quaternion Documentation

Release 2021.0.0-alpha.0

Michael Boyle

Aug 25, 2020

Contents:

For a quick start, take a look at the usage section of the README.

CHAPTER 1

Quaternions in numpy

This Python module adds a quaternion dtype to NumPy.

The code was originally based on code by Martin Ling (which he wrote with help from Mark Wiebe), but has been rewritten with ideas from rational to work with both python 2.x and 3.x (and to fix a few bugs), and *greatly* expands the applications of quaternions.

1.1 Quickstart

```
conda install -c conda-forge quaternion
```

or

```
pip install --user numpy numpy-quaternion
```

1.2 Dependencies

The basic requirements for this code are reasonably current versions of `python` and `numpy`. In particular, `python` versions 2.7, 3.6, and 3.7 are routinely tested. Also, any `numpy` version greater than 1.13.0 should work, but the tests are run on the most recent release at the time of the test.

However, certain advanced functions in this package (including `squad`, `mean_rotor_in_intrinsic_metric`, `integrate_angular_velocity`, and related functions) require `scipy` and can automatically use `numba`. `Scipy` is a standard python package for scientific computation, and implements interfaces to C and Fortran codes for optimization (among other things) need for finding mean and optimal rotors. `Numba` uses LLVM to compile python code to machine code, accelerating many numerical functions by factors of anywhere from 2 to 2000. It is *possible* to run all the code without `numba`, but these particular functions are roughly 4 to 400 times slower without it.

The only drawback of `numba` is that it is nontrivial to install on its own. Fortunately, the best python installer for scientific python, `anaconda`, makes it trivial. Just install the main `anaconda` package, which installs both `numba` and `scipy`. If you prefer the smaller download size of `miniconda` (which comes with minimal extras), you'll also have to run this command:

```
conda install numpy scipy numba
```

1.3 Installation

Assuming you use `conda` to manage your python installation (which is currently the preferred choice for science and engineering with python), you can install this package simply as

```
conda install -c conda-forge quaternion
```

If you prefer to use `pip` (whether or not you use `conda`), you can instead do

```
pip install numpy numpy-quaternion
```

If you refuse to use `conda`, you might want to install inside your home directory without root privileges. (Conda does this by default anyway.) This is done by adding `--user` to the above command:

```
pip install --user numpy numpy-quaternion
```

Note that pip will attempt to compile the code — which requires a working `C` compiler.

Finally, there's also the fully manual option of just downloading the code, changing to the code directory, and running

```
python setup.py install
```

This should work regardless of the installation method, as long as you have a compiler hanging around. However, note that you will need to have at least `numpy` installed *before* this can compile (because this package uses a header file provided by `numpy`).

1.4 Basic usage

The full documentation can be found on Read the Docs, and most functions have docstrings that should explain the relevant points. The following are mostly for the purposes of example.

```
>>> import numpy as np
>>> import quaternion
>>> np.quaternion(1,0,0,0)
quaternion(1, 0, 0, 0)
>>> q1 = np.quaternion(1,2,3,4)
>>> q2 = np.quaternion(5,6,7,8)
>>> q1 * q2
quaternion(-60, 12, 30, 24)
>>> a = np.array([q1, q2])
>>> a
array([quaternion(1, 2, 3, 4), quaternion(5, 6, 7, 8)], dtype=quaternion)
>>> exp(a)
array([quaternion(1.69392, -0.78956, -1.18434, -1.57912),
       quaternion(138.909, -25.6861, -29.9671, -34.2481)], dtype=quaternion)
```

The following ufuncs are implemented (which means they run fast on numpy arrays):

```
add, subtract, multiply, divide, log, exp, power, negative, conjugate,
copysign, equal, not_equal, less, less_equal, isnan, isinf, isfinite, absolute
```

Quaternion components are stored as doubles. Numpy arrays with `dtype=quaternion` can be accessed as arrays of doubles without any (slow, memory-consuming) copying of data; rather, a `view` of the exact same memory space can be created within a microsecond, regardless of the shape or size of the quaternion array.

Comparison operations follow the same lexicographic ordering as tuples.

The unary tests isnan and isinf return true if they would return true for any individual component; isfinite returns true if it would return true for all components.

Real types may be cast to quaternions, giving quaternions with zero for all three imaginary components. Complex types may also be cast to quaternions, with their single imaginary component becoming the first imaginary component of the quaternion. Quaternions may not be cast to real or complex types.

Several array-conversion functions are also included. For example, to convert an Nx4 array of floats to an N-dimensional array of quaternions, use `as_quat_array`:

```
>>> import numpy as np
>>> import quaternion
>>> a = np.random.rand(7, 4)
>>> a
array([[ 0.93138726,  0.46972279,  0.18706385,  0.86605021],
       [ 0.70633523,  0.69982741,  0.93303559,  0.61440879],
       [ 0.79334456,  0.65912598,  0.0711557 ,  0.46622885],
       [ 0.88185987,  0.9391296 ,  0.73670503,  0.27115149],
       [ 0.49176628,  0.56688076,  0.13216632,  0.33309146],
       [ 0.11951624,  0.86804078,  0.77968826,  0.37229404],
       [ 0.33187593,  0.53391165,  0.8577846 ,  0.18336855]])
>>> qs = quaternion.as_quat_array(a)
>>> qs
array([ quaternion(0.931387262880247, 0.469722787598354, 0.187063852060487, 0.
↪866050210100621),
       quaternion(0.706335233363319, 0.69982740767353, 0.933035590130247, 0.
↪614408786768725),
       quaternion(0.793344561317281, 0.659125976566815, 0.0711557025000925, 0.
↪466228847713644),
       quaternion(0.881859869074069, 0.939129602918467, 0.736705031709562, 0.
↪271151494174001),
       quaternion(0.491766284854505, 0.566880763189927, 0.132166320200012, 0.
↪333091463422536),
       quaternion(0.119516238634238, 0.86804077992676, 0.779688263524229, 0.
↪372294043850009),
       quaternion(0.331875925159073, 0.533911652483908, 0.857784598617977, 0.
↪183368547490701)], dtype=quaternion)
```

