tested, a maximum eigenvalue of 1 corresponds to a valid rotation.

A quaternion q*-1 corresponds to the same rotation as q; thus the sign of the reconstructed quaternion is arbitrary, and we return quaternions with positive w (q[0]).

See notes.

Parameters

M
[array-like] 3x3 rotation matrix

Returns

q
[(4,) array] closest quaternion to input matrix, having positive q[0]

Notes

http://en.wikipedia.org/wiki/Rotation_matrix#Quaternion

Bar-Itzhack, Itzhack Y. (2000), "New method for extracting the quaternion from a rotation matrix", AIAA Journal of Guidance, Control and Dynamics 23(6):1085-1087 (Engineering Note), ISSN 0731-5090

References

- http://en.wikipedia.org/wiki/Rotation_matrix#Quaternion
- Bar-Itzhack, Itzhack Y. (2000), "New method for extracting the quaternion from a rotation matrix", AIAA Journal of Guidance, Control and Dynamics 23(6):1085-1087 (Engineering Note), ISSN 0731-5090

Examples

```
>>> import numpy as np
>>> q = mat2quat(np.eye(3)) # Identity rotation
>>> np.allclose(q, [1, 0, 0, 0])
True
>>> q = mat2quat(np.diag([1, -1, -1]))
>>> np.allclose(q, [0, 1, 0, 0]) # 180 degree rotn around axis 0
True
```

4.8.4 nearly_equivalent

transforms3d.quaternions.**nearly_equivalent**(*q1*, *q2*, *rtol=1e-05*, *atol=1e-08*)

Returns True if *q1* and *q2* give near equivalent transforms

q1 may be nearly numerically equal to *q2*, or nearly equal to *q2* * -1 (because a quaternion multiplied by -1 gives the same transform).

Parameters

q1
[4 element sequence] w, x, y, z of first quaternion

q2
[4 element sequence] w, x, y, z of second quaternion

Returns

equiv
[bool] True if *q1* and *q2* are nearly equivalent, False otherwise

Examples

```
>>> q1 = [1, 0, 0, 0]
>>> nearly_equivalent(q1, [0, 1, 0, 0])
False
>>> nearly_equivalent(q1, [1, 0, 0, 0])
True
>>> nearly_equivalent(q1, [-1, 0, 0, 0])
True
```

4.8.5 qconjugate

transforms3d.quaternions.**qconjugate**(*q*)

Conjugate of quaternion

Parameters

q
[4 element sequence] w, i, j, k of quaternion

Returns

conjq
[array shape (4,)] w, i, j, k of conjugate of *q*

4.8.6 qexp

transforms3d.quaternions.**qexp**(*q*)

Return exponential of quaternion

Parameters

q
[4 element sequence] w, i, j, k of quaternion

Returns

q_exp
[array shape (4,)] The quaternion exponential

Notes

See:

- https://en.wikipedia.org/wiki/Quaternion#Exponential,_logarithm,_and_power
- https://math.stackexchange.com/questions/1030737/exponential-function-of-quaternion-derivation

4.8.7 qeye

transforms3d.quaternions.**qeye**(*dtype=<class 'numpy.float64'>*)

Return identity quaternion

4.8.8 qinverse

transforms3d.quaternions.**qinverse**(*q*)

Return multiplicative inverse of quaternion *q*

Parameters

q
[4 element sequence] w, i, j, k of quaternion

Returns

invq
[array shape (4,)] w, i, j, k of quaternion inverse

4.8.9 qisunit

transforms3d.quaternions.**qisunit**(*q*)

Return True is this is very nearly a unit quaternion

4.8.10 qlog

transforms3d.quaternions.**qlog**(*q*)

Return natural logarithm of quaternion

Parameters

q
[4 element sequence] w, i, j, k of quaternion

Returns

q_log
[array shape (4,)] Natual logarithm of quaternion

Notes

See: https://en.wikipedia.org/wiki/Quaternion#Exponential,_logarithm,_and_power

4.8.11 qmult

transforms3d.quaternions.**qmult**(*q1*, *q2*)

Multiply two quaternions

Parameters

q1
[4 element sequence]

q2
[4 element sequence]

Returns

q12
[shape (4,) array]

Notes

See : http://en.wikipedia.org/wiki/Quaternions#Hamilton_product

4.8.12 qnorm

`transforms3d.quaternions.qnorm(q)`

Return norm of quaternion

Parameters

q
[4 element sequence] w, i, j, k of quaternion

Returns

n
[scalar] quaternion norm

Notes

http://mathworld.wolfram.com/QuaternionNorm.html

4.8.13 qpow

`transforms3d.quaternions.qpow(q, n)`

Return the n th power of quaternion q

Parameters

q
[4 element sequence] w, i, j, k of quaternion

n
[int or float] A real number

Returns

q_pow
[array shape (4,)] The quaternion q to n th power.

Notes

See: https://en.wikipedia.org/wiki/Quaternion#Exponential,_logarithm,_and_power

4.8.14 quat2axangle

transforms3d.quaternions.**quat2axangle**(*quat*, *identity_thresh=None*)

Convert quaternion to rotation of angle around axis

Parameters

quat
[4 element sequence] w, x, y, z forming quaternion.

identity_thresh
[None or scalar, optional] Threshold below which the norm of the vector part of the quaternion (x, y, z) is deemed to be 0, leading to the identity rotation. None (the default) leads to a threshold estimated based on the precision of the input.

Returns

theta
[scalar] angle of rotation.

vector
[array shape (3,)] axis around which rotation occurs.

