Compressed Unit Vectors

David Eberly
Geometric Tools, LLC
http://www.geometrictools.com/

Created: July 30, 2008

Contents

1 Introduction

In some applications, it is desirable to reduced the storage required by unit-length vectors. For example, a graphics application might want to provide an array of normal vectors to a vertex shader program. Each normal vector is naturally stored as a 3-tuple, each component a 32-bit floating-point number for a total of 96 bits. Instead, each normal vector can be compressed into 32-bit storage, 10 bits for each of 3 channels with 2 wasted bits. There are a few ways to do this. One involves taking a component in $[-1, 1]$ and mapping it to an unsigned integer i with $0 \le i \le 1023$. The three mapped components are stored in a 32-bit unsigned integer with the two high-order bits unused.

```
void CompressVariant1 (float x, float y, float z, unsigned int& value)
{
    unsigned int xc = (int)(1023.0f*(0.5f*(x + 1.0f)));
    unsigned int yc = (int)(1023.0f*(0.5f*(y + 1.0f)));
    unsigned int zc = (int)(1023.0f*(0.5f*(z + 1.0f)));
    value = xc | (yc << 10) | (zc << 20);
}
```

In a shader-based graphics system that supports a vertex normal format of `10-10-10-2`, the compressed normals are automatically expanded for the vertex shader. However, if you were to write this yourself, the following suffices; the graphics system probably does not perform the normalization that is shown here.

```
void DecompressVariant1 (unsigned int value, float& x, float& y, float& z)
{
    unsigned int xu = (value & 0x000003FF);
    unsigned int yu = (value & 0x000FFC00) >> 10;
    unsigned int zu = (value & 0x3FF00000) >> 20;
    x = 2.0f*xu/1023.0f - 1.0f;
    y = 2.0f*yu/1023.0f - 1.0f;
    z = 2.0f*zu/1023.0f - 1.0f;
    float invLength = 1.0f/sqrtf(x*x + y*y + z*z);
    x *= invLength;
    y *= invLength;
    z *= invLength;
}
```

An alternative is to map the normal components to 10-bit signed integers using two's complement. Each compressed channel i satisfies $-512 \le i \le 511$. For symmetry, we can require $|i| \le 511$. This is slightly more tedious to program directly using bit masks and shifting, because we must be careful in handling the sign bit. In C++, bit-field members allow the compiler to do this for us. The compression and decompression code is

```
class Compressed
{
public:
    int x : 10;
    int y : 10;
    int z : 10;
};

void CompressVariant2 (float x, float y, float z, Compressed& value)
{
    value.x = (int)(511.0f*x);
    value.y = (int)(511.0f*y);
    value.z = (int)(511.0f*z);
}
```

```
void DecompressVariant2 (Compressed value, float& x, float& y, float& z)
{
    x = ((float)value.x)/511.0f;
    y = ((float)value.y)/511.0f;
    z = ((float)value.z)/511.0f;
    float invLength = 1.0f/sqrtf(x*x + y*y + z*z);
    x *= invLength;
    y *= invLength;
    z *= invLength;
}
```

Naturally, a bit-field class using unsigned integers could have been used for the first variation of compression and decompression.

The preceding examples are simple, but they might not immediately indicate what the main issues are for compression and decompression of unit-length vectors. These issues are:

1. The speed of compression.
2. The speed of decompression.
3. The distribution of unit-length vectors on a circle (2D), on a sphere (3D), or on a hypersphere (4D).

The importance of each issue depends on your application. For example, if your goal is to reduce the disk storage requirements for a large set of vectors, you might not be concerned about the speed of compression. To quickly load the vectors from disk, you would be concerned about the speed of decompression. Another example involves sending vectors over a network, say, for a real-time game where each client needs to know physics data from all the other clients. Most likely you want to minimize bandwidth (compressed size must be small) and the compression and decompression must be fast.

The distribution of vectors might not be an apparent issue at first glance. However, if your vectors are normals and used for lighting of a terrain for which the up direction is $(0, 0, 1)$, a compression scheme that produces a high density of normals near $(1, 0, 0)$ but a low density near $(0, 0, 1)$ is most likely not desirable. More likely is that you want a fairly uniform distribution over the set of normals that your application uses.