[Note that quaternions are printed with full precision, unlike floats, which is why you see extra digits above. But the actual data is identical in the two cases.] To convert an N-dimensional array of quaternions to an Nx4 array of floats, use `as_float_array`:

```
>>> b = quaternion.as_float_array(qs)
>>> b
array([[ 0.93138726,  0.46972279,  0.18706385,  0.86605021],
       [ 0.70633523,  0.69982741,  0.93303559,  0.61440879],
       [ 0.79334456,  0.65912598,  0.0711557 ,  0.46622885],
       [ 0.88185987,  0.9391296 ,  0.73670503,  0.27115149],
       [ 0.49176628,  0.56688076,  0.13216632,  0.33309146],
```

(continues on next page)

(continued from previous page)

```
       [ 0.11951624,  0.86804078,  0.77968826,  0.37229404],
       [ 0.33187593,  0.53391165,  0.8577846 ,  0.18336855]])
```

It is also possible to convert a quaternion to or from a 3x3 array of floats representing a rotation matrix, or an array of N quaternions to or from an Nx3x3 array of floats representing N rotation matrices, using `as_rotation_matrix` and `from_rotation_matrix`. Similar conversions are possible for rotation vectors using `as_rotation_vector` and `from_rotation_vector`, and for spherical coordinates using `as_spherical_coords` and `from_spherical_coords`. Finally, it is possible to derive the Euler angles from a quaternion using `as_euler_angles`, or create a quaternion from Euler angles using `from_euler_angles` — though be aware that Euler angles are basically the worst things ever.*1* Before you complain about those functions using something other than your favorite conventions, please read this page.

1.5 Bug reports and feature requests

Bug reports and feature requests are entirely welcome (with very few exceptions). The best way to do this is to open an issue on this code's github page. For bug reports, please try to include a minimal working example demonstrating the problem.

Pull requests are also entirely welcome, of course, if you have an idea where the code is going wrong, or have an idea for a new feature that you know how to implement.

This code is routinely tested on recent versions of both python (2.7, 3.6, and 3.7) and numpy (>=1.13). But the test coverage is not necessarily as complete as it could be, so bugs may certainly be present, especially in the higher-level functions like `mean_rotor_...`.

1.6 Acknowledgments

This code is, of course, hosted on github. Because it is an open-source project, the hosting is free, and all the wonderful features of github are available, including free wiki space and web page hosting, pull requests, a nice interface to the git logs, etc. Github user Hannes Ovrén (hovren) pointed out some errors in a previous version of this code and suggested some nice utility functions for rotation matrices, etc. Github user Stijn van Drongelen (rhymoid) contributed some code that makes compilation work with MSVC++. Github user Jon Long (longjon) has provided some elegant contributions to substantially improve several tricky parts of this code. Rebecca Turner (9999years) and Leo Stein (duetosymmetry) did all the work in getting the documentation onto Read the Docs.

Every change in this code is automatically tested on Travis-CI. This service integrates beautifully with github, detecting each commit and automatically re-running the tests. The code is downloaded and installed fresh each time, and then tested, on each of the five different versions of python. This ensures that no change I make to the code breaks either installation or any of the features that I have written tests for. Travis-CI also automatically builds the `conda` and `pip` versions of the code hosted on anaconda.org and pypi respectively. These are all free services for open-source projects like this one.

The work of creating this code was supported in part by the Sherman Fairchild Foundation and by NSF Grants No. PHY-1306125 and AST-1333129.

1.6.1 1 Euler angles are awful

Euler angles are pretty much the worst things ever and it makes me feel bad even supporting them. Quaternions are faster, more accurate, basically free of singularities, more intuitive, and generally easier to understand. You can work entirely without Euler angles (I certainly do). You absolutely never need them. But if you really can't give them up, they are mildly supported.

CHAPTER 2

Package API

`quaternion`	Adds a quaternion dtype to NumPy.
`quaternion.calculus`	
`quaternion.means`	
`quaternion.numpy_quaternion`	
`quaternion.quaternion_time_series`	

2.1 quaternion

Adds a quaternion dtype to NumPy.

Functions

`allclose`(a, b[, rtol, atol, equal_nan, verbose])	Returns True if two arrays are element-wise equal within a tolerance.
`as_euler_angles`(q)	Open Pandora's Box
`as_float_array`(a)	View the quaternion array as an array of floats
`as_quat_array`(a)	View a float array as an array of quaternions
`as_rotation_matrix`(q)	Convert input quaternion to 3x3 rotation matrix
`as_rotation_vector`(q)	Convert input quaternion to the axis-angle representation
`as_spherical_coords`(q)	Return the spherical coordinates corresponding to this quaternion
`as_spinor_array`(a)	View a quaternion array as spinors in two-complex representation
`from_euler_angles`(alpha_beta_gamma[, beta, ...])	Improve your life drastically
`from_float_array`(a)	

Continued on next page

Table 2 – continued from previous page

from_rotation_matrix(rot[, nonorthogonal])	Convert input 3x3 rotation matrix to unit quaternion
from_rotation_vector(rot)	Convert input 3-vector in axis-angle representation to unit quaternion
from_spherical_coords(theta_phi[, phi])	Return the quaternion corresponding to these spherical coordinates
isclose(a, b[, rtol, atol, equal_nan])	Returns a boolean array where two arrays are element-wise equal within a tolerance.
rotate_vectors(R, v[, axis])	Rotate vectors by given quaternions

