GSOC 2018 : Implement scipy.spatial.rotation

- **Samyak Jain**

Sub-Org Info

I am applying for **GSOC'18** under the sub org **Scipy** of **Python Software Foundation**.

Personal Information

Name	Samyak Jain
Country	India
Institution	International Institute of Information Technology, Hyderabad
Degree	B.Tech in Computer Science and Research.
Current Year	2nd
Email	smkjain8@gmail.com samyak.j@research.iiit.ac.in
Phone	+91-8179658068
Time Zone	UTC + 5:30 (IST)
Social Profile	Github: https://github.com/samyak0210

About Me

I am Samyak Jain, currently studying at **The International Institute of Information Technology**, Hyderabad. I am a 2nd-year student pursuing B.Tech in Computer Science and MS in Research. I have had prior experience with web-development making different web-apps like a model of currently working bookmyshow and implementing a web quiz game on Ruby-on-rails. I am good at various languages - Python, C, C++, Javascript and PHP. I am a new open source contributor and very keen towards contributing to it.

Contributions towards SciPy

I tried to understand the various modules of scipy through solving some issues.

- Added examples of scipy.integrate.dblquad

  This PR deals with the addition of example related to double integration in the integration module of scipy.
- Added examples of scipy.integrate.tplquad

  This PR deals with the addition of example related to triple integration in the integration module of scipy.
- Improved Docstring of various modules

  This PR deals with changing variables named I, O and l (small L), as followed by pep8/pycodestyle E741, in integrate and matlab module of scipy.

Why choose Scipy?

I chose to contribute to python software foundation because of my great interest in python. I have made various projects in python - a **Bomberman game** in python and a small working client-server model along with a proxy server in python. I chose scipy as mathematics will never stop fascinating me. I got both in a nutshell in scipy and Rotation Formalism project was quite interesting. By contributing towards this project I will be able to learn newer ways and algorithms to perform rotations in two and three-dimensions. I have a good knowledge about two and three-dimensional rotations as I have experience working with OpenGL 3.0. Also, I would love to learn new representations of rotations apart from Euler angles like quaternions and DCMs. Developing a new package for scipy organization will be a good exposure for me.

Time Clash

I may not be available from 18 June to 1 July, as there is a planned vacation trip, due to loss of any internet source. But if I could get the internet connection, I will continue doing my project and will devote 12-15 hours per week during those 2 weeks.

Time Commitments

I will work for 35 hours per week. But I could work more during weekends as per the need of mentors and also cover over my time lost during the time clash.

Proposal Title

SciPy: Rotation Formalism in 3 dimensions

This proposal is regarding the implementation of a new module Rotation in scipy which helps to describe, apply and compose rotations.

Project Abstract

This project aims at implementing and expressing rotations in three dimensions which are extensively used in computer vision and is a missing module in SciPy.
There are various ways to represent a rotation- Euler angles, Quaternions, Direction Cosines Matrices. The main part of the project is to implement conversions between one form of representation to any other form of representation.

Description

There are different ways to represent a rotation. The representations mostly used are -

- **Euler Angles:** According to Euler's rotation theorem any rotation can be represented using three angles. The three angles giving the three rotation matrices are called Euler angles. (ø,θ,ψ) are Euler angles representing rotations about x,y and z-axis. The rotation matrices are multiplied to a single transformation matrix. There exists twelve possible sequences of rotations axes which are divided into two groups :

  - **Proper Euler angles** : (`z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y`)
  - **Tait-Bryan angles** : (`x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z`)

  a. **Pros**: Euler angles can be extended to higher dimensions as well.
  b. **Cons**: These do not solve the problem of **"Gimbal Lock".** Also computationally three 3x3 matrix multiplications are done for every rotation made.

