ICC profile internal mechanics

Introduction

So, profile internal mechanics. What am I going to tell you? If you take a look on the agenda, this small tutorial covers a little bit of everything. Including some items that is hard to imagine how relates with color science. In fact I'm going to explore little about color science, but a lot on the engineering aspect of creating and interpreting ICC profiles. This is about math too. Not complex math, with lots of arcane equations but just simple linear algebra. I am going to detail tricks and traps on how ICC profiles may be encoded internally. How to deal with the issues the 8 or 16 bits encoding may have. And how to maintain precision as well.

Ok, you may argue, I am going to use floating point always when generating or interpreting my ICC profiles, so I have not to worry about all that. Really? Please consider this small piece of C code:

```
int main(void)
{
	int i;
	float a = 0;

	for (i=0; i < 10; i++)
		a = a + 0.1;

	if (a == 1.0)
		printf("all is ok");
	else
		printf("Ops, what the?!?");

	return 0;
}
```

I am just adding 0.1 ten times, so it should work: Let's run it. Well, it doesn't work. Ok, maybe that's because I used float? Let's use double instead. It doesn't work either. Sorry, maybe most of you already know this. It's an old trick, but I hope you get my point. For getting the most of any tool, you have to understand the internals of it, else you may be trapped by completely unexpected and often wrong results.

Now I will introduce some basic techniques. Those are easy, and are the foundation of all complex things we are going to discuss latter.

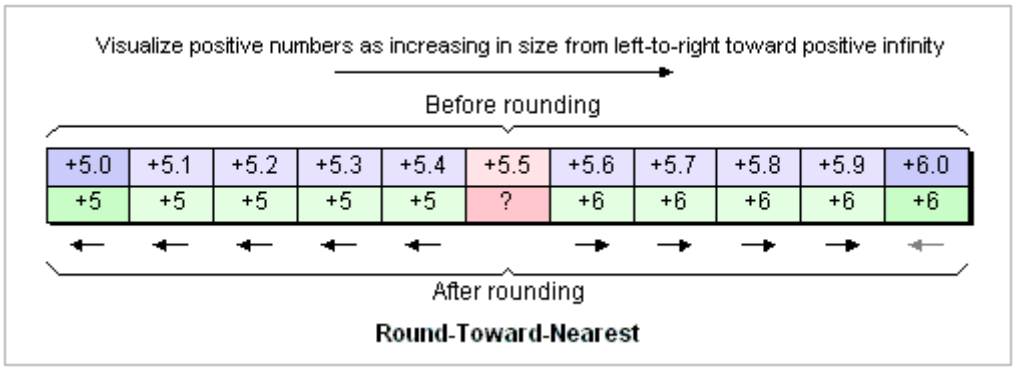
Rounding

First technique I would like to introduce is rounding. In general, Rounding is the process of reducing the number of significant digits in a number. The result of rounding is a "shorter" number having fewer non-zero digits yet similar in magnitude. The result is less precise but easier to use.

ICC profiles do have a severe dependency on rounding. Why? Because ICC profiles are computer artifacts: binary digital files and as such are subjected to quantization. Quantization means a fixed amount of bits, so round-off errors happen. Let's see some ways to round.

Round-Toward-Nearest

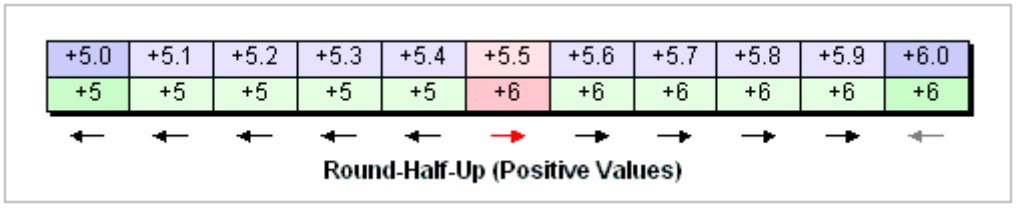
As its name suggests, this algorithm rounds towards the nearest significant value. In many ways, this is the most intuitive of the various rounding algorithms, because values such as 5.1, 5.2, 5.3, and 5.4 will round down to 5, while values of 5.6, 5.7, 5.8, and 5.9 will round up to 6:



But what should we do in the case of a "half-way" value such as 5.5? Well, there are two obvious options: we could round it up to 6 or we could round it down to 5; these schemes, which are introduced below, are known as Round-Half-Up and Round-Half-Down, respectively.

Round-Half-Up (Arithmetic Rounding)

This algorithm, which may also be referred to as arithmetic rounding, is the one that we typically associate with the concept of rounding that we learned at school. In this case, a "half-way" value such as 5.5 will round up to 6:

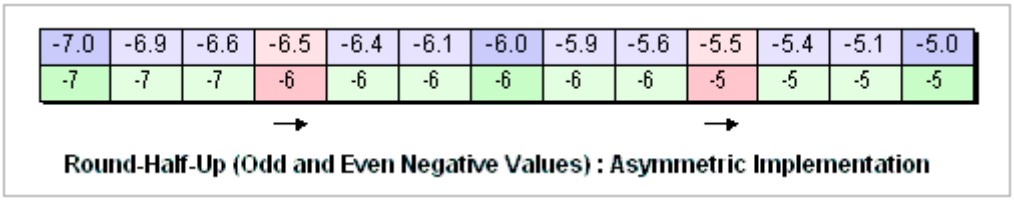


One way to view this is that, at this level of precision and for this particular example, we can consider there to be ten values that begins with a "5" in the most-significant place (5.0, 5.1, 5.2, 5.3, 5.4, 5.5, 5.6, 5.7, 5.8, and 5.9).

