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)
- Current Year : 1st Year
- Telephone: +91 9082079014 (India)
- 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.
- *cspline* function to perform cubic spline interpolation (interpolation with quaternions, angular rate and acceleration vectors as continuous functions of time).
- *slerp* function which would perform Spherical Linear Interpolation (SLERP) given the initial and final quaternions, and the parameter between 0 and 1.
- *absorient* function to solve for Absolute Orientation Problem, given the two sets of vectors and non-negative weights. *absorient*

will provide a general interface to multiple algorithms that can be used to solve Absolute Orientation Problem. Though for this project, only Davenport-Q algorithm will be implemented.

- *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 additional 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).

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 algorithms given in [2], the one involving creating a 4x4 symmetric matrix seems to be the best to me. It has ability to handle significant numerical error.

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 array of euler angles. This method would internally call *transforms3d.euler.euler2quat* of [14].

Parameters
---

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

Returns

None

6. *rotate(vectarray)*

Function will rotate the given array of vectors. Depending upon the cardinality of the *Rotation* instance as well as the dimension of *vectarray* 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].

Which operation to perform when r.rotate(v) is called are decided based on the following rules -

1. When the instance *r* is set to a single rotation and the dimension of *v* is (n,3) i.e. v is an array of n vectors, then the single rotation will be used to rotate all the vectors.

2. When the instance *r* is set to a sequence of n rotations and the dimension of *v* is (1,3) or (3) i.e. v is a single vector, then all of the n rotations will be applied in order to the single vector.

3. When the instance *r* is set to a sequence of n rotations and the dimension of *v* is (n,3) i.e. v is an array of n vectors. 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.

Examples

Let's take the common case of rotating a single vector from a rotation matrix.

```
>>> mat = np.array([0.6190476,  0.1904762, -0.7619048, 0.1904762,  0.9047619,
   0.3809524, 0.7619048, -0.3809524,  0.5238096])

>>> mat.reshape(3,3)
array([[ 0.6190476,  0.1904762, -0.7619048],
       [ 0.1904762,  0.9047619,  0.3809524],
       [ 0.7619048, -0.3809524,  0.5238096]])

>>> vec = np.array([1, 0, 1])

>>> r = Rotation(mat,"matrix")

>>> r.rotate(vec[np.newaxis, :])
array([[-0.1428571455180645, 0.571428582072258, 1.2857143096625805]])
```

Similarly, to rotate a set of vectors from a rotation matrix.

```
>>> mat = np.array([0.6190476,  0.1904762, -0.7619048, 0.1904762,  0.9047619,
   0.3809524, 0.7619048, -0.3809524,  0.5238096])

>>> mat.reshape(3,3)
array([[ 0.6190476,  0.1904762, -0.7619048],
       [ 0.1904762,  0.9047619,  0.3809524],
       [ 0.7619048, -0.3809524,  0.5238096]])

>>> vec = np.array([[ 1,  0,  1],
                    [ 1,  4,  3],
                    [-1,  2,  1]])
>>> vec.shape
(3, 3)

>>> r = Rotation(mat,"matrix")

>>> r.rotate(vec)
array([[-0.1428571455180645, 0.571428582072258,  1.2857143096625805],
       [-0.9047619216144083, 4.952381044626236,  0.8095238246023652],
       [-1.0000000186264515, 2.000000037252903, -1.0000000186264515]])
```

To perform many rotations on one vector -

```
>>> mat = np.array([[[ 0.6190476,  0.1904762, -0.7619048],
                    [ 0.1904762,  0.9047619,  0.3809524],
                    [ 0.7619048, -0.3809524,  0.5238096]],

                   [[ 0.5873016, -0.6031746, -0.5396826],
                    [ 0.7301587,  0.6825397,  0.031746 ],
                    [ 0.3492064, -0.4126984,  0.8412699]],

                   [[ 0.3333333, -0.6666667, -0.6666667],
                    [ 0.8717949,  0.4871795, -0.0512821],
                    [ 0.3589744, -0.5641026,  0.7435898]]])

>>> mat.shape
(3, 3, 3)

>>> vec = np.array([1, 0, 1])

>>> r = Rotation(mat,"matrix")

>>> r.rotate(vec[np.newaxis, :])
array([[-1.1224490269345213, -0.8602826806732822, 0.004709538601820731]])
```