Notes

A quaternion for which x, y, z are all equal to 0, is an identity rotation. In this case we return a 0 angle and an arbitrary vector, here [1, 0, 0].

The algorithm allows for quaternions that have not been normalized.

Examples

```
>>> vec, theta = quat2axangle([0, 1, 0, 0])
>>> vec
array([1., 0., 0.])
>>> np.allclose(theta, np.pi)
True
```

If this is an identity rotation, we return a zero angle and an arbitrary vector:

```
>>> quat2axangle([1, 0, 0, 0])
(array([1., 0., 0.]), 0.0)
```

If any of the quaternion values are not finite, we return a NaN in the angle, and an arbitrary vector:

```
>>> quat2axangle([1, np.inf, 0, 0])
(array([1., 0., 0.]), nan)
```

4.8.15 quat2mat

transforms3d.quaternions.**quat2mat**(*q*)

Calculate rotation matrix corresponding to quaternion

Parameters

q
[4 element array-like]

Returns

M
[(3,3) array] Rotation matrix corresponding to input quaternion *q*

Notes

Rotation matrix applies to column vectors, and is applied to the left of coordinate vectors. The algorithm here allows quaternions that have not been normalized.

References

Algorithm from http://en.wikipedia.org/wiki/Rotation_matrix#Quaternion

Examples

```
>>> import numpy as np
>>> M = quat2mat([1, 0, 0, 0]) # Identity quaternion
>>> np.allclose(M, np.eye(3))
True
>>> M = quat2mat([0, 1, 0, 0]) # 180 degree rotn around axis 0
>>> np.allclose(M, np.diag([1, -1, -1]))
True
```

4.8.16 rotate_vector

`transforms3d.quaternions.`**`rotate_vector`**(*v*, *q*, *is_normalized=True*)

Apply transformation in quaternion *q* to vector *v*

Parameters

v
[3 element sequence] 3 dimensional vector

q
[4 element sequence] w, i, j, k of quaternion

is_normalized
[{True, False}, optional] If True, assume *q* is normalized. If False, normalize *q* before applying.

Returns

vdash
[array shape (3,)] *v* rotated by quaternion *q*

Notes

See: http://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation#Describing_rotations_with_quaternions

4.9 reflections

Functions to work with reflections

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *aff* : affine array shape (4, 4) (3D homogenous coordinates)
- *rfnorm* : reflection in plane defined by normal vector and optional point.

aff2rfnorm(aff)	Mirror plane normal vector and point from affine *aff*
mat2rfnorm(mat)	Mirror plane normal vector from *mat* matrix.
rfnorm2aff(normal[, point])	Affine to mirror at plane defined by point and normal vector.
rfnorm2mat(normal)	Matrix to reflect in plane through origin, orthogonal to *normal*

4.9.1 aff2rfnorm

transforms3d.reflections.**aff2rfnorm**(*aff*)

Mirror plane normal vector and point from affine *aff*

Parameters

aff
[array-like, shape (4,4)]

Returns

normal
[array shape (3,)] vector normal to point (and therefore mirror plane).

point
[array shape (3,)] x, y, x coordinates of point that, together with normal, define the reflection plane.

Raises

ValueError
If there is no eigenvector for `aff[:3,:3]` with eigenvalue -1

ValueError
If determinant of `aff[:3, :3]` is not close to -1

ValueError
If there is no eigenvector for *aff* with eigenvalue 1.

Examples

```
>>> v0 = np.random.random(3) - 0.5
>>> v1 = np.random.random(3) - 0.5
>>> M0 = rfnorm2aff(v0, v1)
>>> normal, point = aff2rfnorm(M0)
>>> M1 = rfnorm2aff(normal, point)
>>> np.allclose(M0, M1)
True
```

4.9.2 mat2rfnorm

transforms3d.reflections.**mat2rfnorm**(*mat*)

Mirror plane normal vector from *mat* matrix.

Parameters

mat
[array-like, shape (3,3)]

Returns

normal
[array shape (3,)] vector normal to point (and therefore mirror plane)

Raises

ValueError
If there is no eigenvector with eigenvalue -1

ValueError
If determinant of *mat* is not close to -1

Examples

```
>>> normal = np.random.random(3) - 0.5
>>> M0 = rfnorm2mat(normal)
>>> normal = mat2rfnorm(M0)
>>> M1 = rfnorm2mat(normal)
>>> np.allclose(M0, M1)
True
```

4.9.3 rfnorm2aff

transforms3d.reflections.**rfnorm2aff**(*normal*, *point=None*)

Affine to mirror at plane defined by point and normal vector.

Parameters

normal
[3 element sequence] vector normal to point (and therefore mirror plane)

point
[3 element sequence] x, y, x coordinates of point

Returns

aff
[array shape (4,4)]

Notes

See *rfnorm2mat()*

Examples

```
>>> normal = np.random.random(3) - 0.5
>>> point = np.random.random(3) - 0.5
>>> R = rfnorm2aff(normal, point)
>>> np.allclose(2., np.trace(R))
True
>>> np.allclose(point, np.dot(R[:3,:3], point) + R[:3,3])
True
```

4.9.4 rfnorm2mat

transforms3d.reflections.**rfnorm2mat**(*normal*)

Matrix to reflect in plane through origin, orthogonal to *normal*

Parameters

normal
[array-like, shape (3,)] vector normal to plane of reflection

Returns

mat
[array shape (3,3)]

Notes

http://en.wikipedia.org/wiki/Reflection_(mathematics)

The reflection of a vector *v* in a plane normal to vector *a* is:

$$\mathrm{Ref}_a(v) = v - 2\frac{v \cdot a}{a \cdot a}a$$

The entries of the corresponding orthogonal transformation matrix *R* are given by:

$$R_{ij} = I_{ij} - 2\frac{a_i a_j}{\|a\|^2}$$

where I is the identity matrix.

