CHSH_inequality_Qs

April 17, 2020

1 Quaternion Series QM Proof of CHSH inequalites

Doug Sweetser sweetser@alum.mit.edu

1.1 Background

Einstein had issues with quantum mechanics. With Podolsky and Rosen, (particularly Podolsky) he wrote a paper in 1935 that proposed a hidden variable model with all the needed information in the past time-like light conde could predict the outcome of measurements of an entangled state (a modern term). The hidden variable model makes the same predictions as quantum mechanics when an entangled system is measured the same way.

In 1964, John Bell thought carefully about an entangled system where the measurements were made independently by two observers. He focused on measurements that were not made the same way. By being "off" in a known way, one looks to see how correlated measurements are. Quantum mechanics predicts stronger correlations of measurements than hidden variable models.

The John Clauser, Michael Horne, Abner Shimony, and Richard Holt (CHSH) inequality describes a particular way to test Bell's inequality. There are two observers and two states. The name of the game is to calculate the correlation between measurements.

The next section is cut and pasted from the notebook e91_quantum_key_distribution_protocol.ipynb. It will be the basis of the calculations to be done in this notebook.

1.2 *CHSH inequality*

The entangled wave function used:

$$|\psi_s\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B) = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle),$$

In the framework of classical physics, it is impossible to create a correlation inherent in the singlet state $|\psi_s\rangle$. Indeed, let us measure the observables X, Z for qubit A and observables $W = \frac{1}{\sqrt{2}}(X+Z)$, $V = \frac{1}{\sqrt{2}}(-X+Z)$ for qubit B. Performing joint measurements of these observables, the following expectation values can be obtained:

$$\begin{aligned} \langle X \otimes W\rangle_{\psi_s} &= -\tfrac{1}{\sqrt{2}}, \quad \langle X \otimes V\rangle_{\psi_s} = \frac{1}{\sqrt{2}}, \\ \langle Z \otimes W\rangle_{\psi_s} &= -\tfrac{1}{\sqrt{2}}, \quad \langle Z \otimes V\rangle_{\psi_s} = -\frac{1}{\sqrt{2}}. \end{aligned} \tag{2}$$

Exercise: Given the singlet state described in the previous section, show that

$$\langle X \otimes W \rangle_{\psi_s} \quad = -\frac{1}{\sqrt{2}}$$

Now we can costruct the *Clauser-Horne-Shimony-Holt (CHSH) correlation value*:

$$C = \langle X \otimes W \rangle - \langle X \otimes V \rangle + \langle Z \otimes W \rangle + \langle Z \otimes V \rangle = -2\sqrt{2}. \tag{3}$$

The local hidden variable theory which was developed in an attempt to explain the quantum correlations with a classical theory gives that $|C| \leqslant 2$. But Bell's theorem states that "no physical theory of local hidden variables can ever reproduce all of the predictions of quantum mechanics." Thus, the violation of the CHSH inequality (i.e. $C = -2\sqrt{2}$ for the singlet state), which is a generalized form of Bell's inequality, can serve as an *indicator of quantum entanglement*. This fact finds its application in the E91 protocol.

1.3 Quaternion Series and Quantum Mechanics

Apparently there is work by Joy Christian claiming that quaternions can be used to disprove Bell's inequality. Christian's views were dismissed in technical papers and in blogs by leaders in quantum information. The conflict apparently ended a possible career at establishment institutions, so he has set up his own.

That is an unfortunate step in physics history. Let's just *do the right thing*.

Quaternions are a normed division algebra. Because a complex number is a subgroup of quaternions, any and all tasks done with complex numbers will be possible with quaternions of the form (a, b, 0, 0). One needs to take small steps to generalize from there.

A quaternion series is a vector space of quaternions with two additional pieces of information, integers for rows and columns. Quaternion series are neither normed nor a division algebra. Two quaternion series can be orthogonal, showing the norm is not preserved. There is exactly one additive identity, zero for each of the n states. There are 2^n multiplicative inverses for a quaternion series with each state being either zero or one. Quaternion series can be thought of as a semi-group with inverses. The author has spent some effort showing quaternion series may have enough required properties to do quantum mechanics. Given the magnitude of the subject, much remains to be done.

The next section is cut and pasted from the notebook e91_quantum_key_distribution_protocol.ipynb and has the relations that my tools must necessarily recreate.