To perform many rotations on many vectors (1 to 1 mapping) -

```
>>> mat = np.array([[[ 0.6190476,  0.1904762, -0.7619048],
                    [ 0.1904762,  0.9047619,  0.3809524],
                    [ 0.7619048, -0.3809524,  0.5238096]],

                   [[ 0.5873016, -0.6031746, -0.5396826],
                    [ 0.7301587,  0.6825397,  0.031746 ],
                    [ 0.3492064, -0.4126984,  0.8412699]],

                   [[ 0.3333333, -0.6666667, -0.6666667],
                    [ 0.8717949,  0.4871795, -0.0512821],
                    [ 0.3589744, -0.5641026,  0.7435898]]])

>>> mat.shape
(3, 3, 3)

>>> vec = np.array([[ 1,  0,  1],
                   [ 1,  4,  3],
                   [-1,  2,  1]])
>>> vec.shape
(3, 3)

>>> r = Rotation(mat,"matrix")
```

```
>>> r.rotate(vec)
array([[-0.14285715,  0.57142858,  1.28571431],
       [-3.44444453,  3.55555564,  1.22222237],
       [-2.33333338,  0.05128206, -0.74358978]])
```

7. *__mul__(self, other)*

* operator will be overloaded to provide a way to combine two sequence of rotations. The result of multiplication of two instances of Rotation will depend upon the cardinality of sequence of rotations they represent. The algorithm for combining two quaternions is given in [4], which essentially is 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 combinations will generate an error.

8. *__eq__(self, other)*

== operator will be overloaded to provide a way to check if two instances of *Rotation* are set to the same sequence of rotations or not.

Returns

Boolean

9. *append(other)*

Function will allow to append the sequence of rotations represented by a *Rotation* instance to the current *Rotation* instance.

Parameters
---
other : instance of *Rotation*

Returns

None

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

Parameters

None

Returns

An ndarray of dimension (n,4)

11. *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)

12. *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)

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

returns the sequence of rotations the instance is set to, using euler angles. For conversion from quaternions to euler angles, the available implementation of this in [14] will be used.

Parameters

None

Returns

An ndarray of dimension (n,3)

14. *cardinality()*
returns the cardinality/size/length of sequence of rotations the instance is set to.

Parameters

None

Returns

An integer

Spherical Linear Interpolation (SLERP)

For performing a spherical linear interpolation between quaternions, always using the "short way" between quaternions, a function `slerp` will be implemented. The function will take *Rotation* instances as parameters to denote starting and ending quaternions. Implementation of this seems to be easy and quite straightforward. *Rotation* internally uses quaternions. [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(start,end,t)

Parameters
----
start : *Rotation* instance set to a single rotation
    initial orientation

end : *Rotation* instance set to a single rotation
    final orientation

```
t   : int
    interpolation parameter 0 < t < 1

Returns
-------
interpolated orientation as Rotation instance
```

Quaternion Cubic Spline Interpolation

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

For performing a cubic spline interpolation, a function `cspline` will be implemented. The function will take quaternions at specific times, the initial and the final angular rate along with number of output points. The function will interpolate at equally spaced 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.

```
cspline(t,orients,wi,wf,o)

Parameters
----------
t : ndarray of dim (n)
  array of input times
orients : Rotation instance set to a sequence of n rotations.
  sequence of quaternions for input
wi : float
  initial angular rate
wf : float
  final angular rate
o : Int
  number of output points

Return
------
A tuple of size 3 - Rotation instance set to output quaternions, ndarray of
angular rates and ndarray of angular acceleration.

Notes
-----
A brief description of internal functions, the function `cspline` will have -
```

1. **_b**

Will implement transformation from coefficient vector to angular rate vector.