On this basis, it intuitively makes sense for five of the values to round down and for the other five to round up; that is, for the five values 5.0 through 5.4 to round down to 5, and for the remaining five values 5.5 through 5.9 to round up to 6.

The tricky point with this *round-half-up* algorithm arrives when we come to consider negative numbers. There is no problem in the case of the values like -5.1, -5.2, -5.3, and -5.4, because these will all round to the nearest integer, which is -5. Similarly, there is no problem in the case of values like -5.6, -5.7, -5.8, and -5.9, because these will all round to -6. The problem arises in the case of "half-way" values like -5.5 and -6.5 and our definition as to what "up" means in the context of "round-half-up." Based on the fact that positive values like +5.5 and +6.5 round up to +6 and +7, respectively, most of us would intuitively expect their negative equivalents of -5.5 and -6.5 to round to -6 and -7, respectively. In this case, we would say that our algorithm was *symmetric* (with respect to zero) for positive and negative values.

The point is that some applications (and some mathematicians) would regard "up" as referring to positive infinity. Based on this, -5.5 and -6.5 would actually round to -5 and -6, respectively, in which case we would class this as being an *asymmetric* (with respect to zero) implementation of the *round-half-up* algorithm:



Round-Half-Up (Odd and Even Negative Values) : Asymmetric Implementation

The problem is that different applications may treat things differently. For example, the round method of the Java Math Library provides an asymmetric implementation of the round-half-up algorithm, while the round function in MATLAB® provides a symmetric implementation. Just for giggles and grins, the round function in Visual Basic for Applications 6.0 actually implements the Round-Half-Even algorithm.

Round-Half-Even

If half-way values are always rounded in the same direction (for example, if +5.5 rounds up to +6 and +6.5 rounds up to +7, as is the case with the Round-Half-Up algorithm presented earlier), the result can be a bias that grows as more and more rounding operations are performed. One solution toward minimizing this bias is to sometimes round up and sometimes round down.

In the case of the round-half-even algorithm, which is often referred to as "Bankers Rounding" because it is commonly used in financial calculations, "half-way" values are rounded toward the nearest even number. Thus, +5.5 will round up to +6 and +6.5 will round down to +6:

+5.0	+5.1	+5.4	+5.5	+5.6	+5.9	+6.0	+6.1	+6.4	+6.5	+6.6	+6.9	+7.0
+5	+5	+5	+6	+6	+6	+6	+6	+6	+6	+7	+7	+7

Round-Half-Even (Odd and Even Positive Values)

This algorithm is, by definition, symmetric for positive and negative values, so both -5.5 and -6.5 will round to the nearest even value, which is -6:

-7.0	-6.9	-6.6	-6.5	-6.4	-6.1	-6.0	-5.9	-5.6	-5.5	-5.4	-5.1	-5.0
-7	-7	-7	-6	-6	-6	-6	-6	-6	-6	-5	-5	-5

Round-Half-Even (Odd and Even Negative Values)

In the case of data sets that feature a relatively large number of "half-way" values, the round-half-even algorithm performs significantly better than the Round-Half-Up scheme in terms of total bias. It is for this reason that the use of round-half-even is a legal requirement for many financial calculations around the world.

Converting between domains

Next basic technique is how to convert between domains. This somehow controversial, and I really don't understand why. Ok, let's assume we have two different domains. Say 0...1 and 0...255. Both beginning with zero. If we want to convert a value from source domain to destination domain it is as easy as:

$$Y = \frac{Destination\ domain\ max}{source\ domain\ max} * X$$

In our example, 0.5 in domain 0..1 is converted to:

$$\frac{255}{1} * 0.5 = 127.5$$

Easy uh? Ok, please consider converting from 8 bits to 16 bits. In this case we go from 0...255 to 0...65535.

$$\frac{65535}{255} * X = Y$$

So far so good, but wait… the fraction gives 257! So actually, to convert a 8 bits value to 16 bits you need to multiply times 257. Ok, that is the same as:

$$X * 257 = X * 256 + X = (X \ll 8) | X$$

That's fast, as it only uses a shift left plus bitwise or operators. But what happens in the inverse situation? How to convert from 16 bits to 8 bits? Yes, you guessed it. You have to divide by 257. Ops, a division, that's slow… how to do it in a fast way? Many people use a simple shift right X >> 8, which is wrong, as would be same as dividing by 256. Here is a trick to do the 257 division fast. It includes arithmetic rounding too:

$$Y = (((X * 0xFF01) + 0x800000) \gg 24) \& 0xFF$$

Slightly more complex is the situation where both domains do not begin with zero. The expression can be computed as a line that goes across two points and the result is still very simple:

$$y = \alpha * x + \beta$$

$$\alpha = \frac{Dest\,max - Dest\,min}{Src\,max - Src\,min}$$

$$\beta = Dest\,min - \alpha * Src\,min$$

For example, from -127...+128 to 0...65535:

$$a = \frac{65535}{128 - (-127)} = 257$$

$$b = 0 - 257 * (-127) = 32639$$

$$y = 257 * x - 32896$$

Let's check the end points to corroborate all is working as expected:

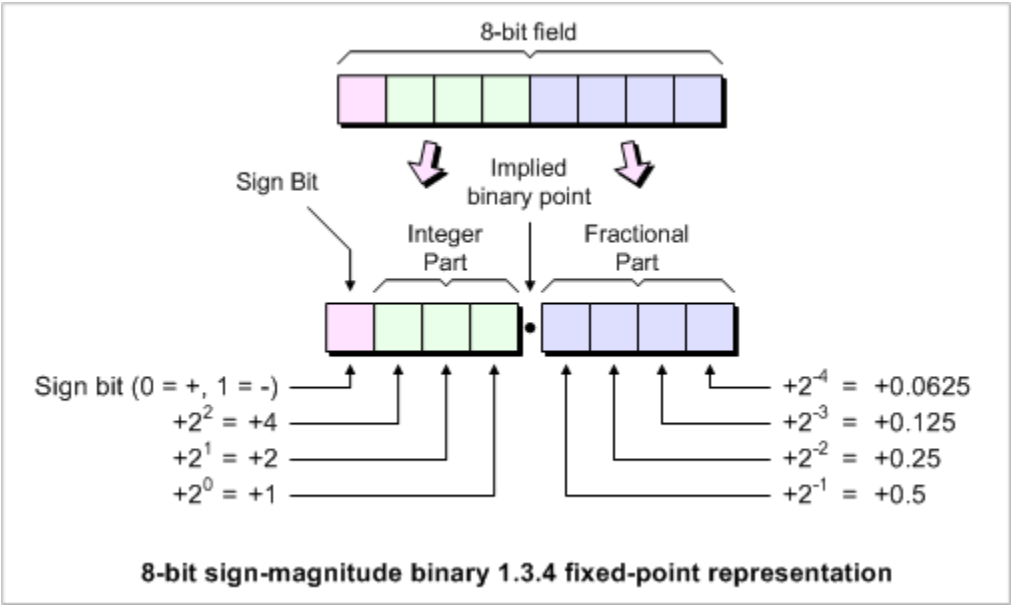
$$Y(-127) = 257 * (-127) + 32639 = 0$$

$$Y(+128) = 257 * 128 + 32639 = 65535$$

That still may sound too simple, but … just hold on to see how complicated things may become just combining those simple tools.

Fixed point

Next item I'm going to explain what is known as fixed point. The idea behind fixed point math is that we pretend that there is a decimal point where really is not.



8-bit sign-magnitude binary 1.3.4 fixed-point representation

Let's take for example this approximation of PI

3.14159

A fixed-point number is an integer that represents a number consisting of a whole part (e.g. 3) and a fractional part (e.g. .14159). The format of a fixed-point number is usually described as "*m*.*n*", where *m* is the number of bits in the whole part and the *n* is the number of bits in the fractional part. *m* + *n* is the total number of bits. For example, "8.24" means a 32-bit integer with an 8-bit whole part and a 24-bit fractional part. Fixed point formats may have sign too.

To convert a number to fixed point you just multiply it by 2^n. To convert the fixed point number back, divide by 2^n So, to represent 3.14159 as a 8.24 fixed point number, we multiply it by 2^{24} Result is 52707134. To return back we divide by $2^{24.}$ It gives 3.141589999

Fixed point math

If you do all of you math without the decimal point, and then add the decimal point, you get the same result.

For example:

$$1.2 * 3.4 = 4.08$$

If I remove the decimal point:

$$12 * 34 = 408$$

That implies some interesting properties. Suppose A, B, and C are the fixed-point versions of a, b, and c.

$$A = a * 2^n$$

$$B = b * 2^n$$

Addition: if c = a + b, then

$$C = (a + b) * 2^n = (a * 2^n) + (b * 2^n)$$

$$C = A + B$$

In other words, to add two fixed point numbers, you just add them. Subtraction works the same way.

Multiplication: if c = a * b, then

$$C = (a * b) * 2^n = (a * 2^n) * \frac{(b * 2^n)}{\square} 2^n$$

$$C = A * B \frac{\square}{\square} 2^n$$

In other words, to multiply two fixed point numbers, you multiply them and then divide by 2^n (or shift right *n* bits). Note that overflow is a problem that has to be dealt with.

Division: if c = a / b, then

$$C = \left(a \frac{\square}{\square} b\right) * 2^n = \frac{(a * 2^n)}{\square} (b * 2^n) * 2^n$$

$$C = A \frac{\square}{\square} B * 2^n$$

In other words, to divide two fixed point numbers, you divide them and then multiply by 2^n (or shift left *n* bits). Note that underflow is a very serious problem which is usually dealt with by rearranging the order of operations like this:

$$C = A * 2^n \frac{\square}{\square} B$$

Again, you have an overflow problem that has to be dealt with. The easiest and safest way to deal with overflow is to use a larger integer size to store intermediate values. For example, you might use 64-bit numbers to do 8.24 fixed-point math.

The drawback is that it can be slower, or there might not be a larger integer size available. You could also be clever with rearranging the math (like I did with the divide operation), but that could have drawbacks.

Now that we know how cool fixed point format is, let's see the issues it has. One not so evident: an exact number in decimal notation is not necessarily exact in fixed point binary. Let's see a practical example.

