A Python module for quaternions.

Abstract:
In modern navigation systems, Direction-Cosine Matrix is widely used to represent rotation in three dimensions. Because of number round in computer and number operation, DCM costs many time and space to compute a little no-accurate output. In real engineering world, this may cause a serious problem. The quaternions are a number system that extents the complex numbers [1]. Compared to the DCMs, quaternions have many advantages in engineering area. To prove our assumption, a Python class called Quaternion is built to compute quaternion object, a function Python file is written to store some useful functions and a test Python file is wrote to test Quaternion class and compare numeric calculation ability between Quaternion and DCM. By using random initial placement, we got outputs with high precision show that quaternion works better than DCM in many cases.

Python data

A. Decimal and Binary

In our daily life, we usually use numbers like 1, 999 or 1024. Do you ever wander how it works in modern society? It works following a simple rule, whenever it hits 10, it adds 1 to a higher level. For example, if we have a number 19, then we add it with a 1, 9+1 hits 10, so it changes to 0 while adds 1 to 1, so 19 + 1 = 20. This in fact a math class in your primary school, so I don't need to explain it more. But computer is based on digital electronics which only has two status, one is 1, another is 0. So using Binary in computer is a more natural way to itself. Like Decimal, whenever it hits 2, it adds 1 to a higher level. While in Decimal, we only use numbers in

$$[1,2,3,4,5,6,7,8,9,0]$$

We only use numbers in

$$[1,0]$$

in Binary.

Now we have Decimal and Binary in hands. What is their relationship between each other?

If we have a number in Decimal as 12345, then it can be written like:

$$12345 = 1\times10^4 + 2\times10^3 + 3\times10^2 + 4\times10^1 + 5\times10^0$$

If we have a binary number as 100011110000, then it can be written like:

$$(100011110000)_2 = 1\times2^{11}+0\times2^{10}+0\times2^9+0\times2^8+1\times2^7+1\times2^6+1\times2^5+1\times2^4+0\times2^3+0\times2^2+0\times2^1+0\times2^0$$

Sum the right part of equal sign, you can get how it is in Decimal.

From the example above, we notice that if a number is represented totally in Binary, it can be very long till the end of the world. In other choices, we can use Octal or Hexadecimal to represent a Binary number in a shorter length. For Octal, from right to left in a Binary number, three numbers can be represented in one of [0,1,2,3,4,5,6,7]. As for Hexadecimal, we use 4 numbers in a Binary number from right to left. It's easy to understand for 8 is three times of 2 while 16 is four times of 2.

Now we know how to convert a Binary number to Decimal and represent Binary number in Octal and Hexadecimal. Let's talk about convert a Decimal number to Binary.

Let's say we have a Decimal number A, if we want to have it in Binary, we take A as numerator and 2 as divisor. Every time we got a quotient and a reminder less than 2. For example, if we want 13 be in Binary number, then:

$$13 \div 2 = 6...1_0$$
$$6 \div 2 = 3...0_1$$
$$3 \div 2 = 1...1_2$$
$$1 \div 2 = 0...1_3$$

So 13 in Binary is 1101.

B. Data type

Our quaternion module is based on Python 3.7. It has basic data types, like integer etc. As one of dynamically-typed languages, we can use one data type without declare it and can even change it data type without a compilation error like in C language. We first introduce integers, then introduce floats, we will talk about long at last.

Integers

In computer science, they are called ***int*** or ***integer***. They can be positive or negative or equal to 0. But they cannot have a decimal part, which means they can have part only before the dot.

In computer, an int number can use 8, 16 or 32 bits to represent a number. No matter how many bits to represent an int number, the ways to represent it are same. They are *sign and magnitude*, *1's complement* and *2's complement*.

Sign and magnitude

If we use 8 bits to represent a integer number, we define the MSB (most significant number) as sign bit and use rest 7 bits to represent its absolute value. If the integer is less than 0, the MSB is 1. If the integer is more than 0, the MSB is 0.

+1 can be represented as 00000001, -1 can be represented as 10000001.

From here, you can tell a problem of this way, the 0 can be represent as 10000000 or 00000000. And in fact, the rules for addition between two integers are complex to follow as well.

1's complement

The rule of 1's complement is little complex. In short, if an integer is less than 0, it would be represented by using 1's complement, if the MSB is full and need have a carry, the carry needs to be added to the LSB (least significant bit).

Here is a example:

$$00001000 - 00000010 \rightarrow 00001000 + (-00000010) \rightarrow 00001000 + 11111101 \rightarrow 00000101$$

MSB add 1 to LSB, then equals:

$$00000110$$

In this way, the rules for addition between two integers are simple, but 0 can be represent as 11111111 or 00000000, which is still not unique.

2's complement

Based on the rule of 1's complement, 2's complement is formed to solve unique 0 problem. If an integer is more than 0, it should not be changed. If an integer is less than 0, using 1's complement and add 1.

In this way, our 0 can only be represent as 00000000.

Floating-point numbers

In computer science, they are called floats. They can be positive or negative or equal to 0.0 with a decimal part.

In computer, float is using scientific notation to present itself. Assuming we have a number in decimal, for example 12345, using scientific notation to present. Then it becomes:

$$1.2345 \times 10^4$$

We call 1.2345 significand while call 4 as exponent. But we know that every number is stored in computer using binary, we need to present a number with radix 2.

Floating-point numbers have a disadvantage. We know that computer uses binary to store numbers. Unfortunately, most decimal fractions cannot be represented exactly as binary fractions [3].

The number as float is approximated by binary number in computer storage. That is the reason why people can not use floats to make comparation.

C. Data structure

List

List is a very useful data structure in Python. Its elements are put into [] and separated by ",".
Unlike in other languages, the elements in a list can be different types:

$$A = [1,2,3,4,5]$$
$$B = ['a','b','c']$$
$$C = ["aa",1,"haha"]$$

In the list A, elements are all integers. In list B, elements are strings, or to be more specific, are characters. In list C, elements are of different types, they are strings or integers.
In the list, each element has a index. In Python, unlike in MATLAB, the index begins from 0.

$$A = [7,77,777,7777,77777]$$

index	0	1	2	3	4
element	7	77	777	7777	77777

Look above, assuming we have a list A with five elements in this order. If you want to get an element with value 7777, you can use A[3] to get it.
There are some operations and functions can be used on list. List structure has some useful methods. We will talk some of them in details below.

A. List operations

+ operation

+ operation is a kind of append. Assuming we have two lists, one is [1, 3, 4], another is $['a','b','c']$, if we do '+ operation' between, it works like:

$$[1,3,4]+['a','b','c'] = [1,3,4,'a','b','c']$$

** operation*

Do a '* operation' within a list is a kind of repeat list. The operation works like that:

$$['ha']*4 = ['ha','ha','ha','ha']$$

B. List functions

In Python, it has some functions can be used on list. Now assuming we have a list A = [1, 2, 3, 9, 0].

len(list)

Using *len(list)*, it would return the length of the list object. In the list A, there are 5 elements. The result should be 5 using *len(list)* function.

$$len(A) = 5$$

max(list)

Using *max(list)*, it would return the maximum element of the list object. In the list A, the maximum element is 9. The result should be 9 using *max(list)* function.

$$\max(A) = 9$$

People may want to say if we have a list with strings, what should be the result of *max(list)*? The rule is simple, it returns the longest length string in the list.
For example, if we have a list like

$$['C','Python','Java']$$

then the result should be *'Python'*.

min(list)

Using *min(list)*, it would return the minimum element of the list object. In the list A, the minimum element is 0. The result should be 0 using *min(list)* function.

$$\min(A) = 0$$

Like in the max(list), if we have a list with strings, the return result should be *'C'*, which has the shortest length.

C. List method

List is a very important data structure in Python. It has some useful methods which can do operations on the list itself or on the element in the list.

list.append(obj)

This method is used to add element into the list. But it has one specific property, it adds the element to the last position of the list. Assuming we have a list like:

$$A = ['C', 'Python', 1, 'Java', 2]$$

We use this method like:

$$A.append('haha')$$

Then we get a list like:

$$A = ['C', 'Python', 1, 'Java', 2, 'haha']$$

From what we did above we can tell, no matter what type of the element is, the element is always added to the last position of the list. This method has no return value but it changes the list.

list.count(obj)

In the Python list, each element has its unique index. If a Python list has two elements with same value, it is accepted because each element has their own index which can be used to separate them from each other. This method is used to count how many times an element shows in the list. Assuming we have a list like:

$$A = ['C', 'haha', 'Python', 1, 'Java', 2, 'haha']$$

we use this method like:

$$A.count('haha')$$

$$A.count('C')$$

Then the first return value should be 2 and the second return value should be 1. And what if we want to find an element that is not in this list, like:

$$A.count(3)$$

the return value should be 0.

list.index(obj)

If we want to know an element's index, we can use this method. *list.index(obj)* can return the index of an element. Assuming we have a list like:

$$A = ['a', 2, 4, 'b']$$

we use method like:

$$A.index(2)$$

Because the Python list begins the index at 0, then the return value would be 1. What if we want to know an index of an element that is not in the list? The return would be a ValueError like:

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: 'd' is not in list
```

list.pop(index)

If we want to delete an element from the list and we already know its index, we can use this method to delete the element. This method will return the element with the index from the original list. Assuming we have a list like

$$A = ['a', 2, 4, 'b']$$

we use method like:

$$A.pop(2)$$

then the method will return the deleted elment like:

$$4$$

list.remove(obj)

If we want to remove an element from the list and we already know what the element is, we can use this method to delete the element. This method will return nothing but change the list it works on.
Assuming we have a list like:

$$A = ['a', 2, 'b', 9]$$

If we want to remove an element like an element 'a', we use method like:

$$A.remove('a')$$

Then the list would be:

$$A = [2, 'b', 9]$$

A special of this method is that if we have some elements of the same value, this method will only delete the first found element, for example, if we have a list like:

$$A = [2, 3, 4, 5, 2, 0, 2]$$

we use method like:

$$A.remove(2)$$

we will get a list like:

$$A = ['a', 2, 4, 'b']$$

The first found element means according to the index order from 0 to 6, then the first found element would be the element with smallest index.

list.reverse()

If we want to have a list with reversed index element, we can use this method. The method doesn't have an input and an output, but it will change the order of elements reversely.
For example, if we have a list like:

$$A = ['a', 2, 4, 'b']$$

we use the method like:

$$A.reverse()$$

we will get a list like:

$$A = ['b', 4, 2, 'a']$$

Array

In our codes, DCM is wrapped by the NumPy array. We will explain the Module called NumPy in Module part. Here we will introduce a part of NumPy called array.
Array definition is almost as the same of a list above. A NumPy array is a N dimensional array object, which is consist of two parts, one is the real data, another is the meta data that is used to describe the real data.
Same as the list structure in Python, the index begins at 0. But there is a difference between list and an array. In a NumPy array, every element must be of the same data type.
As for dimensions of an array, things are different from 1D-array and N-Dimension array. For 1D-array, like *numpy.array([1,2,3,4])*, usually we don't indicate its name of axis. For 2D-array, we indicate rows as '0-axis' and columns as '1-axis'. With dimensions increasing, our arrays can have axis of number 2, 3, 4 etc.
Before we talked about details of arrays, assuming we have initialized three arrays, A is a 1D array, B is a 2D array, C is a 3D array.
They are as below:

```
A = numpy.array([1,2,3,4])
B = numpy.array([[1,2,3,4],[5,6,7,8]])
C = numpy.array([[1,2,3,4],[5,6,7,8],[9,10,11,12]])
```

A. attribute of array

array.shape

By using shape on array object, it will return the shape of the array. For example, an array is a m*n array, the attribute will return (m,n).

```
>>> A.shape
(4,)
>>> B.shape
(2, 4)
>>> C.shape
(3, 4)
```

array.ndim

By using ndim on array object, it will return the dimension of an array.

```
>>> A.ndim
1
>>> B.ndim
2
>>> C.ndim
2
```

array.size

By refer to the size attribute, it will return the number of elements in the array. For instance, if we have a m rows n columns array, then the attribute would be m*n.

```
>>> A.size
4
>>> B.size
8
>>> C.size
12
```

B. slicing and indexing of array

slicing

To understanding the slicing, we can easily take it as a cut of entire array. To slice from a 1D-array, we can use

```
>>> A[2:3]
array([3])
```

Because the array index begins from 0, the 2 indicates third element in the array, which is 3 in our case. Another issue to be emphasized is the slicing includes left boundary but does not take right boundary into consideration. That's why our codes above only have one element in return.

To slice from a ND-array, first slicing indicates the first axis slicing, if you don't give more slicing in it, the return will have all elements in the first axis slicing.

```
>>> B[0:1]
array([1, 2, 3, 4])
```

```
>>> B[0:1,2:3]
array([[3]])
```

Looking at codes above, the B[0:1] indicates first row of B, without give more slicing in second axis, it has all elements in first row in return. While the second line gives slicing in second axis, it means the third element in first row, which is 3 in our case.

You may have a question in mind. From your point, in second line of slicing, the return should be first row to second row and third element to forth element. That is the normal question in even experienced programmer. Please keep in mind that our slicing always includes left hand boundary but usually not the right hand boundary.

indexing

Compare to slicing, indexing is much more straight forward. If you want second row third element, just minus 1 in each axis and you will get what you want in mind. Like slicing, if you offer less dimension number than the real dimensions of array, it will return all elements in that dimension.

C. reshape of an array

By using reshape method of a numpy array, it will return the reshaped array.

```
>>> A = numpy.array([1,2,3,4,5,6,7,8])
>>> A.reshape(2,4)
array([[1, 2, 3, 4],
       [5, 6, 7, 8]])
```

D. broadcasting

It requires arrays of same shape that can be calculated. If you use two arrays with different shape, the output may surprise you for the rule of broadcasting.

For example, if we have an array A of shape (1,5), added by an array B of shape (6,1), the broadcasting works like below.