4.10 shears

Functions for working with shears

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *aff* : affine array shape (4, 4) (3D homogenous coordinates)
- *striu* : shears encoded by vector giving triangular portion above diagonal of NxN array (for ND transformation)
- *sadn* : shears encoded by angle scalar, direction vector, normal vector (with optional point vector)

aff2sadn(aff)	Return shear angle, direction and plane normal from shear matrix.
inverse_outer(mat)	Return scalar t, unit vectors *a*, *b* so *mat = t * np.outer(a, b)*
mat2sadn(mat)	Return shear angle, direction and plane normal from shear matrix.
sadn2aff(angle, direction, normal[, point])	Affine for shear by *angle* along vector *direction* on shear plane.
sadn2mat(angle, direction, normal)	Matrix for shear by *angle* along *direction* vector on shear plane.
striu2mat(striu)	Construct shear matrix from upper triangular vector

4.10.1 aff2sadn

transforms3d.shears.**aff2sadn**(*aff*)

Return shear angle, direction and plane normal from shear matrix.

Parameters

mat
[array-like, shape (3,3)] shear matrix.

Returns

angle
[scalar] angle to shear, in radians

direction
[array, shape (3,)] direction along which to shear

normal
[array, shape (3,)] vector normal to shear plane

point
[array, shape (3,)] point that, with *normal*, defines shear plane.

Notes

The translation part of the affine shear matrix is calculated using:

This holds for the ``i``th coordinate:

Then:

This can be compared with the equation of the plane:

where d is the distance from the plane to the origin.

Examples

```
>>> A = sadn2aff(0.5, [1, 0, 0], [0, 1, 0])
>>> angle, direction, normal, point = aff2sadn(A)
>>> angle, direction, normal
(0.5, array([1., 0., 0.]), array([0., 1., 0.]))
>>> assert np.all(point == [0, 0, 0])
>>> A_again = sadn2aff(angle, direction, normal, point)
>>> np.allclose(A, A_again)
True
```

4.10.2 inverse_outer

transforms3d.shears.**inverse_outer**(*mat*)

Return scalar t, unit vectors *a*, *b* so *mat* = *t* * *np.outer(a, b)*

Parameters

mat
[array-like, shape (3,3)] shear matrix

Returns

t
[float] Scalar such that *mat* = *t* * *np.outer(a, b)*

a
[array, shape (3,)] Unit vector such that *mat = t * np.outer(a, b)*

b
[array, shape (3,)] Unit vector such that *mat = t * np.outer(a, b)*

4.10.3 mat2sadn

`transforms3d.shears.`**`mat2sadn`**(*mat*)

Return shear angle, direction and plane normal from shear matrix.

Parameters

mat
[array-like, shape (3,3)] shear matrix

Returns

angle
[scalar] angle to shear, in radians

direction
[array, shape (3,)] direction along which to shear

normal
[array, shape (3,)] vector defining shear plane, where shear plane passes through origin

Notes

The shear matrix we are decomposing was calculated using:

So the idea is to use an "inverse outer product" to recover the shears. See *inverse_outer()* for the implementation.

Examples

```
>>> M = sadn2mat(0.5, [1, 0, 0], [0, 1, 0])
>>> angle, direction, normal = mat2sadn(M)
>>> angle, direction, normal
(0.5, array([1., 0., 0.]), array([0., 1., 0.]))
>>> M_again = sadn2mat(angle, direction, normal)
>>> np.allclose(M, M_again)
True
```

4.10.4 sadn2aff

`transforms3d.shears.`**`sadn2aff`**(*angle*, *direction*, *normal*, *point=None*)

Affine for shear by *angle* along vector *direction* on shear plane.

The shear plane is defined by a point and normal vector. The direction vector must be orthogonal to the plane's normal vector.

A point P is transformed by the shear matrix into P" such that the vector P-P" is parallel to the direction vector and its extent is given by the angle of P-P'-P", where P' is the orthogonal projection of P onto the shear plane.

Parameters

angle
[scalar] angle to shear, in radians

direction
[array-like, shape (3,)] direction along which to shear

normal
[array-like, shape (3,)] vector normal to shear-plane

point
[None or array-like, shape (3,), optional] point, that, with *normal* defines shear plane. Defaults to None, equivalent to shear-plane through origin.

Returns

aff
[array shape (4,4)] affine shearing matrix

Examples

```
>>> angle = (np.random.random() - 0.5) * 4*math.pi
>>> direct = np.random.random(3) - 0.5
>>> normal = np.cross(direct, np.random.random(3))
>>> S = sadn2mat(angle, direct, normal)
>>> np.allclose(1.0, np.linalg.det(S))
True
```

4.10.5 sadn2mat

transforms3d.shears.**sadn2mat**(*angle*, *direction*, *normal*)

Matrix for shear by *angle* along *direction* vector on shear plane.

The shear plane is defined by normal vector *normal*, and passes through the origin. The direction vector must be orthogonal to the plane's normal vector.

A point P is transformed by the shear matrix into P" such that the vector P-P" is parallel to the direction vector and its extent is given by the angle of P-P'-P", where P' is the orthogonal projection of P onto the shear plane.