Classes

quaternion	Floating-point quaternion numbers

class quaternion.**quaternion**

Floating-point quaternion numbers

Attributes

T transpose

a The complex number (w+i*z)

b The complex number (y+i*x)

base base object

components The components (w,x,y,z) of the quaternion as a numpy array

data pointer to start of data

dtype get array data-descriptor

flags integer value of flags

flat a 1-d view of scalar

imag The vector part (x,y,z) of the quaternion as a numpy array

itemsize length of one element in bytes

nbytes length of item in bytes

ndim number of array dimensions

real The real component of the quaternion

shape tuple of array dimensions

size number of elements in the gentype

strides tuple of bytes steps in each dimension

vec The vector part (x,y,z) of the quaternion as a numpy array

w The real component of the quaternion

x The first imaginary component of the quaternion

y The second imaginary component of the quaternion

z The third imaginary component of the quaternion

Methods

abs	Absolute value (Euclidean norm) of quaternion
absolute	Absolute value of quaternion
all	Not implemented (virtual attribute)
angle	Angle through which rotor rotates
any	Not implemented (virtual attribute)
argmax	Not implemented (virtual attribute)
argmin	Not implemented (virtual attribute)
argsort	Not implemented (virtual attribute)
astype	Not implemented (virtual attribute)
byteswap	Not implemented (virtual attribute)
choose	Not implemented (virtual attribute)
clip	Not implemented (virtual attribute)
compress	Not implemented (virtual attribute)
conj	Return the complex conjugate of the quaternion
conjugate	Return the complex conjugate of the quaternion
copy	Not implemented (virtual attribute)
copysign	Componentwise copysign
cumprod	Not implemented (virtual attribute)
cumsum	Not implemented (virtual attribute)
diagonal	Not implemented (virtual attribute)
dump	Not implemented (virtual attribute)
dumps	Not implemented (virtual attribute)
equal	True if the quaternions are PRECISELY equal
exp	Return the exponential of the quaternion (e**q)
fill	Not implemented (virtual attribute)
flatten	Not implemented (virtual attribute)
getfield	Not implemented (virtual attribute)
greater	Strict dictionary ordering
greater_equal	Dictionary ordering
inverse	Return the inverse of the quaternion
isfinite	True if the quaternion has all finite components
isinf	True if the quaternion has any INF components
isnan	True if the quaternion has any NAN components
item	Not implemented (virtual attribute)
itemset	Not implemented (virtual attribute)
less	Strict dictionary ordering
less_equal	Dictionary ordering
log	Return the logarithm (base e) of the quaternion
max	Not implemented (virtual attribute)
mean	Not implemented (virtual attribute)
min	Not implemented (virtual attribute)
newbyteorder([new_order])	Return a new *dtype* with a different byte order.
nonzero	True if the quaternion has all zero components
norm	Cayley norm (square of the absolute value) of quaternion
normalized	Return a normalized copy of the quaternion
not_equal	True if the quaternions are not PRECISELY equal

Continued on next page

Table 4 – continued from previous page

parity_antisymmetric_part	Part anti-invariant under negation of all vectors (note spinorial character)
parity_conjugate	Reflect all dimensions (note spinorial character)
parity_symmetric_part	Part invariant under negation of all vectors (note spinorial character)
prod	Not implemented (virtual attribute)
ptp	Not implemented (virtual attribute)
put	Not implemented (virtual attribute)
ravel	Not implemented (virtual attribute)
reciprocal	Return the reciprocal of the quaternion
repeat	Not implemented (virtual attribute)
reshape	Not implemented (virtual attribute)
resize	Not implemented (virtual attribute)
round	Not implemented (virtual attribute)
searchsorted	Not implemented (virtual attribute)
setfield	Not implemented (virtual attribute)
setflags	Not implemented (virtual attribute)
sort	Not implemented (virtual attribute)
sqrt	Return the square-root of the quaternion
square	Return the square of the quaternion
squeeze	Not implemented (virtual attribute)
std	Not implemented (virtual attribute)
sum	Not implemented (virtual attribute)
swapaxes	Not implemented (virtual attribute)
take	Not implemented (virtual attribute)
tofile	Not implemented (virtual attribute)
tolist	Not implemented (virtual attribute)
tostring	Not implemented (virtual attribute)
trace	Not implemented (virtual attribute)
transpose	Not implemented (virtual attribute)
var	Not implemented (virtual attribute)
view	Not implemented (virtual attribute)
x_parity_antisymmetric_part	Part anti-invariant under reflection across y-z plane (note spinorial character)
x_parity_conjugate	Reflect across y-z plane (note spinorial character)
x_parity_symmetric_part	Part invariant under reflection across y-z plane (note spinorial character)
y_parity_antisymmetric_part	Part anti-invariant under reflection across x-z plane (note spinorial character)
y_parity_conjugate	Reflect across x-z plane (note spinorial character)
y_parity_symmetric_part	Part invariant under reflection across x-z plane (note spinorial character)
z_parity_antisymmetric_part	Part anti-invariant under reflection across x-y plane (note spinorial character)
z_parity_conjugate	Reflect across x-y plane (note spinorial character)
z_parity_symmetric_part	Part invariant under reflection across x-y plane (note spinorial character)

tobytes	

a

The complex number (w+i*z)

abs()
Absolute value (Euclidean norm) of quaternion

absolute()
Absolute value of quaternion

angle()
Angle through which rotor rotates

b
The complex number (y+i*x)