```
>>> A = numpy.array([1,2,3,4,5])
>>> B = numpy.array([[0],[1],[2],[3],[4],[5]])
>>> A+B
array([[ 1,  2,  3,  4,  5],
       [ 2,  3,  4,  5,  6],
       [ 3,  4,  5,  6,  7],
       [ 4,  5,  6,  7,  8],
       [ 5,  6,  7,  8,  9],
       [ 6,  7,  8,  9, 10]])
```

The outcome is an array of shape (6,5). Look close to the outcome, you'll find that the first row of outcome equals the first element adds each element in the row of A, the second row of outcome is the second element adds each element in the row of A. Now you may have the idea of how the broadcasting works in arrays calculation.

Module

Math

It is a module made for mathematical functions. It has six parts, one part for number-theoretic and representation functions, one for power and logarithmic functions, one for trigonometric functions, one for angular conversion, one for hyperbolic functions, one for special functions and the last is for constants.

For math module used in our Quaternion module, we mainly use the trigonometric functions part to produce our rotation quaternion and square function in power and logarithmic part to get some square root.

NumPy

It is a functional package made for scientific computing. In quaternion implement, it is used as a short term np to rearrange Direction-Cosine matrix into an array object.

We have already introduce some attributes and methods in NumPy array part. We will introduce some data type in NumPy. For NumPy captures mainly in numerical part, it supports more data types than the primitive data type in Python.

Data type in NumPy []

data type	Desciption
np.bool	Boolean (True or False) stored as a byte
np.byte	Platform-defined
np.int8	Byte (-128 to 127)
np.int16	Integer (-32768 to 32767)
np.int32	Integer (-2147483648 to 2147483647)
np.int64	Integer (-9223372036854775808 to 9223372036854775807)
np.uint8	Unsigned integer (0 to 255)
np.uint16	Unsigned integer (0 to 65535)
np.uint32	Unsigned integer (0 to 4294967295)
np.uint64	Unsigned integer (0 to 18446744073709551615)

random

For scientific work, it may need some random number to test you codes, so does our work here. Usually we use *random.random()* method to produce a float random number between 0 to 1, while the number can equal to zero but it will never get to 1. To produce a float in a specific range, we can use method like *random.uniform(a,b)*, which will produce a random number bigger or equal to the smaller one in a and b and smaller or equal to the bigger one in a and b. Maybe you want a integer number rather than float number? Then the *random.randint(a,b)* will meet your requirement. If you only want odds or evens, the *random.randrange(a,b,step)* can produce a random chose from a list between a and b (including a not b) with step.

No matter what type of random number you want, you can use methods in random module to get it. If the module does not meet your requirement, you can combine some method to get what you want. It is a really useful module in scientific field.

Object-oriented programming

Object-oriented programming is a programming paradigm based on the concept of "objects", which may contain data, in the form of fields, often known as attributes, in the form of procedures, often known as methods [3]. By using the thinking of OOP, a Quaternion class is made to represent quaternion. Assuming a quaternion is in form of

$$a + b\,\underset{\sim}{i} + c\,\underset{\sim}{j} + d\,\underset{\sim}{k}$$

then its attributes are real part a, imagery part b, c and d. For quaternion methods, they would be explained in Quaternion part.

In general, there are many advantages to use OOP in your programming.

1. It hides many things that user do not need to know. For example, user do not need to know the details how a method works. User only cares about what is this method doing and its output/input.
2. Easy to extent. If someone want a new function of this class, programmer do not need to rewrite whole class but to add what user needs.
3. Reuse. While it doesn't show the inheritance in this Quaternion codes, it is still a really important advantage of OOP.
   Assuming we make a People class. As a human, people have many similarities like they both have hair or sights in most cases. After hard work, we finally get a People class with hair() and sights() methods. Then our boss said, hey, we need another class of Children and third class

Old. Both of them are people, so it can inherit hair() and sights() methods without write same methods again. This advantage will cut codes much shorter and easier to understand.
To make each subclass unique, every subclass can add their own method, like in Children() class may have a method called drink_milk() which may not show up in Old() class, and Old() may have a method called experiences() which may not useful in Children() class.

4. Codes will be easy to read and easy to rewrite.

Once again, OOP has many advantages which make it one of the most important thinking in programming. And in our Quaternion class, some advantages will be shown, some would not be shown. Please remember that codes can be reused, so it would be clear if people decide to extent this class or write a class inherit from Quaternion class.

Quaternion

Theory part

Quaternion

DCM

Implementation part

In our implementation of quaternion, there are 12 basic methods for quaternion. They would be explained one by one as below.

.__init__(self, a, b, c, d)：

If a quaternion object can be represented as below:

$$q = a+b\underset{\sim}{i}+c\underset{\sim}{j}+d\underset{\sim}{k}$$

To initialize a Quaternion object, 4 numbers are required to represent the real part a, the imagery part parameters of b, c and d. In computer science, it is important to protect data itself. A method called encapsulation is invented to protect data and hide details of designed class. To encapsulize the data, 4 arguments are given to the object attributes. The basic formula of this method as below:

$$self.a = a\ ;$$

If user want to use some encapsulated data, the designed class need to have some methods like ***setter()*** or getter(). By their names, user can set data using ***setter()*** methods and get data using getter() methods. In this thesis Quaternion class, we have no need for specific single parameter of a Quaternion object, so the author didn't implement these normal methods in the class. If user find they are useful in future work, this method can be added without any correction to other codes. Thanks to encapsulation thinking, we could do this work without any pain.

.__add__(self, other)

Addition is a normal operation in Quaternion objects. Assuming we have two Quaternion objects A and B, their sum is C. All of them are Quaternion objects. If we have them in:

$$A = a_1 + a_2\underset{\sim}{i} + a_3\underset{\sim}{j} + a_4\underset{\sim}{k}$$
$$B = b_1 + b_2\underset{\sim}{i} + b_3\underset{\sim}{j} + b_4\underset{\sim}{k}$$
$$C = c_1 + c_2\underset{\sim}{i} + c_3\underset{\sim}{j} + c_4\underset{\sim}{k}$$

then the addition works like:

$$c_1 = a_1 + b_1$$
$$c_2 = a_2 + b_2$$
$$c_3 = a_3 + b_3$$
$$c_4 = a_4 + b_4$$

In this rule, we can conclude easily that *C* can be written as:

$$C = (a_1 + b_1) + (a_2 + b_2)\underset{\sim}{i} + (a_3 + b_3)\underset{\sim}{j} + (a_4 + b_4)\underset{\sim}{k}$$

Addition rule between two Quaternion objects is clear, then we meet a difficulty to translate the rule into programming language, in our case, in Python.
It has two ways to implement addition in Quaternion. One is to create a method and used as a dot method like other methods. In this way, once we want to add two Quaternion objects together, we need to initial a Quaternion object first, then initial another Quaternion object. Let's called the first one A, then the second one B. we can do addition like this:

$$A.addition(B)$$

Above is quite normal in OOP, but what if the user wants to add two objects like what we do when we have two natural number. In fact, another way makes it possible. In this way, user with little programming experiences can use it without any pain and it looks like this:

$$A + B;$$

Some people with programming experiences may think it as *overloading*. As Python is a dynamic language, it doesn't support *overloading* as many other languages do. In Python, they are called methods, they are *magic methods* [4]. Long in short, magic method let it possible to do addition like above in a more human readable way.

.__sub__(self, other)
Subtraction is almost the same as addition. It can be seen as a Quaternion object add a negative Quaternion object.
Assuming we have two Quaternion objects A and B, their sum is C. All of them are Quaternion objects. If we have them in:

$$A = a_1 + a_2\underset{\sim}{i} + a_3\underset{\sim}{j} + a_4\underset{\sim}{k}$$
$$B = b_1 + b_2\underset{\sim}{i} + b_3\underset{\sim}{j} + b_4\underset{\sim}{k}$$
$$C = c_1 + c_2\underset{\sim}{i} + c_3\underset{\sim}{j} + c_4\underset{\sim}{k}$$

then the subtraction works like:

$$c_1 = a_1 - b_1$$
$$c_2 = a_2 - b_2$$
$$c_3 = a_3 - b_3$$
$$c_4 = a_4 - b_4$$

In this rule, we can conclude easily that *C* can be written as:

$$C = (a_1 - b_1) + (a_2 - b_2)\underset{\sim}{i} + (a_3 - b_3)\underset{\sim}{j} + (a_4 - b_4)\underset{\sim}{k}$$

In Quaternion class, it implemented as a magic method too. User can use it as below:

$$A - B;$$

.__mul__(self, other)

The multiplication of two Quaternion objects is not as easy as above. Before we dive into how to multiply two quaternions, we need to know how to compute multiplication product between each element.

As said before, the quaternion can be seen as an extent to complex number. So in some area, they have same rules.

A number multiply a imagery part follows the rule of scalar product between number and imagery part of complex number:

$$2 \bullet \underset{\sim}{i} = 2\underset{\sim}{i}$$
$$3 \bullet \underset{\sim}{j} = 3\underset{\sim}{j}$$
$$4 \bullet \underset{\sim}{k} = 4\underset{\sim}{k}$$

The rule applies to products between same imagery part:

$$\underset{\sim}{i}^2 = \underset{\sim}{j}^2 = \underset{\sim}{k}^2$$

The basis elements I, j, and k commute with the real quaternion 1, that is

$$1 \bullet \underset{\sim}{i} = \underset{\sim}{i} \bullet 1 = \underset{\sim}{i}$$
$$1 \bullet \underset{\sim}{j} = \underset{\sim}{j} \bullet 1 = \underset{\sim}{j}$$
$$1 \bullet \underset{\sim}{k} = \underset{\sim}{k} \bullet 1 = \underset{\sim}{k} \quad [5]$$

Then come to the different part:

$$\underset{\sim}{i} \bullet \underset{\sim}{j} = \underset{\sim}{k}, \underset{\sim}{j} \bullet \underset{\sim}{i} = -\underset{\sim}{k}$$
$$\underset{\sim}{k} \bullet \underset{\sim}{i} = \underset{\sim}{j}, \underset{\sim}{i} \bullet \underset{\sim}{k} = -\underset{\sim}{j}$$
$$\underset{\sim}{j} \bullet \underset{\sim}{k} = \underset{\sim}{i}, \underset{\sim}{k} \bullet \underset{\sim}{j} = -\underset{\sim}{i}$$

Using the rules above, we can draw a multiplication table for quaternion:

x	1	i	j	k
1	1	i	j	k
i	i	-1	k	-j
j	j	-k	-1	i
k	k	j	-i	-1

In our Quaternion class, we compute __mul__() method as a Hamilton product. Assuming we have two Quaternion objects, one is called A, another is called B. Assuming A is in form of

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k}$$

B is in form of

$$B = a_2 + b_2\underset{\sim}{i} + c_2\underset{\sim}{j} + d_2\underset{\sim}{k}$$

The Hamilton product is determined by the product of the basis elements and the distributive law[6]. Then the Hamilton Product as below:

$$\begin{aligned}
&a_1a_2 + a_1b_2\underset{\sim}{i} + a_1c_2\underset{\sim}{j} + a_1d_2\underset{\sim}{k} + \\
&b_1a_2\underset{\sim}{i} + b_1b_2\underset{\sim}{i}^2 + b_1c_2\underset{\sim}{ij} + b_1d_2\underset{\sim}{ik} + \\
&c_1a_2\underset{\sim}{j} + c_1b_2\underset{\sim}{ji} + c_1c_2\underset{\sim}{j}^2 + c_1d_2\underset{\sim}{jk} + \\
&d_1a_2\underset{\sim}{k} + d_1b_2\underset{\sim}{ki} + d_1c_2\underset{\sim}{kj} + d_1d_2\underset{\sim}{k}^2
\end{aligned}$$

By using the rules in multiplication table, it can be rewritten as:

$$\begin{aligned}
&a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 + \\
&(a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\underset{\sim}{i} + \\
&(c_1a_2 - b_1d_2 + a_1c_2 + d_1b_2)\underset{\sim}{j} + \\
&(a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)\underset{\sim}{k}
\end{aligned}$$

But in our Quaternion class, we use Quaternion objects in angle rotations. In case like this, we can see it as rotation B followed by the rotation A, that is rotation

$$B = a_2 + b_2\underset{\sim}{i} + c_2\underset{\sim}{j} + d_2\underset{\sim}{k}$$

followed by rotation

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k}$$

.scalar_mul(self, scalar)

To scale a Quaternion object, the rule is a little different from normal multiplication between two Quaternion objects. In the class, method scalar_mul() handles this.

Assuming we have a Quaternion object called A and a scalar number B. A is in the form of

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k}$$

the scalar product can be computed as below:
scalar part :

$$a_1 B$$

vector part:

$$(b_1 \underset{\sim}{i} + c_1 \underset{\sim}{j} + d_1 \underset{\sim}{k}) \times B = b_1 B \underset{\sim}{i} + c_1 B \underset{\sim}{j} + d_1 B \underset{\sim}{k}$$