This document describes a couple of algorithms for compressing unit-length vectors. I mentioned vectors in 2D, 3D, and 4D. The most common need for compression of unit-length vectors in 4D is for applications that use quaternions to represent rotations and orientations.

2 Unit Vectors in 2D

The compression algorithms in this document are motivated by considering unit-length vectors in 2D. In particular, the three main issues mentioned previously are illustrated here.

Given a unit-length vector (x, y), we will use two bits to represent which quadrant contains the vector. Effectively, the bits are sign bits. A bit-value of 0 denotes a nonnegative number and a bit-value of 1 denotes a negative number. For a vector (x, y), a bit-pair `00` indicates that $x \geq 0$ and $y \geq 0$. A bit-pair `10` indicates that $x < 0$ and $y \geq 0$. The remaining bits are used to compress the x-values for vectors in the first quadrant. The x-values are obtained by decompression and the y-values are computed by $y = \sqrt{1 - x^2}$. The sign bits are applied appropriately.

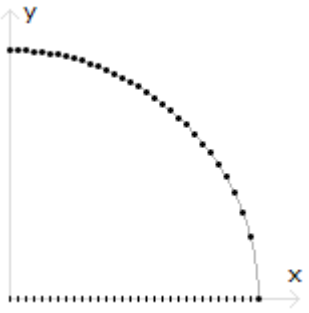
A simple scheme is the following. The number of bits, B, is specified by the user. The x-values in $[0,1]$ are discretized, producing at most $N = 2^B$ possibilities. The code shown next assumes that the two sign bits are the high-order bits of the result. It also assumes that the number of bits for the x-component (call these "mantissa bits") satisfies $B \leq 30$. This guarantees that the sign bits and mantissa bits fit into a 32-bit unsigned integer. The ideas can be extended to handle more than 30 bits, but the coding will be slightly more complicated because the bits are distributed across multiple unsigned integer members.

```
// The number of bits must satisfy 1 <= B <= 30.
#define B <user-specified number of bits>

const unsigned int N = (1 << (B-1));
const float Nm1 = (float)(N - 1);
const float invNm1 = 1.0f/Nm1;

class Compressed2
{
public:
    unsigned int mantissa : B;
    unsigned int ySign : 1;
    unsigned int xSign : 1;
};

void Compress2a (float x, float y, Compressed2& value)
{
    if (x >= 0.0f)
    {
        value.xSign = 0;
    }
    else
    {
        value.xSign = 1;
        x = -x;
    }

    if (y >= 0.0f)
    {
        value.ySign = 0;
    }
    else
    {
        value.ySign = 1;
        y = -y;
    }

    value.mantissa = (unsigned int)floorf(Nm1*x);
}

void Decompress2a (Compressed2 value, float& x, float& y)
{
    x = ((float)value.mantissa)*invNm1;
    y = sqrtf(1.0f - x*x);

    if (value.xSign > 0)
    {
        x = -x;
    }

    if (value.ySign > 0)
    {
        y = -y;
    }
}
```

Both compression and decompression are relatively fast, especially these days when the square-root and inverse-square-root functions have been optimized to use only a small number of cycles. However, the distribution of vectors on the arc of the circle in the first quadrant is not uniform. The density of vectors is

high for points near $(0, 1)$ but low for points near $(1, 0)$. Figure 2.1 illustrates this.

Figure 2.1 The nonuniform density of vectors for the simple compression algorithm in 2D.



The x-samples are uniformly spaced along the x-axis but the circle points are nonuniformly distributed.

