Tetravolumes with Quadrays

I'm testing a new Qvectors enhancement, suggested to me by Tom Ace:

$$V_{ivm} = (1/4)\begin{vmatrix} a0 & a1 & a2 & a3 & 1 \\ b0 & b1 & b2 & b3 & 1 \\ c0 & c1 & c2 & c3 & 1 \\ d0 & d1 & d2 & d3 & 1 \\ 1 & 1 & 1 & 1 & 0 \end{vmatrix}$$

Take the absolute value of that determinant if you wish only positive values for the tetrahedron's tetravolume.

What tetrahedron?

The one defined by the four Quadrays or Qvectors embedded as the first four rows in the matrix.

What's a Quadray?

Quadrays

Each Qvector is defined using 4-tuples representing 4 positively scaled and additively combined elementary rays.

These four elementary rays originate from the origin (0,0,0,0) and go to the 4 corners of a regular tetrahedron: (1,0,0,0) (0,1,0,0) (0,0,1,0) and (0,0,0,1). See Figure 1.

Tools we'll need:

In [1]:
```
import numpy as np
from qrays import Qvector
from random import choice
from itertools import permutations
import tetravolume as tv
```

The four elementary Qvectors:

In [2]:
```
a = Qvector((1,0,0,0))
b = Qvector((0,1,0,0))
c = Qvector((0,0,1,0))
d = Qvector((0,0,0,1))
```

Linear combinations of these four elementary rays yield a unique positive 4-tuple in canonical form -- representing an equivalence class -- corresponding to every (x, y, z) coordinate in $\mathbb{R}^3$.

In this implementation of Qvectors, the resulting home base tetrahedron is what has edges 1, not the elementary Qvectors. An XYZ unit cube will be introduced below with half this edge length, i.e. 1R instead of 1D.

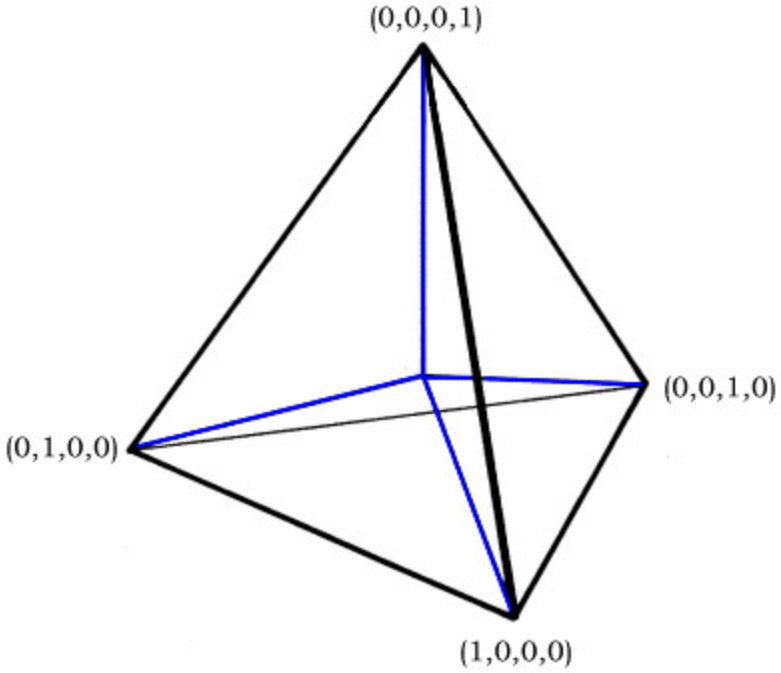


Figure 1

In [3]:
```
(a-b).length()
```

Out[3]:
```
1.0
```

Tetravolumes

What about "tetravolume"? That's where we take a tetrahedron of edges 1, as unit volume, or edges 2 if measuring in radians (radii).

The Quadrays in question have been geared to operate in such a namespace, i.e. the balls below have radius R, diameter D, D = 2R. That's our home base tetrahedron with (0,0,0,0) at its center of gravity.