Put them together:

$$A.scalar\_mul(B) = a_1 B + b_1 B \underset{\sim}{i} + c_1 B \underset{\sim}{j} + d_1 B \underset{\sim}{k}$$

For scalar multiplication, it can be seen as each part of Quaternion object A is scaled B times.
if 0 < B < 1, then A becomes smaller; if B = 1, A is of the same as the original A; if B > 1, then A becomes bigger.

.dot(self, other)

From what we did above in Quaternion multiplication, we know that it's not easy to compute Hamilton product between two Quaternion objects. Like any other mathematical class, there is a operation called dot product between two Quaternion object and its rule is straight forward. To compute the dot production of two Quaternion objects, a dot() method is used. A multiplication between two objects is a Hamilton product while a dot product means each element multiply with same corresponding element in another quaternion object.
Assuming we have two Quaternion objects, one is called A, another is called B. Let A in form of

$$A = a_1 + b_1 \underset{\sim}{i} + c_1 \underset{\sim}{j} + d_1 \underset{\sim}{k}$$

and B in form of

$$B = a_2 + b_2 \underset{\sim}{i} + c_2 \underset{\sim}{j} + d_2 \underset{\sim}{k}$$

then the dot product rule is like:

$$A \bullet B$$
$$= (a_1 + b_1 \underset{\sim}{i} + c_1 \underset{\sim}{j} + d_1 \underset{\sim}{k}) \bullet (a_2 + b_2 \underset{\sim}{i} + c_2 \underset{\sim}{j} + d_2 \underset{\sim}{k})$$
$$= a_1 a_2 + b_1 b_2 \underset{\sim}{i} + c_1 c_2 \underset{\sim}{j} + d_1 d_2 \underset{\sim}{k}$$

.norm(self)

To understand norm of a Quaternion object, let's explain a simple instance in complex number. As we could treat quaternion as extent to complex number, the rule applies in complex number can be used in quaternion field as well. In the complex number system, assuming we have a complex number

$$a = b + c \underset{\sim}{i}$$

then we can get its norm following this rule

$$|a| = \sqrt{b^2 + c^2}$$

One way to understand the norm in complex number is to think about $b + c\boldsymbol{i}$ as a point (b, c) in complex plain. Then the square root of $b^2 + c^2$ is in fact nothing more than the length from the origin to the point (b, c).

Thinking in the same way, we can take a quaternion $A = a_1 + b_1 \underset{\sim}{i} + c_1 \underset{\sim}{j} + d_1 \underset{\sim}{k}$ as in four dimensions, one axis is for the scalar part, other three axis are for the complex part. It's a hard work to explain 4D one a 2D plain frame. At this step, we need some imagination. Assuming we have a 4D frame, one is scalar axis, one is for i-axis, one is for j-axis, the last one is for k-axis. Refer to what we talked above in

complex number, we can take the norm of a quaternion as the length from origin to the point (a_1, b_1, c_1, d_1). the norm of Quaternion object A should be:

$$|A| = \sqrt{a_1^{\,2} + b_1^{\,2} + c_1^{\,2} + d_1^{\,2}}$$

Another way to think about norm of complex number, is to think it in a more geometric perspective. Assuming we have a complex number like $a = b + c\underset{\sim}{i}$, then its conjugate is $\overline{a} = b - c\underset{\sim}{i}$. We can draw a norm formula like this $a \times \overline{a} = \sqrt{(b + c\underset{\sim}{i}) \times (b - c\underset{\sim}{i})}$. The final output is the same, but the way of thinking it is different. We can rewrite it as matrix multiplication like:

$$\begin{bmatrix} b & c \\ b & -c \end{bmatrix} \begin{bmatrix} 1 \\ i \end{bmatrix} = \begin{bmatrix} b + ci \\ b - ci \end{bmatrix}$$

As it in number theory, the norm is the square of determinant of above matrix. for any two quaternions *p* and *q, m*ultiplicativity is a consequence of the formula for the conjugate of a product [7]. Then it follows the rule by using the determinant like:

$$det \begin{bmatrix} a + ib & id + c \\ id - c & a - ib \end{bmatrix}$$

after calculate the determinant of above, we could know that $|A| = \sqrt{a_1^{\,2} + b_1^{\,2} + c_1^{\,2} + d_1^{\,2}}$.

.norm_q(self)

This method is to compute unit quaternion object. In its definition, the unit quaternion is the quaternion of norm 1. In other word, if we have a Quaternion object like

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k},$$

then $a_1 \quad + b_1 \quad + c_1 \quad + d_1 \quad = 1$. User may treat it as a useless method at first, but as we dive deeper into rotation test, we need to normalize the output quaternion object each time to make sure we are in the way of same scale.

The definition is not complex, for each element of a Quaternion object, divide them by norm of this Quaternion object, then we are done.

$$|A| = \sqrt{a_1^{\,2} + b_1^{\,2} + c_1^{\,2} + d_1^{\,2}}$$

$$A.norm\_q() = \frac{a_1}{|A|} + \frac{b_1}{|A|}\underset{\sim}{i} + \frac{c_1}{|A|}\underset{\sim}{j} + \frac{d_1}{|A|}\underset{\sim}{k}$$

The tricky is, we calculate norm of this object in the method rather than calculate four times in return operation. It may have little effect when we get small number normalizations, but when we test for million times, it makes huge difference. There are some other tricky in computing this class and author will explain it when we come to the part.

.conj(self)

Conjugation of quaternions is analogous to conjugation of complex numbers and to transposition of elements of Clifford algebras [8]. If a quaternion object can be represented as

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k}$$

then the conjugation of this quaternion object is

$$\overline{A} = a_1 - b_1\underset{\sim}{i} - c_1\underset{\sim}{j} - d_1\underset{\sim}{k}$$

We have already talked about some conjugation definition in $.norm(self)$ method part, now in this part we first talked about complex conjugation then we will talk about quaternion conjugation. Please keep in mind, we can treat quaternion as an extent in 4D space of complex number, so many attributes are in the same form but of different dimensions.

Conjugation of complex number

In mathematics, the complex conjugate of a complex number is the number with an equal real part and an imaginary part equal in magnitude but opposite in sign [9].

Complex conjugation has some useful properties as follow:

Assuming we have two complex numbers, z and w.

The conjugation of sum of them equals conjugation of z add conjugation of w.

$$\overline{z + w} = \overline{z} + \overline{w};$$

The conjugation of subtraction of them equals conjugation of z minus conjugation of w.

$$\overline{z - w} = \overline{z} - \overline{w};$$

The conjugation of multiplication of them equals conjugation of z multiply conjugation of w.

$$\overline{zw} = \overline{z}\,\overline{w};$$

The norm of conjugation z equals to the norm of conjugation.

$$(\overline{z}) = (z);$$

We have many other properties in complex conjugation. But our aim here is to see how similar a complex number and a quaternion alike. Now we come to the quaternion part.

Assuming we have two Quaternion objects, one is A in form of

$$A = a_1 + b_1\underset{\sim}{i} + c_1\underset{\sim}{j} + d_1\underset{\sim}{k}$$

another is B in form of

$$B = a_2 + b_2\underset{\sim}{i} + c_2\underset{\sim}{j} + d_2\underset{\sim}{k}$$

According to the conjugation rule of quaternion, the conjugation of A is:

$$\overline{A} = a_1 - b_1\underset{\sim}{i} - c_1\underset{\sim}{j} - d_1\underset{\sim}{k}$$

the conjugation of B is:

$$\overline{B} = a_2 - b_2\underset{\sim}{i} - c_2\underset{\sim}{j} - d_2\underset{\sim}{k}$$

the conjugation of sum of A and B is:

$$\overline{A+B} = \overline{a_1 + b_1 i + c_1 j + d_1 k + a_2 + b_2 i + c_2 j + d_2 k}$$

$$= \overline{(a_1 + a_2) + (b_1 + b_2)i + (c_1 + c_2)j + (d_1 + d_2)k}$$

$$= (a_1 + a_2) - (b_1 + b_2)i - (c_1 + c_2)j - (d_1 + d_2)k$$

$$= (a_1 - b_1 i - c_1 j - d_1 k) + (a_2 - b_2 i - c_2 j - d_2 k)$$

$$= \overline{A} + \overline{B};$$

the conjugation of subtraction of A and B is:

$$\overline{A-B} = \overline{(a_1 + b_1 i + c_1 j + d_1 k) - (a_2 + b_2 i + c_2 j + d_2 k)}$$

$$= \overline{(a_1 - a_2) + (b_1 - b_2)i + (c_1 - c_2)j + (d_1 - d_2)k}$$

$$= (a_1 - a_2) - (b_1 - b_2)i - (c_1 - c_2)j - (d_1 - d_2)k$$

$$= (a_1 - b_1 i - c_1 j - d_1 k) - (a_2 - b_2 i - c_2 j - d_2 k)$$

$$= \overline{A} - \overline{B};$$

the conjugation of multiplication of A and B is:

$$\overline{AB}$$

$$= \overline{(a_1 + b_1 i + c_1 j + d_1 k) * (a_2 + b_2 i + c_2 j + d_2 k)}$$

$$= \overline{(a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2) + (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2)i}$$

$$\overline{+(a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2)j + (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2)k}$$

$$= (a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2) - (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2)i$$

$$-(a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2)j - (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2)k$$

while multiplication of conjugation A and conjugation B is:

$$\overline{A} * \overline{B}$$

$$= \overline{a_1 + b_1 i + c_1 j + d_1 k} * \overline{a_2 + b_2 i + c_2 j + d_2 k}$$

$$= (a_1 - b_1 i - c_1 j - d_1 k) * (a_2 - b_2 i - c_2 j - d_2 k)$$

$$= a_1 a_2 - b_1 b_2 - c_1 c_2 - d_1 d_2 - (a_1 b_2 + b_1 a_2 + c_1 d_2 - d_1 c_2)i$$

$$-(a_1 c_2 - b_1 d_2 + c_1 a_2 + d_1 b_2)j - (a_1 d_2 + b_1 c_2 - c_1 b_2 + d_1 a_2)k$$

$$= \overline{AB}$$

The norm of A is:

$$(A) = \sqrt{a_1^2 + b_1^2 + c_1^2 + d_1^2}$$

The norm of conjugation of A is:

$$(\overline{A}) = \sqrt{(a_2^2 + (-b_2)^2 + (-c_2)^2 + (-d_2)^2)} = \sqrt{(a_2^2 + b_2^2 + c_2^2 + d_2^2)}$$

From above, we could see that in quaternion field, the rules can be used as well.

.rotator(self, theta, vectors)

The aim of rotator() method is to rotate the Quaternion object for **theta** angle around the **vectors**. From what we say above, we could see that this function has two steps, the first is to compute the needed rotator, the second is to calculate the rotated Quaternion object.

But less is more [10]. In the rule of function design, usually only one thing can be done in a single function. According to this rule, in usual way, we need to write an independent function to build he rotator, then use it in a Quaternion object rotation. There are two advantages using this way, one is the strengthen of codes reusable, another is to make each function has its specific aim which would make others easy to understand.

Author chose another way to build this function for the reason we only care about rotated Quaternion object in this thesis. Every time we compute a Quaternion object to rotate another Quaternion object, we used it immediately. In other words, we only care about aimed Quaternion object, so it can be written like this.

.toDCM(self)

Both quaternion and DCM can represent rotations. But what are their differences?

In quaternion, the advantages are:

1. The advantage of Quaternion over DCM in computer computing, is in matrix representing of step by step rotation the floats is rounded every time and would cause a huge difference from real output.
2. For quaternion structure, it operates less multiplication than DCM.
3. Easy to compute a normalized one
4. Represent pure rotation
5. Avoid a problem called gimbal lock.

The disadvantages are:

1. hard to understand
2. sometimes you had to do normalization to get a pure rotation.

As for DCM, advantages are:

1. widely used and easy to understand
2. can be split into three simpler constitutive rotations.

Disadvantages are:

1. because it's depended on axis about which rotation made and their sequence, the sequence of multiply is important and can be misused easily.
2. The inverse to get rotation angles is not unique

3. may meet some problem called gimbal lock.

4. It has six constraints. For using three parameters we can describe a rotation, while DCM use 9 parameters.

Sometimes you cannot compare which one is better than another one only by words, author write a series of test codes to test both DCM and Quaternion. These tests would be explained later in this thesis.

Now let's focus on the method of ***.toDCM()***.

Assuming we have a Quaternion object like

$$A = q_1 + q_2\underset{\sim}{i} + q_3\underset{\sim}{j} + q_4\underset{\sim}{k}$$

and our aimed DCM is in form of

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} \\ C_{21} & C_{22} & C_{23} \\ C_{31} & C_{32} & C_{33} \end{bmatrix}$$

Then we would compute each element as below

$$\begin{aligned} C_{11} &= {q_0}^2 + {q_1}^2 - {q_2}^2 - {q_3}^2 \\ C_{12} &= 2\times(q_1q_2 + q_0q_3) \\ C_{13} &= 2\times(q_1q_3 - q_0q_2) \\ C_{21} &= 2\times(q_1q_2 - q_0q_3) \\ C_{22} &= {q_0}^2 - {q_1}^2 + {q_2}^2 - {q_3}^2 \\ C_{23} &= 2\times(q_3q_2 + q_0q_1) \\ C_{31} &= 2\times(q_1q_3 + q_0q_2) \\ C_{32} &= 2\times(q_2q_3 - q_0q_1) \\ C_{33} &= {q_0}^2 - {q_1}^2 - {q_2}^2 + {q_3}^2 \end{aligned}$$

Then we need to compute norm of each row, let each element divided by row norm, group them in a array. The array is what we need for using this method.

.__str__(self)

Now we already have many useful methods in Quaternion class, what if we want to print a Quaternion object? Assuming we have initialized a Quaternion object like

$$A = Quaternion(1,2,3,4)$$

What we want to have it to be printed on screen is:

$$A = 1 + 2\underset{\sim}{i} + 3\underset{\sim}{j} + 4\underset{\sim}{k}$$

But we have in fact on console window is:

```
<Quaternion.Quaternion at 0x145c5002780>
```

That is not we want on screen. In short, this line shows where your Quaternion in computer is which we do not care about. What we care about is what this Quaternion object exactly is, that is the reason why we need this method called __str__().
The __str__() method is used to print user readable output on screen by using a function called *print()*. What if user want the address of this Quaternion object in computer? Then we need console and typed A then enter, the output is the address we need. If users think we don't need the address at all, the class can add a method called __repr__(), by using

$$\_\_repr\_\_() = \_\_str\_\_()$$

In both ways we will get user readable output on screen.

functions

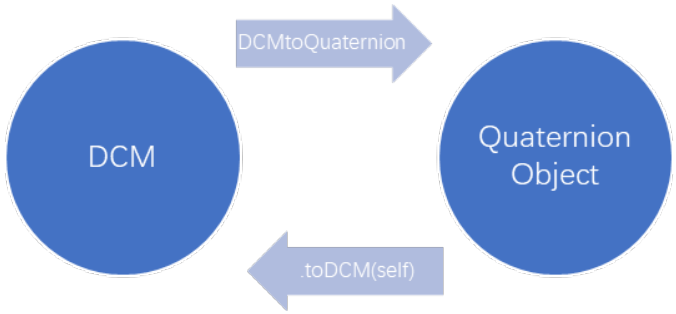
Apart from what we have implemented in Quaternion class, in order to compare our Quaternion class and to build our DCM, author implemented a file of independent functions to do these works. Some of them are related to DCM, some of them are related to Quaternion object, the rest are the test functions for measuring the ability of our Quaternion class and comparing our Quaternion class with DCM. Before we dive into the details of our functions, we will have a look at some rules of function in Python briefly.

function in Python

You may come across to a situation at which we need to achieve same aims for many times. Then you may write same codes here and there by clicking copy and paste. We can see this behavior anywhere but you'd better not to do so in programming field or you'll be labelled as green hand in this world. The meaning of function life is to reduce repeated codes and let your codes have good structure.
We begin to definite functions by using 'def', a short term stands for definition. Followed by parentheses, you need to make your name of function to be meaningful and readable. In the parentheses, you need to give referred names of your parameters. Unlike other languages, you don't need to indicate their type, Python will find them by itself. Adding a colon and press enter, you can add comments or codes in next a few lines.

implementation part

Our implementation part can be divided into three parts. The first part is about building DCM in two different ways and convert it to a Quaternion object. Recall from what we have implemented above in Quaternion class, we write a method for Quaternion object called Quaternion.*toDCM(self)*. By now, we have already built a cycle for DCM and Quaternion transformation.



The second part is a test about a point rotated by Quaternion and DCM. Assuming we rotate the point around the imaginary axis until it gets to the original position, the point cannot be as the exact same as before. By calculating the differences between original point and the rotated ones, we can compare the ability differences between Quaternion and DCM rotation for a single point.

The third part aims to get differences from another prospect. We have got the influences from second part both in DCM and Quaternion. Now we want to know how do the DCM and Quaternion influence our rotation factor in the rotation process. At this stage, we will take care of differences from four aspect. one is the diagonal check. We know that when we rotate around an axis, one of three axis is fixed. The diagonal check can tell us how far the rotator is from the original axis. And the off-diagonal check can tell us how different the rotator is from rotation angles. In our orthonormality check, because of our way build DCM and Quaternion object, the length would always be 1 and we only check its orthogonality. The next check is to calculate mean difference of nine elements. The last check is about angle differences. We drive three angles and compare them with original angles. The output would be the total difference between rotated ones and original ones.

A. building DCM

computeDCM(theta, vectors)

According to Euler's Theorem, two arbitrarily oriented coordinate systems can always be transformed into each other by a single rotation about a well-defined rotation axis, represented by a unit vector []. According to the formula of building a DCM with angle and rotation axis。

$$C=\begin{bmatrix} \cos\phi + f_1^2(1-\cos\phi) & f_1f_2(1-\cos\phi)+f_3\sin\phi & f_1f_3(1-\cos\phi)-f_2\sin\phi \\ f_1f_2(1-\cos\phi)-f_3\sin\phi & \cos\phi + f_2^2(1-\cos\phi) & f_2f_3(1-\cos\phi)+f_1\sin\phi \\ f_1f_3(1-\cos\phi)+f_2\sin\phi & f_2f_3(1-\cos\phi)-f_1\sin\phi & \cos\phi + f_3^2(1-\cos\phi) \end{bmatrix}$$

Users need to define the angle and a unit vector, then the function will give back a numpy array of 3 lists, each list has three elements and in total the array has 9 elements.

computeDCM_angle(alpha, beta, gamma)

If user want to build an array with three angles, like

$$C=C(1,\alpha)\bullet C(2,\beta)\bullet C(3,\gamma)$$

That means we rotate around 3-axis with gamma angle first, then we rotate around 2-axis with beta angle, we rotate around 1-axis with alpha angle at last. You can tell above we multiply these three angle rotations in a reverse way and it finally looks like

$$C=\begin{bmatrix} \cos\beta\cos\gamma & \cos\beta\sin\gamma & -\sin\beta \\ -\sin\gamma\cos\alpha+\sin\beta\cos\gamma\sin\alpha & \cos\gamma\cos\alpha+\sin\beta\sin\gamma\sin\alpha & \cos\beta\sin\alpha \\ \sin\gamma\sin\alpha+\sin\beta\cos\gamma\cos\alpha & -\cos\gamma\sin\alpha+\sin\beta\sin\gamma\cos\alpha & \cos\beta\cos\alpha \end{bmatrix}$$

with

$$C(1,\alpha)=\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha \\ 0 & -\sin\alpha & \cos\alpha \end{bmatrix} C(2,\beta)=\begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 1 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix} C(3,\gamma)=\begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When you want to rotate an angle around an axis, the axis is fixed and has no influence to other two axis, that is the reason why we have 1 and 0s in our matrix of each sequential rotation.

DCMtoQuaternion(aListOfLists)

If we have a DCM in hand and we want it in another form of Quaternion, things can be different.

If your DCM is defined directly by Euler Symmetric Parameters, then the conversion is much easier.

$$q0 = \cos\frac{\phi}{2}, q1 = f_1 \sin\frac{\phi}{2}, q2 = f_2 \sin\frac{\phi}{2}, q3 = f_3 \sin\frac{\phi}{2}$$

But if your DCM is built by three subsequent rotations, then things can be a little difficult. Because we need to calculate an element of Quaternion, it can be q0, q1, q2 or q3. Then to calculate other elements using the calculated one as denominator. So if the calculated one is near to zero, we may get a big error in division. In this case, the best choice would be calculate each element by using diagonal of DCM, then compare these four elements, choose the biggest one, then treat it as denominator and calculate other three elements.

As the parameter of this function is a DCM, there no need to care about if it is built by Euler Symmetric Parameters. Every DCM is a N by N matrix, which means it will always have a main diagonal. From the main diagonal, we can use the second idea above to calculate each element of a Quaternion object.

At first, we need to calculate the biggest one of four elements

$$q0 = \sqrt{\frac{1}{4} \bullet (1 + C_{11} + C_{22} + C_{33})}$$

$$q1 = \sqrt{\frac{1}{4} \bullet (1 + C_{11} - C_{22} - C_{33})}$$

$$q2 = \sqrt{\frac{1}{4} \bullet (1 - C_{11} + C_{22} - C_{33})}$$

$$q3 = \sqrt{\frac{1}{4} \bullet (1 - C_{11} - C_{22} + C_{33})}$$

compare these four elements and choose the biggest one, then follow the table below to calculate other elements of a Quaternion object.

$\max \rightarrow$	$q0$	$q1$	$q2$	$q3$
$q0$	$q0$	$\frac{C_{32} - C_{23}}{4 \bullet q0}$	$\frac{C_{13} - C_{31}}{4 \bullet q0}$	$\frac{C_{21} - C_{12}}{4 \bullet q0}$
$q1$	$\frac{C_{32} - C_{23}}{4 \bullet q1}$	$q1$	$\frac{C_{21} + C_{12}}{4 \bullet q1}$	$\frac{C_{13} + C_{31}}{4 \bullet q1}$
$q2$	$\frac{C_{13} - C_{31}}{4 \bullet q2}$	$\frac{C_{21} + C_{12}}{4 \bullet q2}$	$q2$	$\frac{C_{32} + C_{23}}{4 \bullet q2}$
$q3$	$\frac{C_{21} - C_{12}}{4 \bullet q3}$	$\frac{C_{13} + C_{31}}{4 \bullet q3}$	$\frac{C_{32} + C_{23}}{4 \bullet q3}$	$q3$

To calculate four elements, first fix the biggest one of four elements by using main diagonal elements in a DCM. Then you can calculate other three elements by using corresponding column.

B. point rotation precision

By now, we have already built our class for Quaternion and functions for DCM. Then we want to know at which level our outputs. The basic thinking behind this part is a point rotation. At first, we test our Quaternion class, then we test our DCM functions, to compare their ability under same situation, we combined two test together trying to find their differences in precision.

Quaternion_rotation_precision(N, R, x_theta, y_theta, z_theta)