We can compensate by compressing instead the angle θ for which $(x, y) = (\cos\theta, \sin\theta)$. The uniformly spaced angle samples are $\theta_i = (1 - i/(N-1))\pi/2$ for $0 \le i \le N-1$. The compression and decompression code is shown next.

```
const float piDivTwo = PI/2.0f;
const float twoDivPi = 2.0f/PI;

void Compress2b (float x, float y, Compressed2& value)
{
    if (x >= 0.0f) { value.xSign = 0; } else { value.xSign = 1; x = -x; }
    if (y >= 0.0f) { value.ySign = 0; } else { value.ySign = 1; y = -y; }
    float theta = acosf(x);
    value.mantissa = (unsigned int)floorf(Nm1*(1.0f - twoDivPi*theta));
}

void Decompress2b (Compressed2 value, float& x, float& y)
{
    float angle = piDivTwo*(1.0f - ((float)value.mantissa)*invNm1));
    x = cosf(angle);
    y = sinf(angle);
    if (value.xSign > 0) { x = -x; }
    if (value.ySign > 0) { y = -y; }
}
```

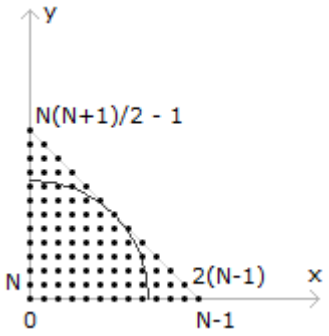
Observe that `Compress2b` is more expensive than `Compress2a` and `Decompress2b` is more expensive than `Compress2b` because of the additional trigonometric function calls. Assuming that `sqrtf` is faster than `sinf`, you may compute y in `Decompress2b` instead using a square-root call.

Obtaining a uniform distribution of vectors generally comes at the cost of more expensive compression and decompression.

3 Unit Vectors in 3D

A simple extension of `Compress2a` and `Decompress2a` is discussed next. Let (x, y, z) be a unit-length vector. We need three sign bits to determine which octant the point is in. We compress the (x, y) components in the first octant to a single index i of B bits and for which $0 \le i \le M - 1$ for some $M > 0$. A triangular array of indices is used, as shown in Figure 3.1.

Figure 3.1 The triangular array of indices that are used for compression of (x, y).



The row of points on the x-axis has N points numbered from 0 to $N-1$. The adjacent row has $N-1$ points numbered from N to $2(N-1)$. The numbering continues for each row, terminating in the last row that has 1 point. The number of indices is

$$1 + 2 + \cdot + (N-1) + N = \sum_{k=1}^{N} k = \frac{N(N+1)}{2}$$

which explains why the last index is $N(N+1)/2-1$ in the figure. For B bits per index, choose N to be the largest integer for which

$$\frac{N(N+1)}{2} < 2^B$$

For example, to store a 96-bit floating-point unit-length vector (32 bits per channel) in a 16-bit quantity, we need 3 bits for the signs. The remaining 13 bits are used for the mantissa, so $B = 13$ ($2^B = 8192$) and $N = 127$ ($N(N+1)/2 = 8128$).

The triangle points cover the quarter circle, but only the indices lying inside or on the circle are used. The distribution of the triangle points in the plane is uniform, but the distribution on the sphere is not uniform, as in the 2-dimensional case. The points have higher density near the north pole $(0, 0, 1)$ with decreasing density as you move towards the equator $z = 0$.

The compression algorithm is relatively simple. The (x, y) component of the unit-length vector is mapped into the triangle array as follows. Notice that the last triangle point on the x-axis occurs at $\sqrt{2}$. Thus, we need to map $x \in [0, \sqrt{2}]$ to the integer-valued $\bar{x} \in \{0, \ldots, N-1\}$; that is, $\bar{x} = \text{floor}((N-1)\sqrt{0.5}x)$. A similar argument applies along the y-axis, so we need to map $y \in [0, \sqrt{2}]$ to the integer-valued $\bar{y} \in \{0, \ldots, N-1\}$.