- **Direction Cosine Matrices (DCM):** The Direction Cosines of a vector are the cosines of the angles between the vector and the three coordinate axes. Direction cosines of the vector **v (v_x, v_y, v_z)** are represented by α,β,γ is given by -

$$\alpha = v_x / (v_x + v_y + v_z)$$
$$\beta = v_y / (v_x + v_y + v_z)$$
$$\gamma = v_z / (v_x + v_y + v_z)$$

  a. **Pros:** Computationally speaking it is quite cheap as a single 3x3 matrix multiplication occurs.
  b. **Cons:** It doesn't cover the edge cases and doesn't solve the problem of **"Gimbal Lock".**

❑ **Quaternions:** These are a number system that extends the complex numbers. They are represented as -

$$a + b\mathbf{i} + c\mathbf{j} + d\mathbf{k}$$

where *a*, *b*, *c*, and *d* are real numbers, and *i*, *j*, and *k* are the *fundamental quaternion units*. Any vector **v** can be represented as quaternion as (0,**v**) where 0 is the scalar part and **v** is the vector part.

a. **Pros**: Quaternions takes care of all edge cases of Euler angles and solves the problem of **"Gimbal Lock".** Spherical interpolation of quaternions gives better results than interpolation of Euler angles.
b. **Cons**: Due to their non-commutative nature, it becomes mathematically tough to compute the rotation using this representation. Talking computationally, 4x4 matrix multiplications takes place which is quite expensive.

In this context, quaternions seem to be the best option to represent the vectors and compute the rotation matrix.

Quaternions in 3 dimension is represented as a 4-tuple - (a,b,c,d)
Multiplication of two quaternions (a,b,c,d) and (e,f,g,h) in matrix domain is given by -

```
[a   -b    -c   -d]       [ e ]
[b    a    -d    c]   *    [ f ]
[c    d     a   -b]        [ g ]
[d   -c     b    a]      [ h ]
```

We can perform rotations using quaternions as follows -

v is the vector about which rotation is to be performed in 3d and α is the angle of rotation then -

$$\mathbf{u} = \cos\alpha/2 + \sin\alpha/2 * \mathbf{v}/|v|$$

To perform rotation of the vector **q** represented by `(a,b,c)` then the rotated vector is given by -

$$\mathbf{u}*(a\mathbf{i} + b\mathbf{j} + c\mathbf{k})*\mathbf{u}^{-1}$$

where -

$$\mathbf{i}*\mathbf{j} = \mathbf{k},$$

$$\mathbf{j}*\mathbf{k} = \mathbf{i},$$

$$\mathbf{k}*\mathbf{i} = \mathbf{j},$$

$$\mathbf{i}^2 = \mathbf{j}^2 = \mathbf{k}^2 = \mathbf{i}*\mathbf{j}*\mathbf{k} = -1$$

Vector `(a,b,c)` can be represented as quaternion as `(0,a,b,c)` and usual matrix multiplication can be used to compute the rotated vector. **Alternatively,** a single transformation matrix can also be used to compute the rotation.

$$\mathbf{q} = (\cos(\theta/2), \sin(\theta/2)\vec{a}) = (w, (x, y, z))$$

$$R_{\mathbf{q}} = \begin{pmatrix} 1-2y^2-2z^2 & 2xy-2wz & 2xz+2wy & 0 \\ 2xy+2wz & 1-2x^2-2z^2 & 2yz-2wx & 0 \\ 2xz-2wy & 2yz+2wx & 1-2x^2-2y^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The rotated vector is

$$\mathbf{q'} = R_q.\mathbf{q}$$

where **q** is **homogenized** vector (`a,b,c,1`).

In this context to perform rotations using quaternions, I will use **numpy.array** function to define the matrix and perform fast vectorized multiplications to enhance the cost of computation.

API Structure Design

The **Rotation** class internal structure will use **Quaternions** to represent rotations as they are easy to compute and have a simple 4-tuple representation.

The vector to be rotated by convention is taken to be a column vector but in certain conditions, it is given as row vector. Internally computations will be done considering it a column vector.

`Initializers`

The following are the methods for **initializing** the rotation matrix and defining the internal representation of the rotation -

- **`euler2quat(choice='zyx',pitch=0.0, roll=0.0, yaw=0.0)`:**
  Convert Euler form to quaternion form
  Parameters -
  a. **choice:** *String*

It represents about which axis the rotation needs to be performed in sequence.

b. pitch, roll, yaw: *Float*

These represent the Euler angles about which the rotation matrices are formed.

Returns normalized quaternion.

- **`dcm2quat(matrix=`*`numpy.array`*`):`**

  Convert direction cosines matrix to quaternion 4-tuple.