Parameters

angle
[scalar] angle to shear, in radians

direction
[array-like, shape (3,)] direction along which to shear

normal
[array-like, shape (3,)] vector defining shear plane, where shear plane passes through origin

Returns

mat
[array shape (3,3)] shear matrix

Examples

```
>>> angle = (np.random.random() - 0.5) * 4*math.pi
>>> direct = np.random.random(3) - 0.5
>>> normal = np.cross(direct, np.random.random(3))
>>> S = sadn2aff(angle, direct, normal)
>>> np.allclose(1.0, np.linalg.det(S))
True
```

4.10.6 striu2mat

transforms3d.shears.**striu2mat**(*striu*)

Construct shear matrix from upper triangular vector

Parameters

striu
[array, shape (N,)] vector giving triangle above diagonal of shear matrix.

Returns

SM
[array, shape (N, N)] shear matrix

Notes

Shear lengths are triangular numbers.

See http://en.wikipedia.org/wiki/Triangular_number

Examples

```
>>> S = [0.1, 0.2, 0.3]
>>> striu2mat(S)
array([[1. , 0.1, 0.2],
       [0. , 1. , 0.3],
       [0. , 0. , 1. ]])
>>> striu2mat([1])
array([[1., 1.],
       [0., 1.]])
>>> striu2mat([1, 2])
Traceback (most recent call last):
   ...
ValueError: 2 is a strange number of shear elements
```

4.11 taitbryan

Euler angle rotations and their conversions for Tait-Bryan zyx convention

See `euler` for general discussion of Euler angles and conventions.

This module has specialized implementations of the extrinsic Z axis, Y axis, X axis rotation convention.

The conventions in this module are therefore:

- axes i, j, k are the z, y, x axes respectively. Thus an Euler angle vector $[\alpha, \beta, \gamma]$ in our convention implies a α radian rotation around the z axis, followed by a β rotation around the y axis, followed by a γ rotation around the x axis.
- the rotation matrix applies on the left, to column vectors on the right, so if `R` is the rotation matrix, and `v` is a 3 x N matrix with N column vectors, the transformed vector set `vdash` is given by `vdash = np.dot(R, v)`.
- extrinsic rotations - the axes are fixed, and do not move with the rotations.
- a right-handed coordinate system

The convention of rotation around `z`, followed by rotation around `y`, followed by rotation around `x`, is known (confusingly) as "xyz", pitch-roll-yaw, Cardan angles, or Tait-Bryan angles.

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *euler* : (sequence of) rotation angles about the z, y, x axes (in that order)
- *axangle* : rotations encoded by axis vector and angle scalar
- *quat* : quaternion shape (4,)

axangle2euler(vector, theta)	Convert axis, angle pair to Euler angles
euler2axangle(z, y, x)	Return angle, axis corresponding to these Euler angles
euler2mat(z, y, x)	Return matrix for rotations around z, y and x axes
euler2quat(z, y, x)	Return quaternion corresponding to these Euler angles
mat2euler(M[, cy_thresh])	Discover Euler angle vector from 3x3 matrix
quat2euler(q)	Return Euler angles corresponding to quaternion *q*

4.11.1 axangle2euler

`transforms3d.taitbryan.`**`axangle2euler`**`(`*vector*, *theta*`)`

Convert axis, angle pair to Euler angles

Parameters

vector
[3 element sequence] vector specifying axis for rotation.

theta
[scalar] angle of rotation

Returns

z
[scalar]

y
[scalar]

x
[scalar] Rotations in radians around z, y, x axes, respectively

Notes

It's possible to reduce the amount of calculation a little, by combining parts of the `angle_axis2mat` and `mat2euler` functions, but the reduction in computation is small, and the code repetition is large.

Examples

```
>>> z, y, x = axangle2euler([1, 0, 0], 0)
>>> np.allclose((z, y, x), 0)
True
```

4.11.2 euler2axangle

`transforms3d.taitbryan.`**`euler2axangle`**`(`*z*, *y*, *x*`)`

Return angle, axis corresponding to these Euler angles

Uses the z, then y, then x convention above

Parameters

z
[scalar] Rotation angle in radians around z-axis (performed first)

y
[scalar] Rotation angle in radians around y-axis

x
[scalar] Rotation angle in radians around x-axis (performed last)

Returns

vector
[array shape (3,)] axis around which rotation occurs

theta
[scalar] angle of rotation

Examples

```
>>> vec, theta = euler2axangle(0, 1.5, 0)
>>> np.allclose(vec, [0, 1, 0])
True
>>> theta
1.5
```

4.11.3 euler2mat

transforms3d.taitbryan.**euler2mat**(*z*, *y*, *x*)

Return matrix for rotations around z, y and x axes

Uses the convention of static-frame rotation around the z, then y, then x axis.

Parameters

z
[scalar] Rotation angle in radians around z-axis (performed first)

y
[scalar] Rotation angle in radians around y-axis

x
[scalar] Rotation angle in radians around x-axis (performed last)

Returns

M
[array shape (3,3)] Rotation matrix giving same rotation as for given angles

Notes

The direction of rotation is given by the right-hand rule. Orient the thumb of the right hand along the axis around which the rotation occurs, with the end of the thumb at the positive end of the axis; curl your fingers; the direction your fingers curl is the direction of rotation. Therefore, the rotations are counterclockwise if looking along the axis of rotation from positive to negative.