In the row $\bar{y} = 0$, the index is

$$i_0 = \bar{x}$$

In the next row $\bar{y} = 1$, the indexing starts at one more than the last index in the previous row. In this case,

$$i_1 = \bar{x} + N = \bar{x} + \frac{1(2N + 1 - 1)}{2}$$

Generally, the number of indices in rows 0 through $\bar{y} - 1$ is

$$N + (N - 1) + \cdots + (N - \bar{y}) = \sum_{k=0}^{N-\bar{y}} k = \frac{\bar{y}(2N + 1 - \bar{y})}{2}$$

The indexing in row $\bar{y}$ starts with this value offset by the $\bar{x}$ value,

$$i = \bar{x} + \frac{\bar{y}(2N + 1 - \bar{y})}{2} \tag{1}$$

Given a $\bar{x}$ and a $\bar{y}$, it is easy to compute i. Given an i, it is more difficult to extract $\bar{x}$ and $\bar{y}$. One possibility is to use a binary search on a sorted array of indices

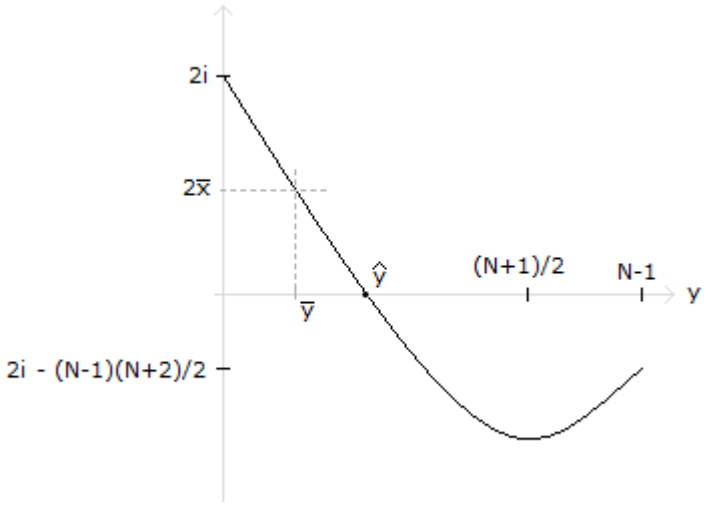
$$i_0 = 0, i_1 = N, \ldots, i_{N-1} = (N - 1)(N + 2)/2 = N(N + 1)/2 - 1$$

The search produces an index $\bar{y}$ for which $i_{\bar{y}} \le i < i_{\bar{y}+1}$; then $\bar{x} = i - i_{\bar{y}}$. Another possibility is to treat Equation (1) as a quadratic equation in $\bar{y}$,

$$\bar{y}^2 - (2N + 1)\bar{y} + 2i = 2x \tag{2}$$

The graph in Figure 3.2 illustrates.

Figure 3.2 The graph of the left-hand side of Equation (2).



The index $\bar{y}$ is the largest integer smaller than the smallest real-valued root $\hat{y}$ to $y^2 - (2N+1)y + 2i = 0$. That is,

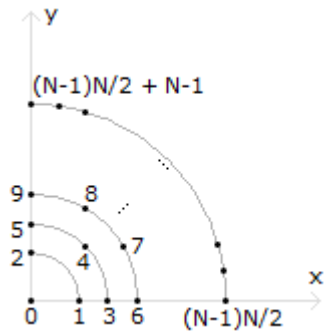
$$\bar{y} = \left\lfloor \frac{(2N+1) - \sqrt{(2N+1)^2 - 8i}}{2} \right\rfloor$$

The pseudocode is listed next, where for simplicity it is assumed that the number of bits B is no larger than 29, in which case the compressed result fits into a 32-bit unsigned integer.

```
// The number of bits must satisfy 1 <= B <= 29.
#define B <user-specified number of bits>
const unsigned int N = <largest value for which N(N+1)/2 is smaller than 2^B>;
const unsigned int twoNp1 = 2*N+1;
const float twoNp1Sqr = twoNp1*twoNp1;
const float factor = ((float)(N - 1))*sqrtf(0.5f);
const float invFactor = 1.0f/factor;

class Compressed3
{
public:
    unsigned int mantissa : B;
    unsigned int zSign : 1;
    unsigned int ySign : 1;
    unsigned int xSign : 1;
};

void Compress3a (float x, float y, float z, Compressed3& value)
{
    if (x >= 0.0f) { value.xSign = 0; } else { value.xSign = 1; x = -x; }
    if (y >= 0.0f) { value.ySign = 0; } else { value.ySign = 1; y = -y; }
    if (z >= 0.0f) { value.zSign = 0; } else { value.zSign = 1; z = -z; }
    unsigned int xu = (unsigned int)floorf(factor*x);
    unsigned int yu = (unsigned int)floorf(factor*y);
    value.mantissa = xu + yu*(twoNp1 - yu)/2;
}

void Decompress3a (Compressed3 value, float& x, float& y, float& z)
{
    float discr = twoNp1Sqr - 8.0f*value.mantissa;
    unsigned int yu = (unsigned int)floorf(0.5f*(twoNp1 - sqrtf(discr)));
    unsigned int xu = value.mantissa - yu*(twoNp1 - yu)/2;
    x = invFactor*xu;
    y = invFactor*yu;
    z = sqrtf(1.0f - x*x - y*y);
    if (value.xSign > 0) { x = -x; }
    if (value.ySign > 0) { y = -y; }
    if (value.zSign > 0) { z = -z; }
}
```