components
The components (w,x,y,z) of the quaternion as a numpy array

conj()
Return the complex conjugate of the quaternion

conjugate()
Return the complex conjugate of the quaternion

copysign()
Componentwise copysign

equal()
True if the quaternions are PRECISELY equal

exp()
Return the exponential of the quaternion (e**q)

greater()
Strict dictionary ordering

greater_equal()
Dictionary ordering

imag
The vector part (x,y,z) of the quaternion as a numpy array

inverse()
Return the inverse of the quaternion

isfinite()
True if the quaternion has all finite components

isinf()
True if the quaternion has any INF components

isnan()
True if the quaternion has any NAN components

less()
Strict dictionary ordering

less_equal()
Dictionary ordering

log()
Return the logarithm (base e) of the quaternion

nonzero()
True if the quaternion has all zero components

norm()
Cayley norm (square of the absolute value) of quaternion

normalized()
Return a normalized copy of the quaternion

not_equal()
True if the quaternions are not PRECISELY equal

parity_antisymmetric_part()
Part anti-invariant under negation of all vectors (note spinorial character)

parity_conjugate()
Reflect all dimensions (note spinorial character)

parity_symmetric_part()
Part invariant under negation of all vectors (note spinorial character)

real
The real component of the quaternion

reciprocal()
Return the reciprocal of the quaternion

sqrt()
Return the square-root of the quaternion

square()
Return the square of the quaternion

vec
The vector part (x,y,z) of the quaternion as a numpy array

w
The real component of the quaternion

x
The first imaginary component of the quaternion

x_parity_antisymmetric_part()
Part anti-invariant under reflection across y-z plane (note spinorial character)

x_parity_conjugate()
Reflect across y-z plane (note spinorial character)

x_parity_symmetric_part()
Part invariant under reflection across y-z plane (note spinorial character)

y
The second imaginary component of the quaternion

y_parity_antisymmetric_part()
Part anti-invariant under reflection across x-z plane (note spinorial character)

y_parity_conjugate()
Reflect across x-z plane (note spinorial character)

y_parity_symmetric_part()
Part invariant under reflection across x-z plane (note spinorial character)

z
The third imaginary component of the quaternion

z_parity_antisymmetric_part()
Part anti-invariant under reflection across x-y plane (note spinorial character)

z_parity_conjugate()
Reflect across x-y plane (note spinorial character)

z_parity_symmetric_part()
Part invariant under reflection across x-y plane (note spinorial character)

quaternion.**as_quat_array**(*a*)
View a float array as an array of quaternions

The input array must have a final dimension whose size is divisible by four (or better yet *is* 4), because successive indices in that last dimension will be considered successive components of the output quaternion.

This function is usually fast (of order 1 microsecond) because no data is copied; the returned quantity is just a "view" of the original. However, if the input array is not C-contiguous (basically, as you increment the index into the last dimension of the array, you just move to the neighboring float in memory), the data will need to be copied which may be quite slow. Therefore, you should try to ensure that the input array is in that order. Slices and transpositions will frequently break that rule.

We will not convert back from a two-spinor array because there is no unique convention for them, so I don't want to mess with that. Also, we want to discourage users from the slow, memory-copying process of swapping columns required for useful definitions of the two-spinors.

quaternion.**as_spinor_array**(*a*)
View a quaternion array as spinors in two-complex representation

This function is relatively slow and scales poorly, because memory copying is apparently involved – I think it's due to the "advanced indexing" required to swap the columns.

quaternion.**as_float_array**(*a*)
View the quaternion array as an array of floats

This function is fast (of order 1 microsecond) because no data is copied; the returned quantity is just a "view" of the original.

The output view has one more dimension (of size 4) than the input array, but is otherwise the same shape.

quaternion.**from_float_array**(*a*)

quaternion.**as_rotation_matrix**(*q*)
Convert input quaternion to 3x3 rotation matrix

Parameters

q: quaternion or array of quaternions The quaternion(s) need not be normalized, but must all be nonzero

Returns

rot: float array Output shape is q.shape+(3,3). This matrix should multiply (from the left) a column vector to produce the rotated column vector.

Raises

ZeroDivisionError If any of the input quaternions have norm 0.0.

quaternion.**from_rotation_matrix**(*rot*, *nonorthogonal=True*)
Convert input 3x3 rotation matrix to unit quaternion

By default, if scipy.linalg is available, this function uses Bar-Itzhack's algorithm to allow for non-orthogonal matrices. [J. Guidance, Vol. 23, No. 6, p. 1085 <http://dx.doi.org/10.2514/2.4654>] This will almost certainly be quite a bit slower than simpler versions, though it will be more robust to numerical errors in the rotation matrix. Also note that Bar-Itzhack uses some pretty weird conventions. The last component of the quaternion appears to represent the scalar, and the quaternion itself is conjugated relative to the convention used throughout this module.

If scipy.linalg is not available or if the optional *nonorthogonal* parameter is set to *False*, this function falls back to the possibly faster, but less robust, algorithm of Markley [J. Guidance, Vol. 31, No. 2, p. 440 <http://dx.doi.org/10.2514/1.31730>].

Parameters

rot: (…Nx3x3) float array Each 3x3 matrix represents a rotation by multiplying (from the left) a column vector to produce a rotated column vector. Note that this input may actually have ndims>3; it is just assumed that the last two dimensions have size 3, representing the matrix.

nonorthogonal: bool, optional If scipy.linalg is available, use the more robust algorithm of Bar-Itzhack. Default value is True.

Returns

q: array of quaternions Unit quaternions resulting in rotations corresponding to input rotations. Output shape is rot.shape[:-2].

Raises

LinAlgError If any of the eigenvalue solutions does not converge

`quaternion.`**`as_rotation_vector`**(*q*)

Convert input quaternion to the axis-angle representation

Note that if any of the input quaternions has norm zero, no error is raised, but NaNs will appear in the output.