Before we dive into the details of this function, let me spend some words in explaining the meaning of each parameters. N is our first parameter. It defines the range of our steps, in our case, the steps start

from 1 and end with N-1. R indicates the number of rounds for each step parameter. In our case, the number of rounds is always be the R-1. Because we want to compare the coordinate differences between the original point and rotated one, in our case, we are stick to the π. After three rotations around each axis with angle π, the point should be in same position as the original one, in theory. In fact, because of the cut down operation in computer, we will never reach exact the same coordinate as the original one and we want to know the difference between them.
In this function, we calculate the mean difference, record the minimum difference and the maximum difference. The final performance will be discussed in the output part in this thesis.

DCM_rotation_precision(N, R, x_theta, y_theta, z_theta)
Like what we have done in the function above, this function changes its aim from Quaternion object to DCM. By using a DCM, we rotate a random point in 3D space. All parameters are defined as the same as above. Our aim is to see the coordinate difference between original point and the rotated one.

Quaternion_DCM_rotation_precision(N, R, x_theta, y_theta, z_theta)
After we test point rotation by using Quaternion and DCM, we want to explore the differences in precision between Quaternion and DCM. In this function, we defined a random point and represented it both in quaternion way and as a vector. All parameters are defined as before. At each single step, we rotated the point as Quaternion and point as a vector at the same time. After each round, we calculated the mean difference, recorded the minimum difference and the maximum difference.

C. rotator precision

after check the influence by Quaternion and DCM by rotating a random point in three-Dimension space, we are now want to know the influences to the rotator, in our case is the quaternion object and a DCM array. We will check the rotator in four different ways. One is to check the influence to the diagonal elements and the off-diagonal elements. One is to check the orthonormality between the columns and rows in DCM. Because in our test, we always use unit vector and normalized quaternion object, we didn't check for the length again in our function and we check for orthogonal. If user used non-unit vector or non-normalized quaternion object, please remember to make sure you have check for the length yourself.
To check the differences, author chose the mean difference. But there still are many ways to describe the differences by using different distance.

1. Euclidean distance
   Assuming we have two points, each element represents a feature:
   $$A = (x_1, x_2, ..., x_n)$$
   $$\mathrm{B} = (y_1, y_2, ..., y_n)$$
   The euclidean distance is defined as followed:
   $$euclidean\_dis = \sqrt{(x_1 - y_1)^2 + (x_2 - y_2)^2 + ... + (x_n - y_n)^2}$$
   In our case, we can use Euclidean distance because all our elements are numbers without units. If some elements have different units, the euclidean distance may not fit this situation. For example, we have two objects in hand. The first element descirbes its weight, the second element descirbe its height. By using formula of Euclidean distance, we can not delete its units. So the addition will be meaning less.
2. Chebyshev distance
   This distance is draw from the chess board and is used mainly with two feature or two dimensions. Assuming we have two points

$$A = (x_1, y_1)$$
$$B = (x_2, y_2)$$

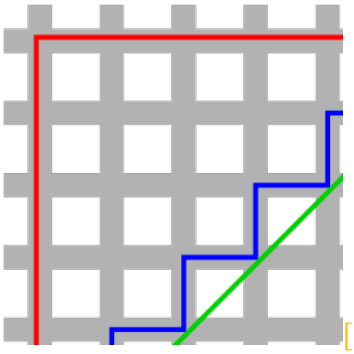
The Chebyshev distance is defined as below:

$$Chebyshev\_dis = \max(|x_1 - x_2|, |y_1 - y_2|)$$

It is defined by the maximum difference in different features.

3. Manhattan distance

   Manhattan distance is one of my favorite distance. It was born based on the concept of Manhattan city blocks. According to the urban planning of Manhattan area, each block is like a square and they are of the same side length. In 2D plane, the Euclidean Distance is the length of straight line between two points and the Manhattan distance is the sum of differences. To make myself clear, the image below can help a lot.



   The green line is the Euclidean distance, the red/green/yellow lines are the Manhattan Distance. The Manhattan distance is a way to represent the sum of points reflection differences on all axis. In 1D space, it equals to the Euclidean distance.

4. Minkovski distance

   The Minkovski distance is a mean of measuring Euclidean space.

   Assuming we have two points in the space like

$$A = (x_1, x_2, ..., x_n)$$
$$B = (y_1, y_2, ..., y_n)$$

   Then the Minkovski distance is defined as

$$Minkovski\_dis = \left(\sum_{i=1}^{n} |x_i - y_i|^p \right)^{\frac{1}{p}}$$

   From the formula of Minkovski distance, we can know that it is a summarized formula for both Euclidean distance and Manhattan distance. When p = 1, the Minkovski distance becomes the Manhattan distance, when p = 2, the Minkovski distance becomes the Euclidean distance, and when the p = positive infinity, then the Minkovski distance becomes the Chebyshev distance.

DCM_diagonal_check(aDCM, bDCM)

This function is to calculate the mean differences of the three diagonal elements in DCMs. There are many ways to measure the differences between the two objects and the author chose a most used one, the mean value of differences. If user want to show differences by using a different measuring way, they can define the function of their own.

In the prospect of space, the differences between the original diagonal elements and the rotated ones is in fact the influence on the rotator axis.

DCM_off_diagonal_check(aDCM, bDCM)

This function is to calculate the mean differences of the six off-diagonal elements in DCMs. Like the function above, author chose the mean value of differences.

Unlike the influences on the axis, the off-diagonal differences are the differences cause by the rotation angles.

DCM_Quaternion_rotator_check0(N, R, x_theta, y_theta, z_theta)

This function is used to produce a DCM with three angles randomly and change DCM into a Quaternion object. By rotating the DCM and the Quaternion object with same angle and round, we can compare the rotation influence on rotator axis and rotator angle.

DCM_orthonormality_check(aDCM, direction)

This function is built to check the orthogonal property in DCM. The first parameter is a DCM, the second parameter indicates the direction of a DCM. The function will check the columns orthogonal when direction parameter equals to 0. If the direction parameter equals to 1 the function will check the row orthogonal. Because we built the DCM with unit vector, the function would not check the length. Once again, if you want to built the DCM with a non-unit vector, please make sure you have normalized it before apply this function. In general, the orthonormality means orthogonal plus normalized.

The thought behind this function is that we built the DCM in a three dimension space, no matter what parameters to be used, the initialized DCM is based on the three axis that are orthogonal to each other. Because of the number length cut down by the machine, the rotation operation will have impacts on the rotator orthogonal appearances.

DCM_Quaternion_rotator_check1(N, R, x_theta, y_theta, z_theta)

This function is used to produce a DCM with three angles randomly and change DCM into a Quaternion object. By rotating the DCM and the Quaternion object with same angle and round, we can compare the rotation influence on rotator orthogonal ability both in columns and rows. Because we built the DCM with a unit length vector and we normalized the Quaternion object after every rotation, the DCM and the Quaternion object are always to be normalized. Considering these two factors together, we have this orthonormality test.

DCM_check (aDCM,bDCM)

To measuring the mean difference between two DCMs, this function applies the theory of Manhattan distance. This function can show us with a overview of the rotation influence on the rotator.

DCM_Quaternion_rotator_check2(N, R, x_theta, y_theta, z_theta)

This function is used to produce a DCM with three angles randomly and change DCM into a Quaternion object. By rotating the DCM and the Quaternion object with same angle and round, we can compare the rotation influence on rotator components.

extracAngleQuaternion(aQuaternion)

In theory, the three Euler angles can be calculated by the formulas as followed

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = \begin{bmatrix} \arctan \dfrac{2(q_0 q_1 + q_2 q_3)}{1 - 2(q_1^2 + q_2^2)} \\ \arcsin(2(q_0 q_2 - q_3 q_1)) \\ \arctan \dfrac{2(q_0 q_3 + q_1 q_2)}{1 - 2(q_2^2 + q_3^2)} \end{bmatrix}$$