Just consider we try to encode gamma 2.2 in a RedTRC tag. This particular tag uses 8.8 fixed point type. The encoded number may be computed by multiplying times 2^8 (256), which give 2.2 * 256 = 563.2, ops! You got some fractional part! Ok, we know how to round so let's round it, you get 563. Wait, rounding means something has been lost… let's check what happens if I try to recover the original value.

$$563 \frac{\square}{\square} 256 = 2.1992$$

So… 2.2 cannot be encoded in 8.8 fixed point format! This is a real problem, and I guess this is the reason AdobeRGB is defined to have a gamma of exactly 2.1992 instead of 2.2 How to fix that?

You can't, but In V4 there are parametric curves using 15.16 encoding for parameters. In this case 2.2 would turn to 2.2 * 65536 = 144179.2, still not exact but the roundtrip is 2.199997, pretty close to 2.2 the learning is: use parametric curves whatever possible.

Interpolation

Interpolation is a sort of "educated guess". Basically it is trying to get a value based on how the neighboring behaves. If there is consensus, interpolation works great. If not, well… Linear interpolation is quick and easy, but it is not very precise.

The simplest form of interpolation is across a single segment, holding two values. Again this is just a linear algebra problem:

$$Y = \alpha x + \beta$$

We know two points (X_a, Y_a) and (X_b, Y_b) . So, solving the system

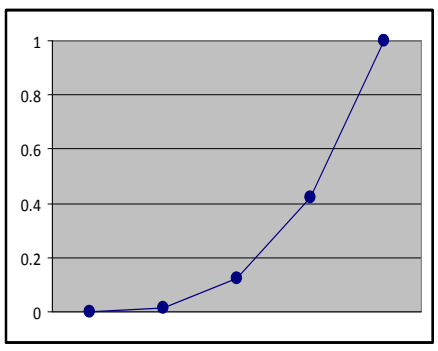
$$Y_a = \alpha X_a + \beta$$

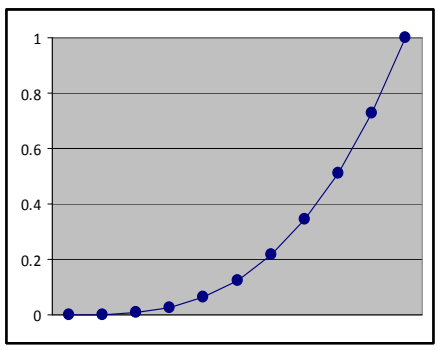
$$Y_b = \alpha X_b + \beta$$

Which give:

$$y = y_a + \frac{(x - x_a)(y_b - y_a)}{(x_b - x_a)}$$

Of course this only works on straight lines. What if we want to approximate a non-linear function? We could break non-linear curves into sets of linear trams. For each tram, we have two endpoints, so we can interpolate the value of any 1D function using this technique. On more nodes, more precise is the approximation.





To interpolate using this tables, for any given input, we have first to isolate which are the surrounding nodes. Hard? Not really, we have a value that goes from 0…65535 and want a value that goes from 0 to the maximum number of nodes. Sounds familiar? Yep, it is just a matter of changing domain. And here comes one of the myths about profiles: some number of nodes has inherent advantages. Let's see. We index in 16 bits and want to convert to nodes n-1, since there is one final node at the very end.

$$y = \frac{n-1}{65535}x = \frac{n-1}{3*5*17*257}x$$

That means for n-1 = 3, 5, 17, 257 or any combination of these we have some advantages. For example on n=256 (3*5*17 + 1), we have:

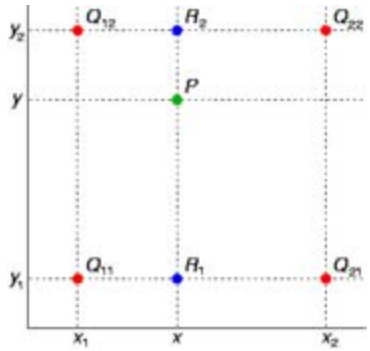
$$y = \frac{256-1}{255*257}x = \frac{x}{257}$$

So, for 256 nodes, that's same as converting our 16 bits to 8 bits and then providing a node for each of the obtained 256 entries.

Other interesting combinations are 2 points (for a straight line), 18 points (17+1), 52 points (17*3+1), 86 points (17*5+1), etc. Each one of these node configurations has an exact number of values affected by each grid point.

Bilinear interpolation

A logical extension of interpolation of 1D would be to use in 2D environments. But at this point, the number of nodes that may influence the point are not just two but four:

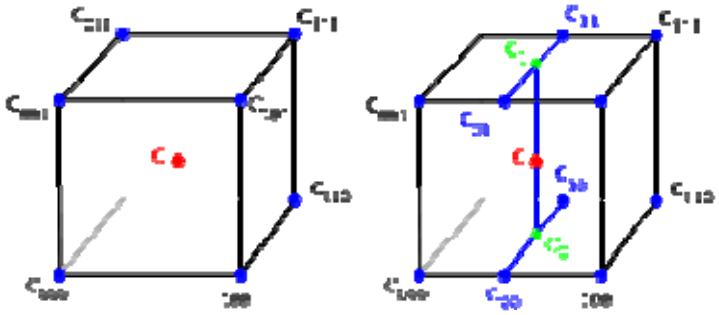


The key idea is to perform linear interpolation first in one direction, and then in the other direction. Suppose that we want to find the value of the unknown function f at the point P = (x, y). It is assumed that we know the value of f at the four points Q11 = (x1, y1), Q12 = (x1, y2), Q21 = (x2, y1), and Q22 = (x2, y2).