Examples

```
>>> zrot = 1.3 # radians
>>> yrot = -0.1
>>> xrot = 0.2
>>> M = euler2mat(zrot, yrot, xrot)
>>> M.shape == (3, 3)
True
```

The output rotation matrix is equal to the composition of the individual rotations

```
>>> M1 = euler2mat(zrot, 0, 0)
>>> M2 = euler2mat(0, yrot, 0)
>>> M3 = euler2mat(0, 0, xrot)
>>> composed_M = np.dot(M3, np.dot(M2, M1))
>>> np.allclose(M, composed_M)
True
```

When applying M to a vector, the vector should column vector to the right of M. If the right hand side is a 2D array rather than a vector, then each column of the 2D array represents a vector.

```
>>> vec = np.array([1, 0, 0]).reshape((3,1))
>>> v2 = np.dot(M, vec)
>>> vecs = np.array([[1, 0, 0],[0, 1, 0]]).T # giving 3x2 array
>>> vecs2 = np.dot(M, vecs)
```

Rotations are counter-clockwise.

```
>>> zred = np.dot(euler2mat(np.pi/2, 0, 0), np.eye(3))
>>> np.allclose(zred, [[0, -1, 0],[1, 0, 0], [0, 0, 1]])
True
>>> yred = np.dot(euler2mat(0, np.pi/2, 0), np.eye(3))
>>> np.allclose(yred, [[0, 0, 1],[0, 1, 0], [-1, 0, 0]])
True
>>> xred = np.dot(euler2mat(0, 0, np.pi/2), np.eye(3))
>>> np.allclose(xred, [[1, 0, 0],[0, 0, -1], [0, 1, 0]])
True
```

4.11.4 euler2quat

transforms3d.taitbryan.**euler2quat**(*z*, *y*, *x*)

Return quaternion corresponding to these Euler angles

Uses the z, then y, then x convention above

Parameters

z
: [scalar] Rotation angle in radians around z-axis (performed first)

y
: [scalar] Rotation angle in radians around y-axis

x
: [scalar] Rotation angle in radians around x-axis (performed last)

Returns

quat
[array shape (4,)] Quaternion in w, x, y z (real, then vector) format

Notes

Formula from Sympy - see `eulerangles.py` in `derivations` subdirectory

4.11.5 mat2euler

`transforms3d.taitbryan.`**`mat2euler`**(*M*, *cy_thresh=None*)

Discover Euler angle vector from 3x3 matrix

Uses the conventions above.

Parameters

M
[array-like, shape (3,3)]

cy_thresh
[None or scalar, optional] threshold below which to give up on straightforward arctan for estimating x rotation. If None (default), estimate from precision of input.

Returns

z
[scalar]

y
[scalar]

x
[scalar] Rotations in radians around z, y, x axes, respectively

Notes

If there was no numerical error, the routine could be derived using Sympy expression for z then y then x rotation matrix, (see `eulerangles.py` in `derivations` subdirectory):

```
[                      cos(y)*cos(z),                       -cos(y)*sin(z),       ␣
↪ sin(y)],
[cos(x)*sin(z) + cos(z)*sin(x)*sin(y), cos(x)*cos(z) - sin(x)*sin(y)*sin(z), -
↪cos(y)*sin(x)],
[sin(x)*sin(z) - cos(x)*cos(z)*sin(y), cos(z)*sin(x) + cos(x)*sin(y)*sin(z), ␣
↪cos(x)*cos(y)]
```

This gives the following solutions for `[z, y, x]`:

```
z = atan2(-r12, r11)
y = asin(r13)
x = atan2(-r23, r33)
```

Problems arise when `cos(y)` is close to zero, because both of:

```
z = atan2(cos(y)*sin(z), cos(y)*cos(z))
x = atan2(cos(y)*sin(x), cos(x)*cos(y))
```

will be close to `atan2(0, 0)`, and highly unstable.

The cy fix for numerical instability in this code is from: *Euler Angle Conversion* by Ken Shoemake, p222-9 ; in: *Graphics Gems IV*, Paul Heckbert (editor), Academic Press, 1994, ISBN: 0123361559. Specifically it comes from `EulerAngles.c` and deals with the case where cos(y) is close to zero:

- http://www.graphicsgems.org/
- https://github.com/erich666/GraphicsGems/blob/master/gemsiv/euler_angle/EulerAngles.c#L68

The code appears to be licensed (from the website) as "can be used without restrictions".

4.11.6 quat2euler

`transforms3d.taitbryan.`**`quat2euler`**`(`*`q`*`)`

Return Euler angles corresponding to quaternion *q*

Parameters

q
[4 element sequence] w, x, y, z of quaternion

Returns

z
[scalar] Rotation angle in radians around z-axis (performed first)

y
[scalar] Rotation angle in radians around y-axis

x
[scalar] Rotation angle in radians around x-axis (performed last)

Notes

It's possible to reduce the amount of calculation a little, by combining parts of the `quat2mat` and `mat2euler` functions, but the reduction in computation is small, and the code repetition is large.