In our math module, the arctan function is defined as *math.atan(aParameter)* and *math.atan2(aParameter, bParameter)*. In Python module, both methods can return the angle in radians. We can see from the above that the math.atan() takes in one parameter while the math.atan2() takes in two parameters. We chose math.atan2() because it gaters information of signs on x and y and defines the sign of the return value. With only one parameter in math.atan(), we cannot achieve this aim. The return value of math.atan2() is always between $-\pi$ and π.

extracAngleDCM(aDCM)
If you want to extract three angles from a single DCM, the following formula can be used.

$$\alpha = \arctan(\frac{C_t^s[2,3]}{C_t^s[3,3]});$$
$$\beta = \arcsin(-C_t^s[1,3]);$$
$$\gamma = \arctan(\frac{C_t^s[1,2]}{C_t^s[1,1]})$$ []

Using the formula above, a problem raises, the angle we calculated according to the formula is not unique.
Considering we now extract β now from the matrix element $C_t^s[1,3]$, from our case is $-\sin\beta$. If we use the anti-triangle formula, we can get the angle of β. But it's not unique. For angle β, both β and $\pi - \beta$ have same value for triangle formula. If you used the anti-triangle formula, the answer for the

$$\beta = \sin^{-1}(-C_t^s[1,3])$$

can be

$$\beta_1 = \sin^{-1}(-C_t^s[1,3]) = -\sin^{-1}(C_t^s[1,3])$$
$$\beta_2 = \pi - \beta_1 = \pi - (\sin^{-1}(-C_t^s[1,3])) = \pi + \sin^{-1}(C_t^s[1,3])$$

Same things can happen to the solution of α,

$$\gamma - \alpha = a\tan 2(-C[3,2], -C[3,1])$$
$$\gamma = \alpha + a\tan 2(-C[3,2], -C[3,1])$$

using the atan2 method in math module, we can compute the α like this:

$$\alpha = math.a\tan 2(C[2,3], C[3,3])$$

For author using random method to compute the three angles, all angles of radian would not exceed the 1, so the $\cos\beta$ will always more than 0 and the formula above is adequate for using. But things are not always working like that, if $\cos\beta$ was less than 0, we can compute the α like this:

$$\alpha = math.a\tan 2(-C[2,3], -C[3,3])$$

In summary, to compute α, we can calculate it with help of $\cos\beta$, the formula would be

$$\alpha = math.a\tan 2(\frac{C[2,3]}{\cos\beta}, \frac{C[3,3]}{\cos\beta})$$

for the two versions of our β, the formula can be written as

$$\alpha_1 = math.a\tan 2(\frac{C[2,3]}{\cos\beta_1}, \frac{C[3,3]}{\cos\beta_1})$$
$$\alpha_2 = math.a\tan 2(\frac{C[2,3]}{\cos\beta_2}, \frac{C[3,3]}{\cos\beta_2})$$

For the compute of our γ, because the non-uniqueness of β, our outputs are not unique as well. And they look like

$$\gamma_1 = math.a\tan 2(\frac{C[1,2]}{\cos\beta_1}, \frac{C[1,1]}{\cos\beta_1})$$

$$\gamma_2 = math.a\tan 2(\frac{C[1,2]}{\cos\beta_2}, \frac{C[1,1]}{\cos\beta_2})$$

By now, we have two sets of solution to one DCM, one is $(\alpha_1, \beta_1, \gamma_1)$, another one is $(\alpha_2, \beta_2, \gamma_2)$. The differences between depend on the different β. In our codes, the β always belongs to $[0rad, 1rad)$, we only get one valid set.

What we talked above including almost all the situation we would like to meet when we compute angles from a DCM, excepting for the $\cos\beta = 0$, which in fact will not happened in our case but is likely to appear in reality.

Assuming we have $\cos\beta = 0$, then $\beta = \frac{\pi}{2} or -\frac{\pi}{2}$. If we calculate the α and γ using the formula above, we will face to a problem like

$$C[1,2] = 0, C[2,3] = 0, C[3,3] = 0, C[1,1] = 0, \cos\beta = 0$$

$$\alpha = \tan^{-1}(\frac{C_t^s[2,3]}{C_t^s[3,3]}) = \tan^{-1}(\frac{0}{0})$$

$$\gamma = \tan^{-1}(\frac{C_t^s[1,2]}{C_t^s[1,1]}) = \tan^{-1}(\frac{0}{0})$$

From the above we can know that if we still use the same elements, the angles can not be extract correctly. We need to use another way to do compute them. In our case, DCM is computed in

$$C=\begin{bmatrix} \cos\beta\cos\gamma & \cos\beta\sin\gamma & -\sin\beta \\ -\sin\gamma\cos\alpha + \sin\beta\cos\gamma\sin\alpha & \cos\gamma\cos\alpha + \sin\beta\sin\gamma\sin\alpha & \cos\beta\sin\alpha \\ \sin\gamma\sin\alpha + \sin\beta\cos\gamma\cos\alpha & -\cos\gamma\sin\alpha + \sin\beta\sin\gamma\cos\alpha & \cos\beta\cos\alpha \end{bmatrix}$$

then it becomes like

$$C = \begin{bmatrix} 0 & 0 & -1 \\ -\sin\gamma\cos\alpha + \cos\gamma\sin\alpha & \cos\gamma\cos\alpha + \sin\gamma\sin\alpha & 0 \\ \sin\gamma\sin\alpha + \cos\gamma\cos\alpha & -\cos\gamma\sin\alpha + \sin\gamma\cos\alpha & 0 \end{bmatrix}$$

we can find that in the equation above,

$$C[2,1] = -C[3,2] = -\sin(\gamma - \alpha)$$
$$C[3,1] = C[2,2] = \cos(\gamma - \alpha)$$

then

$$\gamma - \alpha = a\tan 2(-C[3,2], -C[3,1])$$
$$\gamma = \alpha + a\tan 2(-C[3,2], -C[3,1])$$

From the above that, we can conclude the if $\cos\beta = 0$, then the other two angles are linked. No matter which case you had in hand, the situations above can help you.

DCM_Quaternion_rotator_check3(N, R, x_theta, y_theta, z_theta)

This function is used to produce a DCM with three angles randomly and change DCM into a Quaternion object. By rotating the DCM and the Quaternion object with same angle and round, we can compare the rotation influence on rotator angles. The differences are the total difference for three angles.

outputs

Now we come to the outputs part, our aim is to compare the advantages of Quaternion over the DCM. According to our output, the Quaternion works better than DCM at most cases. There are cases our outputs worked not as we had assumed. At those special cases, author will give the reason for the output and give some improvement to our codes.

Our outputs are of five different step and round combinations, they are

step	round	outputs
30	10	Quaternion rotation precision Quaternion rotation time DCM rotation precision DCM rotation time rotation precision of DCM and Quaternion in log rotation precision of DCM and Quaternion in semi-log Diagonal check in DCM and Quaternion off-diagonal check in DCM and Quaternion Orthonormality check in rows Orthonormality check in columns element differences in rotator angle differences in rotator
300	10	
300	100	
300	1000	
3000	1000	

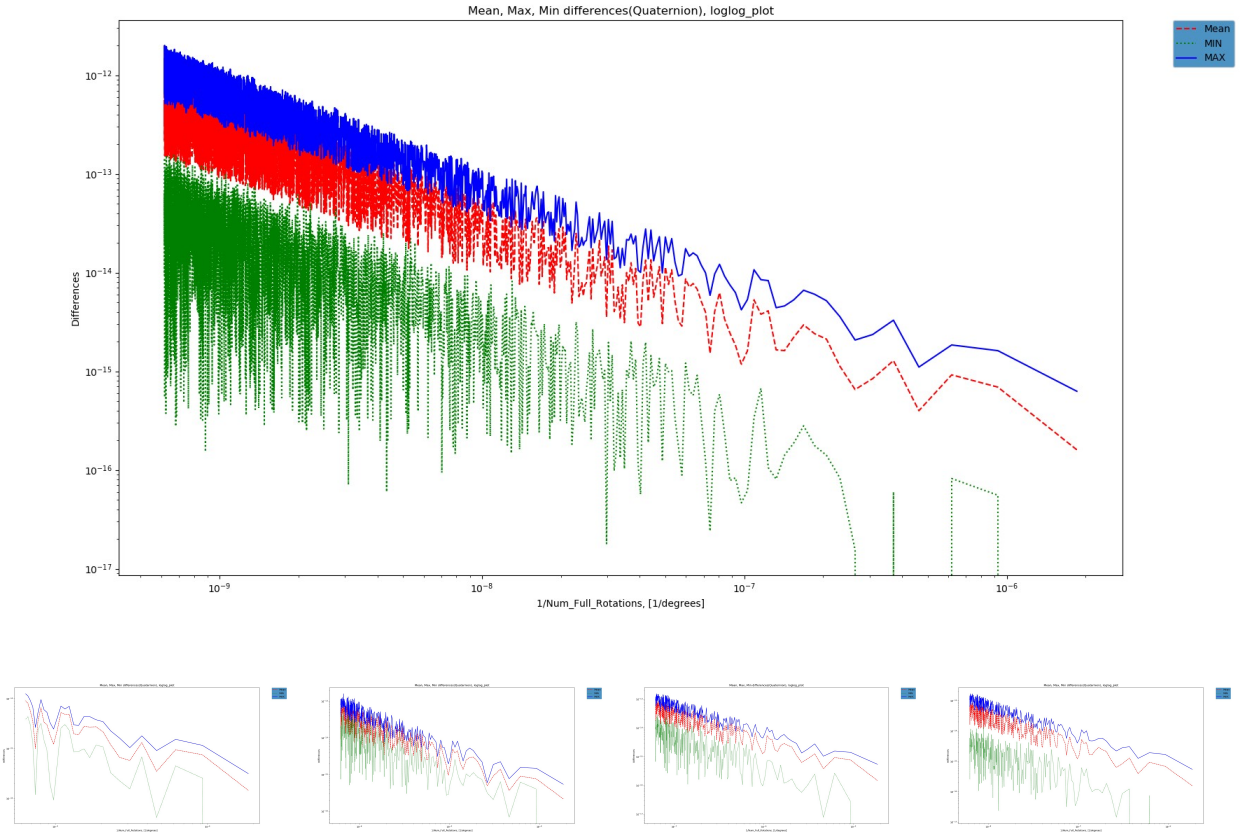
To make our outputs clear and easy to understand, we will show readers the images with largest step and round, other four pictures will in a smaller shape as a reference to the main figure.

All our outputs in same test were produced at same computer and same complier, this control method will make our outputs more reliable.

computer	Lenovo	Asus
complier		
operation system		

Before we dive into the details of our output, we first have a look at the Python module called Matplotlib. This module is an extension of Panda module, it is used widely in two dimensions plot. By using Matplotlib, you can make scatter plot, histogram plot and even three dimensions plot. Sometimes data is not clear to be understood until we use tools to draw them. That is the reason why we use Matplotlib in our implementations.

Quaternion rotation precision

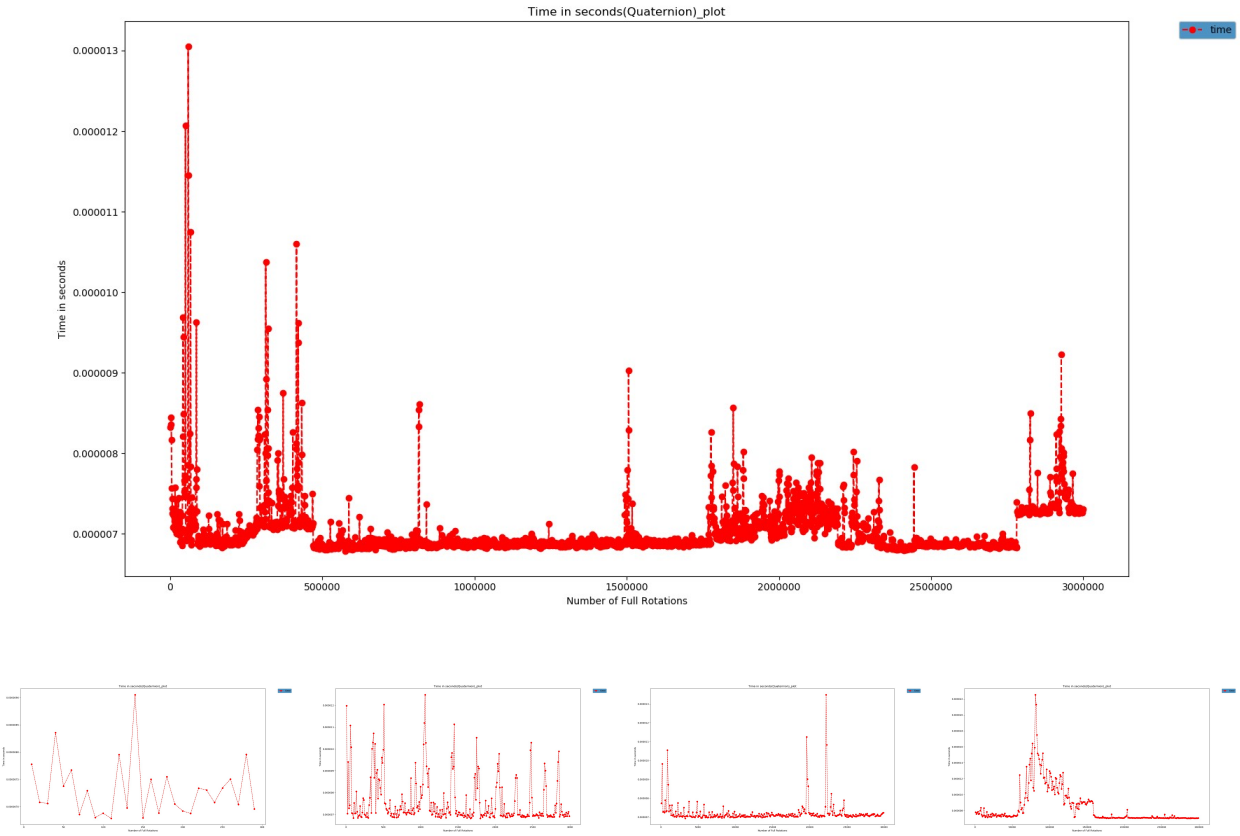


In this plot, we use log scalar to deal with both x and y axis values. Because the rotation differences are very small and hard to plot in a small image, using log can help us to scale them in a reasonable range. In the plot, the blue line represents the maximum differences, the green line shows the variation of minimum differences and the red line explains the mean of maximum and minimum differences. Our x-axis is one over the number of full rotations, our y-axis is the differences.

The maximum differences range from $\left(2^{10^{-15}},2^{10^{-12}}\right)$, the minimum differences range from $\left(2^{10^{-17}},2^{10^{-15}}\right)$. As we can see from the images, we could know that the value of differences goes up when the number of full rotations goes up. In our case, the increase of number of full rotations means at each step, we turned random object with a smaller angle. In a word, the images showed that when we decrease our rotation angle at each step, our error goes up sharply. When we test our codes with small dataset like N=30 and R=10, the output looks not stable. From this stage, we cannot draw any conclusion from the image. Until we get more data from intensive random choice, we can see the trend of our test.

Before we talk about time measurement in Quaternion. Let us have some time talk about how to measurement time in Python. In fact, there are many ways to measure time from time point A to time point B.
We will introduce two of them, one is the time module and another one is timeit module.
Both modules can measure the time differences, there are some differences. In the time module, you can use time.time() to return the time as floats. The float dates back to the epoch. The epoch is depended on platform, most of them is the January 1, 1970, 00:00:00 (UTC) and leap seconds are not counted towards the time in seconds since the epoch. [] It doesn't matter if you use the module to calculate the time differences. But there is one problem that matters, the precision of some platforms would not better than 1 second. Considering our needs, we need to know the time of each step which would not exceed 1 second in most cases. That is the reason why we chose second module called timeit. The timeit module is designed to measure the time of small part of codes. It avoids a number of common traps for measuring execution times. [] This is what we need to measure our time in each step.