Parameters

q: quaternion or array of quaternions The quaternion(s) need not be normalized, but must all be nonzero

Returns

rot: float array Output shape is q.shape+(3,). Each vector represents the axis of the rotation, with norm proportional to the angle of the rotation in radians.

`quaternion.`**`from_rotation_vector`**(*rot*)

Convert input 3-vector in axis-angle representation to unit quaternion

Parameters

rot: (Nx3) float array Each vector represents the axis of the rotation, with norm proportional to the angle of the rotation in radians.

Returns

q: array of quaternions Unit quaternions resulting in rotations corresponding to input rotations. Output shape is rot.shape[:-1].

`quaternion.`**`as_euler_angles`**(*q*)

Open Pandora's Box

If somebody is trying to make you use Euler angles, tell them no, and walk away, and go and tell your mum.

You don't want to use Euler angles. They are awful. Stay away. It's one thing to convert from Euler angles to quaternions; at least you're moving in the right direction. But to go the other way?! It's just not right.

Assumes the Euler angles correspond to the quaternion R via

R = exp(alpha*z/2) * exp(beta*y/2) * exp(gamma*z/2)

The angles are naturally in radians.

NOTE: Before opening an issue reporting something "wrong" with this function, be sure to read all of the following page, *especially* the very last section about opening issues or pull requests. <https://github.com/moble/quaternion/wiki/Euler-angles-are-horrible>

Parameters

q: quaternion or array of quaternions The quaternion(s) need not be normalized, but must all be nonzero

Returns

alpha_beta_gamma: float array Output shape is q.shape+(3,). These represent the angles (alpha, beta, gamma) in radians, where the normalized input quaternion represents *exp(alpha*z/2) * exp(beta*y/2) * exp(gamma*z/2)*.

Raises

AllHell ...if you try to actually use Euler angles, when you could have been using quaternions like a sensible person.

`quaternion.`**`from_euler_angles`**(*alpha_beta_gamma*, *beta=None*, *gamma=None*)

Improve your life drastically

Assumes the Euler angles correspond to the quaternion R via

R = exp(alpha*z/2) * exp(beta*y/2) * exp(gamma*z/2)

The angles naturally must be in radians for this to make any sense.

NOTE: Before opening an issue reporting something "wrong" with this function, be sure to read all of the following page, *especially* the very last section about opening issues or pull requests. <https://github.com/moble/quaternion/wiki/Euler-angles-are-horrible>

Parameters

alpha_beta_gamma: float or array of floats This argument may either contain an array with last dimension of size 3, where those three elements describe the (alpha, beta, gamma) radian values for each rotation; or it may contain just the alpha values, in which case the next two arguments must also be given.

beta: None, float, or array of floats If this array is given, it must be able to broadcast against the first and third arguments.

gamma: None, float, or array of floats If this array is given, it must be able to broadcast against the first and second arguments.

Returns

R: quaternion array The shape of this array will be the same as the input, except that the last dimension will be removed.

`quaternion.`**`as_spherical_coords`**(*q*)

Return the spherical coordinates corresponding to this quaternion

Obviously, spherical coordinates do not contain as much information as a quaternion, so this function does lose some information. However, the returned spherical coordinates will represent the point(s) on the sphere to which the input quaternion(s) rotate the z axis.

Parameters

q: quaternion or array of quaternions The quaternion(s) need not be normalized, but must be nonzero

Returns

vartheta_varphi: float array Output shape is q.shape+(2,). These represent the angles (vartheta, varphi) in radians, where the normalized input quaternion represents *exp(varphi*z/2) * exp(vartheta*y/2)*, up to an arbitrary inital rotation about *z*.

`quaternion.`**`from_spherical_coords`**(*theta_phi*, *phi=None*)

Return the quaternion corresponding to these spherical coordinates

Assumes the spherical coordinates correspond to the quaternion R via

R = exp(phi*z/2) * exp(theta*y/2)

The angles naturally must be in radians for this to make any sense.

Note that this quaternion rotates *z* onto the point with the given spherical coordinates, but also rotates *x* and *y* onto the usual basis vectors (theta and phi, respectively) at that point.

Parameters

theta_phi: float or array of floats This argument may either contain an array with last dimension of size 2, where those two elements describe the (theta, phi) values in radians for each point; or it may contain just the theta values in radians, in which case the next argument must also be given.

phi: None, float, or array of floats If this array is given, it must be able to broadcast against the first argument.

Returns

R: quaternion array If the second argument is not given to this function, the shape will be the same as the input shape except for the last dimension, which will be removed. If the second argument is given, this output array will have the shape resulting from broadcasting the two input arrays against each other.

`quaternion.`**`rotate_vectors`**(*R*, *v*, *axis=-1*)

Rotate vectors by given quaternions

For simplicity, this function simply converts the input quaternion(s) to a matrix, and rotates the input vector(s) by the usual matrix multiplication. However, it should be noted that if each input quaternion is only used to rotate a single vector, it is more efficient (in terms of operation counts) to use the formula

v' = v + 2 * r x (s * v + r x v) / m

where x represents the cross product, s and r are the scalar and vector parts of the quaternion, respectively, and m is the sum of the squares of the components of the quaternion. If you are looping over a very large number of quaternions, and just rotating a single vector each time, you might want to implement that alternative algorithm using numba (or something that doesn't use python).

Parameters

R: quaternion array Quaternions by which to rotate the input vectors

v: float array Three-vectors to be rotated.

axis: int Axis of the *v* array to use as the vector dimension. This axis of *v* must have length 3.

Returns

vprime: float array The rotated vectors. This array has shape R.shape+v.shape.