4.12 utils

Utilities for transforms3d

inique(iterable)	Generate unique elements from *iterable*
normalized_vector(vec)	Return vector divided by Euclidean (L2) norm
permuted_signs(seq)	Generate permuted signs for sequence *seq*
permuted_with_signs(seq)	Return all permutations of *seq* with all sign permutations
random_unit_vector([rng])	Return random normalized 3D unit vector
vector_norm(vec)	Return vector Euclidaan (L2) norm

4.12.1 inique

`transforms3d.utils.`**`inique`**`(`*`iterable`*`)`

Generate unique elements from *iterable*

Parameters

iterable
[iterable]

Returns

gen
[generator] generator that yields unique elements from *iterable*

Examples

```
>>> tuple(inique([0, 1, 2, 0, 2, 3]))
(0, 1, 2, 3)
```

4.12.2 normalized_vector

transforms3d.utils.**normalized_vector**(*vec*)

Return vector divided by Euclidean (L2) norm

See *unit vector* and *Euclidean norm*

Parameters

vec
[array-like shape (3,)]

Returns

nvec
[array shape (3,)] vector divided by L2 norm

Examples

```
>>> vec = [1, 2, 3]
>>> l2n = np.sqrt(np.dot(vec, vec))
>>> nvec = normalized_vector(vec)
>>> np.allclose(np.array(vec) / l2n, nvec)
True
>>> vec = np.array([[1, 2, 3]])
>>> vec.shape
(1, 3)
>>> normalized_vector(vec).shape
(3,)
```

4.12.3 permuted_signs

transforms3d.utils.**permuted_signs**(*seq*)

Generate permuted signs for sequence *seq*

Parameters

seq
[sequence]

Returns

gen
[generator] generator returning *seq* with signs permuted

Examples

```
>>> tuple(permuted_signs([1, -2, 0]))
((1, -2, 0), (1, -2, 0), (1, 2, 0), (1, 2, 0), (-1, -2, 0), (-1, -2, 0), (-1, 2, 0),
↪ (-1, 2, 0))
```

4.12.4 permuted_with_signs

transforms3d.utils.**permuted_with_signs**(*seq*)

Return all permutations of *seq* with all sign permutations

Parameters

seq
[sequence]

Returns

gen
[generator] generator returning permutations and sign permutations

Examples

```
>>> tuple(permuted_with_signs((1,2)))
((1, 2), (1, -2), (-1, 2), (-1, -2), (2, 1), (2, -1), (-2, 1), (-2, -1))
```

4.12.5 random_unit_vector

transforms3d.utils.**random_unit_vector**(*rng=None*)

Return random normalized 3D unit vector

Parameters

rng
[None or random number generator, optional] *rng* must have function / method *normal* that allows *size=* keyword.

Returns

vec
[shape (3,) array] Vector at random on unit sphere.

Notes

https://mathworld.wolfram.com/SpherePointPicking.html

4.12.6 vector_norm

transforms3d.utils.**vector_norm**(*vec*)

Return vector Euclidaan (L2) norm

See *unit vector* and *Euclidean norm*

Parameters

vec
[array-like shape (3,)]

Returns

norm
[scalar]

Examples

```
>>> vec = [1, 2, 3]
>>> l2n = np.sqrt(np.dot(vec, vec))
>>> nvec = vector_norm(vec)
>>> np.allclose(nvec, np.sqrt(np.dot(vec, vec)))
True
```

4.13 zooms

Functions for working with zooms (scales)

Terms used in function names:

- *mat* : array shape (3, 3) (3D non-homogenous coordinates)
- *aff* : affine array shape (4, 4) (3D homogenous coordinates)
- *zfdir* : zooms encoded by factor scalar and direction vector

aff2zfdir(aff)	Return scaling factor, direction and origin from scaling matrix.
mat2zfdir(mat)	Return scaling factor and direction from zoom (scaling) matrix
zfdir2aff(factor[, direction, origin])	Return affine to scale by *factor* around *origin* in *direction*.
zfdir2mat(factor[, direction])	Return matrix to scale by factor around origin in direction.

4.13.1 aff2zfdir

`transforms3d.zooms.`**aff2zfdir**(*aff*)

Return scaling factor, direction and origin from scaling matrix.

Parameters

aff
[array-like shape (4,4)] 4x4 *affine transformation* matrix.

Returns

factor
[scalar] zoom (scale) factor as for `zfdir2mat`

direction
[None or array, shape (3,)] direction of zoom as for `zfdir2mat`. None if scaling is uniform.

origin
[array, shape (3,)] origin of zooms

Examples

```
>>> factor = np.random.random() * 10 - 5
>>> direct = np.random.random(3) - 0.5
>>> origin = np.random.random(3) - 0.5
>>> S0 = zfdir2aff(factor)
>>> f2, d2, o2 = aff2zfdir(S0)
>>> np.allclose(S0, zfdir2aff(f2, d2, o2))
True
>>> S0 = zfdir2aff(factor, direct)
>>> f2, d2, o2 = aff2zfdir(S0)
>>> np.allclose(S0, zfdir2aff(f2, d2, o2))
True
>>> S0 = zfdir2aff(factor, direct, origin)
```

4.13.2 mat2zfdir

transforms3d.zooms.**mat2zfdir**(*mat*)

Return scaling factor and direction from zoom (scaling) matrix

Parameters

mat
[array-like shape (3,3)] 3x3 zoom matrix

Returns

factor
[scalar] zoom (scale) factor as for `zfdir2mat`

direction
[None or array, shape (3,)] direction of zoom as for `zfdir2mat`. None if scaling is uniform.

Examples

Roundtrip may not generate same factor, direction, but the generated transformation matrices will be the same

```
>>> factor = np.random.random() * 10 - 5
>>> S0 = zfdir2mat(factor, None)
>>> f2, d2 = mat2zfdir(S0)
>>> S1 = zfdir2mat(f2, d2)
>>> np.allclose(S0, S1)
True
>>> direct = np.random.random(3) - 0.5
>>> S0 = zfdir2mat(factor, direct)
>>> f2, d2 = mat2zfdir(S0)
>>> S1 = zfdir2mat(f2, d2)
>>> np.allclose(S0, S1)
True
```

4.13.3 zfdir2aff

transforms3d.zooms.**zfdir2aff**(*factor*, *direction=None*, *origin=None*)

Return affine to scale by *factor* around *origin* in *direction*.