  Parameters -

  a. **matrix:** *numpy.array/numpy.ndarray*

  The direction cosines matrix can be represented as a numpy array instance.

  Returns normalized quaternion.

Converters

The Class will also have different **conversion** methods -

- **`quat2euler():`** This function will return the yaw, pitch, roll (Euler Angles) of the quaternion used internally. There will be some problems-
  - There are two ways to represent the same rotation using Euler angles which has to be gracefully handled.
  - The problem of **“Gimbal Lock”** also has to be taken care of.

- **`quat2dcm()`:** This function will return a *numpy.array* instance. The matrix will represent the direction cosines of the rotated vector.

Class Methods

Various other **class** methods defined are-

- **`__init__()`:** It will initialize our class with a 4-tuple quaternion which will be a private object of our class.

- **`__mul__()`:** Overloading the multiplication operator to define multiplication between two quaternions.

- **`axis_rotate(axis='z',angle=0.0)`:** The function will rotate about the given axis with the given angle.

  Parameters-

  a. **axis:** *String*

     This argument represents the axis about which the rotation is to occur. It can be {**'x','y','z'**}.

  b. **angle:** *Float*

     This argument represents the angle of rotation.

  Returns normalized quaternion for internal representation.

- **`rotate(vector=numpy.array)`:** The function will perform the actual rotation between the internally computed quaternion tuple

and the vector to be rotated. The vector is assumed to be a column vector.

Parameters-

a. **vector:** *numpy.array/numpy.ndarray*
   This will be a (3,n) vector to perform rotation to all ‘n’ points in space.

- **`slerp(quat1, quat2, num)`:** It will define the interpolation between different orientations.

  Parameters-

  a. **quat1 , quat2:** *4-tuple*
     The arguments represents the quaternions for interpolation.

  b. **num:** *int*
     It represents the number of rotations between the given quaternions.

  Returns normalized quaternion wrapped in the Rotation instance.

- **`random_sample()`:** This function will be used for uniform random sampling of rotations.

- **`spline(quat, ws, we)`:** It will perform the cubic interpolation between the set of rotations.

  Parameters-

  a. **quat:** *4-tuple*
     The set of rotations represented by quaternion.

  b. **ws, we:** *float*
     It defines the angular rates at the end points.

- **`wahba_estimate(p1, p2)`:** It will return the estimate of rotation between the set of points.

  Parameters-

  a. **p1, p2:** *numpy.array/numpy.ndarray*
     The set of 3 dimensional vectors to get the estimate rotation of the given points.

Prior Available Implementations

---

1. **Conversion Algorithms:** The conversions between direction cosines, Euler angles, quaternion and axis angles currently exists in matlab. Functions to convert DCMs to quaternion, Euler to quaternion and vice-versa.

2. **SLERP Algorithm:** Interpolations between quaternions is already defined in matlab, quatinterp function defines quaternion interpolation between two normalized quaternions.

   These available implementations can form the basis for further development. Increasing accuracy and correctness of the algorithms used in the functions is a major concern here. These implementations in matlab can be used to verify our results and improve the accuracy of the class methods.

Code Snippets

- **__init__**: Initializes the class with an arbitrary private tuple.

```
def __init__(self):

    self._quat = tuple([0, 0, 0, 0])
```

- **__mul__**: Defines multiplication between two quaternions.

```
def __mul__(self, other):

        a = self._quat[0]; b = self._quat[1]
        c = self._quat[2]; d = self._quat[3]
        e = other._quat[0]; f = other._quat[1]
        g = other._quat[2]; h = other._quat[3]

        q1 = np.array([
                       [a, -b, -c, d],
                       [b, a, -d, c],
                       [c, d, a, -b],
                       [d, -c, b, a]
                                   ])

        q2 = np.array([e, f, g, h]).reshape((4,1))
```

```
        return tuple(np.matmul(q1,q2).reshape(1,4))
```

- **euler2quat:** Conversion of Euler angles to quaternions.

```
def euler2quat(self, choice = 'zyx', pitch=0.0,
               roll = 0.0, yaw = 0.0):

        """ Implemented for a single case of zyx
              Pitch, yaw, roll are in radians"""

        cy = math.cos(yaw * 0.5); sy = math.sin(yaw * 0.5)
        cr = math.cos(roll * 0.5); sr = math.sin(roll * 0.5)
        cp = math.cos(pitch * 0.5); sp = math.sin(pitch *
0.5)

        w = cy * cr * cp + sy * sr * sp
        x = cy * sr * cp - sy * cr * sp
        y = cy * cr * sp + sy * sr * cp
        z = sy * cr * cp - cy * sr * sp

        return tuple([w,x,y,z])
```

- **quat2euler:** Conversion of quaternions to Euler angles

```
def quat2euler(self):

        w = self._quat[0]
        x = self._quat[1]
        y = self._quat[2]
        z = self._quat[3]

        a = 2.0*(w * x + y * z)
        b = 1.0 - 2.0*(x**2 + y**2)
        pitch = math.degrees(math.atan2(a, b))

        a = 2.0 * (w * y - z * x)
        a = 1.0 if a > 1.0 else a
        a = -1.0 if a < -1.0 else a

        roll = math.degrees(math.asin(a))

        a = 2.0 * (w * z + x * y)
        b = 1.0 - 2.0 * (y**2 + z**2)
        yaw = math.degrees(math.atan2(a, b))

        return [pitch,roll,yaw]
```

Timeline

Dates	Work to be done
Community Bonding Phase	
Week-1 April 24 - May 1	● Interaction with the mentors discussing about the meetings. ● Discussion about weekly updates.
Week-2 May 2 - May 9	● Complete setup of scipy(if anything extra is required) and of the blog as per the mentor.
Week-3 May 10 - May 17	● Finalising the design with the mentors. ● Gain more knowledge about Euler angles, DCMs, and quaternions. ● Also focus more on the various conversion algorithms available.
Coding Phase - I	
Week-4 May 18 - May 24	● Rotation class will be initialized with all the required parameters after discussing it with the mentors. ● Rotations will be defined internally as quaternions.
~ Week-5 May 25 - May 31	● Implement initializing methods for the Rotation class. ● Implement **__mul__, rotate** methods for the class. ● Add tests, examples, and docstrings.
Week-6 June 1 - June 7	● Read about conversion algorithms for converting Euler angles to/from quaternion. ● Implement **quat2euler , euler2quat** functions for conversion of quaternion to/from euler. ● Add tests, examples, and docstrings.
~Week-7	● Read about conversion algorithms for converting

June 8 - June 13	DCMs to/from quaternion. • Implement **quat2dcm , dcm2quat** functions for conversion of quaternion to/from DCMs. • Add tests, examples, and docstrings.
June 14 - June 17	• Buffer period • Discuss the project status with the mentor for the first evaluation.
Week 8,9 June 18 - July 1	• Buffer period. • Planned vacation trip.
Coding Phase - II	
~Week 10 July 2 - July 8	• Implement **axis_rotate** function to define rotations about a given axis. • Add tests, examples, and docstrings.
July 9 - July 12	• Buffer period. • Discuss the project status with the mentor for the second evaluation.
Week 11 July 13 - July 19	• Have a thorough reading about the **SLERP** algorithm and uniform random sampling of rotations. • Implement the **slerp and random_sample**. • Add tests, examples, and docstrings.
Week 12 July 20 - July 27	• Implement the **wahba_estimate** to estimate rotation between two quaternions.. • Add tests, examples, and docstrings
Week 13 July 28 - August 3	• Implement **spline** using cubic spline interpolation. • Add tests, examples, and docstrings.
Integration Phase	

~ Week 14 August 4 - August 10	• Refactor the code according to current PEP8 codestyle , final integration of the code. • Compatibility testing with other modules. • Updating docstrings, adding more examples at the integration phase. • Final evaluation of the code with the mentors and code submission.

References

---

1. https://en.wikipedia.org/wiki/Quaternions_and_spatial_rotation
2. https://en.wikipedia.org/wiki/Quaternion#Matrix_representations
3. Understanding Various Rotation Formalisms in 3 dimensions
4. https://en.wikipedia.org/wiki/Slerp#Quaternion_Slerp
5. https://en.wikipedia.org/wiki/Gimbal_lock
6. https://matthew-brett.github.io/transforms3d/
7. http://qspline.sourceforge.net/qspline.pdf