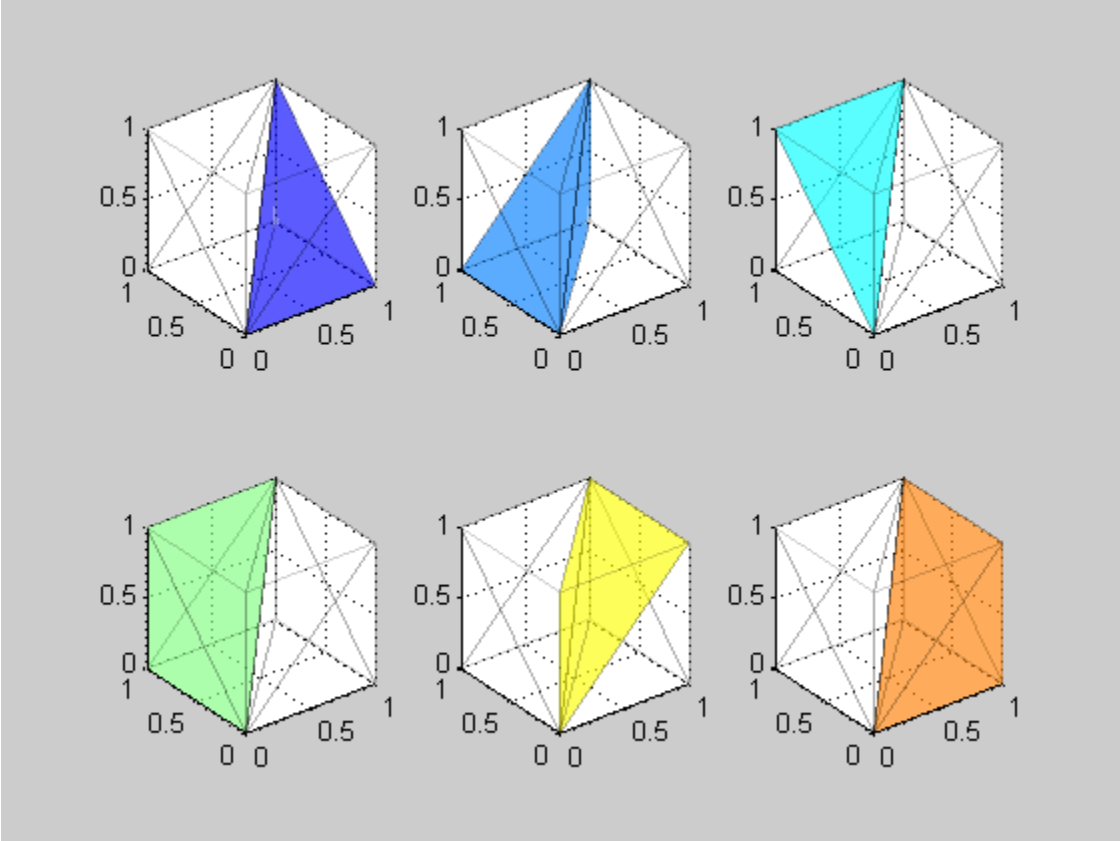
If we choose a coordinate system in which the four points where f is known are (0, 0), (0, 1), (1, 0), and (1, 1), then the interpolation formula simplifies to

$$f(x,y) \approx \begin{bmatrix} 1-x & x \end{bmatrix} \begin{bmatrix} f(0,0) & f(0,1) \\ f(1,0) & f(1,1) \end{bmatrix} \begin{bmatrix} 1-y \\ y \end{bmatrix}.$$

Bilinear interpolation is not used on ICC profiles, but the logic extension to 3 or more dimensions is certainly used. The generic method is known as trilinear interpolation. Again we do interpolation in one direction and then to the remaining ones.



There are different techniques for optimizing that. One common method is tetrahedral interpolation. In both tetrahedral and trilinear interpolation the grid points along each of the three axes serve to divide the volume into a set of rectangular hexahedra. To interpolate at a particular point, the first step is determine which hexahedron the point is in. In tetrahedral interpolation, the hexahedron is further subdivided into six tetrahedra. There is more than one possible subdivision. Here's the subdivision typically used for color space conversion.



Once the tetrahedron containing the interpolation point is identified, the interpolation is computed as a weighted sum of the grid values at the vertices of that tetrahedron.

If you look carefully at each of the tetrahedra, you can see that each one shares a common edge that is the diagonal from (0,0,0) to (1,1,1). That's why this particular tetrahedral subdivision is typically used for color space conversions. In an RGB space, this diagonal contains the neutral (or gray) colors, which are particularly important. Tetrahedral interpolation can be used to convert colors more accurately and with lower computational cost.

Encoding of some common color spaces

Now that we have the basic tools, we can explore how are encoded the color spaces. First come RGB. Scientific apps often use 0...1.0 range for RGB. Others use 0...255 because historic reasons, although they may use decimal numbers too. We are going to use 16 bits profiles, so we can apply the rules we have learnt for converting between domains.

$$0...1.0 \ (0..0xffff : rgb * 65535$$

$$0...255.0 \ (0..0xffff : rgb * 257$$

Second in our ranking is CMYK. Cmyk is slightly different in the sense many people uses a percentage for inks, that would mean 100% is the maximum amount of ink. So, using the same rules:

$$0..1.0 \ (0..0xffff : cmyk * 65535$$

$$0..100.0 \ (0.0xffff : cmyk * 65535/100 = cmyk * 655.35$$

So far, so good. Now we are going to get in some trouble with XYZ. For XYZ, ICC uses a notation normalized to 1.0, so D50 is represented as (0.9642, 1.0, 0.8249) instead of traditional (96, 100, 82). To fit that in 16 bits, the values are encoded in fixed point notation, specifically 1.15 unsigned.

Which is then the encodeable range? Well, zero equals to zero in fixed point, and since the maxim encoded number would be 0xffff in 1.15 fixed point, we can convert this back to rational by dividing by 2^n

$$65535 \frac{\square}{\square} 2^{15} = 1.9999694824218375$$

Weird number isn't it? Is not two, but almost!

Things goes more complex when dealing with Lab. XYZ allows only 16 bits encoding, but Lab can be encoded in either 8 or 16 bits. And then, there are two different, incompatible ways to encode Lab on 16 bits. Let's see each one in detail.

Lab8

The first one was for V2 8-bit profiles. Since in Lab, the a/b part can be negative, we need to somehow encode this. We can use 1 bit for sign or just change domain as we have seen before. Using a bit for sign has a bad side effect: since signed numbers are using two's complement, values close to zero in both positive and negative sides are quite different when encoded and transition on zero is not smooth.

a/b to encode	Signed	Change domain
-2	0xFE	0x7E
-1	0xFF	0x7F
0	0x00	0x80
1	0x01	0x81
2	0x02	0x82

Very bad for interpolation. To prevent that, we can use a domain translation, so we define our destination domain as 0 … 255 (remember this is 8 bits), the source domain as 0…100 for L*, and 127…-128 for a/b. That's how comes the conversions.

For L*:

$$L^*_{encoded} = \frac{255}{100} * L^*_{unencoded}$$

$$L^*_{unencoded} = 100\frac{\square}{\square}255 * L^*_{encoded}$$

And then for the ab part:

$$ab_{encoded} = ab_{unencoded} + 128$$

$$ab_{unencoded} = ab_{encoded} - 128$$

There is at least one major flow in this design. Do you see what is? Look at the ab range; it goes from -128 to 127. So where is zero? Yep, zero is not centered! Why is this important? Lab is used as the PCS, so it will be used as index for our 3D LUTs. Suppose we want to place all neutral axis, over specific nodes, ab=0 in the encoded notation is:

$$ab_{encoded} = ab_{unencoded} + 128 = 0 + 128 = 128$$

Which node configuration would make ab=0 to fall over specific nodes?

$$\frac{n-1}{255} * 128 = \frac{n-1}{3*5*17} * 2^7$$

Ops, all are prime factors, so the only possible configuration is 3*5*17+1 = 256, but that is a huge number of nodes for a 3D grid isn't it?

Lab16 (V2)

Then for 16 bits, a particular encoding was described in the v2 ICC spec. To add more precision, the format was defined as Lab8, but in fixed point, being the upper byte same as Lab8 and the lower byte the added fractional part. As seen in the fixed point discussion, we can convert to fixed point 8.8 encoding by multiplying times 256.

Let's see how some common numbers are encoded in this particular format.

$$L* = 100 \ (0xFF00 = 65280$$

$$a/b = 0 \ (0x8000 = 32768$$

Unfortunately, that made the issues on centering even worst. Now for getting L*=100 (white point) on an exact node:

$$\frac{n-1}{65535} * 65280 = \frac{n-1}{3*5*17*257} * 3*5*7*2^8 = \frac{256}{257} * (n-1)$$

So, only 256 nodes would meet this requirement, and for the ab part:

$$\frac{n-1}{65535} * 32768 = \frac{n-1}{3*5*17*257} * 2^{15}$$

There is no way to center this!

Lab16 (V4)

Version 4 of the ICC spec tried to fix that, Lab16 encoding was redefined to be Lab8 times 257, that is, we can convert from/to 8 and 16 bits in the same way for all color spaces. As a result, now L*=100 is encoded as 0xFFFF, which is always the last node. It works on all number of nodes. Well done!

For the ab part we still have some issues. Now zero is encoded as 0x8080 = 32896

$$\frac{n-1}{65535} * 65280 = \frac{n-1}{3*5*17*257} * 257 * 2^7 = \frac{128}{3*5*17} * (n-1)$$

It is still not possible to make zero to fall exactly on a node. But almost, and then, there are some tricks that may be very handy to fix things.

LUT types

Now for the LUTS. ICC spec offers 4 different LUT types, and with the addition of MPE that would be 5. Actually we have LUT8, LUT16, LutAToB and LutBToA. LUT8 and LUT16 were used in versions previous to V4, LutAToB and LutBToA are used in V4 and above.

If you care about precision, you should avoid, whenever possible LUT8 on BToAx direction. Why? Well, the only allowed PCS in this case is Lab8, and then a digital count of difference is worth of 1 ΔE. That is not longer true for Lab16 which has about 0.004 ΔE. LUT16 is twice the size of LUT8, but has 250 times more precision. I think that's a big deal!

LUT8 and LUT16 have this structure:

In LUT8 and LUT16, Matrix element can only be used when PCS is XYZ. In LUT8, both curves are fixed to 256 entries.

LutAToB may have more elements, but only certain combinations are allowed

Allowed combinations
B
M - Matrix – B
A - CLUT – B
A - CLUT - M - Matrix – B

Then comes LutBToA, which have same elements but in inverse order

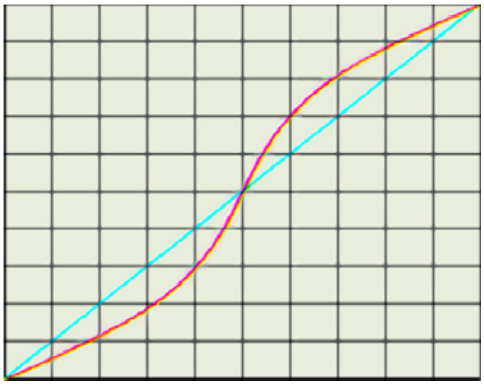
Allowed combinations:
B
B - Matrix – M
B - CLUT – A
B - Matrix - M - CLUT - A

Matrices

One of the many possible uses of matrices is to use a sort of RGB as intermediate space. Since PCS can be XYZ, we can use the matrix to convert this XYZ to a gamma 1.0 RGB, this has the advantage of no negative numbers and huge gamut. This RGB may then be accommodated to a more perceptually uniform RGB by means of the 1D curves. Finally we can use the CLUT to fine tune the obtained RGB. With this approach we can map white and black on a node, and let the CMM to do the necessary clipping on out-of gamut XYZ values before even entering the prelinearization.

Pre linearization curves

These are very handy to fit Lab encoding to the 3D CLUT. Using 258 entries, we can map the 0xff00 white of Lab16 to the upper node. We can also center ab = 0 on gray axis. Furthermore, we can implement different node densities by using non-linear curves. Take a look on this configuration. The blue line is the prelinearization on L*, red curve is applied to ab. Using this sigmoidal, we add more resolution near neutral axis at the expense of highly saturated colors. But that is ok, as highly saturated colors are probably out of our device gamut anyway.



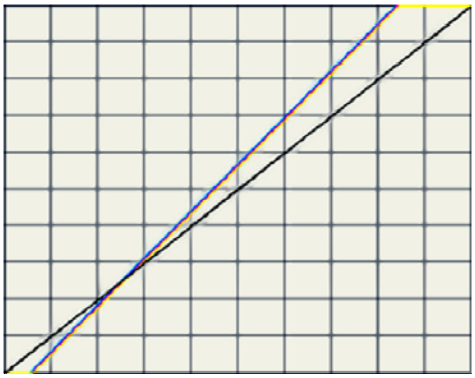
Another use would be to decouple gamma. If we want to create a devicelink that goes from AppleRGB, which has gamma 1.8 to sRGB, which has gamma near 2.2, we can use a 1.2 exponential as prelinearization. Then when AppleRGB goes across the curve, the gamma is adjusted and we can complete the transform with a simple 3x3 matrix.

Post linearization curves

Those are available on both v2 and V4, and I am going to explain here only one of its multiple uses. Let's first know what the issue is. Imagine we are implementing relative colorimetric intent. For simplicity sake in this example I would use a RGB space with a black point on Lab=(10, 0, 0). Even using prelinearization curves, we can end in a situation where black point falls between two nodes, say node 0 and node 1. There is no problem on computing RGB for node 1, which would be RGB=(5, 5, 5) and then for node 0… oh, well, since that is below real black point the theoretical RGB value is out of gamut: (-4, -4, -4) Ok, since LUT cannot contain negative numbers I clamp it to zero. Then I certainly have an issue here. Let's see what happens when the black point Lab=(10, 0, 0) arrives at the LUT. Yep, instead of the zero I got a value slightly superior (2,2,2). Why? Because the interpolation goes from (0, 0, 0) to (5, 5, 5) and then I end with RGB=(2,2,2) instead of RGB=(0, 0, 0)

This is a severe issue on certain situations, black point compensation, for example, since black point compensation may amplify this effect and as a result all your precious shadow detail is lost!

How to solve this? You guessed it: using post linearization. We may use the curve to add an offset for all values. In our example that is -4 for each RGB component. Then the values I put in the nodes are (0, 0, 0) and (9, 9, 9). When interpolation takes place, L*=10 gets interpolated between 0 and 9, that gives 4, then the curve subtracts 4 and the result is 0. For a L* falling on node 1, the value is still correct, since 9-4 = 5, and for an absolute zero value is already clamped to zero by the curve.



Multiprocessing elements

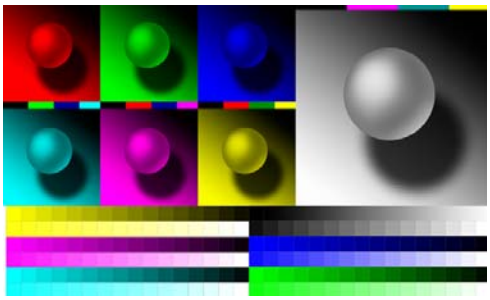
A new structure, MPE for multiprocessing elements is being added to the ICC spec. This would greatly simplify the task of profile creators and make things a little bit harder for CMM implementers, but overall is a good improvement. Basically an MPE as a type is a bag that can hold any combination of curves, matrices and LUTS, and the better part is, you can use floating point. So no fixed point tricks, no limited precision? Not really. Because that means we are no longer restricted to a 0..0xffff range but from minus infinite to infinite in middle stages, and only at the very last stage the CMM clamps the values. MPE is right now limited to DToB/BToD tags, but it may evolve in the future. Curves in MPE may be specified in several segments, with each segment may be sampled or specified as a math formula. This is a very powerful approach, but it has certainly some subtleties that must be fully understood if you want to use such constructs.