Figure 2

In [4]:
```
for q in a,b,c,d:
```

```
    print(q.coords)
```

```
ivm_vector(a=1, b=0, c=0, d=0)
ivm_vector(a=0, b=1, c=0, d=0)
ivm_vector(a=0, b=0, c=1, d=0)
ivm_vector(a=0, b=0, c=0, d=1)
```

In [5]:
```
for q in a,b,c,d:
    print(q.xyz.xyz)
```

```
xyz_vector(x=0.35355339059327373, y=0.35355339059327373, z=0.35355339059327373)
xyz_vector(x=-0.35355339059327373, y=-0.35355339059327373, z=0.35355339059327373)
xyz_vector(x=-0.35355339059327373, y=0.35355339059327373, z=-0.35355339059327373)
xyz_vector(x=0.35355339059327373, y=-0.35355339059327373, z=-0.35355339059327373)
```

Now lets get down to the business of testing:

In [6]:
```
def volume(q0, q1, q2, q3):
    """
    Construct a 5x5 matrix per Tom Ace
    """
    A = np.ones((5,5)) # shape, all 1s except..
    A[4,4] = 0 # zero in lower right corner
    A[0,0:4] = q0.coords
    A[1,0:4] = q1.coords
    A[2,0:4] = q2.coords
    A[3,0:4] = q3.coords
    print(A) # comment out?
    return abs(np.linalg.det(A))/4 # that's it!
```

Do we get back the expected unit-volume?

In [7]:
```
volume(a,b,c,d)
```

```
[[1. 0. 0. 0. 1.]
 [0. 1. 0. 0. 1.]
 [0. 0. 1. 0. 1.]
 [0. 0. 0. 1. 1.]
 [1. 1. 1. 1. 0.]]
```

Out[7]:
```
1.0
```

Double all Qvector lengths...

In [8]:
```
e, f, g, h = map(lambda q: 2 * q, (a,b,c,d))
```

Volume is 8-folded...

In [9]:
```
volume(e, f, g, h)
```

```
[[2. 0. 0. 0. 1.]
 [0. 2. 0. 0. 1.]
 [0. 0. 2. 0. 1.]
 [0. 0. 0. 2. 1.]
 [1. 1. 1. 1. 0.]]
```

Out[9]:
```
8.0
```

Random Tetrahedrons in the IVM

Create the 12 directions to neighoring balls in a ball packing.

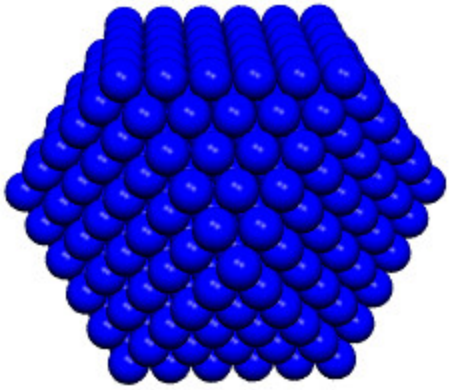


Figure 3: CCP

In [10]:
```
moves = [Qvector(p) for p in set(permutations((2,1,1,0)))]
moves
```

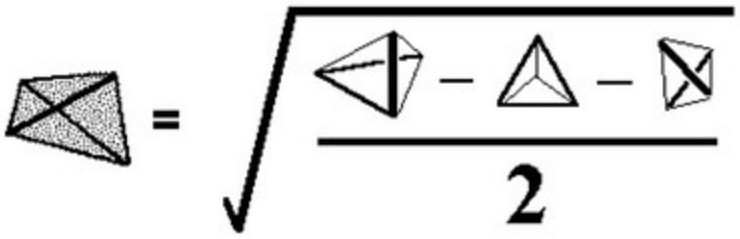
Out[10]:
```
[ivm_vector(a=0, b=1, c=1, d=2),
 ivm_vector(a=1, b=0, c=1, d=2),
 ivm_vector(a=2, b=0, c=1, d=1),
 ivm_vector(a=0, b=2, c=1, d=1),
 ivm_vector(a=0, b=1, c=2, d=1),
 ivm_vector(a=1, b=2, c=1, d=0),
 ivm_vector(a=1, b=1, c=2, d=0),
 ivm_vector(a=2, b=1, c=1, d=0),
 ivm_vector(a=1, b=0, c=2, d=1),
 ivm_vector(a=1, b=2, c=0, d=1),
 ivm_vector(a=2, b=1, c=0, d=1),
 ivm_vector(a=1, b=1, c=0, d=2)]
```