Use factor -1 for point symmetry.

Parameters

factor
[scalar] factor to zoom by (see direction)

direction
[None or array-like shape (3,)] If None, simply apply uniform scaling by *factor*. Otherwise, apply scaling along direction given by vector *direction*. We convert direction to a *unit vector* before application.

origin
[None or array-like shape (3,)] point at which to apply implied zooms

Returns

aff
[array shape (4,4)] 4x4 transformation matrix implementing zooms

Examples

```
>>> v = (np.random.rand(3, 5) - 0.5) * 20.0
>>> S = zfdir2aff(-1.234)[:3,:3]
>>> np.allclose(np.dot(S, v), -1.234*v)
True
>>> factor = np.random.random() * 10 - 5
>>> direct = np.random.random(3) - 0.5
>>> origin = np.random.random(3) - 0.5
>>> S = zfdir2aff(factor, None, origin)
>>> S = zfdir2aff(factor, direct, origin)
```

4.13.4 zfdir2mat

transforms3d.zooms.**zfdir2mat**(*factor*, *direction=None*)

Return matrix to scale by factor around origin in direction.

Use factor == -1 for point symmetry.

Parameters

factor
[scalar] factor to zoom by (see *direction*)

direction
[None or array-like shape (3,), optional] If None, simply apply uniform scaling by *factor*. Otherwise, apply scaling along direction given by vector *direction*. We convert direction to a *unit vector* before application.

Returns

mat
[array shape (3,3)] 3x3 transformation matrix implementing zooms

Examples

```
>>> v = (np.random.rand(3, 5) - 0.5) * 20.0
>>> S = zfdir2mat(-1.234)
>>> np.allclose(np.dot(S, v), -1.234*v)
True
```

(continues on next page)

(continued from previous page)

```
>>> factor = np.random.random() * 10 - 5
>>> direct = np.random.random(3) - 0.5
>>> S = zfdir2mat(factor, direct)
```

CHAPTER

FIVE

GLOSSARY

Affine matrix
A matrix implementing an *affine transformation* in *homogenous coordinates*. For a 3 dimensional transform, the matrix is shape 4 by 4.

Affine transformation
See wikipedia affine definition. An affine transformation is a *linear transformation* followed by a translation.

Axis angle
A representation of rotation. See: wikipedia axis angle . From Euler's rotation theorem we know that any rotation or sequence of rotations can be represented by a single rotation about an axis. The axis $\hat{\boldsymbol{u}}$ is a *unit vector*. The angle is θ. The *rotation vector* is a more compact representation of θ and $\hat{\boldsymbol{u}}$.

Euclidean norm
Also called Euclidean length, or L2 norm. The Euclidean norm $\|\mathbf{x}\|$ of a vector $\mathbf{x}$ is given by:

$$\|\mathbf{x}\| := \sqrt{x_1^2 + \cdots + x_n^2}$$

Pure Pythagoras.

Euler angles
See: wikipedia Euler angles and Mathworld Euler angles.

Gimbal lock
See *Gimbal lock*

Homogenous coordinates
See wikipedia homogenous coordinates

Linear transformation
A linear transformation is one that preserves lines - that is, if any three points are on a line before transformation, they are also on a line after transformation. See wikipedia linear transform. Rotation, scaling and shear are linear transformations.

Quaternion
See: wikipedia quaternion. An extension of the complex numbers that can represent a rotation. Quaternions have 4 values, w, x, y, z. w is the *real* part of the quaternion and the vector x, y, z is the *vector* part of the quaternion. Quaternions are less intuitive to visualize than *Euler angles* but do not suffer from *gimbal lock* and are often used for rapid interpolation of rotations.

Reflection
A transformation that can be thought of as transforming an object to its mirror image. The mirror in the transformation is a plane. A plan can be defined with a point and a vector normal to the plane. See wikipedia reflection.

Rotation matrix
See wikipedia rotation matrix. A rotation matrix is a matrix implementing a rotation. Rotation matrices are square and orthogonal. That means, that the rotation matrix R has columns and rows that are *unit vector*, and

where $R^T R = I$ (R^T is the transpose and I is the identity matrix). Therefore $R^T = R^{-1}$ (R^{-1} is the inverse). Rotation matrices also have a determinant of 1.

Rotation vector

A representation of an *axis angle* rotation. The angle θ and unit vector axis $\hat{\boldsymbol{u}}$ are stored in a *rotation vector* $\boldsymbol{u}$, such that:

$$\theta = \|\boldsymbol{u}\|$$
$$\hat{\boldsymbol{u}} = \frac{\boldsymbol{u}}{\|\boldsymbol{u}\|}$$

where $\|\boldsymbol{u}\|$ is the *Euclidean norm* of $\boldsymbol{u}$

Shear matrix

Square matrix that results in shearing transforms - see wikipedia shear matrix.

Unit vector

A vector $\hat{\boldsymbol{u}}$ with a *Euclidean norm* of 1. Normalized vector is a synonym. The "hat" over the $\hat{\boldsymbol{u}}$ is a convention to express the fact that it is a unit vector.

CHAPTER

SIX

GUIDE TO MAKING A TRANSFORMS3D RELEASE

A guide for developers who are doing a Transforms3d release

6.1 Release checklist

- Review the open list of transforms3d issues. Check whether there are outstanding issues that can be closed, and whether there are any issues that should delay the release. Label them !
- Review and update the release notes. Review and update the `Changelog` file. Get a partial list of contributors with something like:

```
git shortlog -ns 0.6.0..
```

  where `0.6.0` was the last release tag name.