1.4 A complex-valued analysis of the CHSH correlation value

I have written a library, Qs, to do the work with quaternions and quaternion series. There are two important classes: 1. Q, for quaternions the normed divison algebra 2. Qs, for quaternion series, the semi-group with inverses

In the work of this section, although the tools can take quaternion values, the third and fourth slots of the quaternions are always equal to zero making the quaternions formally identical to complex numbers. This should show that the libraries do as they promise.

Load the needed resources.

[31]:
```
%%capture
%matplotlib inline
import numpy as np
import sympy as sp
import matplotlib.pyplot as plt
import math

# To get equations the look like, well, equations, use the following.
from sympy.interactive import printing
printing.init_printing(use_latex=True)
from IPython.display import display

# Tools for manipulating quaternions.
from Qs import *
from IPython.core.display import display, HTML, Math, Latex
display(HTML("<style>.container { width:100% !important; }</style>"))
```

The assignment does not dictate what basis should be used for the spin states |0> and |1>. There is one exceptionally popular one, up and down:

[2]:
```
q_0, q_1, q_i, q_j, q_k = q0(), q1(), qi(), qj(), qk()

u = Qs([q_1, q_0])
d = Qs([q_0, q_1])

u.print_state("|u>")
d.print_state("|d>")
```

```
|u>
n=1: (1.0, 0.0, 0.0, 0.0)
n=2: (0, 0, 0, 0)
ket: 2/1

|d>
n=1: (0, 0, 0, 0)
n=2: (1.0, 0.0, 0.0, 0.0)
ket: 2/1
```

To demonstrate these tools are robust, another basis will also be used that uses imaginary values.

[32]:
```
r2 = sp.sqrt(1/2)
q_2 = Qs([Q([r2, 0, 0, 0])])
q_2i = Qs([Q([0, r2, 0, 0])])

q_2_op = diagonal(q_2, 2)
q_2i_op = diagonal(q_2i, 2)
```

```
#i = q_2.product(u).add(q_2i.product(d)).ket()
#o = q_2.product(u).dif(q_2i.product(d)).ket()

i = adds(products(q_2_op, u), products(q_2i_op, d))
o = difs(products(q_2_op, u), products(q_2i_op, d))

i.print_state("|i>")
o.print_state("|o>")
```

```
|i>
n=1: (0.707106781186548, 0, 0, 0)
n=2: (0, 0.707106781186548, 0, 0)
ket: 2/1

|o>
n=1: (0.707106781186548, 0, 0, 0)
n=2: (0, -0.707106781186548, 0, 0)
ket: 2/1
```

Now define the four operators to be used as quaternion series Qs.

[33]:
```
# The numbers
q_1r2 = q1(sp.sqrt(1/2))
q_1r2_n = q1(- sp.sqrt(1/2))
q_12 = q1(1/2)
q_12_n = q1(-1/2)

# The operators
 _x = Qs([q_0, q_1r2, q_1r2, q_0], qs_type="op")
 _z = Qs([q_1r2, q_0, q_0, q_1r2_n], qs_type="op")
W = Qs([q_12, q_12, q_12, q_12_n], qs_type="op")
V = Qs([q_12, q_12_n, q_12_n, q_12_n], qs_type="op")

# Print out
 _x.print_state(" _x")
 _z.print_state(" _z")
W.print_state("W")
V.print_state("V")

norm_squareds( _x).print_state("norm of  _x")
norm_squareds(W).print_state("norm of W")
```

```
 _x
n=1: (0, 0, 0, 0)
n=2: (0.707106781186548, 0, 0, 0)
n=3: (0.707106781186548, 0, 0, 0)
```

```
n=4: (0, 0, 0, 0)
op: 2/2

 _z
n=1: (0.707106781186548, 0, 0, 0)
n=2: (0, 0, 0, 0)
n=3: (0, 0, 0, 0)
n=4: (-0.707106781186548, 0, 0, 0)
op: 2/2

W
n=1: (0.5, 0.0, 0.0, 0.0)
n=2: (0.5, 0.0, 0.0, 0.0)
n=3: (0.5, 0.0, 0.0, 0.0)
n=4: (-0.5, 0.0, 0.0, 0.0)
op: 2/2

V
n=1: (0.5, 0.0, 0.0, 0.0)
n=2: (-0.5, 0.0, 0.0, 0.0)
n=3: (-0.5, 0.0, 0.0, 0.0)
n=4: (-0.5, 0.0, 0.0, 0.0)
op: 2/2

norm of  _x
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

norm of W
n=1: (1.0, 0.0, 0.0, 0.0)
scalar_q: 1/1
```

All the needed players are here: a way to represent the spin 2 states and the operators. The additional function one needs is called "bracket" which works like so:

[34]:
```
bracket = Qs.bracket
bracket(u, identity(2, operator=True), u).print_state("<u|I|u>")
bracket(u,  _x, u).print_state("<u| _x|u>")
```

```
fed 2 bras or kets, took a conjugate. Double check.
<u|I|u>
n=1: (1.0, 0.0, 0.0, 0.0)
scalar_q: 1/1

fed 2 bras or kets, took a conjugate. Double check.
<u| _x|u>
n=1: (0, 0, 0, 0)
```

```
scalar_q: 1/1
```

Operators mix things around!

Here is the first task:

Given a wave function:

$$|\psi_s\rangle = \frac{1}{\sqrt{2}}(|0\rangle_A \otimes |1\rangle_B - |1\rangle_A \otimes |0\rangle_B) = \frac{1}{\sqrt{2}}(|01\rangle - |10\rangle),$$

show:

$$\langle X \otimes W \rangle_{\psi_s} \quad = -\frac{1}{\sqrt{2}}$$

There are eight brackets that have to be calculated: 1. $< 0X0 >< 1W1 >$ 1. $< 1X1 >< 0W0 >$ 1. $- < 0X1 >< 1W0 >$ 1. $- < 1X0 >< 0W1 >$

These will have to be repeated a bunch of times too. Time to write a function for the task.

[35]:
```python
def bracket_bracket(ket_0, ket_1, op_1, op_2, verbose=False):
    """Form the inner product for a superposition of states."""

    b_010 = bracket(ket_0.bra(), op_1, ket_0.ket())
    b_111 = bracket(ket_1.bra(), op_1, ket_1.ket())
    b_011 = bracket(ket_0.bra(), op_1, ket_1.ket())
    b_110 = bracket(ket_1.bra(), op_1, ket_0.ket())

    b_121 = bracket(ket_1.bra(), op_2, ket_1.ket())
    b_020 = bracket(ket_0.bra(), op_2, ket_0.ket())
    b_120 = bracket(ket_1.bra(), op_2, ket_0.ket())
    b_021 = bracket(ket_0.bra(), op_2, ket_1.ket())

    b_0011 = products(b_010, b_121)
    b_1100 = products(b_111, b_020)
    b_0110 = products(b_011, b_120)
    b_1001 = products(b_110, b_021)

    bb = difs(difs(adds(b_0011, b_1100), b_0110), b_1001)

    if verbose:
        b_010.print_state("b_010", quiet=True)
        b_011.print_state("b_011", quiet=True)
        b_110.print_state("b_110", quiet=True)
        b_111.print_state("b_111", quiet=True)
        b_121.print_state("b_121", quiet=True)
        b_120.print_state("b_120", quiet=True)
        b_021.print_state("b_021", quiet=True)
```

```
        b_020.print_state("b_020", quiet=True)

        b_0011.print_state("b_0011", quiet=True)
        b_1100.print_state("b_1100", quiet=True)
        b_0110.print_state("b_0110", quiet=True)
        b_1001.print_state("b_1001", quiet=True)

    return bb
```

[36]:
```
XxW = bracket_bracket(d, u,  _x, W)
XxW.print_state("XxW")
```

```
XxW
n=1: (-0.707106781186548, 0, 0, 0)
scalar_q: 1/1
```

Bingo, bingo. This was not a trivial calculation.

Demonstrate the expression is true even if the basis is switched to |i> and |o> which use imaginary values.

[37]:
```
bracket_bracket(i, o,  _x, W).print_state("XxW")
```

```
XxW
n=1: (-0.707106781186548, 0, 0, 0)
scalar_q: 1/1
```

Now calculate the other 3: XxV, ZxW, and ZxV.

[38]:
```
XxV = bracket_bracket(i, o,  _x, V)
XxV.print_state("XxV")

ZxW = bracket_bracket(i, o,  _z, W)
ZxW.print_state("ZxV")

ZxV = bracket_bracket(i, o,  _z, V)
ZxV.print_state("ZxV")
```

```
XxV
n=1: (0.707106781186548, 0, 0, 0)
scalar_q: 1/1

ZxV
n=1: (-0.707106781186548, 0, 0, 0)
scalar_q: 1/1

ZxV
n=1: (-0.707106781186548, 0, 0, 0)
```

```
scalar_q: 1/1
```

Add 'em all up but XxV which gets subtracted.

[39]:
```
CHSH = difs(adds(adds(XxW, ZxW), ZxV), XxV)
CHSH.print_state("CHSH")
-2 * sp.sqrt(2.0)
```

```
CHSH
n=1: (-2.82842712474619, 0, 0, 0)
scalar_q: 1/1
```

[39]:

-2.82842712474619

This is the correct answer, a good thing.

1.5 Redo with quaternion-valued states

Take the state vectors |i> and |o> and add in a non-zero j and k. The normalization **has to be tweeked** so that the states remain ortho-normal.

[28]:
```
n = sp.sqrt(6)
i3 = Qs([Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, 1/n, 1/n, 1/n])])
o3 = Qs([Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, -1/n, -1/n, -1/n])])

norm_squareds(i3).print_state("i3 norm")
norm_squareds(o3).print_state("o3 norm")
products(o3.bra(), i3).print_state("<o3|i3>")
```

```
i3 norm
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

o3 norm
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

<o3|i3>
n=1: (1.11022302462516E-16, 0, 0, 0)
scalar_q: 1/1
```

There is a bit of rounding error, but I will count this as a quaternion-valued ortho-normal spin 2 states. Calculate the XxV as before.

[40]:
```
XxV = bracket_bracket(i3, o3, _x, W)
XxV.print_state("XxV", quiet=True)
```

```
XxV
n=1: (-0.707106781186548, 0, 0, 0)
scalar_q: 1/1
```

There better not be anything special about the direction 1, 1, 1. Show a different direction also works.

[43]:
```
n = sp.sqrt(28)
i123 = Qs( [ Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, 1/n, 2/n, 3/n])])
o123 = Qs( [ Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, -1/n, -2/n, -3/n])])

norm_squareds(i123).print_state("i123 norm")
norm_squareds(o123).print_state("o123 norm")
products(o123.bra(), i123).print_state("<o123|i123>")

XxV = bracket_bracket(i123, o123,  _x, W)
XxV.print_state("XxV", quiet=True)
```

```
i123 norm
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

o123 norm
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

<o123|i123>
n=1: (1.11022302462516E-16, 0, 0, 0)
scalar_q: 1/1

XxV
n=1: (-0.707106781186548, 5.55111512312578E-17, -2.77555756156289E-17, 0)
scalar_q: 1/1
```

[44]:
```
n = sp.sqrt(28)
i123 = Qs([Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, 1/n, 2/n, 3/n])])
o123 = Qs([Q([sp.sqrt(1/2), 0, 0, 0]), Q([0, -1/n, -2/n, -3/n])])

norm_squareds(i123).print_state("i123 norm")
norm_squareds(o123).print_state("o123 norm")
products(o123.bra(), i123).print_state("<o123|i123>")

XxV = bracket_bracket(i123, o123,  _x, W)
XxV.print_state("XxV")
```

```
i123 norm
n=1: (1.00000000000000, 0, 0, 0)
```

```
scalar_q: 1/1

o123 norm
n=1: (1.00000000000000, 0, 0, 0)
scalar_q: 1/1

<o123|i123>
n=1: (1.11022302462516E-16, 0, 0, 0)
scalar_q: 1/1

XxV
n=1: (-0.707106781186548, 5.55111512312578E-17, -2.77555756156289E-17, 0)
scalar_q: 1/1
```

The rounding error is worse, but that is trivial.

1.6 Why this *had* to work

Complex numbers are a subgroup of quaternions. Nature and mathematics are logically consistent, so it follows that any expression written using complex numbers can be rewritten using quaternions that have a pair of zeros. The majority of this notebook showed the trivial double-zero quaternion process worked.

In a brief exchange with the famous blogger, he refused to call this an exercise in quaternion quantum mechanics since it so closely aligned with complex-valued quantum mechanics as to not justify having a separate name. To me, that indicates a bias - one hopes for something radically new. There is an entire book devoted to quaternion quantum mechanics by Stephen Adler, yet its author admits the work is a failure since it allows for superluminal transformation of information.

Quaternion series quantum mechanics may be boring, but at least it works. Prosals that work are worthy of further investigation.

If one interprets the imaginary numbers of quaternions as a spatial thing, then one can argue on physical grounds that space is homogeneous, so it should not matter what direction one points in: i, j, k, or any combination of those. By convention, we have always chosen the exactly same direction in space, i since the 1920s, with $j = k = 0$. Why was this so?

I have a name for this requirement: **Don't point like a drunk, point with precision**. Think about a typical quantum system experiment done on a light table. Months can go into tweaking the layout of the layers, mirrors, and detectors. Experimental physicists are famous about being compulsive about pointing with precision, isolating vibrations from the table inself. That attitude of precision is reflected in the numbers used: the quaternions in space point the same way.