In [11]:
```
def random_walk(start, steps):
    """
    Randomly move in the 12 CCP directions
    for steps times, starting at start
    """
    end = start
    for _ in range(steps):
        end += choice(moves)
    return end

vA = random_walk(Qvector((0,0,0,0)), 1000)
vB = random_walk(Qvector((0,0,0,0)), 1000)
vC = random_walk(Qvector((0,0,0,0)), 1000)
vD = random_walk(Qvector((0,0,0,0)), 1000)
```

In [12]:
```
ab = (vA-vB).length()
ac = (vA-vC).length()
ad = (vA-vD).length()
bc = (vB-vC).length()
cd = (vC-vD).length()
bd = (vB-vD).length()
```

The volume formula used in tv.Tetrahedron operates on the six edge lengths only.



Here's a Python version of the algorithm:

```
def vol2(edges):
    """Return the volume of a tetrahedron, 6 edge lengths

    Thanks to Leonhard Euler and Gerald de Jong
    """
    a2,b2,c2,d2,e2,f2 = map(mul,edges,edges)

    open   = ( f2 * a2 * b2 + d2 * a2 * c2
             + a2 * b2 * e2 + c2 * b2 * d2
             + e2 * c2 * a2 + f2 * c2 * b2
             + e2 * d2 * a2 + b2 * d2 * f2
             + b2 * e2 * f2 + d2 * e2 * c2
             + a2 * f2 * e2 + d2 * f2 * c2 )
    closed = ( a2 * b2 * d2 + d2 * e2 * f2
             + b2 * c2 * e2 + a2 * c2 * f2 )
    oppos  = ( a2 * e2 * (a2 + e2)
             + b2 * f2 * (b2 + f2)
             + c2 * d2 * (c2 + d2))

    return (abs(open - closed - oppos)/2.0)**0.5
```

Figure 4: Gerald's Volume Formula

The formula originally comes from Gerald de Jong, a tensegrity master and an inventor of "elastic interval geometry".

My taking the absolute value in the last line is unnecessary in this case as the result is positive if the tetrahedron is possible at all.