Quaternion rotation time

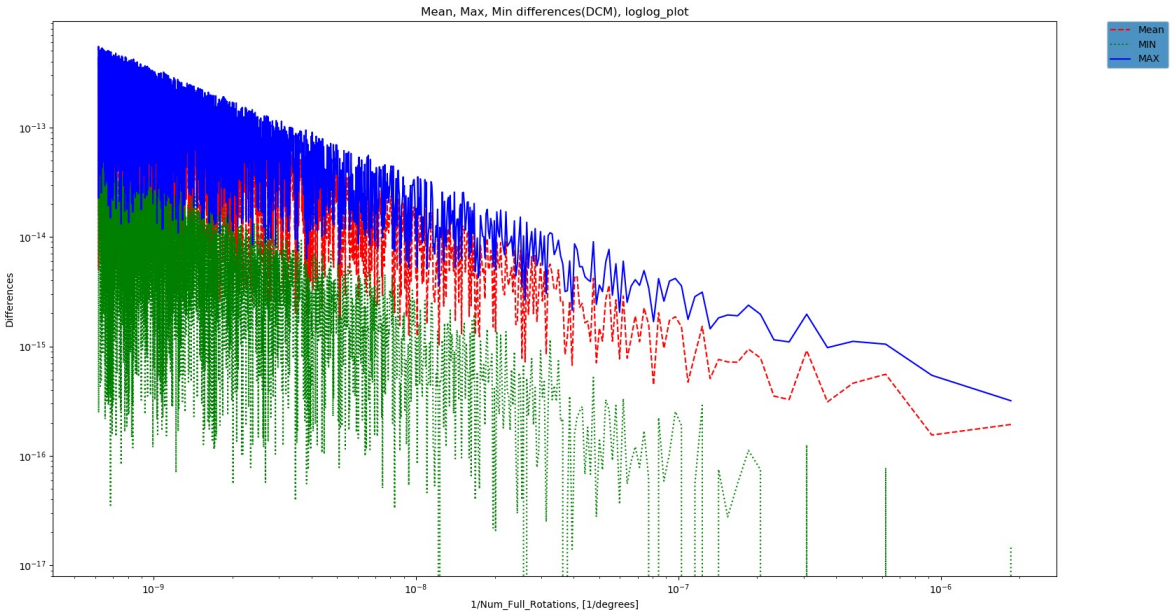


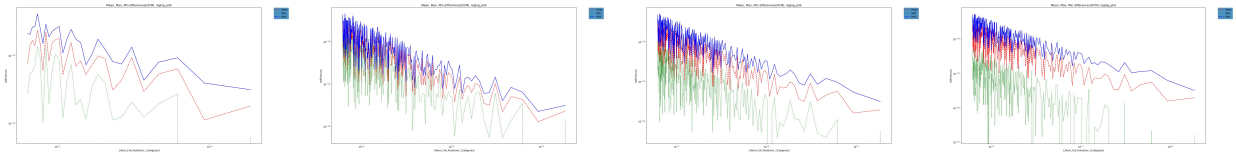
In our Quaternion time plot, our x-axis is on behalf of the number of full rotations which equals the multiplication of two loop numbers. We measure the time of a full rotation and append the time of calculated time of each step into our record. The time in our y-axis means the mean time of each step in a single full rotation. Each red dot is one of our recorded data and they are lined by a red line.

From our largest image we could see that our mean time of each step is around 7×10^{-6}.

From all images above, we could know that no matter how many rotations we run, the mean time is almost the same and this is a good result. But we should also notice about two problems. One is that if we have small data sample like what we have in N=30, R=10 and N=300, R=100, the output is not stable. Another problem is that the time measurement is sensitive to our computer processor. We could see from the image with N=300, R=1000, there is an unpredictable wave between 50000 and 150000 rotation. It might because the computer had some other tasks while processing with codes. You could see that when N=300, R=100 and N=3000, R=1000, the outputs are more reliable and stable.

DCM rotation precision

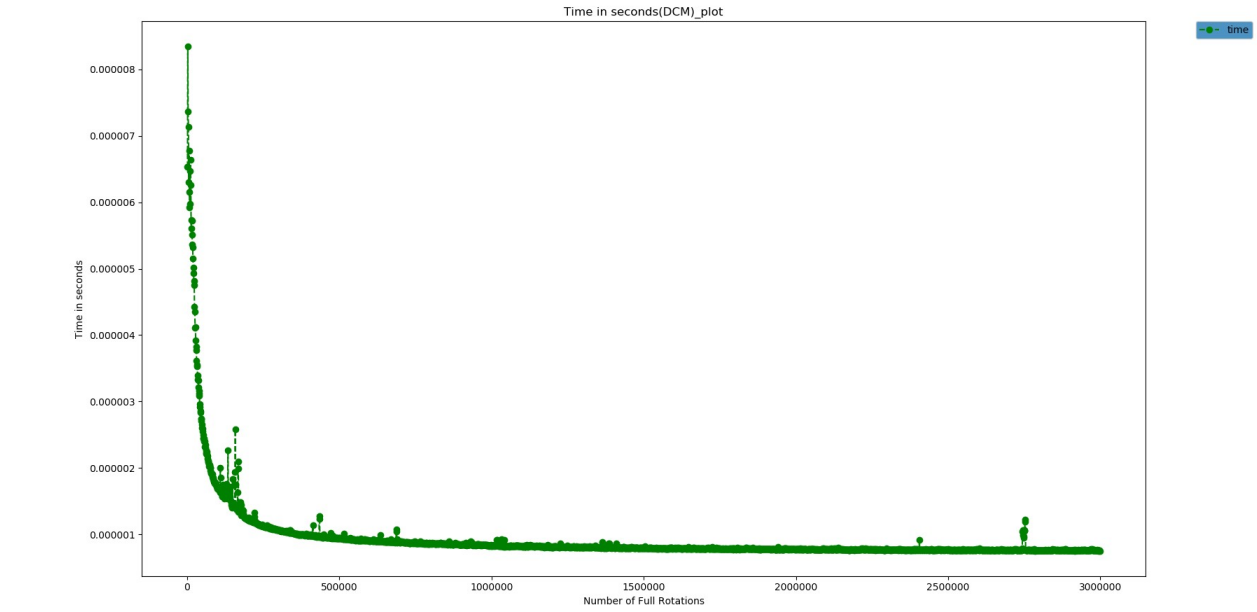


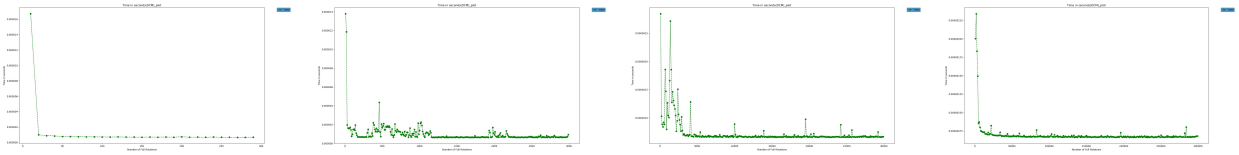


In our images, x-axis is on behalf of 1 over number of full rotations in degrees, y-axis represents the differences. Each image has three lines with different color, they are blue, red and green. Known from the label, the blue line is the maximum differences, the green line is the minimum differences and the red one is the mean of all real differences. See from the image, the blue line is one the top, the green line is in the bottom and the red line is between above two. The location shows our labels are correct. Now focus on the numbers in our images. First thing to emphasis on is that the data were scaled by log function. All data that showed on images are magnitudes. From the largest image we know that the maximum differences in log function range in $\left(10^{-15},10^{-12}\right)$, the minimum differences in log range in $\left(10^{-17},10^{-14}\right)$, the mean difference in log range in $\left(10^{-15},10^{-13}\right)$.

See the images from right to left, all three lines goes up with the value of our x-axis becomes less. Values in x-axis goes down means the number of full rotations goes up, which means each step we calculate the differences using smaller rotation angles.

DCM rotation time





In our DCM rotation time plots, our x-axis is on behalf of the number of full rotations which equals the multiplication of two loop numbers. We measure the time of a full rotation and append the time of calculated time of each step into our record list. The time in our y-axis means the mean time of each step in a single full rotation. Each green dot is one of our recorded data and they are lined by a green line.

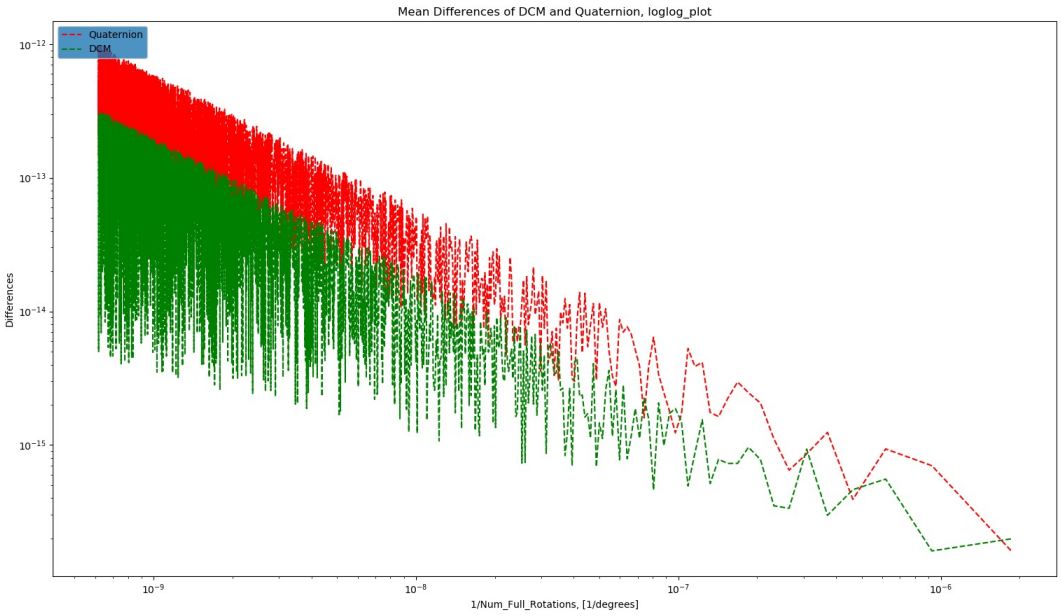
From our largest image we could see that our mean time of each step is around 1×10^{-6} .

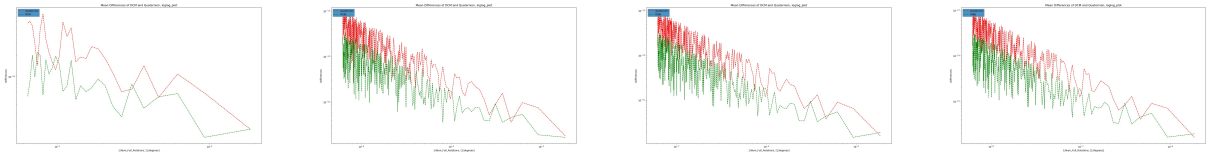
From the five images above, we can find some problems. One is that every we begin our time measurement, the first few results are unreliable. Another one is that the time measurement is not strong enough to ignore other processor influences.

For the first problem, we could focus the data one the left side of our images, when the value of x is near 0, the value of y is bigger compare to the right-side data and it goes down very quickly. It might be a systematic problem, author left them to keep the completeness of the dataset. Because we did not need them to do further investigation, then this action is acceptable. But if we need to handle this dataset for further investigation, the deletion of outliers is necessary.

For the second problem, we already talked about it in Quaternion time plot. It happened because the time measurement is sensitive to the computer core. If core had many tasks in hand, the time measurement would wave like we have in third and forth images. To avoid it, Windows users can close all application down and to get better outputs. For outliers in this problem, simply deletion is not a good idea. Depended on cases, but it would be a better way to keep them in most cases.

rotation precision of DCM and Quaternion in log



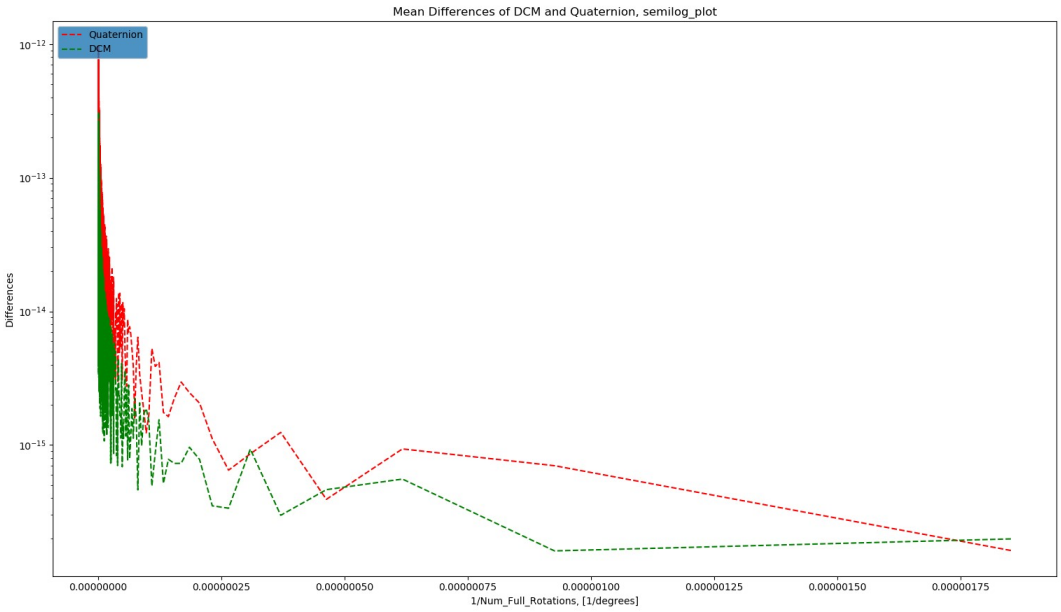


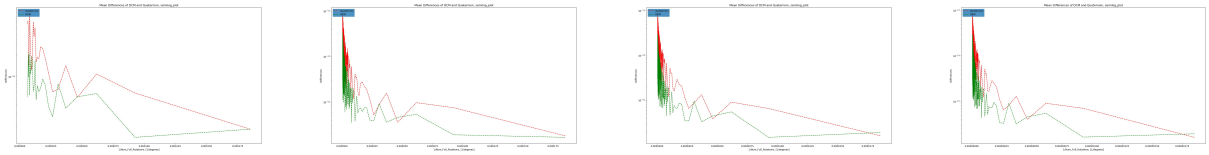
In our images, x-axis is on behalf of 1 over number of full rotations in degrees, y-axis represents the differences. In Quaternion, our differences are the norm of differences between original vector represented in Quaternion object and the rotated one. In our DCM, the difference is the Euclidean distance of original vector and the rotated one. Each image has two lines with different color, they are red and green. Known from the label, the red line is the differences when rotated using Quaternion, the green line is the differences when rotated using DCM. From five images above, the differences caused by Quaternion are always bigger than those caused by DCM and the difference in log is about 10 times than those in DCM.

Now focus on the numbers in our images. First thing to emphasis on is that the data were scaled by log function. All data that showed on images are magnitudes. From the largest image we know that the Quaternion differences in log function range in $\left(10^{-15},10^{-12}\right)$, the DCM differences in log range in $\left(10^{-15},10^{-13}\right)$.

See the images from right to left, all two lines goes up with the value of our x-axis becomes less. Values in x-axis goes down means the number of full rotations goes up, which means each step we calculate the differences using smaller rotation angles.

rotation precision of DCM and Quaternion in semi-log



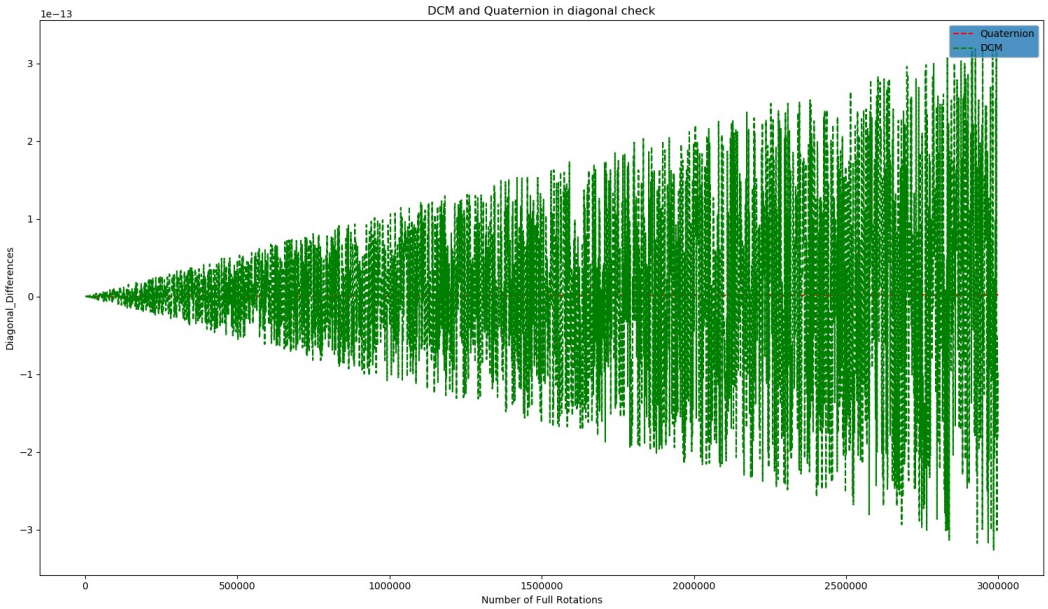


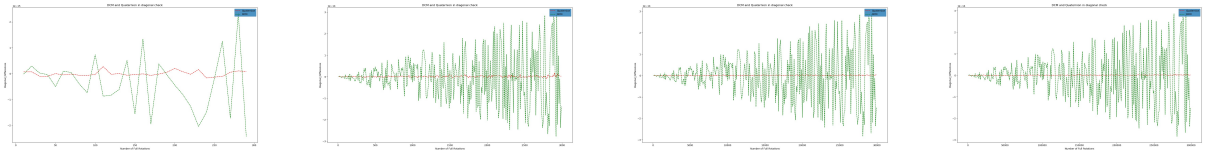
These five plots use same data as the plot in log function before. Without apply log function in both x and y axes, we only apply the log function on y axis. Or if you like, you can choose to use log function only on x axis. Comparing to the log function applied on both axes, these plots here only scale the y axis. We could see that semilog may not very suitable for our differences because it a little hard to figure out the trend and the comparison.
Author tried this plot had two aims. one is to see if we could get a better plot that clear to see and easy to understand. And we now know that is not good for our dataset. Another aim is to show that in Matplotlib, there are many ways to handle your dataset and make your plots more beautiful and clearer. If one plot is not good enough, then try another one and it may work well.

From the point where we stand, we have showed all outputs we get in the influences on a vector rotated by a DCM or a Quaternion. The outputs show that in some cases the DCM works better than Quaternion which is not what we had supposed to see. We will talk about this problem later in comments part.

From here, we will leave the rotation influence on a vector and focus on rotation influence on rotator itself. In our case, the rotators are DCM and Quaternion. To compare their precision, we will compare them in different aspects.

Diagonal check in DCM and Quaternion





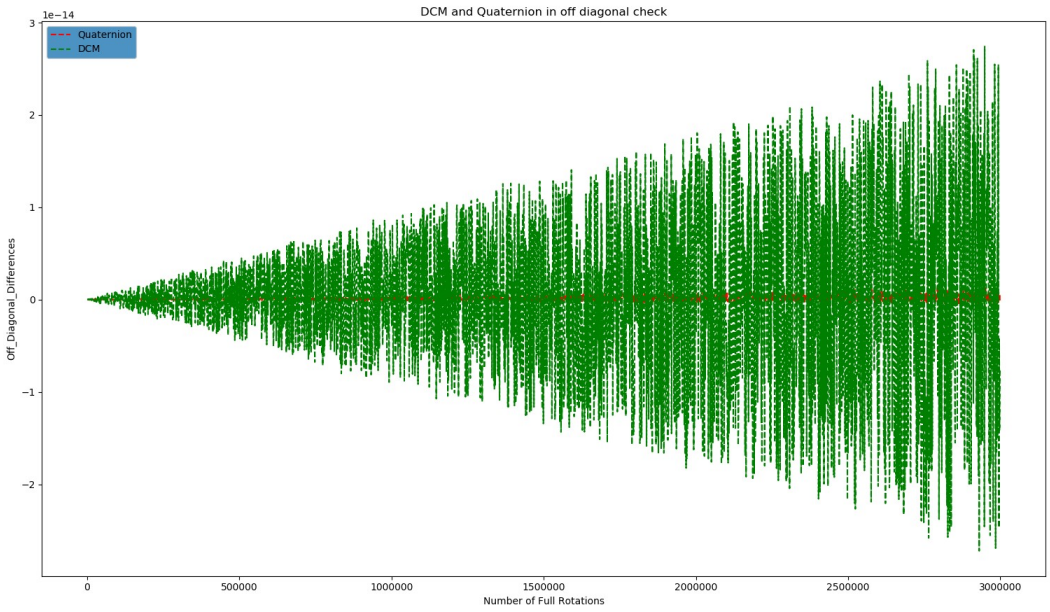
You may have a question when you first came into this problem, each DCM has a diagonal but the Quaternion object doesn't. In our case, we first get a DCM with random choice of three angles, then we convert it to a Quaternion object. After rotation in both with same angle, we convert the Quaternion object back to a DCM again. From there, we can compute the diagonal and off-diagonal check.

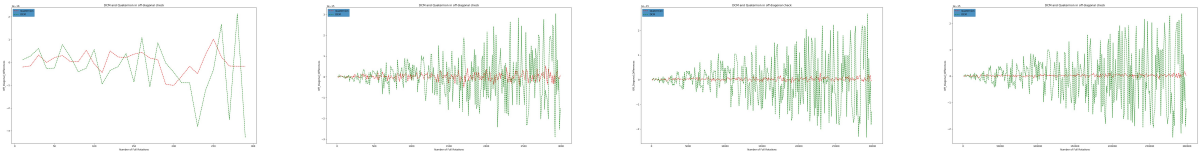
This process has a negative impact on Quaternion object for we add two more steps to Quaternion objects, which may cause more decimal lose in computer.

See from our images, the x-axis represents the number of full rotations and the y-axis means the mean differences in diagonal elements. The magnitude in y-axis is 10^{-13}.

From the largest image we can see that the differences in diagonal elements are around zero, but the green line which represents DCM is fluctuating intensively, while the red line that represents the Quaternion is more stable than DCM. As we increase the number of full rotations, which in fact a decrease of the rotation angle in each step, the difference in Quaternion diagonal elements seems stay around the zero for the red line is very thin and almost be overwritten by the green line. As for the DCM, the difference increases quickly and has a tend to become bigger and bigger as we rotated with a smaller angle. Though the differences in DCM are range in $\left(-3\times10^{-13},3\times10^{-13}\right)$, which is very small in normal aspect, the Quaternion is much more stable and reliable in the same situation.

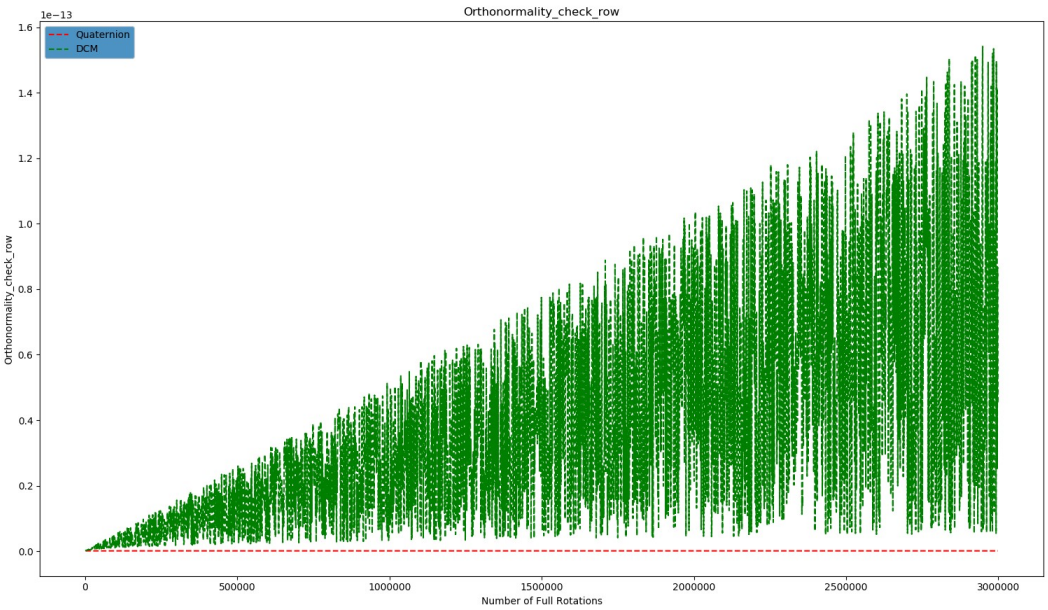
off-diagonal check in DCM and Quaternion

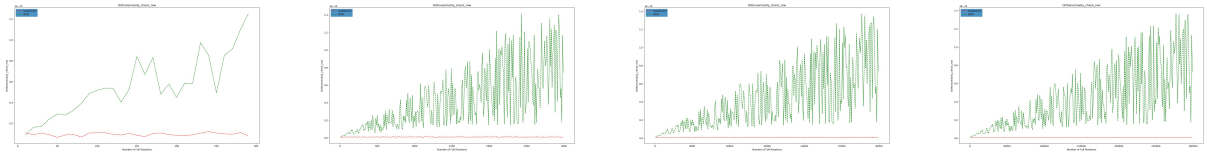




After we compared the differences in diagonal check, here we come to the point to check the off-diagonal elements in both DCM and Quaternion.
The x-axis shows the number of full rotations, with the number increases, the rotated angle in each step decreases. The y-axis represents the mean difference in off-diagonal elements with magnitude of 10^{-14}.
From the five images above, we can see the red line compared to the diagonal check images, which means the off-diagonal difference fluctuates more intensive than those in diagonal difference check.
Our off-diagonal differences in DCM range in $\left(-3\times10^{-14},3\times10^{-14}\right)$ and it has a trend to increase as the rotation angle becomes smaller.

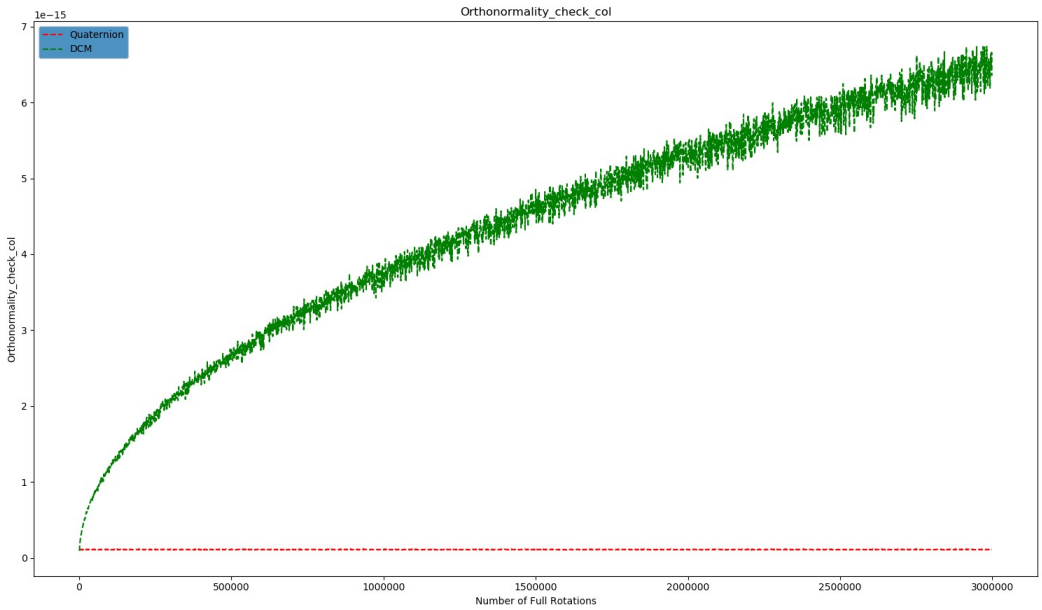
Orthonormality check in rows

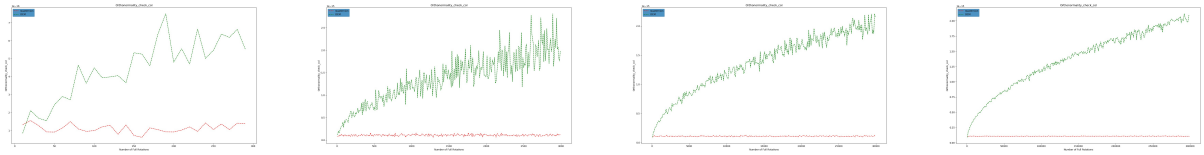




As five images show, the x-axis means the number of full rotations, the y-axis presents the outputs of orthogonal check with a magnitude of 10^{-13}. Here we have the output of rows orthogonal check. The red line that shows the Quaternion object orthogonal check is at zero line without fluctuations. The green line ranges from zero to a larger upper band as we decreased the rotation angle. Conclude from previous data, we can say that if the angle becomes smaller and smaller the range of DCM diagonal row check would be bigger and bigger and the Quaternion diagonal row check would be zero.

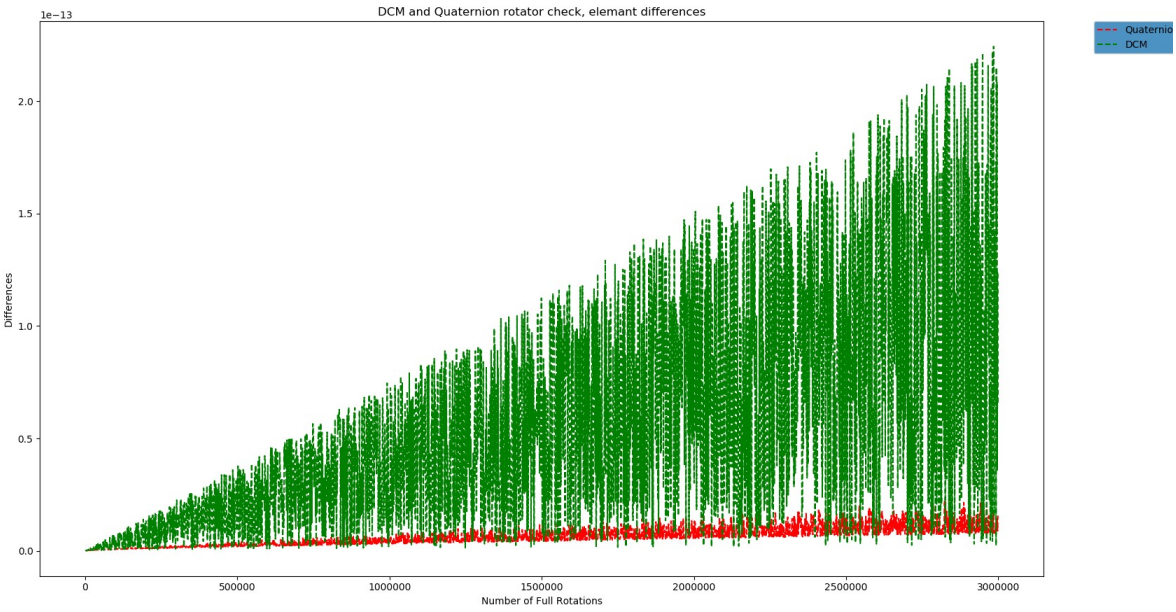
Orthonormality check in columns

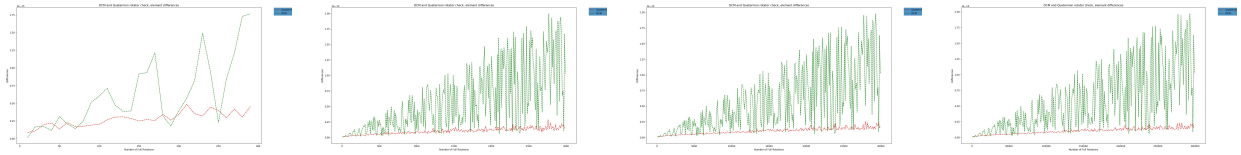




As five images show, the x-axis means the number of full rotations, the y-axis presents the outputs of orthogonal check with a magnitude of 10^{-15}. Here we have the output of columns orthogonal check. The red line that shows the Quaternion object orthogonal check is at zero line without fluctuations. The green line ranges from zero to a larger upper band as we decreased the rotation angle.
Unlike what we have seen in the row check, here we have to notice two differences. One is the magnitude of our y-axis, which is about one over hundred less than those in rows check. Another one is the fluctuation of our DCM doesn't as severely as before. As the rotation angle decreased, the column check had a increasing trend. But we can see from the largest image, the derivative goes down as our angle is shrinking.

element differences in rotator



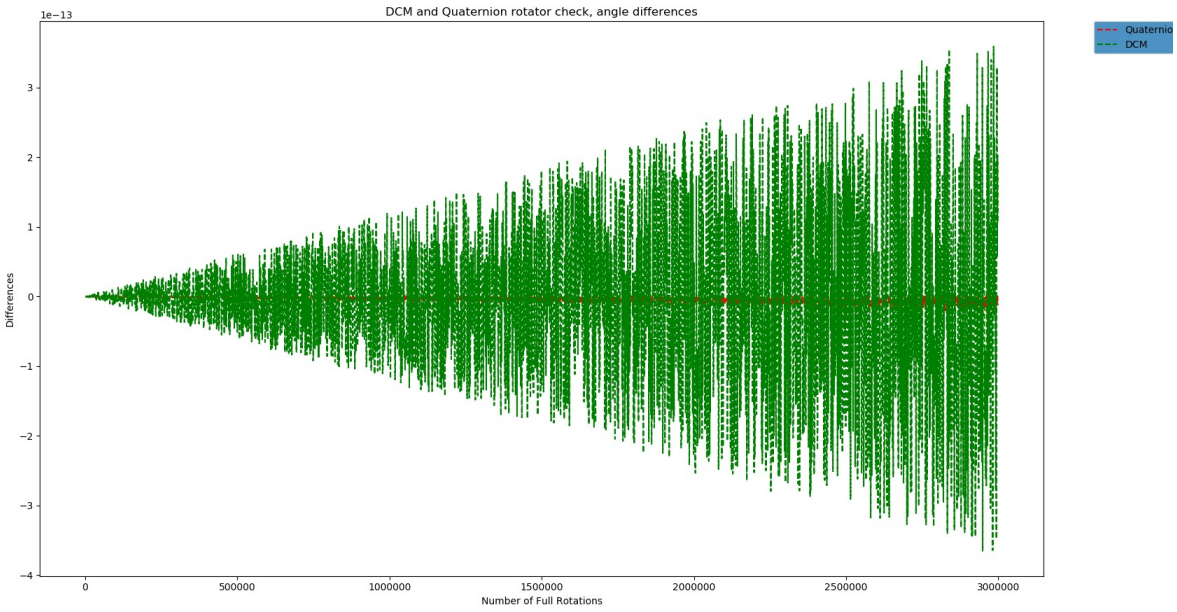


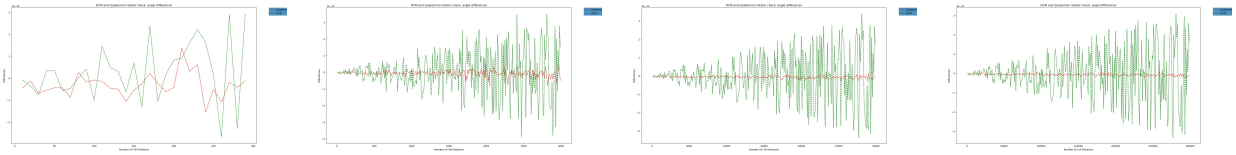
Here we come to the elements check in rotator. The x-axis means the number of full rotations, the y-axis means the mean differences of each elements in two DCMs with magnitude of 10^{-13}. We computed a DCM with three random angles, then used the transformational function to change DCM to Quaternion. After we rotated the DCM and Quaternion, we transformed Quaternion to DCM again. We compared them with original DCMs at last.

The red line is on behalf of Quaternion and the green line represents the DCM. We can notice that both of them are fluctuating with different ranges. As the angle decreases, the DCM has a larger range in fluctuation, the Quaternion also has a fluctuation but with a smaller range.

From what we have said above, we can say in element differences check, the Quaternion rotator is more stable than the DCM's.

angle differences in rotator





Here we come to the angle check in DCM and Quaternion. We extracted angles from both DCM and Quaternion object, then we compared angles with the original random chosen angles. Our x-axis shows the number of full rotations and y-axis represents the angle differences with a magnitude of 10^{-13}.
As the rotation angles became smaller, the differences fluctuated more violently both in DCM and Quaternion. We could see that the plot of green line jumped around zero with bigger and bigger steps from left side to right side. The red line worked as the same way as the green line. Though the red line is nearly overwritten by the green plot, we still can tell that the range of its fluctuation becomes bigger from left to right as the absolute length of red plot increased in y-direction.

improvements

We have seen the time measurements in the outputs chapter, in which the DCM shows a better performs than Quaternion object. Before we dive into what the author did to improve the time measurements, let's see the operation time in a theoretical way.

Assuming we have a Quaternion object $A_1 = a_0 + a_1\underset{\sim}{i} + a_2\underset{\sim}{j} + a_3\underset{\sim}{k}$, derived from a random produced vector $A_2 = (a_1, a_2, a_3)$. We get a rotator in Quaternion is $R_1 = r_0 + r_1\underset{\sim}{i} + r_2\underset{\sim}{j} + r_3\underset{\sim}{k}$, then we convert it to a DCM $R_2 = \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix}$.

At first, we have a look at Quaternion rotation. To rotate a Quaternion object with an angle, we need to compute like:

$$R_1 * A_1 * \overline{R}_1$$

To compute a Hamilton product in two Quaternion objects, we need to

$$\begin{aligned} & R_1 * A_1 \\ &= (r_0 + r_1\underset{\sim}{i} + r_2\underset{\sim}{j} + r_3\underset{\sim}{k}) * (a_0 + a_1\underset{\sim}{i} + a_2\underset{\sim}{j} + a_3\underset{\sim}{k}) \\ &= a_0r_0 - a_1r_1 - a_2r_2 - a_3r_3 + \\ &(a_1r_0 + a_0r_1 + a_2r_2 - a_3r_3)\underset{\sim}{i} + \\ &(a_2r_0 - a_3r_1 + a_0r_2 + a_1r_3)\underset{\sim}{j} + \\ &(a_3r_0 + a_2r_1 - a_1r_2 + a_0r_3)\underset{\sim}{k} \end{aligned}$$

From the equation above we can know we need do 19 multiplications and 15 additions to compute a Hamilton product. To compute a rotation, we need do Hamilton product twice. In that situation, we have to do 32 multiplications and 15 additions.

Look into dot product in two DCMs. Each DCM has 9 elements in three by three matrix. To produce their dot product:

$$\begin{aligned} R_2 \bullet A_2 &= \begin{bmatrix} c_{11} & c_{12} & c_{13} \\ c_{21} & c_{22} & c_{23} \\ c_{31} & c_{32} & c_{33} \end{bmatrix} \bullet \begin{bmatrix} a_1 \\ a_2 \\ a_3 \end{bmatrix} \\ &= \begin{bmatrix} c_{11}a_1 + c_{12}a_2 + c_{12}a_3 \\ c_{21}a_1 + c_{22}a_2 + c_{23}a_3 \\ c_{31}a_1 + c_{32}a_2 + c_{33}a_3 \end{bmatrix} \end{aligned}$$

In each rotation, we need 9 multiplications and 6 additions.

Recall from the implementation part, we used a structure in NumPy called array to store our DCM. It might have nothing special at first glance. But when it came to time measurement, we need to be really careful about this module.

In fact, the whole module called NumPy is underlying built most in C and Fortran and has its own command set. Both of these factors make the operation in NumPy run faster than ordinary operation. Author cares about the speed and precision. For speed comparison, we used a static complier called Cython to convert our codes into C language. And then we compare them in time measurement. The second part we used a module called Decimal that gives us a number with more precision to see if it influences our time measurement.

Cython

If you have ever touched some codes written by C languages, you might find that the codes run faster than those in other languages. It is because the C language is the one called that is near the machine. In other words, it won't check array bound itself or check memory leak issues and so on. But some dynamic languages would do, like Python. That partly gives the reason why Python codes runs slower than C codes in most cases.
To obtain advantages of both languages, many developers contribute to build a static compiler called Cython.
Decimal
comments
conclusion

[1] https://en.wikipedia.org/wiki/Quaternion
[2] https://docs.python.org/2/library/decimal.org
[3] https://docs.python.org/3.7/tutorial/floatingpoint.html
[3] https://en.wikipedia.org/wiki/Object-oriented-programmings
[4] https://github.com/RafeKettler/magicmethods
[5] https://en.m.wikipedia.org/wiki/Quaternion
[6] https://en.m.wikipedia.org/wiki/Quaternion
[7] https://en.m.wikipedia.org/wiki/Quaternion
[8] https://en.wikipedia.org/wiki/Quaternion
[9] https://en.m.wikipedia.org/wiki/Complex_conjugate
[10]
[] https://www.numpy.org/devdocs/user/basics.types.html
[] Parameterization of the DCM
[] https://en.wikipedia.org/wiki/Taxicab_geometry
[] VO02
[]Computing Euler angles from a rotation matrix, Gregory G. Slabaugh
[]https://docs.python.org/3/library/time.html
[]https://docs.python.org/2/library/timeit.html