In an attempt to obtain an approximately uniform distribution, and also to use all the indices in the triangular array, the next algorithm is quite useful. Figure 3.3 shows the mapping of points to the triangular array.

Figure 3.3 An approximately uniform distribution of points that uses all the available indices in the triangle array.



There are N circular shells labeled 0 (the origin) through $N-1$ (the unit circle). Shell s has $s+1$ angular samples.

Spherical coordinates for the unit-length vectors in the first octant are

$$x = \cos\theta \sin\phi, \quad y = \sin\theta \sin\phi, \quad z = \cos\phi$$

where $0 \le \theta \le \pi/2$ and $0 \le \phi \le \pi/2$. We discretize θ and ϕ. First, $x^2 + y^2 = 1 - z^2 = 1 - \cos^2\phi = \sin^2\phi$, so

$$\phi = \text{asin}(\sqrt{x^2 + y^2}) \in [0, \pi/2]$$

The circular shell index is

$$s = \text{Round}\left((N-1)\frac{2}{\pi}\text{asin}(\sqrt{x^2 + y^2})\right) \in \{0, 1, \ldots, N-1\}$$

where the Round function returns the nearest integer to its argument. The starting index on the shell is $s(s+1)/2$. The angular index is

$$a = \text{Round}\left(s\frac{2}{\pi}\text{atan2}(y, x)\right) \in \{0, 1, \ldots, s\}$$

where $\text{atan2}(y, x) \in [0, \pi/2]$. The combined index is

$$i = a + \frac{s(s+1)}{2} \in \{0, 1, \ldots, N(N+1)/2 - 1\}$$

The pseudocode for compression and decompression is

```
// The number of bits must satisfy 1 <= B <= 29.
#define B <user-specified number of bits>
const unsigned int N = <largest value for which N(N+1)/2 is smaller than 2^B>;
const float piDivTwo = PI/2.0f;
const float twoDivPi = 2.0f/PI;
const float factor = ((float)(N - 1))*twoDivPi;
```

```
const float invFactor = 1.0f/factor;

class Compressed3
{
public:
    unsigned int mantissa : B;
    unsigned int zSign : 1;
    unsigned int ySign : 1;
    unsigned int xSign : 1;
};

void Compress3b (float x, float y, float z, Compressed3& value)
{
    if (x >= 0.0f) { value.xSign = 0; } else { value.xSign = 1; x = -x; }
    if (y >= 0.0f) { value.ySign = 0; } else { value.ySign = 1; y = -y; }
    if (z >= 0.0f) { value.zSign = 0; } else { value.zSign = 1; z = -z; }
    unsigned int s = (unsigned int)floorf(factor*asinf(sqrtf(x*x+y*y)));
    if (s > 0)
    {
        unsigned int a = (unsigned int)floorf(s*twoDivPi*atan2f(y,x));
        value.mantissa = a + s*(s+1)/2;
    }
    else
    {
        value.mantissa = 0;
    }
}

void Decompress3b (Compressed3 value, float& x, float& y, float& z)
{
    float discr = 1.0f + 8.0f*value.mantissa;
    unsigned int s = (unsigned int)floorf(0.5f*(-1.0f + sqrtf(discr)));
    if (s > 0)
    {
        unsigned int a = value.mantissa - s;
        float theta = piDivTwo*a/s;
        float phi = invFactor*s;
        float sinPhi = sinf(s);
        x = cosf(theta)*sinPhi;
        y = sinf(theta)*sinPhi;
        z = cosf(phi);
    }
    else
    {
        x = 0;
        y = 0;
        z = 1;
    }

    if (value.xSign > 0) { x = -x; }
    if (value.ySign > 0) { y = -y; }
    if (value.zSign > 0) { z = -z; }
}
```

The extraction of s from i is similar to the previous algorithm, where we analyzed the roots of a quadratic equation (this time in s).

4 Unit Vectors in 4D

An extension of `Compress3a` and `Decompress3a` is discussed next. Let (x, y, z, w) be a unit-length vector. We need four sign bits to determine which hextant the point is in. We compress the (x, y, z) components in the first hextant to a single index i of B bits and for which $0 \le i \le M - 1$ for some $M > 0$. A tetrahedral array of indices is used. The spherical domain of the first hextant of the hypersphere is embedded in the

tetrahedron, much like the circular domain of the first octant of the sphere was embedded for 3-tuples; see Figure 3.1.

Let $(\bar{x}, \bar{y}, \bar{z})$ be the integer-valued coordinate indices that are used in the tetrahedron index i. Along each coordinate axis we have N tetrahedron points. The index tuples $(0,0,0)$, $(N-1,0,0)$, $(0,N-1,0)$, and $(0,0,N-1)$ are the tetrahedron vertices and occur at the real-valued locations $(0,0,0)$, $((N-1)/\sqrt{3},0,0)$, $(0,(N-1)/\sqrt{3},0)$, and $(0,0,(N-1)/\sqrt{3})$.

The triangle face of the tetrahedron in the $\bar{z}=0$ plane has $N(N+1)/2$ indices indexed by

$$i_0 = \bar{x} + \frac{\bar{y}[2N+1-\bar{y}]}{2}, \;\; \bar{x}+\bar{y} \le N-1$$

The $\bar{z}=1$ plane has $(N-1)(N)/2$ indices indexed by

$$i_1 = \bar{x} + \frac{\bar{y}[2(N-1)+1-\bar{y}]}{2} + \frac{(N-1)[2N+1-(N-1)]}{2} + 1, \;\; \bar{x}+\bar{y} \le N-2$$

where the last term on the right-hand side of the i_1 equation is the maximum value (plus 1) for i_0. Generally, the index for the $\bar{z}$ plane is

$$i_{\bar{z}} = \bar{x} + \frac{\bar{y}[2(N-\bar{z})+1-\bar{y}]}{2} + \sum_{k=0}^{\bar{z}-1} \left(\frac{(N-1-k)[2(N-k)+1-(N-1-k)]}{2} + 1 \right), \;\; \bar{x}+\bar{y} \le N-1-\bar{z}$$

This simplifies to

$$i_{\bar{z}} = \bar{x} + \frac{\bar{y}[2(N-\bar{z})+1-\bar{y}]}{2} + \frac{\bar{z}}{6}\left(\bar{z}^2 - 3(N+1)\bar{z} + (3N^2+6N+2)\right)$$

The total number of indices is the sum of the number of indices for the $\bar{z}$ triangles,

$$\sum_{\bar{z}=0}^{N-1} \frac{(N-\bar{z})(N+1-\bar{z})}{2} = \frac{N(N+1)(2N+1)}{6}$$

To have B bits of compression, choose N to be the largest integer for which

$$\frac{N(N+1)(2N+1)}{6} < 2^B$$

Compression is straightforward. As before, I assume the simplest case of storing the compressed value in a 32-bit unsigned integer. Decompression can use a binary search in the ordered array of partial sums of the number of points in the triangle slices, similar to what was described for 3 dimensions. Alternatively, the trick involving quadratic equations can be extended to cubic equations, but the computations involved in finding the smallest root of a cubic polynomial is probably not worth the effort. I leave it as an exercise. Hint: Find the largest integer $\bar{z}$ smaller than the minimum root of $z(z^2-3(N+1)z+(3N^2+6N+2))-6i=0$. Subtract $\bar{z}(\bar{z}^2-3(N+1)\bar{z}+(3N^2+6N+2))/6$ from i to obtain an equation in $\bar{x}$ and $\bar{y}$. Use the quadratic equation trick to extract $\bar{y}$ and $\bar{x}$.

```
// The number of bits must satisfy 1 <= B <= 28.
#define B <user-specified number of bits>
const unsigned int N = <largest value for which N(N+1)(2N+1)/6 is smaller than 2^B>;
```

```
const float factor = ((float)(N - 1))/sqrtf(3.0f);
const float invFactor = 1.0f/factor;
const float poly1 = 3*(N + 1);
const float poly2 = 3N*(N + 2) + 2;
const float table[N] =
{
    0,
    (N-1)(N-2)/2,
    :
    <remaining partial sums>
};

unsigned int GetZBar (unsigned int mantissa)
{
    <implement a binary search in 'table'>;
}

class Compressed4
{
public:
    unsigned int mantissa : B;
    unsigned int wSign : 1;
    unsigned int zSign : 1;
    unsigned int ySign : 1;
    unsigned int xSign : 1;
};

void Compress4a (float x, float y, float z, float w, Compressed4& value)
{
    if (x >= 0.0f) { value.xSign = 0; } else { value.xSign = 1; x = -x; }
    if (y >= 0.0f) { value.ySign = 0; } else { value.ySign = 1; y = -y; }
    if (z >= 0.0f) { value.zSign = 0; } else { value.zSign = 1; z = -z; }
    if (w >= 0.0f) { value.wSign = 0; } else { value.wSign = 1; w = -w; }
    unsigned int xu = (unsigned int)floorf(factor*x);
    unsigned int yu = (unsigned int)floorf(factor*y);
    unsigned int zu = (unsigned int)floorf(factor*z);
    unsigned int ycomp = yu*(2*(N-zu)+1-yu)/2;
    unsigned int zcomp = zu*(zu*(zu - poly1) + poly2)/6;
    value.mantissa = xu + ycomp + zcomp;
}

void Decompress3a (Compressed3 value, float& x, float& y, float& z)
{
    unsigned int zu = GetZBar(value.mantissa);
    unsigned int zcomp = zu*(zu*(zu - poly1) + poly2)/6;
    unsigned int adjusted = value.mantissa - zcomp;
    unsigned int tmp = 2*(N - zu) + 1;
    float discr = tmp*tmp - 8.0f*adjusted;
    unsigned int yu = (unsigned int)floorf(0.5f*(tmp - sqrtf(discr)));
    unsigned int xu = adjusted - yu*(tmp - y)/2;
    x = invFactor*xu;
    y = invFactor*yu;
    z = invFactor*zu;
    w = sqrtf(1.0f - x*x - y*y - z*z);
    if (value.xSign > 0) { x = -x; }
    if (value.ySign > 0) { y = -y; }
    if (value.zSign > 0) { z = -z; }
}
```

We may similarly extend the ideas from 3D to have an approximately uniform distribution of points and to use all the tetrahedron indices. The idea is to have N spherical shells in the (x, y, z) domain. Each shell has its set of tetrahedron indices, all such indices within a plane of the form $\bar{x} + \bar{y} + \bar{z} = j$, where $0 \le j \le N - 1$. The shell index is

$$s = \text{Round}\left((N-1)\frac{2}{\pi}\text{asin}(x^2 + y^2 + z^2)\right) \in \{0, 1, \ldots, N-1\}$$

Once you compute the closest shell for the (x, y, z) component of the unit-vector, you know the j value for

which the three "vertices" of the shell are in the plane $\bar{x}+\bar{y}+\bar{z}=j$. Project the closest shell point (x', y', z') onto this plane and locate the nearest triangle point t within that plane. This gives you an index $s+t$ into the entire tetrahedron array. Constructing this mapping is the tediuos part of the algorithm, which I leave as an exercise for now. (Once I regain some spare time, I'll list the implementation here.)