Fixed-point log function

The algorithm

In general, we want to find:

$y = \log(x) = ?$

We know that:

$\log(1) = 0$

So, if we find numbers {k1, k2, …, Km} that their multiplication with x will close to 1, then:

$0 = \log(x * k_1 * k_2 * \ldots * k_m)$

Let's simplify using logarithm rules:

$0 = \log(x) + \log(k_1) + \log(k_1) + \cdots + \log(k_m)$

And replace $y = \log(x)$:

$0 = y + \log(k_1) + \log(k_1) + \cdots + \log(k_m)$

$\Rightarrow \quad y = 0 - \log(k_1) - \log(k_1) - \cdots - \log(k_m)$

So, in our algorithm we start with y=0 and then for each k that brings us closer to 1, we subtract its logarithm from y.

The closer the multiplication of $x * k_1 * k_2 * \ldots * k_m$ to 1, the closer y to log(x).

Note: the idea is to perform multiplication only with nice numbers (easy to multiply) and subtract not necessarily nice numbers.

This algorithm explained in detail @ https://www.quinapalus.com/efunc.html

C code

A modification to the function given @ https://www.quinapalus.com/efunc.html

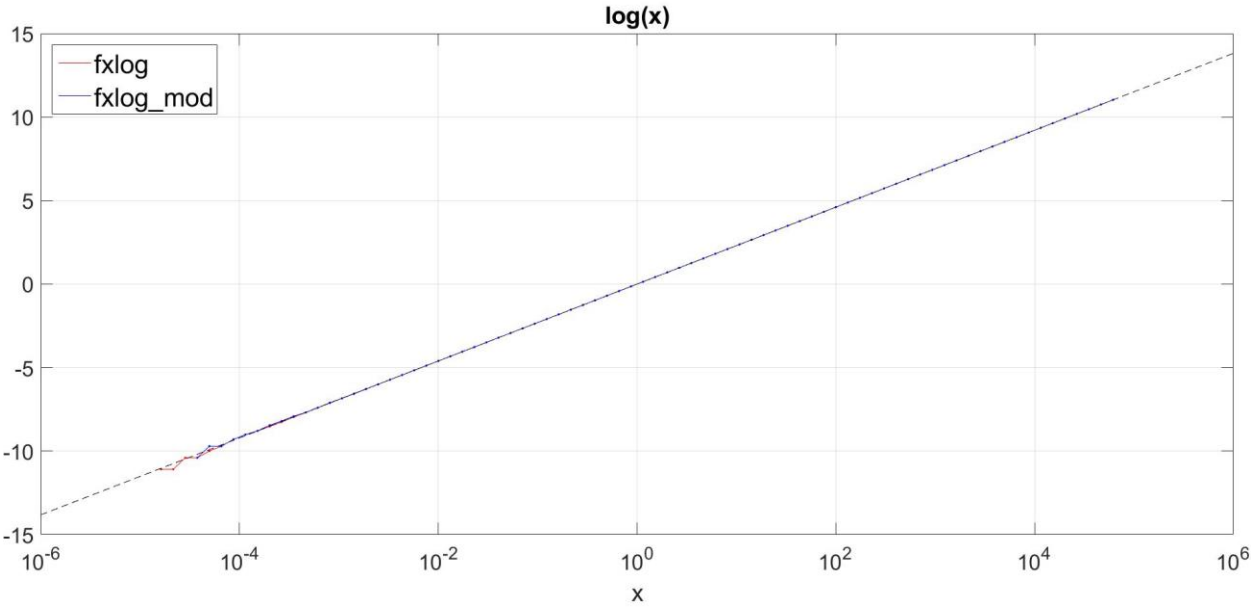
- The code assumes integers are at least 32 bits long.
- The (positive) argument expressed as fixed-point values with 15 fractional bits.
- The result of the function expressed as fixed-point values with 24 fractional bits.
- Intermediates are kept with 31 bits of precision to avoid loss of accuracy during shifts.

```
/*
fxlog_mod: implementation for Q15 input and Q24 output
*/
s32 fxlog_mod(s32 x) {
	s32 t, y;

	y = 0xb17217f;
	if (x<0x00008000) x <<= 16, y -= 0xb17217f;
	if (x<0x00800000) x <<= 8, y -= 0x58b90c0;
	if (x<0x08000000) x <<= 4, y -= 0x2c5c860;
	if (x<0x20000000) x <<= 2, y -= 0x162e430;
	if (x<0x40000000) x <<= 1, y -= 0xb17218;
	t = x + (x >> 1); if ((t & 0x80000000) == 0) x = t, y -= 0x67cc90;
	t = x + (x >> 2); if ((t & 0x80000000) == 0) x = t, y -= 0x391ff0;
	t = x + (x >> 3); if ((t & 0x80000000) == 0) x = t, y -= 0x1e2707;
	t = x + (x >> 4); if ((t & 0x80000000) == 0) x = t, y -= 0xf8518;
	t = x + (x >> 5); if ((t & 0x80000000) == 0) x = t, y -= 0x7e0a7;
	t = x + (x >> 6); if ((t & 0x80000000) == 0) x = t, y -= 0x3f815;
	t = x + (x >> 7); if ((t & 0x80000000) == 0) x = t, y -= 0x1fe03;
	x = 0x80000000 - x;
	y -= x >> 7;
	return y;
}
```