`quaternion.`**`allclose`**(*a*, *b*, *rtol=8.881784197001252e-16*, *atol=0.0*, *equal_nan=False*, *verbose=False*)

Returns True if two arrays are element-wise equal within a tolerance.

This function is essentially a wrapper for the *quaternion.isclose* function, but returns a single boolean value of True if all elements of the output from *quaternion.isclose* are True, and False otherwise. This function also adds the option.

Note that this function has stricter tolerances than the *numpy.allclose* function, as well as the additional *verbose* option.

Parameters

a, b [array_like] Input arrays to compare.

rtol [float] The relative tolerance parameter (see Notes).

atol [float] The absolute tolerance parameter (see Notes).

equal_nan [bool] Whether to compare NaN's as equal. If True, NaN's in *a* will be considered equal to NaN's in *b* in the output array.

verbose [bool] If the return value is False, all the non-close values are printed, iterating through the non-close indices in order, displaying the array values along with the index, with a separate line for each pair of values.

Returns

allclose [bool] Returns True if the two arrays are equal within the given tolerance; False otherwise.

See also:

`isclose`, `numpy.all`, `numpy.any`, `numpy.allclose`

Notes

If the following equation is element-wise True, then allclose returns True.

absolute(a - b) <= ($atol$ + $rtol$ * absolute(b))

The above equation is not symmetric in a and b, so that *allclose(a, b)* might be different from *allclose(b, a)* in some rare cases.

`quaternion.`**`slerp_evaluate`**`()`

Interpolate linearly along the geodesic between two rotors

See also *numpy.slerp_vectorized* for a vectorized version of this function, and *quaternion.slerp* for the most useful form, which automatically finds the correct rotors to interpolate and the relative time to which they must be interpolated.

`quaternion.`**`squad_evaluate`**`()`

Interpolate linearly along the geodesic between two rotors

See also *numpy.squad_vectorized* for a vectorized version of this function, and *quaternion.squad* for the most useful form, which automatically finds the correct rotors to interpolate and the relative time to which they must be interpolated.

`quaternion.`**`integrate_angular_velocity`**`(`*Omega*, *t0*, *t1*, *R0=None*, *tolerance=1e-12*`)`

Compute frame with given angular velocity

Parameters

Omega: tuple or callable

Angular velocity from which to compute frame. Can be

1) a 2-tuple of float arrays (t, v) giving the angular velocity vector at a series of times,

2) a function of time that returns the 3-vector angular velocity, or

3) a function of time and orientation (t, R) that returns the 3-vector angular velocity

In case 1, the angular velocity will be interpolated to the required times. Note that accuracy is poor in case 1.

t0: float Initial time

t1: float Final time

R0: quaternion, optional Initial frame orientation. Defaults to 1 (the identity orientation).

tolerance: float, optional Absolute tolerance used in integration. Defaults to 1e-12.

Returns

t: float array

R: quaternion array

`quaternion.`**`squad`**(*R_in*, *t_in*, *t_out*)

Spherical "quadrangular" interpolation of rotors with a cubic spline

This is the best way to interpolate rotations. It uses the analog of a cubic spline, except that the interpolant is confined to the rotor manifold in a natural way. Alternative methods involving interpolation of other coordinates on the rotation group or normalization of interpolated values give bad results. The results from this method are as natural as any, and are continuous in first and second derivatives.

The input *R_in* rotors are assumed to be reasonably continuous (no sign flips), and the input *t* arrays are assumed to be sorted. No checking is done for either case, and you may get silently bad results if these conditions are violated. The first dimension of *R_in* must have the same size as *t_in*, but may have additional axes following.

This function simplifies the calling, compared to *squad_evaluate* (which takes a set of four quaternions forming the edges of the "quadrangle", and the normalized time *tau*) and *squad_vectorized* (which takes the same arguments, but in array form, and efficiently loops over them).

Parameters

R_in: array of quaternions A time-series of rotors (unit quaternions) to be interpolated

t_in: array of float The times corresponding to R_in

t_out: array of float The times to which R_in should be interpolated

`quaternion.`**`slerp`**(*R1*, *R2*, *t1*, *t2*, *t_out*)

Spherical linear interpolation of rotors

This function uses a simpler interface than the more fundamental *slerp_evaluate* and *slerp_vectorized* functions. The latter are fast, being implemented at the C level, but take input *tau* instead of time. This function adjusts the time accordingly.

Parameters

R1: quaternion Quaternion at beginning of interpolation

R2: quaternion Quaternion at end of interpolation

t1: float Time corresponding to R1

t2: float Time corresponding to R2

t_out: float or array of floats Times to which the rotors should be interpolated

`quaternion.`**`derivative`**(*f*, *t*, *derivative_order=1*, *axis=0*)

`quaternion.`**`definite_integral`**(*f*, *t*, *t1=None*, *t2=None*, *axis=0*) `quaternion.`**`indefinite_integral`** (*f*, *t*, *integral_order=1*, *axis=0*)

2.2 quaternion.calculus

Functions

antiderivative(f, t[, integral_order, axis])	
definite_integral(f, t[, t1, t2, axis])	
derivative(f, t[, derivative_order, axis])	
fd_definite_integral(f, t)	
fd_derivative(f, t)	Fourth-order finite-differencing with non-uniform time steps
indefinite_integral(f, t[, integral_order, axis])	
spline(f, t[, t_out, axis, spline_degree, ...])	Approximate input data using a spline and evaluate
spline_definite_integral(f, t[, t1, t2, axis])	
spline_derivative(f, t[, derivative_order, axis])	
spline_evaluation(f, t[, t_out, axis, ...])	Approximate input data using a spline and evaluate
spline_indefinite_integral(f, t[, ...])	

`quaternion.calculus.`**`antiderivative`** (*f*, *t*, *integral_order=1*, *axis=0*)

`quaternion.calculus.`**`definite_integral`** (*f*, *t*, *t1=None*, *t2=None*, *axis=0*)

`quaternion.calculus.`**`derivative`** (*f*, *t*, *derivative_order=1*, *axis=0*)

`quaternion.calculus.`**`fd_definite_integral`** (*f*, *t*)

`quaternion.calculus.`**`fd_derivative`** (*f*, *t*)

Fourth-order finite-differencing with non-uniform time steps

The formula for this finite difference comes from Eq. (A 5b) of "Derivative formulas and errors for non-uniformly spaced points" by M. K. Bowen and Ronald Smith. As explained in their Eqs. (B 9b) and (B 10b), this is a fourth-order formula – though that's a squishy concept with non-uniform time steps.

TODO: If there are fewer than five points, the function should revert to simpler (lower-order) formulas.

`quaternion.calculus.`**`fd_indefinite_integral`** (*f*, *t*)

`quaternion.calculus.`**`indefinite_integral`** (*f*, *t*, *integral_order=1*, *axis=0*)

`quaternion.calculus.`**`spline`** (*f*, *t*, *t_out=None*, *axis=None*, *spline_degree=3*, *derivative_order=0*, *definite_integral_bounds=None*)

Approximate input data using a spline and evaluate

Note that this function is somewhat more general than it needs to be, so that it can be reused for closely related functions involving derivatives, antiderivatives, and integrals.

Parameters

- **f: (..., N, ...) array_like** Real or complex function values to be interpolated.
- **t: (N,) array_like** An N-D array of increasing real values. The length of f along the interpolation axis must be equal to the length of t. The number of data points must be larger than the spline degree.
- **t_out: None or (M,) array_like [defaults to None]** The new values of *t* on which to evaluate the result. If None, it is assumed that some other feature of the data is needed, like a derivative or antiderivative, which are then output using the same *t* values as the input.

axis: None or int [defaults to None] The axis of *f* with length equal to the length of *t*. If None, this function searches for an axis of equal length in reverse order – that is, starting from the last axis of *f*. Note that this feature is helpful when *f* is one-dimensional or will always satisfy that criterion, but is dangerous otherwise. Caveat emptor.

spline_degree: int [defaults to 3] Degree of the interpolating spline. Must be 1 <= spline_degree <= 5.

derivative_order: int [defaults to 0] The order of the derivative to apply to the data. Note that this may be negative, in which case the corresponding antiderivative is returned.

definite_integral_bounds: None or (2,) array_like [defaults to None] If this is not None, the *t_out* and *derivative_order* parameters are ignored, and the returned values are just the (first) definite integrals of the splines between these limits, along each remaining axis.

`quaternion.calculus.`**`spline_definite_integral`**(*f*, *t*, *t1=None*, *t2=None*, *axis=0*)

`quaternion.calculus.`**`spline_derivative`**(*f*, *t*, *derivative_order=1*, *axis=0*)

`quaternion.calculus.`**`spline_evaluation`**(*f*, *t*, *t_out=None*, *axis=None*, *spline_degree=3*, *derivative_order=0*, *definite_integral_bounds=None*)

Approximate input data using a spline and evaluate

Note that this function is somewhat more general than it needs to be, so that it can be reused for closely related functions involving derivatives, antiderivatives, and integrals.

Parameters

f: (. . . , N, . . .) array_like Real or complex function values to be interpolated.

t: (N,) array_like An N-D array of increasing real values. The length of f along the interpolation axis must be equal to the length of t. The number of data points must be larger than the spline degree.

t_out: None or (M,) array_like [defaults to None] The new values of *t* on which to evaluate the result. If None, it is assumed that some other feature of the data is needed, like a derivative or antiderivative, which are then output using the same *t* values as the input.

axis: None or int [defaults to None] The axis of *f* with length equal to the length of *t*. If None, this function searches for an axis of equal length in reverse order – that is, starting from the last axis of *f*. Note that this feature is helpful when *f* is one-dimensional or will always satisfy that criterion, but is dangerous otherwise. Caveat emptor.

spline_degree: int [defaults to 3] Degree of the interpolating spline. Must be 1 <= spline_degree <= 5.

derivative_order: int [defaults to 0] The order of the derivative to apply to the data. Note that this may be negative, in which case the corresponding antiderivative is returned.

definite_integral_bounds: None or (2,) array_like [defaults to None] If this is not None, the *t_out* and *derivative_order* parameters are ignored, and the returned values are just the (first) definite integrals of the splines between these limits, along each remaining axis.

`quaternion.calculus.`**`spline_indefinite_integral`**(*f*, *t*, *integral_order=1*, *axis=0*)

2.3 quaternion.means

Functions

mean_rotor_in_chordal_metric(R[, t])	Return rotor that is closest to all R in the least-squares sense
mean_rotor_in_intrinsic_metric(R[, t])	
optimal_alignment_in_chordal_metric(Ra, Rb)	Return Rd such that Rd*Rb is as close to Ra as possible

quaternion.means.**mean_rotor_in_chordal_metric**(*R*, *t=None*)

Return rotor that is closest to all R in the least-squares sense

This can be done (quasi-)analytically because of the simplicity of the chordal metric function. It is assumed that the input R values all are normalized (or at least have the same norm).

Note that the *t* argument is optional. If it is present, the times are used to weight the corresponding integral. If it is not present, a simple sum is used instead (which may be slightly faster). However, because a spline is used to do this integral, the number of input points must be at least 4 (one more than the degree of the spline).

quaternion.means.**mean_rotor_in_intrinsic_metric**(*R*, *t=None*)

quaternion.means.**optimal_alignment_in_chordal_metric**(*Ra*, *Rb*, *t=None*)

Return Rd such that Rd*Rb is as close to Ra as possible

This function simply encapsulates the mean rotor of Ra/Rb.

As in the *mean_rotor_in_chordal_metric* function, the *t* argument is optional. If it is present, the times are used to weight the corresponding integral. If it is not present, a simple sum is used instead (which may be slightly faster).

2.4 quaternion.numpy_quaternion

Functions

slerp_evaluate	Interpolate linearly along the geodesic between two rotors
squad_evaluate	Interpolate linearly along the geodesic between two rotors

quaternion.numpy_quaternion.**slerp_evaluate**()

Interpolate linearly along the geodesic between two rotors

See also *numpy.slerp_vectorized* for a vectorized version of this function, and *quaternion.slerp* for the most useful form, which automatically finds the correct rotors to interpolate and the relative time to which they must be interpolated.

quaternion.numpy_quaternion.**squad_evaluate**()

Interpolate linearly along the geodesic between two rotors

See also *numpy.squad_vectorized* for a vectorized version of this function, and *quaternion.squad* for the most useful form, which automatically finds the correct rotors to interpolate and the relative time to which they must be interpolated.

2.5 quaternion.quaternion_time_series

Functions

`angular_velocity`(R, t)	
`integrate_angular_velocity`(Omega, t0, t1[, ...])	Compute frame with given angular velocity
`minimal_rotation`(R, t[, iterations])	Adjust frame so that there is no rotation about z' axis
`slerp`(R1, R2, t1, t2, t_out)	Spherical linear interpolation of rotors
`squad`(R_in, t_in, t_out)	Spherical "quadrangular" interpolation of rotors with a cubic spline

Classes

`appending_array`(shape[, dtype, initial_array])	

Attributes

quaternion.quaternion_time_series.**angular_velocity**(*R*, *t*)

class quaternion.quaternion_time_series.**appending_array**(*shape*, *dtype=<class 'float'>*, *initial_array=None*)

Attributes

a

Methods

append	

a

append(*row*)

quaternion.quaternion_time_series.**frame_from_angular_velocity_integrand**(*rfrak*, *Omega*)

quaternion.quaternion_time_series.**integrate_angular_velocity**(*Omega*, *t0*, *t1*, *R0=None*, *tolerance=1e-12*)

Compute frame with given angular velocity

Parameters

Omega: tuple or callable

Angular velocity from which to compute frame. Can be

1) a 2-tuple of float arrays (t, v) giving the angular velocity vector at a series of times,
2) a function of time that returns the 3-vector angular velocity, or
3) a function of time and orientation (t, R) that returns the 3-vector angular velocity

In case 1, the angular velocity will be interpolated to the required times. Note that accuracy is poor in case 1.

t0: float Initial time

t1: float Final time

R0: quaternion, optional Initial frame orientation. Defaults to 1 (the identity orientation).

tolerance: float, optional Absolute tolerance used in integration. Defaults to 1e-12.

Returns

t: float array

R: quaternion array

quaternion.quaternion_time_series.**minimal_rotation**(*R*, *t*, *iterations=2*)

Adjust frame so that there is no rotation about z' axis

The output of this function is a frame that rotates the z axis onto the same z' axis as the input frame, but with minimal rotation about that axis. This is done by pre-composing the input rotation with a rotation about the z axis through an angle gamma, where

dgamma/dt = 2*(dR/dt * z * R.conjugate()).w

This ensures that the angular velocity has no component along the z' axis.

Note that this condition becomes easier to impose the closer the input rotation is to a minimally rotating frame, which means that repeated application of this function improves its accuracy. By default, this function is iterated twice, though a few more iterations may be called for.

Parameters

R: quaternion array Time series describing rotation

t: float array Corresponding times at which R is measured

iterations: int [defaults to 2] Repeat the minimization to refine the result

quaternion.quaternion_time_series.**slerp**(*R1*, *R2*, *t1*, *t2*, *t_out*)

Spherical linear interpolation of rotors

This function uses a simpler interface than the more fundamental *slerp_evaluate* and *slerp_vectorized* functions. The latter are fast, being implemented at the C level, but take input *tau* instead of time. This function adjusts the time accordingly.

Parameters

R1: quaternion Quaternion at beginning of interpolation

R2: quaternion Quaternion at end of interpolation

t1: float Time corresponding to R1

t2: float Time corresponding to R2

t_out: float or array of floats Times to which the rotors should be interpolated

quaternion.quaternion_time_series.**squad**(*R_in*, *t_in*, *t_out*)

Spherical "quadrangular" interpolation of rotors with a cubic spline

This is the best way to interpolate rotations. It uses the analog of a cubic spline, except that the interpolant is confined to the rotor manifold in a natural way. Alternative methods involving interpolation of other coordinates on the rotation group or normalization of interpolated values give bad results. The results from this method are as natural as any, and are continuous in first and second derivatives.

The input *R_in* rotors are assumed to be reasonably continuous (no sign flips), and the input *t* arrays are assumed to be sorted. No checking is done for either case, and you may get silently bad results if these conditions are violated. The first dimension of *R_in* must have the same size as *t_in*, but may have additional axes following.

This function simplifies the calling, compared to *squad_evaluate* (which takes a set of four quaternions forming the edges of the "quadrangle", and the normalized time *tau*) and *squad_vectorized* (which takes the same arguments, but in array form, and efficiently loops over them).

Parameters

R_in: array of quaternions A time-series of rotors (unit quaternions) to be interpolated

t_in: array of float The times corresponding to R_in

t_out: array of float The times to which R_in should be interpolated

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

q

Index

A

B

C

D

E

F

G

I