  Then manually go over `git shortlog 0.6.0..` to make sure the release notes are as complete as possible and that every contributor was recognized.
- Use the opportunity to update the `.mailmap` file if there are any duplicate authors listed from `git shortlog -ns`.
- Add any new authors to the `AUTHORS` file. Add any new entries to the `THANKS` file.
- Check the copyright years in `doc/conf.py` and `LICENSE`
- Clean:

```
git clean -fxd
```

- Run tests after installing into a virtualenv, to test that installing works correctly:

```
mkvirtualenv transforms3d-test
pip install pytest wheel
git clean -fxd
python setup.py install
mkdir for_test
cd for_test
pytest --doctest-modules transforms3d
```

- Make sure all tests pass on your local machine (from the Transforms3d root directory):

```
pytest --doctest-modules transforms3d
```

- Check the documentation Doctests:

```
pip install -r doc-requirements.txt
cd doc
make doctest
cd ..
```

- The release should now be ready.

6.2 Doing the release

You might want to make tag the release commit on your local machine, push to pypi, review, fix, rebase, until all is good. Then and only then do you push to upstream on github.

- Make a signed annotated tag for the release with tag of form `0.6.0`:

```
git tag -sm 'Fifth public release' 0.6.0
```

  Because we're using versioneer it is the tag which sets the package version.

- Once everything looks good, upload the source release to PyPi:

```
pip install build twine
git clean -fxd
python -m build . --sdist
twine upload -s dist/transforms*gz
```

- Remember you'll need your `~/.pypirc` file set up right for this to work. See setuptools intro. If you have 2-factor authentication, the file may look something like this:

```
[pypi]
username = __token__
```

- Check how everything looks on Pypi - the description, the packages. If necessary delete the release and try again if it doesn't look right.
- Push the tag with `git push --tags`
- Upload the docs with:

```
pip install -e .  # if you haven't done this already
pip install -r doc-requirements.txt
cd doc
make github
```

- Announce to the mailing lists. With fear and trembling.

CHAPTER

SEVEN

REFACTORING PLAN

Note the *Naming conventions*.

Base the module distinctions on the naming conventions.

In general, routines that create or break down affines, rather than, say 3x3 matrices, go with the - say - 3x3 routines, and are named `aff2something`, `something2aff`.

Routines that only apply to affines, like `compose`, `decompose`, go into the `affines` module.

Different decompositions might have different names - such as `decompose_szrt`, returning transformations in order that they are performed - here shears, zooms, rotations, translations.

Move the SPM (ND affine) decompose to `decompose_szrt_nd`.

Maybe move the comments from Euler code out into `doc` tree.

7.1 To integrate

- projection_matrix, projection_from_matrix: move to own module, rename to `projpn2aff`. `aff2projpn`.
- clip_matrix: move to own module, rename to `clip2aff`
- decompose_matrix, compose_matrix: move to `affines` module, rename to `decompose_zsrtp`, `compose_zsrtp`
- orthogonalization_matrix: move to `misc` module, rename to `orth_aff`
- superimposition_matrix: move to `misc` module, rename to `vecs2aff`
- quaterion_about_axis, quaternion_matrix, quaternion_from_matrix, quaternion_multiply, quaternion_conjugate, quaternion_inverse: check against my code
- quaternion_real, quaternion_imag: needed?
- quaternion_slerp, random_quaternion: move to `quaternions` module, rename `qslerp`, `rand_quat`.
- random_rotation_matrix: move to some module as `rand_rmat`
- Arcball class, arcball_map_to_sphere, arcball_constrain_to_axis, arcball_nearest_axis: to own module
- vector_norm: move to `utils` module, review use after refactoring
- unit_vector: replace with `utils.normalized_vector`
- random_vector, inverse_matrix, concatenate_matrices: not obviously used, very simple code, remove?
- _import_module: for preferring C functions. Move to `utils` for now, consider C / python cooexistence strategy.

7.2 C / python integration

See scipy.spatial - parallel C and Python routines, but there in a very simple case.

How about:

1. Import all the Python names via the __init__.py
2. Import all the C names (after renaming with agreed scheme as above or differently) into a `c.py` __init__ -like module. Thus something like:

```
>>> import transforms3d as t3d
>>> M = t3d.euler3rmat(0.1, 0.3, 0.2)
```

for the python code, and:

```
>>> import transforms3d.c as t3dc
>>> M = t3dc.euler2mat(0.1, 0.3, 0.2)
```

for the C code? Or the other way round of course:

```
>>> import transforms3d as t3d
>>> import transforms3d.python as t3dpy
```

or both, and switch the __init__ to import from the C or Python namespace with a one-liner.

7.3 Questions for Christoph

Does this refactoring plan make sense?

Does the naming scheme make sense (drawing distinction for example between 3x3 rotation matrix and affine)?

Now about the naming scheme for `compose` - e.g `compose_szrtp`?

OK to return a 3x3 rotation matrix from `decompose` rather then Euler angles?

How to classify `clip_matrix`, `orthogonalization_matrix`, `superimposition_matrix`?

OK to remove: `quaternion_real`, `quaternion_imag`, `random_vector`, `inverse_matrix`, `concatenate_matrices` ?

What do you think of the C / Python scheme (it's about the same as your current one)?

How do you see the relationship between your current code and this code?

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

t

INDEX

Symbols

A

C

D

E

R

S

T

U

V

X

Y

Z