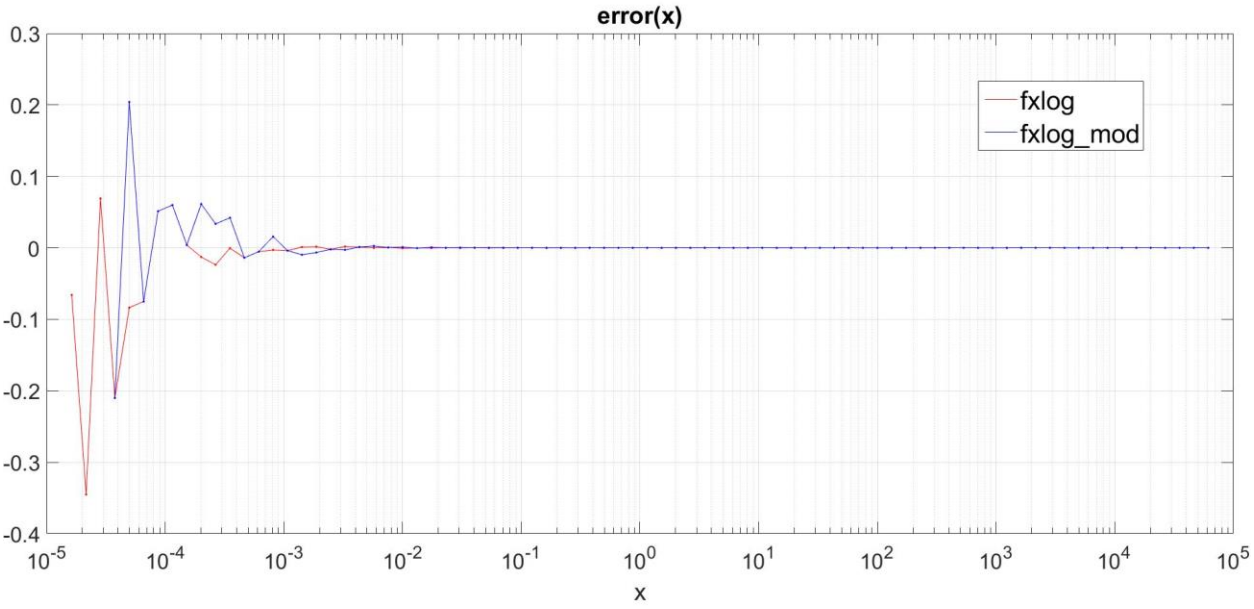
Analysis

The modified implementation tested against the reference code for Q16 and against MATLAB implementation for type double.
The C-code was exported as a dynamic library that in turn loaded from a MATLAB script for convenient visualization.

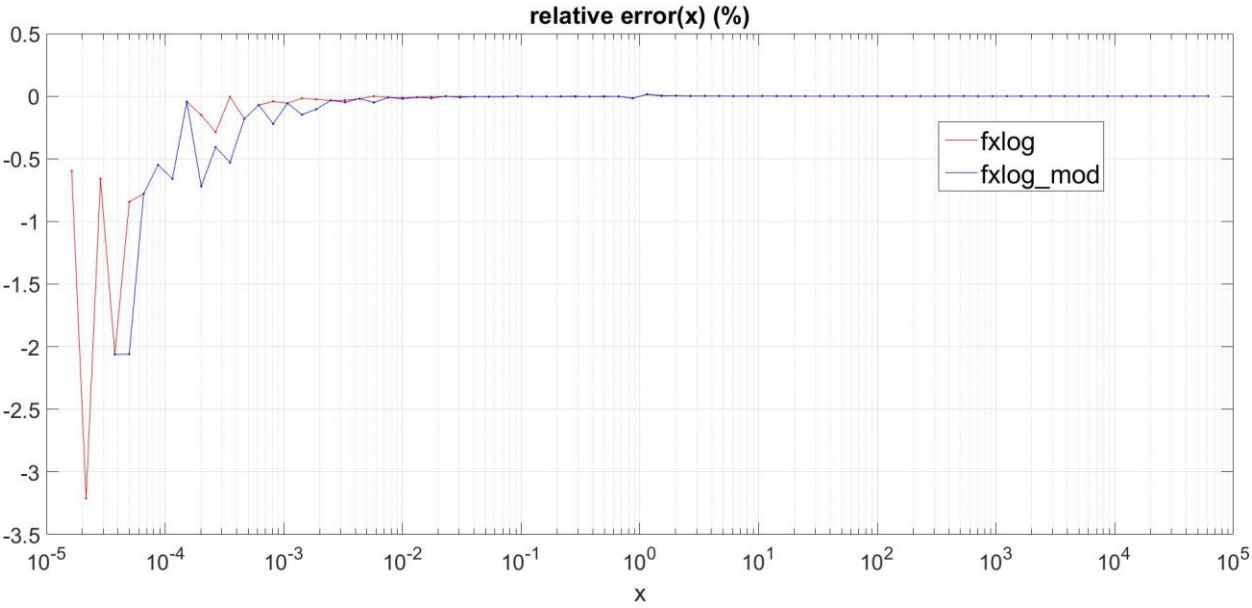
log result:



log error:



log relative error:



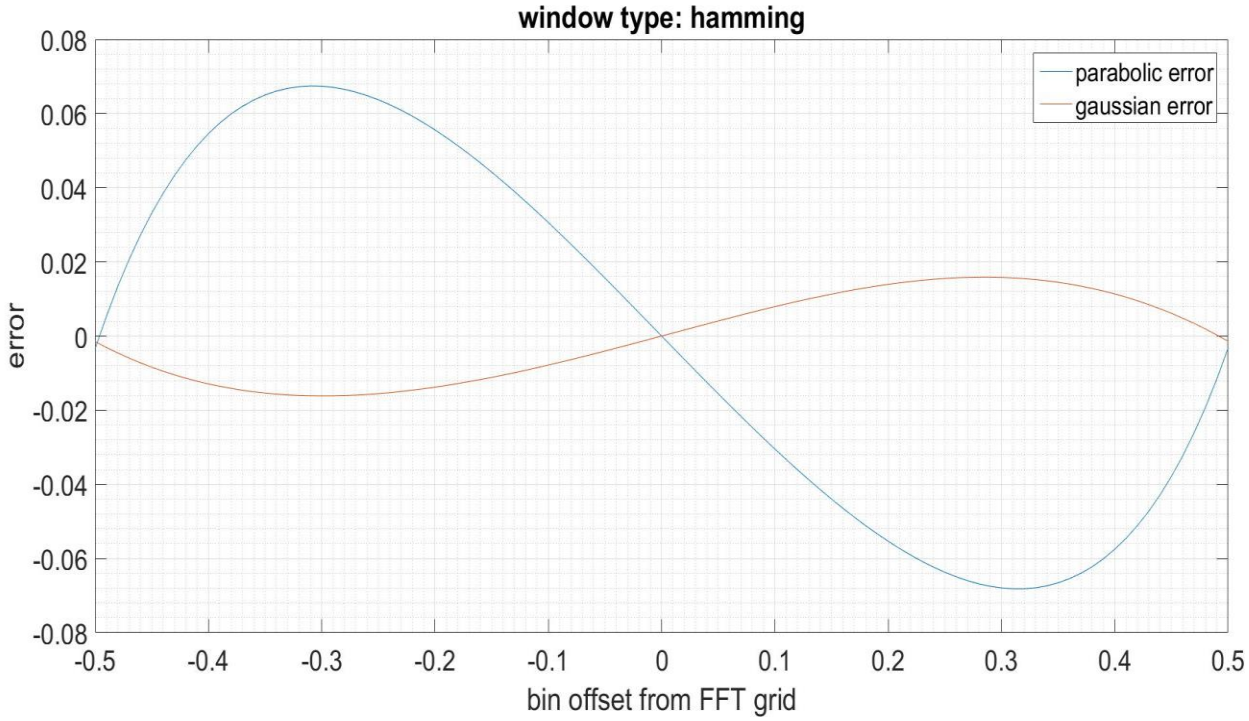
From the graphical analysis it seems reasonable to conclude that the two implementations are comparable in term of accuracy (and identical in terms of performance).

Fixed-point Gaussian interpolation function

Background

The frequency estimation process can utilize a gaussian interpolation to improve its accuracy.
It can be shown mathematically that this king of interpolation is superior to the parabolic method when applied to the FFT outputs.

When true peak of the Fourier transform is located in a [-0.5, 0.5] offset from the closest integer bin index. And can be restored using parabolic or Gaussian interpolation:



As can be seen from the graph, the Gaussian interpolation introduces smaller error when estimating the true peak location.

