**Well, ok. This idea about using uint64 for complex formulas was abandoned because of power function. To raise an int to an arbitrary fractional power you have to use Taylor series. To calculate terms of this series you have to multiply and divide several numbers for each term. For arbitrary power coefficients cannot be pre calculated. Therefore whole new data type of uint64 with exponent component should be developed along with algorithms for multiplication, addition, division. While this would be a good exercise, first – it should be already existing in some library and inventing it from scratch is a bad practice, second – takes too much effort. At this point conversion will be performed with doubles and that’s it.**

**sRGB Gamma Conversion**

The formula for transferring from sRGB color space to linear (CIE XYZ) brightness space is from Wikipedia article <https://en.wikipedia.org/wiki/SRGB>.

In this formula input and output values are normalized to be between 0 and 1:

**Integer conversion explanations**

Transformations are performed over integer numbers to avoid precision loss of big floating-point numbers. And resulting numbers are integer anyway, so we avoid floating point altogether.

In the application linear color space values are of 16 bit depth. Input images (sRGB) can be 8, or 16 bit deep.

Intermediate number variables are 64 bit unsigned.

Max value of 64 bit unsigned number is .

1. **sRGB 8 bit to linear XYZ 16 bit:**

Denoting:

– input sRGB value 8 bit,

– signal - normalized sRGB value,

– linear brightness value, 16 bit (result),

There are two segments to input value, and . Let’s find that produces signal values over . We call it threshold and denote .

**A.1 Therefore first 10 values are calculated with**

If we want to talk about only integer numbers, we must multiply and divide by 100:

Now we must divide while saving as much precision as possible. To do that we multiply upper value by number while is as big as possible. And after the division we divide by back. We find by respecting max value of 64 bit integer:

Because max value of n is 10:

Therefore we can write:

And we can calculate with 4 steps that avoid overflow while keeping precision:



The result of this division will have 13 decimal places of precision after the integer part, but the integer part will not be separated by decimal dot because we are dealing only with the integer numbers.

If we want to find final integer value, we should divide by and round to the nearest integer:

Rounding to the nearest integer can be done by adding 0.5 and then taking floor:

In reverse order we add and then divide by which in integers performs floor of the resulting value:

**A.2 Now we calculate for input values :**

We can calculate this formula in 3 steps. Each step is performed with maximum available precision, and then rounded to match precision allowed by the next step. Steps are:

**A.2.1 Step 1:**

must conform to while having as much precision as possible.

We perform

while complying with

**A.2.2 Step 2:**

must conform to while having as much precision as possible.

We perform

while complying with

**A.2.3 Step 3:**

must conform to while having as much precision as possible.

We perform:

While

**A.2.4 Final step:**

Now we have the number and should divide it by . Since is not an integer power we represent the divisor

Since all values should be integer we multiply by (arbitrarily) and perform division first by and then division with rounding by

**A.2.5 Order of calculations is:**