2. **_binverse**
   Will implement transformation from angular rate to coefficient vector.

3. **_r**
   Computes contribution to final angular acceleration using final angular rate vector.

4. **_rates**
   Computes intermediate angular rates.

5. **_coeff_calc**
   Calculate coefficients for interpolation function and its derivatives.

6. **_slew**
   Computes orientation, angular rate and angular acceleration.

Absolute Orientation Solver

For solving Absolute Orientation Problem, a function `absorient` will be implemented. `absorient` will provide a general interface to different method/algorithm for solving absolute orientation problem. Algorithms that can be used to solve absolute orientation are SVD (Singular Value Decomposition), Davenport-Q, QUEST and FOAM. SVD & Davenport-Q are robust algorithm, but involve matrix factorisation and are quite slow. QUEST and FOAM are computationally efficient, but use estimations.

`absorient` will take two set of vectors as parameters and a array of non-negative weights. Optional parameters to decide the method/algorithm and the direction of rotation transform.

For this project, `absorient` will be implemented with only one method/algorithm `davenportq`, in future support for other methods/algorithms can be added.

For implementation of davenportq, [15] provides a algorithm to convert absolute orientation problem into equivalent wahba's problem. Thereafter the function will find the Davenport matrix and then use numpy to calculate the maximum eigenvalue and corresponding eigenvector. This part of the algorithm is described in [10] and a elaborated proof in [11].

absorient(p,q,w,meth="davenportq",p2q=True)

Parameters

```
----------
p    : ndarray of dim (n,3)
       array of vectors in first frame
q    : ndarray of dim (n,3)
       array of vectors in second frame
w    : ndarray of dim (n)
       array of non-negative weights
meth : string, optional
       to specify which algorithm to use
p2q  : boolean, optional
       computed rotation transform will convert vectors of p to q, if True
       computed rotation transform will convert vectors of q to p, if False

Returns
-------
A tuple of size 2 - a instance of Rotation and ndarray representing translation
transform.
```

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.
- Actively participate in regular meetings.
- Finalise API.
- Finalise deadlines and milestones.
- Increase familiarity with community practices and processes.
- Contribute bug fixes/patches.

- Setup blog for GSoC.

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__, __eq__, cardinality* and *append* of *Rotation* class.
- Documentation and tests for the same.

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 *absorient* function with Davenport-Q algorithm.
- Documentation and tests for *absorient*.

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

- Complete documentation and tests for *absorient*.
- Complete any leftover work.

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

- Implement internal functions of cspline necessary to compute intermediate angular rates - _b, _binverse, _r.

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

- Implement internal function to calculate intermediate angular rates and coefficient vectors of interpolating polynomial - _rates, _calccoeff.

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

- Complete implementation of cspline
- Documentation and tests for cspline.

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

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

Additional Commitments

I might not be able to maintain the same hours/week for the following periods -

- For the period of - **April 20 to April 30,** I will have my university examination (End-Semester Examination).
- From **July 1 to August 6,** I will have my regular university classes.

Previous Contribution to Scipy

- (**Open**) https://github.com/scipy/scipy/pull/8584
- (**Open**) https://github.com/scipy/scipy/pull/8612
- (**Merged**) https://github.com/scipy/scipy/pull/8613

Additional Code Samples

- I had created this test to see the possible improvement in speed with numpy vectorized operations (in matrix to quaternion conversion). https://gist.github.com/anubhavp28/e79645a544c6e7b16b408a172e522134

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

[15]
https://www.researchgate.net/profile/Manolis_Lourakis/publication/316445722_An_efficient_solution_to_absolute_orientation/links/59fc8915458515d07064cccf/An-efficient-solution-to-absolute-orientation.pdf