Rotation Formalism in 3 Dimension

- Name: Anubhav Dinesh Patel
- Email: anubhavp28@gmail.com
- Github: anubhavp28
- University: Indian Institute of Information Technology Kalyani
- Course: Bachelor of Technology in Computer Science and Engineering
- Course Term: 2017/18 - 2020/21 (4 Year)
- Timezone: IST (GMT +5:30)

Proposal

This project is regarding adding a module scipy.spatial.rotation that would allow for easy description, creation and application of rotations in 3 dimensions.

Deliverables

At the end of this project, scipy will have a *scipy.spatial.rotation* module with:

- *Rotation* class which would provide ability to take a rotation or a sequence of rotations in 3d represented using any of the following formalisms - quaternions, axis-angle, rotation matrix, euler angles and apply them to 3d vectors or a sequence of 3d vectors. It would also provide support for conversion from one formalism to another.

- *qspline* function to perform quaternion cubic spline interpolation (interpolation with quaternion, angular rate and acceleration vectors as continuous functions of time).

- *slerp* function which would perform quaternion interpolation given the ends and the parameter between 0 and 1 using Spherical Linear Interpolation (SLERP) algorithm.

- *davenportq* function to solve for Wahba's Problem, given the set of body frame vectors, set of reference vectors and non-negative weights.

- *orthogonalize* function to orthogonalize a approximately orthogonal 3x3 matrix using modified Shepperd's algorithm.

Implementation

The implementation language will be Python. For any implementation of method/algorithm, numpy vector operations will be prefered, making sure computation happen at C level. Loops inside python would be avoided as much as possible. The implementation of any method/algorithm will accompany tests and documentation for the same.

Discussion

Rather than supporting multiple convention of a formalism, in my opinion it is better to support just a single widely used convention. Our choice should be consistent across all the methods. For this project, I have decided on a convention and way of representation (using *ndarray*) for each formalism -

1. Quaternions - will be represented by a *ndarray* of dim (4) with its scalar part first, for a sequence of quaternions *ndarray* of dim (n,4) will be used.
2. Euler Angles - will be represented by a *ndarray* of dim (3). For a sequence of euler angles *ndarray* of dim (n,3) would be used. Euler angles have 24 different conventions. To implement support for all of the conventions would be a difficult task. An already available implementation is given in [14], which can be used in SciPy to support all 24 conventions of euler angles. Though the implementation is not vectorised we may miss out on performance gain, but considering the limited time period of GSoC, it is our best choice. In the future we can add a vectorised implementation.

   For user to be able to choose among different conventions, every euler function will have an addition parameter *axes* - 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.

3. Angle-Axis - will be represented by a *ndarray* of dim (3), as a vector r = θê where θ is the angle of rotation (in radians) and ê is a unit vector representing the eigen-axis.

4. Rotation Matrices - will be represented by a *ndarray* of dim (3,3) and sequence of rotation matrices by *ndarray* of dim (n,3,3).

Details

Rotation

Rotation class will provide a common convenient interface to represent a rotation, compose and apply them. *Rotation* will internally use quaternions, with ability to store a single rotation or a sequence of rotations. It will be implemented with the following methods -

1. *__init__(self, rarray, formalism = "quat")*

Sets the instance to the given rotation(s). Internally, it will call the appropriate method among *setfromquat, setfromvector, setfrommatrix and setfromeangles,* depending upon the *formalism* supplied.

```
Parameters
----------

rarray : ndarray
    Array of rotations. Its is expected to be of specific dimension depending
    upon the value of `formalism` parameter. Below is the list of valid
    formalism and the expected dimensions of rarray for it.

    “quat”    --> (n,4)   or (4)                      : for quaternions
    “matrix”  --> (n,3,3) or (3,3)                    : for rotation matrices or DCM
    “vector”  --> (n,3) or (3)                        : for axis-angle
    any valid `axes` sequence  --> (n,3) or (3)     : for euler angles

formalism : string, optional
    The formalism used to represent rotations in rarray. Defaults to quaternion
    representation.
```

2. *setfromquat(self, rarray)*

Sets the *Rotation* instance to rotations of the given quaternions. Since the *Rotation* class uses quaternions internally, it would require no conversion. I feel we need to discussion the behaviour when the method is given a non-unit quaternion - should it throw an exception or silently convert it to a unit quaternion.

```
Parameters
----------
rarray : ndarray with shape of (n,4)
    Array of quaternions with their scalar part first, ie in form of (w,x,y,z).
    Quaternions are assumed to be of unit magnitude.

Returns
-------
None
```

3. ***setfromvector(self, rarray)***

Sets the `Rotation` instance to rotations of the given rotation vectors (axis-angle vectors). The algorithm described by [1] can be used here.

```
Parameters
----------

rarray : ndarray with shape of (n,3)
    Array of rotation vectors(axis-angle vectors) assumed to be of the form
    r = theta * e^, where e^ is a unit vector in the direction of the axis of
    rotation and theta is the angle in radians. The rotation occurs in the
    sense prescribed by the right-hand rule.

Returns
-------
None
```