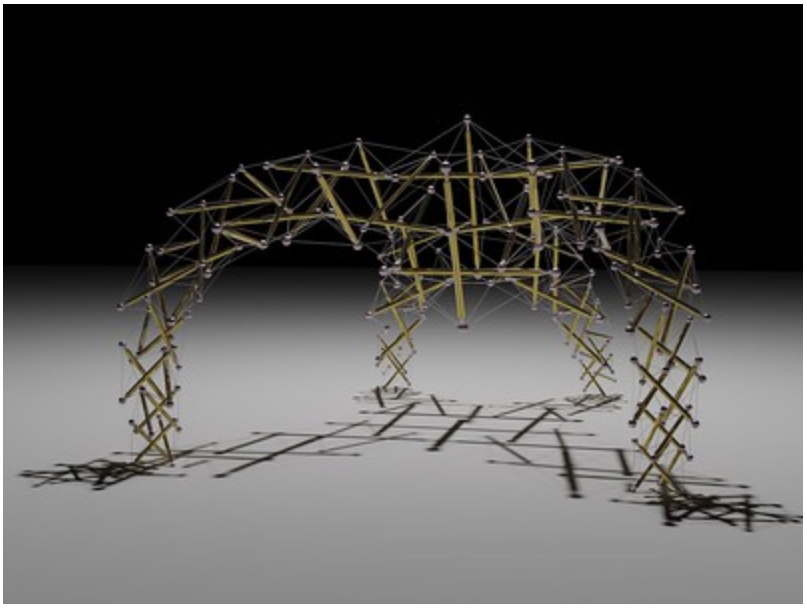


Figure 5: Computer Rendered Tensegrity by Gerald de Jong

In [13]:

```
t = tv.Tetrahedron(ab,ac,ad,bc,cd,bd)
t.ivm_volume()
```

Out[13]:
```
5235.0
```

In [14]:
```
volume(vA, vB, vC, vD)
```

```
[[59.  0. 51. 18.  1.]
 [ 0.  6. 37.  5.  1.]
 [24. 31.  0. 25.  1.]
 [ 3. 14.  0.  3.  1.]
 [ 1.  1.  1.  1.  0.]]
5235.000000000003
```

Out[14]:

These vestigal rounding errors are annoying. Lets employ some arbitrary precision numbers, allowing 200 bits per number object. If you grab a copy of this Notebook and install gmpy2, you will be able to adjust the precision.

In [15]:
```
import gmpy2
from gmpy2 import mpfr
gmpy2.get_context().precision=200
z0 = mpfr('0')
vA = random_walk(Qvector((z0,z0,z0,z0)), 1000)
vB = random_walk(Qvector((z0,z0,z0,z0)), 1000)
vC = random_walk(Qvector((z0,z0,z0,z0)), 1000)
vD = random_walk(Qvector((z0,z0,z0,z0)), 1000)
ab = (vA-vB).length()
ac = (vA-vC).length()
ad = (vA-vD).length()
bc = (vB-vC).length()
cd = (vC-vD).length()
bd = (vB-vD).length()
t = tv.Tetrahedron(ab,ac,ad,bc,cd,bd)
t.ivm_volume()
```

Out[15]:
```
mpfr('64935.0',200)
```

Since the inputs to the determinant below will be only integers in this situation, it's OK to force an integer result. If the algorithm is correct, its output should match the above. Why float answers in the first place? numpy speeds determinant finding, using time-saving short cuts that introduce some entropy.

Since we know from other homework that tetrahedrons randomly generated by the method described have integer volumes, we're fine with rounding and typecasting to int as a last step, in the version of the volume function below.

In [16]:
```
def ivm_volume(q0, q1, q2, q3, dtype=np.float):
    """
    improved:  more flexible as to element type
    """
    A = np.ones((5,5), dtype)
    A[4,4] = 0
    A[0,0:4] = q0.coords
    A[1,0:4] = q1.coords
    A[2,0:4] = q2.coords
    A[3,0:4] = q3.coords
    print(A)
    return abs(np.linalg.det(A))/4
```

In [17]:
```
int(round(ivm_volume(vA, vB, vC, vD, np.int64)))
```

```
[[52  0 39  9  1]
```

```
 [ 0 12 22 22  1]
 [61 77  0 18  1]
 [ 0 57 49 22  1]
 [ 1  1  1  1  0]]
```

Out[17]:
```
64935
```

Lets express three vectors emanating from a common origin (0,0,0,0) as if from the corner of a regular tetrahedron. The Qvectors in question, each of unit length, would be:

In [18]:
```
o = Qvector((0,0,0,0)) # origin
p = Qvector((2,1,0,1)) # reg tet edge
q = Qvector((2,1,1,0)) # ditto
r = Qvector((2,0,1,1)) # ditto
```

In [19]:
```
p.length()
```

Out[19]:
```
1.0
```

In [20]:
```
int(round(ivm_volume(o, p, q, r, np.int64 )))
```

```
[[0 0 0 0 1]
 [2 1 0 1 1]
 [2 1 1 0 1]
 [2 0 1 1 1]
 [1 1 1 1 0]]
```

Out[20]:
```
1
```

Now we can show our "closing the lid" volume operation, by scaling those 3 edges as if they were XYZ elementary basis vectors, for an analagous tetrahedron.