Implementation

The find the bin offset using gaussian interpolation is just taking the logarithm (no matter the base) of the FFT output and then continue with the formula of the parabolic method.

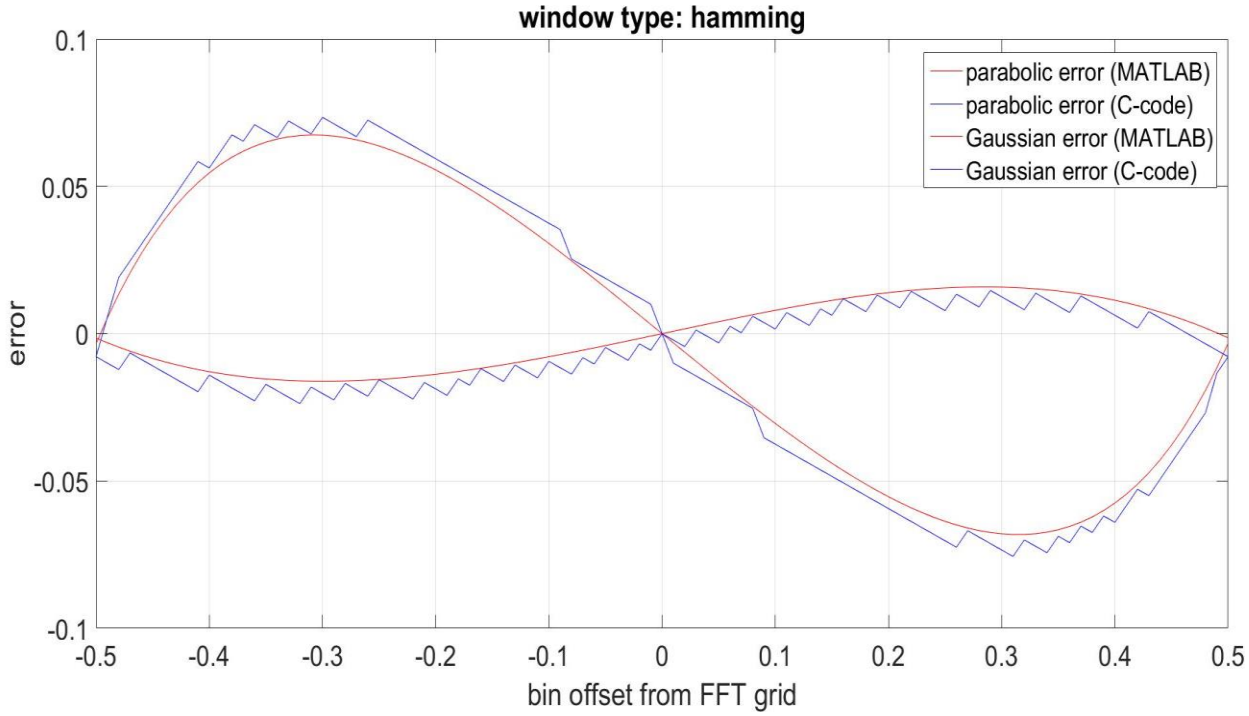
The following function uses the modified fixed-point log function described above to perform the task in hand (Gaussian interpolation):

```
s32 calculateGaussianPeak(u16 index, u16 energy, u16 leftEnergy, u16 rightEnergy)
{
	s32 energySum = (s32)leftEnergy + (s32)rightEnergy;
	s32 doubleEnergy = 2 * (s32)energy;
	s32 dxQ24 = 0;
	s32 dxQ7 = 0;

	s32 logEnergy = fxlog_mod((s32)energy << 15);
	s32 logLeftEnergy = fxlog_mod((s32)leftEnergy << 15);
	s32 logRightEnergy = fxlog_mod((s32)rightEnergy << 15);

	if (energySum < doubleEnergy)
	{
		s32 logEnergyDiff = logLeftEnergy - logRightEnergy;
		s32 logDenom = (s32)((logLeftEnergy + logRightEnergy) - 2 * logEnergy);

		s64 temp = ((s64)logEnergyDiff << (24 - 1)); // shift 24 is due to fix
point. -1 is because we need to multiple by 0.5
		dxQ24 = logDenom != 0 ? (s32)(temp / logDenom) : 0;
	}

	dxQ7 = dxQ24 >> (24 - 7);
	return ((s32)index << p) + dxQ7;
}
```

The key idea here is that the conversion to Q7 in done only on the last step.

Analysis

Using hamming window:



The interpolation can be improved further by applying Blackman-Harris-Nuttall window: