qmath Quick Card

Marco Abrate
abrate.m@gmail.com

Contents

1 Installation

The following software is required before installing qmath:

- Python 2.x+ (http://www.python.org/)
- NumPy 1.x+ (http://new.scipy.org/download.html)

The first step is to download the qmath tarball from http://pypi.python.org/pypi. Open a shell. Unpack the tarball in a temporary directory (not directly in Python's site-packages). Commands:

```
tar zxf qmath-X.Y.Z.tar.gz
```

X, Y and Z are the major and minor version numbers of the tarball. Go to the directory created by expanding the tarball:

```
cd qmath-X.Y.Z
```

Get root privileges:

```
su

(enter root password)
```

To install for python type:

```
python setup.py install
```

If the python executable isn't on your path, you'll have to specify the complete path, such as /usr/local/bin/python.

2 Classes

class quaternion(*attitude*)
Quaternion class.
attitude can be:
- a number (of any type, complex are included);

```
>>> import qmath
>>> qmath.quaternion(1)
(1.0)
>>> qmath.quaternion(1+1j)
(1.0+1.0i)
```

- a list or a numpy array of the components with respect to `1,i,j` and `k`;

```
>>> qmath.quaternion([1,2,3,4])
(1.0+2.0i+3.0j+4.0k)
>>> qmath.quaternion(np.array([1,2,3,4]))
(1.0+2.0i+3.0j+4.0k)
```

- a string of the form `'a+bi+cj+dk'`;

```
>>> qmath.quaternion('1+1i+3j-2k')
(1.0+1.0i+3.0j-2.0k)
```

- a rotation about an axis using pairs (rotation angle, axis of rotation);

```
>>> qmath.quaternion(0.5 * math.pi, [0,0,1])
(0.968912421711+0.247403959255k)
```

- a list whose components are Euler angles;

```
>>> import math
>>> qmath.quaternion([0.0,math.pi / 6,math.pi / 3])
(0.836516303738+0.482962913145i+0.224143868042j-0.129409522551k)
```

- a 3X3 rotation matrix. The matrix must be given as a numpy array.

```
import numpy as np
>>> qmath.quaternion(np.array([[0  ,-0.8 ,-0.6],\\
                               [0.8,-0.36, 0.48],\\
                               [0.6,0.48 ,-0.64]]))
(0.707106781187i+0.565685424949j+0.424264068712k)
```

class hurwitz(*attitude*)
The class of Hurwitz quaternions, i.e. quaternions whose components are integers. *attitude* can be the same as for quaternion class, if the components as a quaternion are integers.

3 Methods

__repr__(*self*)
Quaternions are represented in the algebraic way: $q = a + bi + cj + dk$, and a, b, c, d are floats. Components are of float type if *self* is a quaternion, of integer type if *self* is a Hurwitz quaternion.

__getitem__(*self*, *key*)
Returns one of the four components of the quaternion. This method allows to get the components by `quaternion[key]`.

__setitem__(*self*, *key*, *number*)
Set one of the four components of the quaternion. This method allows to set the components by `quaternion[key] = number`.

__delitem__(*self*, *key*)
Delete (set to zero) one of the four components of the quaternion. This method allows to write `del quaternion[key]`.

__delslice__(*self*, key_1, key_2)
Delete (set to zero) the components of the quaternion from the $key_1 - th$ to the $key_2 - th$. This method allows to write `del quaternion[1:3]`.

__contains__(*self*, *key*)
Returns 0 if the component of the quaternion with respect to *key* is zero, 1 otherwise. This method allows to write `del 'k' in quaternion`.

```
>>> q = qmath.quaternion('1+1i+5k')
>>> 'j' in q
False
>>> 'i' in q
True
```

__eq__(*self*, *other*)
Returns `True` if two quaternion are equal, `False` otherwise. This method allows to write `quaternion1 == quaternion2`.

```
>>> q = qmath.quaternion('1+1k')
>>> q == 0
False
>>> q == '1+1k'
True
```

Also equalities with a tolerance are admitted:

```
>>> q == qmath.quaternion([1,0,1e-15,1])|1e-9
True
>>> q == [1,1,1e-15,0]
False
```

__ne__(*self*, *other*)
Returns `False` if two quaternion are equal, `True` otherwise. This method allows to write `quaternion1 != quaternion2`.

```
>>> q = qmath.quaternion('1+1k')
>>> q != 0
True
>>> q != '1+1k'
False
```

int(*self*)
Converts *self* components into integers.

```
>>> q = qmath.quaternion('1+1i+5k')
>>> q
(1.0+1.0i+5.0k)
>>> q._int_()
(1+1i+5k)
```

__iadd__(*self*, *other*)
__isub__(*self*, *other*)
__imul__(*self*, *other*)
__idiv__(*self*, *other*)
These methods are called to implement the augmented arithmetic assignments. These methods do the operation in-place (modifying self). If *other* is a Hurwitz quaternion but its inverse is not, the __idiv__ method raises an error.

__imod__(*self*, *other*)
This method is called to implement the modular reduction. It is performed only if *self* is a Hurwitz quaternion and *other* is an integer. Otherwise an error is raised.

__add__(*self*, *other*)
__sub__(*self*, *other*)
__mul__(*self*, *other*)
__div__(*self*, *other*)
__mod__(*self*, *other*)
These methods are called to implement the binary arithmetic operations. For instance, to evaluate the expression `x + y`, where `x` is a quaternion, `x.__add__(y)` is called. `y` can either be a quaternion (or a Hurwitz quaternion) or something that can be converted to quaternion.
If *other* is a Hurwitz quaternion but its inverse is not, the __div__ method raises an error.
The __mod__ method is performed only if *self* is a Hurwitz quaternion and *other* is integer (see __imod__).

__rmul__(*self*, *other*)
__rdiv__(*self*, *other*)

These methods are called to implement the binary arithmetic operations with reflected operands. If *other* is a Hurwitz quaternion but its inverse is not, the __rdiv__ method raises an error.

__neg__(*self*)

Return the opposite of a quaternion. You can write `- quaternion`.

__pow__(*self*, *exponent*[, *modulo*])

Implements the operator `**`. The power of a quaternion can be computed for integer power (both positive or negative) and also if the exponent is half or third a number: for example square or cube roots are evaluated (see also `sqrt` and `croot`). If *self* is a Hurwitz quaternion, powers are computed only for natural exponents. Modular reduction is performed for Hurwitz quaternions (if *self* is a Hamilton quaternion, modulo is ignored).

```
>>> base = qmath.quaternion('1+1i+2j-2k')
>>> base ** 3
(-26.0-6.0i-12.0j+12.0k)
>>> base ** (-2)
(-0.08-0.02i-0.04j+0.04k)
>>> qmath.quaternion([-5,1,0,1]) ** (1.0/3)
(1.0+1.0i+1.0k)
>>> qmath.quaternion([-5,1,0,1]) ** (2.0/3)
(-1.0+2.0i+2.0k)
>>> qmath.quaternion('1.0+1.0i+1.0k') ** 2
(-1.0+2.0i+2.0k)
>>> qmathcore.hurwitz('1+1i+1k') ** 2
(-1+2i+2k)
>>> pow(qmathcore.hurwitz('1+1i+1k'),2,3)
(2+2i+2k)
```

__abs__(*self*)

Returns the modulus of the quaternion.

equal(*self*, *other*[, *tolerance*])

Returns quaternion equality with arbitrary tolerance. If no tolerance is admitted it is the same as __eq__(*self*, *other*).

```
>>> a = qmath.quaternion([1,1,1e-15,0])
>>> b = qmath.quaternion(1+1j)
>>> a.equal(b,1e-9)
True
>>> a.equal(b)
False
```

`real`(*self*)
Returns the real part of the quaternion.

`imag`(*self*)
Returns the imaginary part of the quaternion.

`trace`(*self*)
Returns the trace of the quaternion (the double of its real part).

`conj`(*self*)
Returns the conjugate of the quaternion.

`norm`(*self*)
Returns the norm of the quaternion (the square of the modulus).

`delta`(*self*)
Returns the δ of the quaternion, that is the opposite of the norm of the imaginary part of the quaternion.

`inverse`(*self*[, *modulo*])
Quaternionic inverse, if it exists. It is equivalent to `quaternion ** (-1)`. Modular inversion can be performed for Hurwitz quaternions (if self is a Hamilton quaternion, modulo is ignored).

```
>>> a = qmath.quaternion([2,-2,-4,-1])
>>> a.inverse()
(0.08+0.08i+0.16j+0.04k)
>>> b = qmath.hurwitz([0,-2,-2,0])
>>> b.inverse(13)
(10i+10j)
```

`unitary`(*self*)
Returns the normalized quaternion, if different from zero.

`sqrt`(*self*)
Computes the square root of the quaternion. If the quaternion has only two roots, the one with positive trace is given: if this method returns r, also -r is a root.

`croot`(*self*)
Computes the cube root (unique) of a quaternion.

`QuaternonToRotation`(*self*)
Converts the quaternion, if unitary, into a rotation matrix.

4 Additional Functions

`real`(*object*)
The same as `object.real`().

`imag`(*object*)
The same as `object.imag`().

`trace`(*object*)
The same as `object.trace`().

`conj`(*object*)
The same as `object.conj`().

`norm`(*object*)
The same as `object.norm`().

`delta`(*object*)
The same as `object.delta`().

`inverse`(*object*[, *module*])
The same as `object.inverse`([*module*]).

`unitary`(*object*)
The same as `object.unitary`().

`sqrt`(*object*)
The same as `object.sqrt`().

`croot`(*object*)
The same as `object.croot`().

`QuaternonToRotation`(*object*)
The same as `object.QuaternioinToRotation`().

`RotationToQuaternion`(*angle*, *vector*)
Converts a pair angle-vector into a quaternion.

`StringToQuaternion`(*string*)
Converts a string into a quaternion.

`MatrixToEuler`(*matrix*)
Converts a 3X3 matrix into a vector having Euler angles as components.

`EulerToQuaternion`(*list*)
Converts a vector whose components are Euler angles into a quaternion.

`identity`()
Returns 1 as a quaternion.

```
>>> qmath.identity()
(1.0)
```

zero()
Returns 0 as a quaternion.

```
>>> qmath.zero()
(0.0)
```

dot($object_1, object_2$)
Returns the dot product of two quaternions.

```
>>> a = qmath.quaternion('1+2i-2k')
>>> b = qmath.quaternion('3-2i+8j')
>>> qmath.dot(a,b)
-1.0
```

CrossRatio(a, b, c, d)
Returns the cross ratio of four quaternions defined by:

$$\texttt{CrossRatio}(a,b,c,d) = (a-c)\cdot(a-d)^{-1}\cdot(b-d)\cdot(b-c)^{-1}.$$

If $a = d$ or $b = c$ returns the string `'Infinity'`.
The arguments of `CrossRatio` can be passed as a tuple.

```
>>> a = qmath.quaternion([1,0,1,0])
>>> b = qmath.quaternion([0,1,0,1])
>>> c = qmath.quaternion([-1,0,-1,0])
>>> d = qmath.quaternion([0,-1,0,-1])
>>> qmath.CrossRatio(a,b,c,d)
(2.0)
>>> tpl = a,b,c,d
>>> qmath.CrossRatio(tpl)
(2.0)
>>> qmath.CrossRatio(a,b,b,d)
'Infinity'
>>> qmath.CrossRatio(a,a,a,d)
(1.0)
>>> qmath.CrossRatio(a,b,a,b)
(0.0)
```

Moebius(z, a, b, c, d)
Returns the Moebius transformation with parameters a,b,c and d:

$$f(z) = (a\cdot z + b)\cdot(c\cdot z + d)^{-1}.$$

If $c\cdot z + d = 0$ returns the string `'Infinity'`.
The arguments of `Moebius` can be passed as a tuple.

```
>>> a = qmath.quaternion([1,1,1,0])
>>> b = qmath.quaternion([-2,1,0,1])
>>> c = qmath.quaternion([1,0,0,0])
>>> d = qmath.quaternion([0,-1,-3,-4])
>>> z = qmath.quaternion([1,1,3,4])
>>> qmath.Moebius(z,a,b,c,d)
(-5.0+7.0i+7.0k)
>>> d = - z
>>> z = qmath.Moebius(z,a,b,c,d)
>>> z
'Infinity'
>>> qmath.Moebius(z,a,b,c,d)
(1.0+1.0i+1.0j)
```