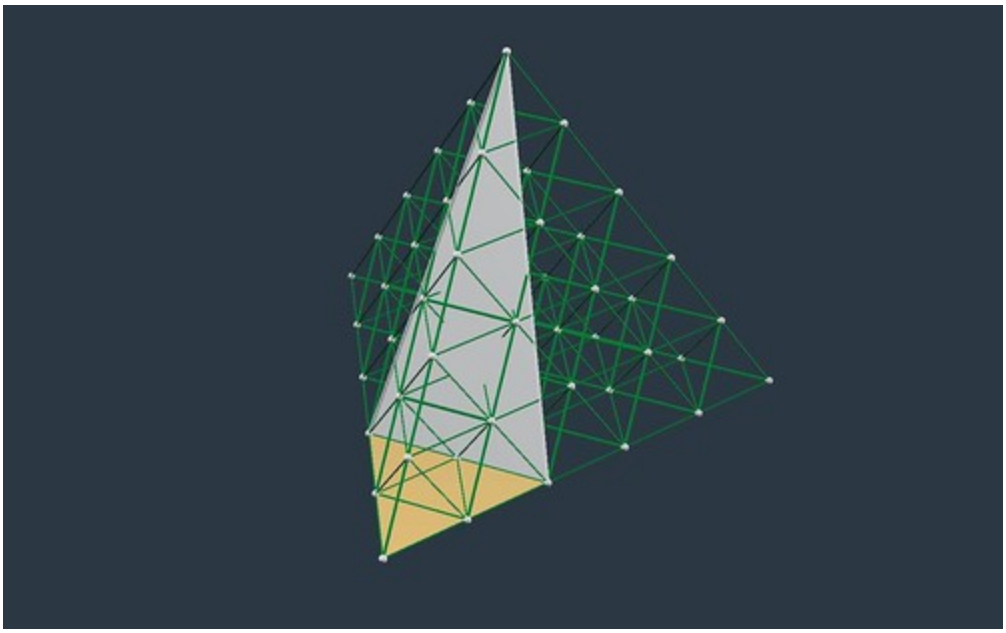


Figure 6: "closing the lid" 2 x 2 x 5 = 20

In [21]:
```
# edges from origin:  2, 2, and 5
tet = (o, 5*p, 2*q, 2*r)
int(round(ivm_volume(*tet, dtype= np.int64)))
```

```
[[ 0  0  0  0  1]
 [10  5  0  5  1]
```

```
 [ 4  2  2  0  1]
 [ 4  0  2  2  1]
 [ 1  1  1  1  0]]
20
```

Out[21]:

XYZ : IVM Conversion (S3)

What happens if we use a more conventional determinant, against the XYZ coordinates of each of simplicial Qvectors?

The formula in question is:

$$V_{xyz} = (1/6) \begin{vmatrix} ax & ay & az & 1 \\ bx & by & bz & 1 \\ cx & cy & cz & 1 \\ dx & dy & dz & 1 \end{vmatrix}$$

The four rows of the matrix corresponding to the four tetrahedron corners, with their XYZ coordinates.

As above, take the absolute value of that determinant if you need always positive values.

In [22]:

```
def xyz_volume(q0, q1, q2, q3, dtype=np.float):
    """
    Takes four Qvectors in D-edge units, outputs
    XYZ volume w/r to R-edge unit cube, D = 2R

    q0.xyz -> XYZ Vector v, v.xyz = (x,y,z) namedtuple
    """
    A = np.ones((4,4), dtype)
    # get xyz namedtuple from qvectors
    A[0,0:3] = (2 * q0).xyz.xyz
    A[1,0:3] = (2 * q1).xyz.xyz
    A[2,0:3] = (2 * q2).xyz.xyz
    A[3,0:3] = (2 * q3).xyz.xyz
    print(A)
    return abs(np.linalg.det(A))/6
```

In [23]:

```
S3 = np.sqrt(9/8) # synergetics constant
```

The thinking behind S3, a conversion constant, is that the edges of the XYZ unit volume cube, are half the length of the unit tetrahedron's.

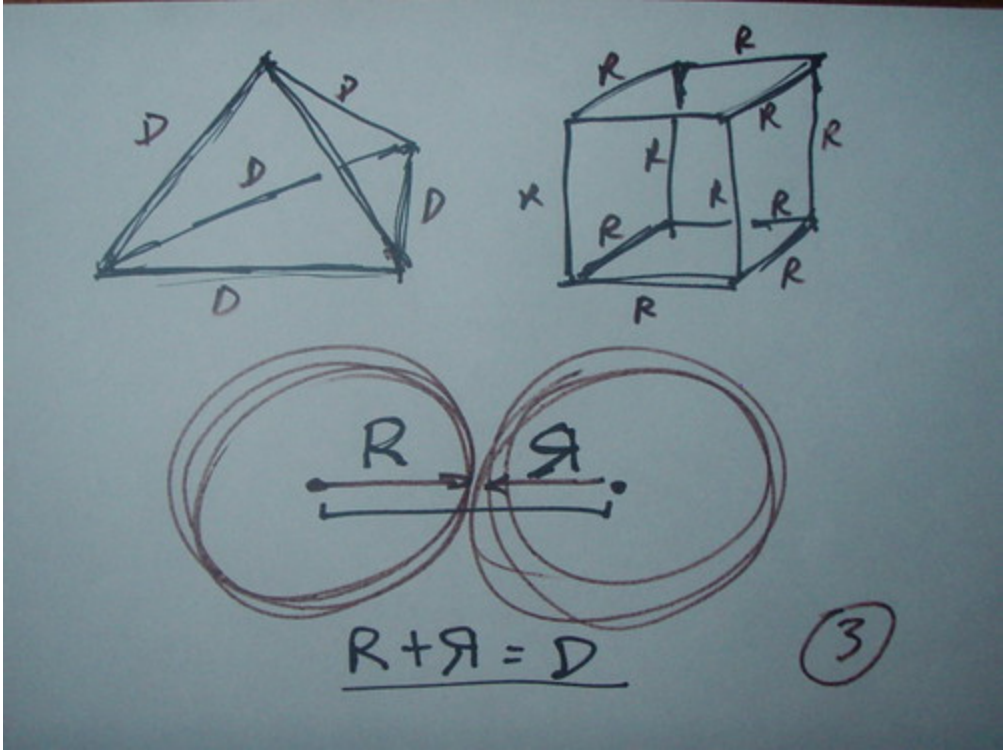


Figure 7: IVM vs XYZ Unit Volumes

In [24]:
```
# XYZ unit cube has edges R, whereas p, q, r were in D units (D = 2R)
vol = xyz_volume(2*q, o, 2*p, 5*r)  # same IVM unit tetrahedron, but in R edge cube units
vol
```

```
[[0.         2.82842712 2.82842712 1.        ]
 [0.         0.         0.         1.        ]
 [2.82842712 0.         2.82842712 1.        ]
 [7.07106781 7.07106781 0.         1.        ]]
```

Out[24]:
```
18.85618083164126
```

In [25]:
```
vol * S3  # i.e. 20, see Fig. 6
```

Out[25]:
```
19.999999999999993
```

In [26]:
```
vol = xyz_volume(a,b,c,d)
vol
```

```
[[ 0.70710678  0.70710678  0.70710678  1.        ]
 [-0.70710678 -0.70710678  0.70710678  1.        ]
 [-0.70710678  0.70710678 -0.70710678  1.        ]
 [ 0.70710678 -0.70710678 -0.70710678  1.        ]]
```

Out[26]:
```
0.9428090415820631
```

In [27]:
```
vol * S3 # i.e. 1
```

Out[27]:
```
0.9999999999999997
```

For further reading:

- Random Walk in the Matrix
- The Quadray Papers
- Quadray Coordinates on Wikipedia

- Cartesian Coordinates and simplicial coordinates