4. ***setfrommatrix(self, rarray)***

Sets the *Rotation* instance to rotations of the given rotation matrices(DCM). Among the conversion algorithm given in [2], the one with copysign seems simple and cleaner for implementation. Behaviour of this method need a discussion for the case when the given matrix is not a pure rotation i.e. is not orthogonal with determinant 1.

```
Parameters
----------
rarray : ndarray with shape of (n,3,3)
    Array of rotation matrices


Returns
-------
None
```

5. *setfromeangles(self, rarray, axes = "rzyx")*

Sets the *Rotation* instance to rotations of the given euler angles. The algorithm is described in [3].

```
Parameters
----------

rarray : ndarray with shape of (n,3)
    Array of euler angles.

Returns
-------
None
```

6. *rotate(vectors)*

Function will rotate the given array of vectors. Depending upon the cardinality of the *Rotation* instance as well as the dimensions of vectors ndarray given, different operation will be performed, i.e. the function will choose among one rotation on many vectors, many rotations on one vector and many rotations on many vectors. The algorithm to rotate a vector by a quaternion is given in [4].

permitted combinations are -

1. When the instance is set to a single rotation and the dimension of vectors is (n,3). The single rotation will be used to rotate all the vectors.

2. When the instance is set to a sequence of n rotations and the dimension of vectors is (1,3). The sequence of n rotations will be applied in order to the single vector.

3. When the instance is set to a sequence of n rotations and the dimension of vectors in (n,3). Then 1-to-1 mapping between rotations and vectors will be used, i.e. the ith vector will be rotated by ith rotation of the instance.

```
Parameters
----------
vectors : ndarray of dim (n,3)
    Array of vectors to rotate.


Returns
-------
An ndarray of rotated vectors.
```

7. *__mul__(self, other)*

* operator will be overloaded to provide a way to combine two sequence of

rotations. The result of multiplication of two instance of Rotation will depend upon the cardinality of sequence of rotations they represent. The algorithm for combining two quaternions is given in [4], basically which just requires a quaternion multiplication.

permitted combinations of A*B, where A and B are instance of *Rotation* will be -

1. When A is set to single rotation and B is set to sequence of n rotations. The resulting instance will have cardinality of n. The ith rotation of the resulting instance will represent a rotation first by A and then by B[i].

2. When A is set to sequence of n rotations and B is set to a single rotation. The resulting instance will have cardinality of n. The ith rotation of the resulting instance will represent a rotation first by A[i] and then by B.

3. When A is set to sequence of n rotations and B is also set to another sequence of n rotations. The resulting instance will have cardinality of n. The ith rotation of the resulting instance will represent a rotation first by A[i] and then by B[i].

All other combination will generate a error.

8. *toquat()*
returns the sequence of rotations the instance is set to, using quaternions.

```
Parameters
----------
None

Returns
-------
An ndarray of dimension (n,4)
```

9. *tomatrix()*
returns the sequence of rotations the instance is set to, using rotation matrices(DCM). To perform conversion from quaternion to rotation matrix, the algorithm described in [5] could be used.

```
Parameters
----------
None

Returns
-------
An ndarray of dimension (n,3,3)
```

10. *tovector()*

returns the sequence of rotations the instance is set to, using rotation vectors (axis-angle representation). The algorithm detailed in [6] could be used for conversion from quaternions to rotation vectors with proper consideration for edge cases of theta = 0 and theta = pi.

```
Parameters
----------
None

Returns
-------
An ndarray of dimension (n,3)
```

11. *toeangles(axes = "rzyx")*

returns the sequence of rotations the instance is set to, using euler angles. [7] details an algorithm for quaternion to euler angles conversion taking into account the singularities.

```
Parameters
----------
None

Returns
-------
An ndarray of dimension (n,3)
```

Quaternion SLERP

For performing a spherical linear interpolation between quaternions, always using the "short way" between quaternions, a function `slerp` will be implemented. Implementation of this seems to be easy and quite straightforward. [8] describes two formulae for quaternion slerp, the second one derived from 4D geometry seems more practical and easier for implementation. For edge cases such as when two quaternions have very small angle between them we can switch to linear interpolation.

```
slerp(q0,q1,t)

Parameters
----------
q0  : ndarray
    initial quaternion (with scalar part first)

q1  : ndarray
    final quaternion (with scalar part first)

t   : int
```

interpolation parameter 0 < t < 1

Returns
----
interpolated quaternion as a ndarray

Quaternion Cubic Spline Interpolation

Spline interpolation will match the value of quaternion at given times and the quaternion, angular rate and acceleration vectors will be continuous functions.

For performing a cubic spline interpolation, a function `qspline` will be implemented. The function will take value of quaternion at specific times, the initial and the final angular rate along with number of output points.

