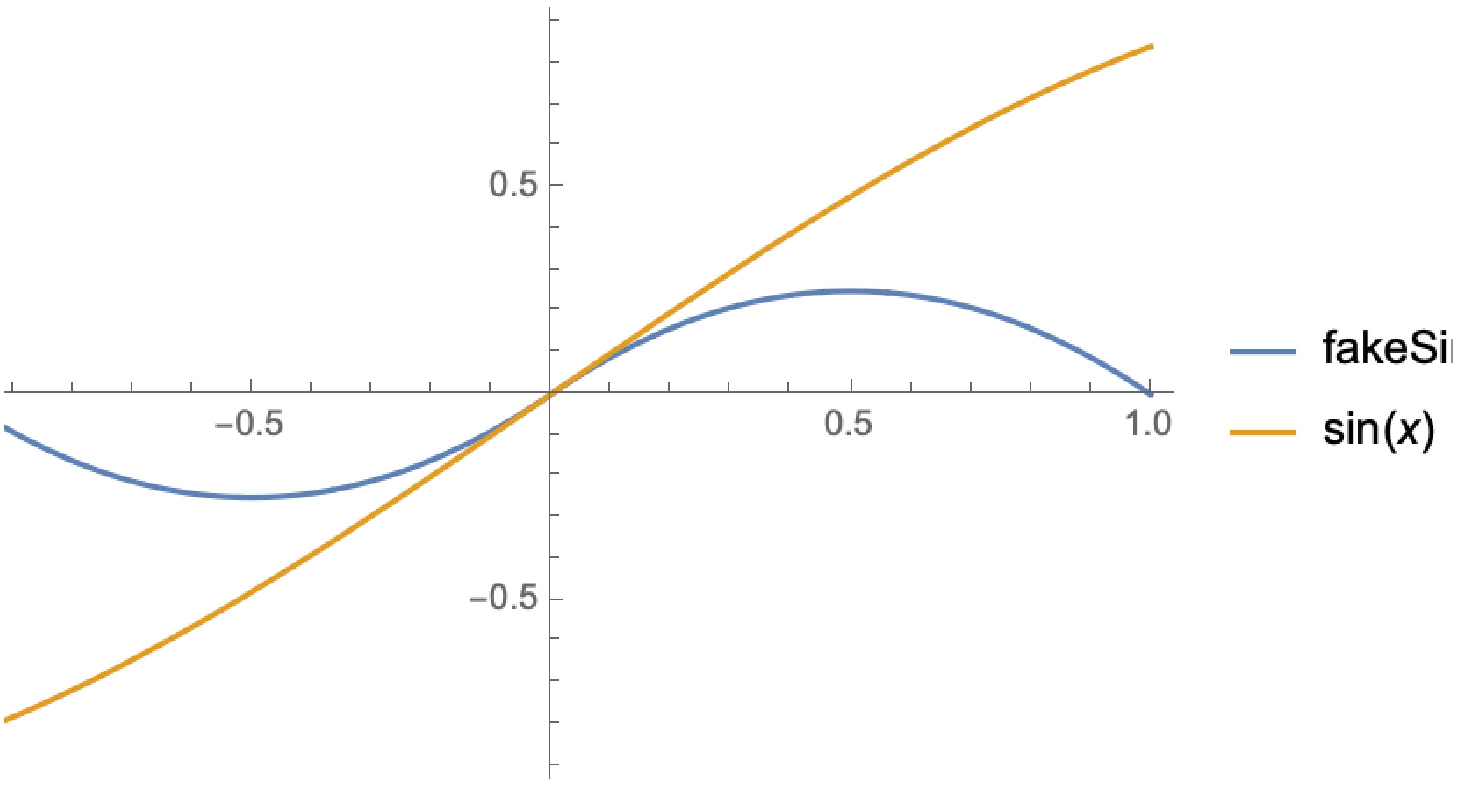


Is this the world’s fastest sine approximation?

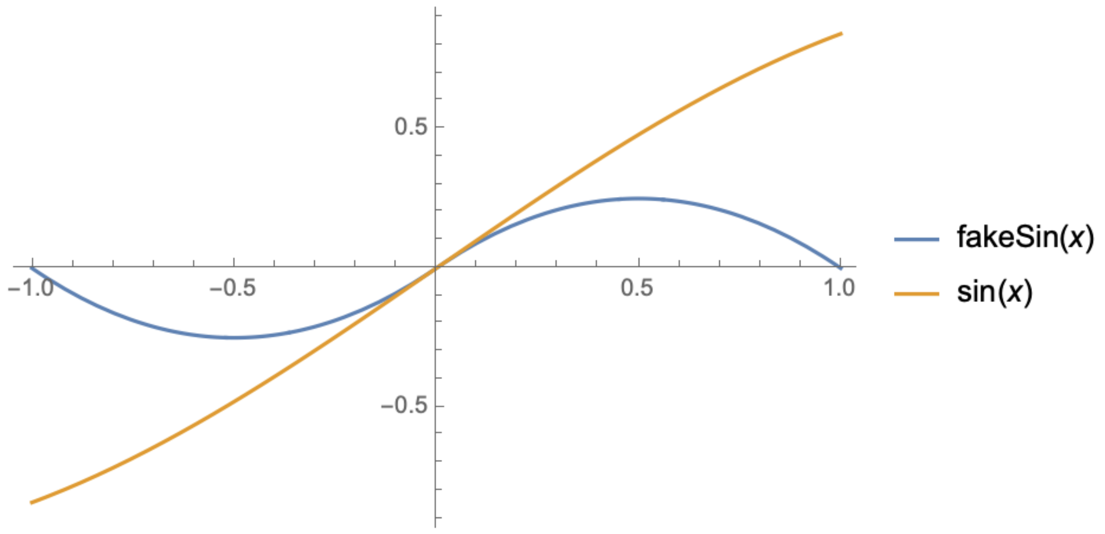
A google search for the simplest, most efficient polynomial approximations to a sine-wave reveals many algorithms of the following two types:

- Quadratic functions that require branching using the **if** keyword to switch between the positive and negative parts of the wave
- Cubic functions that don’t require branching but don’t reach their max and min values at the correct phase corresponding to the pi/4 and 3 pi/4 parts of the wave.

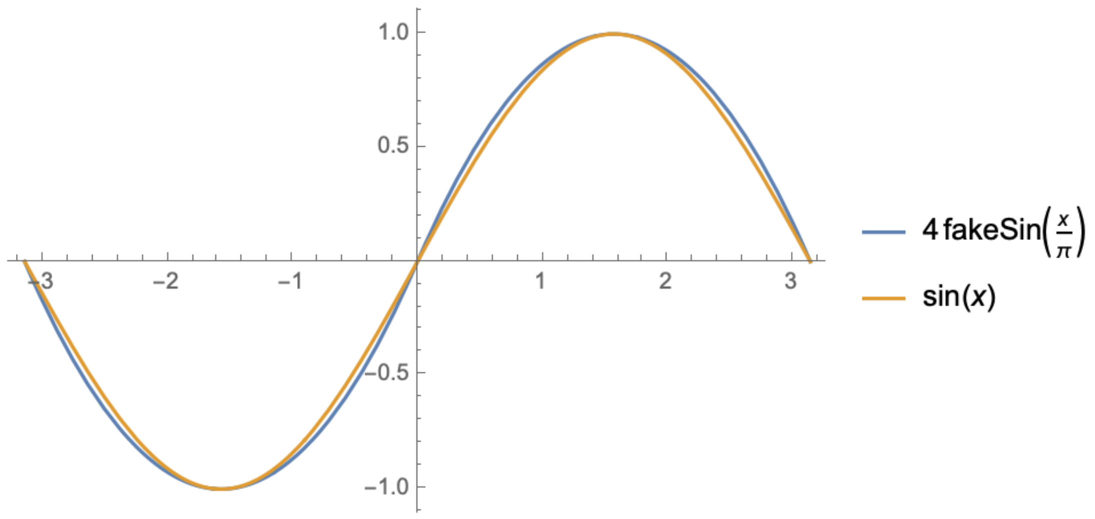
In this article we present an unusual solution that is quadratic, closely resembles a sinusoid, and is extremely simple. Sounds too good to be true? Here it is:

```
// x in [-1,1]
fakeSin(x) = x (1 - abs(x))
```

Note that this function differs from the sin function in magnitude and frequency:



However, we can scale it to match with an additional two multiplications:



However, in audio signal processing applications, when we generate a sinusoids for use in oscillators, we usually end up scaling the frequency and amplitude anyway, so we prefer to leave the fakeSin function as stated above for maximum efficiency.

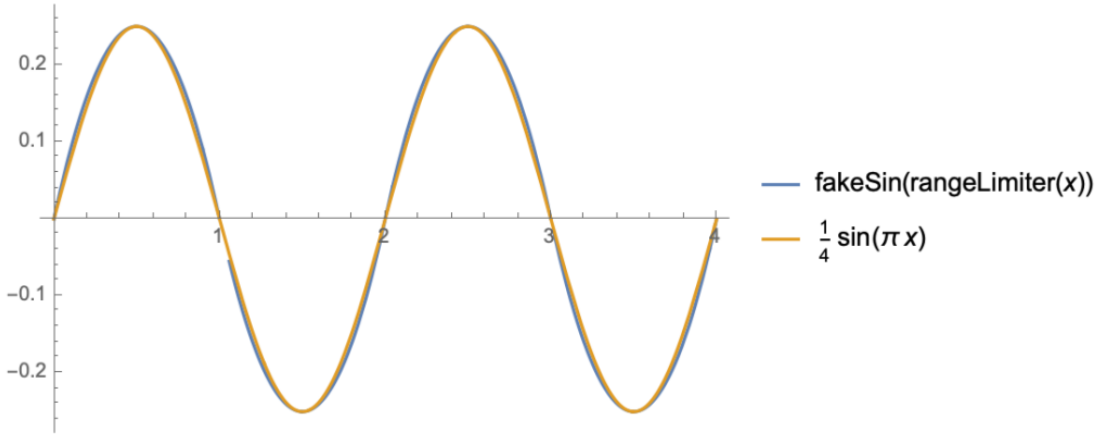
Now, like the real sine function, the fakeSin function algorithm is defined on a limited range of inputs x. To make it work for any floating point value of the input variable, we define the following range limiter function:

```
rangeLimiter(x) = fmod(x + 1, 2) - 1

// if we don't care about the phase alignment with the sin function, we can drop the addition:
rangeLimiter(x) = fmod(x,2) - 1
```

We get a complete fast sinusoid oscillator by combining these two functions:

```
fastSinOscillator(x) = fakeSin(rangeLimiter(x))
```



What about fmod? Isn't it slow?

The use of the floating point modulo operation in the rangeLimiter() function could be problematic because it requires a floating point division. Floating point division is generally considered a no-no in optimised signal processing code because it takes 10 times more clock cycles than multiplication on many processor architectures. (The number 10 is an example, not a measurement.) However…

1. If we are processing the function in vectorised code, as is often the case when we are building an oscillator, we have seen cases where the division is almost as fast as multiplication. We believe this is due to pipelining. In other words, it may take a lot longer to do the first division but if we can cue the second division operation before the first one is completed then for arrays of length 100 or more it doesn't really matter if a single division operation goes slower than multiplication. We have observed this in practice and therefore we do not shy away from vectorised division operations.
2. We can replace the fmod(x,2) function call with this:

```
// replacement for fmod(x,2.0)
fmod2(x) = 2.0 * (x*0.5 - floor(x*0.5))

// obviously the product x*0.5 should be computed only once, so we have:
// multiplications: 2
// subtractions: 1
// integer truncations: 1
```

Whether or not the fmod2 replacement above is actually faster than a vectorised floating point mod using pipelining and SIMD (single-instruction multiple-data) methods depends on the architecture and as usual we should measure it if we are serious about optimising this calculation.

Alternatively, we could redefine the fakeSin function so that it operates on [-0.5, 0.5] instead of [-1,1]. This would allow us to limit the range without doing any multiplications:

```
// x in [-0.5,0.5]
fakeSin2(x) = 2*x * (1 - abs(2*x))

// wrap x to [-0.5, 0.5]
```