Profile qualification

To end the tutorial, I would like to give a short introduction on how to qualify the profiles we have built. Profile qualification is more an art than a pure science. It would take many hours to just explain the whole process on any profile class, so I'm just going to give you some clues. I will use output profiles for my example, and specifically printer profiles.

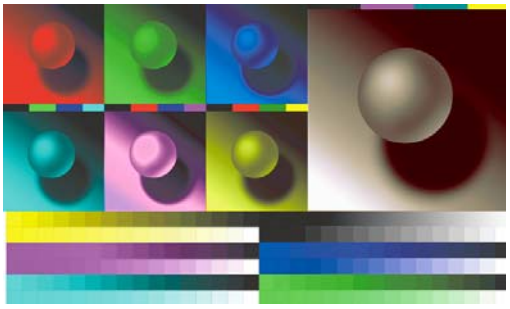
First thing to take in account is the fact you probably want not to assess the quality of the profile, but the quality of the whole workflow, that is the profile when used in conjunction with the printer and the CMM. Why? It is certainly possible to have a profile that has high rating in all metrics, but performs poorly when printing.

To finally render the image on media, printers use, among others, halftoning and separations. Both may introduce unwanted artifacts, so the first thing to check, even before to profile the printer, is the native device space. What we want to check most is the smoothness of the native space. To do that, we can use test plots like this:



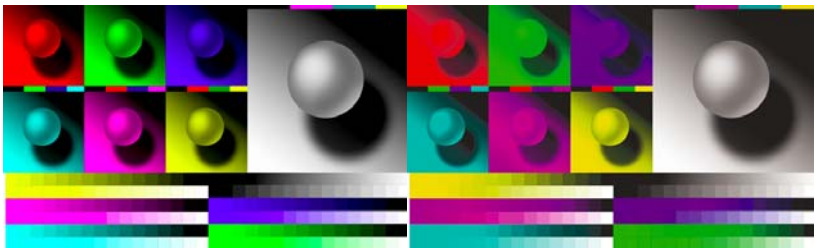
This is only a portion of the real test plot, you have to design it by yourself if you want to follow my advice. Each one of those balls is drawn by using only a primary or a secondary. Channel R from 0 to 255 for example.

If all goes ok, the image comes smooth with no contouring. But some printers do have problems:



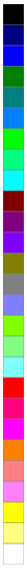
You can see a little bit of everything over here, contouring, inversions, non-neutrality, etc. Look at white zones with a magnifier glass: no ink should be dropped there. Lack of this is known as scum dot, and we want to make sure who to blame if the defect appears.

If the native space has such problems, the profile would try to compensate them and quite probably fail. That is because a profile uses interpolation. It assumes the native behaves smooth between samples. If the printer passes this test, the first intent to check would be the relative colorimetric. So, create the profile and print again this test plot by using a wide gamut matrix-shaper profile as input. AdobeRGB or CIERGB would be a good choice. Look at the plot for continuity problems. Relative colorimetric behavior outside gamut is undefined, so blocked zones are ok; contouring or inversions are not. Double check for white: no scum dot should show.



Once we are happy with continuity, we can check the accuracy. To do that in, for example, RGB, create a target subdividing whole RGB space in equally spaced steps. For example, 3 points:

0	0	0	0
1	0	0	128
2	0	0	255
3	0	128	0
4	0	128	128
5	0	128	255
6	0	255	0
7	0	255	128
8	0	255	255
9	128	0	0
10	128	0	128
11	128	0	255
12	128	128	0
13	128	128	128
14	128	128	255
15	128	255	0
16	128	255	128
17	128	255	255
18	255	0	0
19	255	0	128
20	255	0	255
21	255	128	0
22	255	128	128
23	255	128	255
24	255	255	0
25	255	255	128
26	255	255	255



Than print those patches and measure them by using a spectrophotometer or colorimeter. The obtained Lab values are assured to be inside printer gamut.

Next step would use the profile in the absolute colorimetric intent to print those Lab values. Doing in this way, we check the BToA1 direction, which is what is going to be used in the color workflow. Then measure again the obtained patches. This second list of lab values hides the accuracy of the profile, to uncover it; you need to compute, patch by patch, the color difference by using your favorite ΔE formula. Try to avoid Euclidean dE(76) as CIELab has issues on very saturated colors,

blues turning violet and red turning oranges. ΔE 2000 has issues too, but much less, so that would be a good choice. Another option is to use CAM02 and compute Euclidean distances.

Then sort all dE and take 95% lower. That gets rid of outliers and is a very suitable metric for imaging reproduction. For spot colors you would need all 100% range. The average and central point may be handy as comparative metrics. A couple of clues: use many patches, 3224 is a good amount. Use also different targets for creating the profile and for measuring accuracy. A good idea is to measure plot-to-plot constancy. This is a top maximum and the profile should not give more accuracy than that. If you have several printers in a cluster you may want to measure the printer-to-printer constancy too.

Finally, when it come the qualification of perceptual, sorry but you have to print plots and take a look on them. Use tons of plots. Really, you need to do so. Maybe the profile works well in 999 plots and fails miserably on the 1000^{th}. You can combine several in a single combo and print the combo. Content to try for combos: skin tones, blue skies, foliage, very saturated colors (toys, flowers). Defects to search for: loss of shadow detail. Issues on highlights, inversions. Some profiles favor match-to-screen in the perceptual intent. If this is the case use a good monitor, calibrate it properly and use the ISO viewing conditions. Use D50 as the illuminant, metamerism happens.