The algorithm described in [12] and a sample implementation in [13], could be used. Since, the implementation is complex, internal functions will be created - _b, _binverse, _r, _rates, _coeff_calc, _slew.

qspline(t,q,wi,wf,o)

Parameters
----
t : ndarray of dim (n)
  array of input times
q : ndarray of dim (n)
  array of quaternion at corresponding times
wi : float
  initial angular rate
wf : float
  final angular rate
o : Int
  number of output points

Return

A tuple of 3 ndarrays, for quaternions, angular rates and angular acceleration.

Davenport's Q-Method

For solving Wahba's Problem, a function `daveportq` will be implemented. Attitude quaternion will be calculated to perform transformation from reference frame to body frame. The function will find the Davenport matrix and then use numpy to calculate maximum eigenvalues and corresponding eigenvector. The algorithm is described in [10] and a elaborated proof in [11].

davenportq(p,q,w)

```
Parameters
----------
p : ndarray of dim (n,3)
    array of reference frame vectors
q : ndarray of dim (n,3)
    array of body frame vectors
w : ndarray of dim (n)
    array of non-negative weights

Returns
-------
ndarray of size (4) representing the best fitting attitude
quaternion.
```

Orthogonalization of 3x3 Matrix

For re-orthogonalization of an approximately orthogonal 3x3 matrix, we can implement a function `orthogonalize`. It will implement the modified Shepperd's algorithm described in [9].

```
orthogonalize(mat)

Parameters
----------
mat : ndarray of dim (n,3,3)

Returns
-------
ndarray of dim (n,3,3).
```

Tentative Timeline

Community Bonding Period : April 23, 2018 - May 14, 2018

- Setup development environment
- Active participation in regular meetings
- Finalise API
- Finalise deadlines and milestones
- Increase familiarity with practices and processes
- Bug fixes/patches

Week 1 : May 14, 2018 - May 21, 2018

- Implement setter functions of *Rotation* class - *setfromquat, setfromvector, setfrommatrix, setfromeangles*.
- Documentation for the same.

Week 2 : May 21, 2018 - May 28, 2018

- Implement functions - *toquat, tomatrix, tovector, toeangles* of Rotation class.
- Documentation for above functions.
- Add tests to cover functions implemented in week 1 and week 2.

Week 3 : May 28, 2018 - June 4, 2018

- Implement function *rotate* of *Rotation* class.
- Documentation for *rotate*.
- Add tests for rotate.

Week 4 : June 4, 2018 - June 11, 2018

- Implement function __mul__ of *Rotation* class.
- Documentation for multiplication of *Rotation* instances.
- Add tests for multiplication.

Week 5 : June 11, 2018 - June 18, 2018

- < Buffer Week > Complete leftover work and discussion with mentors.

Week 6 : June 18, 2018 - June 25, 2018

- Implement functions - slerp and orthogonalize.
- Start documenting and writing tests for the same.

Week 7 : June 25, 2018 - July 2, 2018

- Complete documentation and tests for slerp and orthogonalize.
- Implement davenportq function.
- Documentation and tests for davenportq.

Week 8 : July 2, 2018 - July 9, 2018

- Complete documentation and tests for davenportq.
- Complete any leftover work.

Week 9 : July 9, 2018 - July 16, 2018

- Implement internal functions of qspline - _b, _binverse, _r.
- Documentation for the same.

Week 10 : July 16, 2018 - July 23, 2018

- Implement internal function of qspline - _rates.
- Documentation for the same.

Week 11 : July 23, 2018 - July 30, 2018

- Implement internal function of qspline - _calccoeff, _slew.
- Implement qspline
- Documentation and tests for qspline.

Week 12 : July 30, 2018 - August 6, 2018

- < Buffer Week > Complete leftover work and discussion with mentors.

Previous Contribution to Scipy

- (**Open**) https://github.com/scipy/scipy/pull/8584

References

[1] http://www.euclideanspace.com/maths/geometry/rotations/conversions/angleToQuaternion/index.htm

[2] https://en.wikipedia.org/wiki/Rotation_matrix

[3] "Euler Angles, Quaternions and Transformation Matrices", NASA. https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19770024290.pdf

[4] https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation

[5] http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToMatrix/index.htm

[6] https://en.wikipedia.org/wiki/Axis%E2%80%93angle_representation

[7] http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/

[8] http://run.usc.edu/cs520-s15/assign2/p245-shoemake.pdf

[9] "Unit Quaternion from Rotation Matrix", F. Landis Markley, NASA Goddard Space Flight Center.

[10] http://malcolmdshuster.com/FC_MarkleyMortari_Girdwood_1999_AAS.pdf

[11] https://math.stackexchange.com/questions/1634113/davenports-q-method-finding-an-orientation-matching-a-set-of-point-samples

[12] http://qspline.sourceforge.net/qspline.pdf

[13] https://sourceforge.net/projects/qspline/files/qspline/%5BUnnamed%20release%5D/qspline.zip/download

[14] http://matthew-brett.github.io/transforms3d/reference/transforms3d.euler.html