Fast/SIMD Linear/Trigonometric Math

Notes

Digital Signal Processing (DSP) is usually applied on streams, thus processed on signal blocks, consisting of a bunch of (complex) floating-point samples, e.g. with block size 256 .. 2048 samples. In the lab, a big signal file might be processed at once - similar to Octave/Matlab/NumPy/SciPy, but not when it comes to responsive realtime processing.

Operations

Main operations are mixing and filtering.

Mixing

Simply said: multiplication of the input stream with a complex (quadrature) sine carrier. Generation of sine carrier $c(n)$ involves the trigonometric functions $\sin()$ and $\cos()$. And, of course, the complex multiplication is required:
$y(n) = x(n) \cdot c(n)$
with the input signal $x(n)$,
the complex carrier $c(n) = \exp(n \cdot i \cdot \Delta\varphi) = \cos(n \cdot \Delta\varphi) + i \cdot \sin(n \cdot \Delta\varphi)$
and $\Delta\varphi = \frac{shift\_frequency}{samplerate} \cdot 2 \pi$.
Boundless phase $\phi(n) = n \cdot \Delta\varphi$ will reduce accuracy of $\cos()$ and $\sin()$.
$\phi(n)$ should always get wrapped to the range $[-\pi ; +\pi]$, when used as argument to trigonometric functions:
$\phi(n) := \operatorname{wrapToPi}(\phi(n-1) + \Delta\varphi)$

Calculation of $c(n)$ won't need $\sin()$ and $\cos()$ for every sample n, when changing the calculation to
$c(n) := c(n-1) \cdot r$
with complex rotation constant $r := \exp(i \cdot \Delta\varphi) = \cos(\Delta\varphi) + i \cdot \sin(\Delta\varphi) $.
Thus, calculation of $c(n)$ does only need one complex multiplication per sample. An additional side effect is, that $\operatorname{wrapToPi}()$ isn't necessary in the complex plane of $c(n) \in \mathbb{C}$.

Cause of error propagation and rounding towards zero (default in C/C++ programs), $c(n)$ needs to be normalized every bunch of M samples, that $|c(n)| = 1$. Thus $c'(n) := \frac{c(n)}{|c(n)|}$ every $n = k \cdot M$ for all $k \in \mathbb{N}$.

For streaming, the generated carrier's phase needs to be continuous - to the previous block.

Filtering

Filtering involves design, needed once, for *infinite impulse response* (IIR) and *finite impulse response* (FIR) filter types.
FIR type filters to need convolution for application of the filter. The convolution is applied to the input stream (in the time domain).
Fast convolution is performed utilizing the *fast fourier transform* (FFT), with algorithms like overlap/scratch.

Down-conversion and decimation

Mixing the desired signal's (high) frequency near or exactly to zero is called *down-conversion*. In addition, an unnecessarily high samplerate is reduced by lowpass filtering and decimation.
Decimation means taking out 1 sample every N samples. To avoid aliases in this *decimation*, a low-pass filter has to be applied.
There are several algorithms for an optimized combination of low-pass filter and decimation, e.g. *half-band-filter* or cascaded-integrator-comb (*CIC*) structures to name a few.

Spectrum (plot)

Usually, a power (density) spectrum is displayed, showing the logarithmic spectrum of the input, temporary or result signals. This involves windowing, the FFT, power $norm(z) = \operatorname{Re}(z)^2 + \operatorname{Im}(z)^2$, alternatively calculated with $norm(z) = z \cdot \operatorname{conj}(z) = z \cdot {z}^\ast$ with the complex FFT bins z. Then, power is converted to a level with logarithm to base 10 ($\log_{10}()$) and scaling (scalar multiplication).
For short: $10 \cdot \log_{10}(\operatorname{norm}(FFT(x \cdot window)))$
For levels on a plot, a precision and accuracy of 0.1 dB is usually sufficient.

FM: Frequency modulation

FM demodulation requires complex multiplication with previous samples conjugate and complex $ \arctan(z) = atan2(\operatorname{Im}(z), \operatorname{Re}(z))$:
$y(z) = \arctan\{ z(n) \cdot conj(z(n-1)) \}$.

Alternatively $y(z) = \operatorname{diff}(\arctan(z)) = \arctan(z(n)) - \arctan(z(n-1)) $
+ wrapping back to the range $[-\pi ; +\pi]$ of the resulting phase values:
$y(z) = \operatorname{wrapToPi}(\operatorname{diff}(\arctan(z)))$

AM: Amplitude modulation

AM demodulation, with simple *envelope detector*, requires complex multiplication with same samples' conjugate and square root ($\operatorname{sqrt}()$):
$y(z) = |z| = \operatorname{abs}(z) = \sqrt{ \operatorname{norm}(z) } = \sqrt{ z \cdot {z}^\ast } = \sqrt{ z \cdot \operatorname{conj}(z) } = \sqrt{ \operatorname{Re}(z)^2 + \operatorname{Im}(z)^2 }$

Then, filter $y(z)$ by a high pass filter, to remove the DC component ..

Speed

Single memory pass

Some operations need to be applied consecutively onto the stream. Applying the operations sequentially onto the blocks, requires one pass over the memory for each operation. Depending on the block size (number of samples) and the CPU's cache sizes, this might get quite slow - due to limited memory bandwidth - compared to calculating all operations in one single pass. But this requires complex/combined functions for the necessary operations ..

Reduce precision and accuracy

Another optimization is, not to pay for unnecessary precision/accuracy. This allows to use different approximation levels, e.g. at/with math functions like $\log_{10}()$ or $\arctan()$, e.g. cause the spectrum display won't need an accuracy better than 0.5 dB. Quite simpler is using/prefering single precision 'float' over 'double'. Half-precision might be a choice for a small dynamic range.

SIMD

Of course, not to miss, are the capabilities of modern processors for SIMD (single instruction multiple data) like SSE or AVX instruction sets on x64 or NEON on ARM architecture. Now, in 2022, C/C++ compilers still have problems at automatic vectorization to generate these vector instructions. Assembler or compiler intrinsic functions can be used to enforce the usage ..

Fixed Point Arithmetic

For ancient processors without any *floating point unit* (FPU) or slightly more recent, but also outdated SIMD units without floating point support, fixed point arithmetic might be interesting. With many microcontrollers, there's also no FPU available, that the only alternative to a slow floating point emulation library is fixed point arithmetic. You need to be very careful about the varying precision at operations. Of course, all value ranges must be well determined.
Having to program the math operations manually, restricts this approach to simple operations - due to it's complexity. Any operations beyond comparison, addition, subtraction and multiplication will be quite complex.
Furthermore, i'm missing a good freely usable C++ template library - both with and without SIMD.

Reproducibility

All operations should be exactly reproducible. That is quite important for debugging and for regression/unit-tests.
When generating noisy (*AWGN*) test signals, it's advised to use a reproducible pseudo random number generator (*PRNG*) with a known and adjustable *seed* value.

Target

This page should collect links and references to relevant libraries and notes. Ideally, we would develop a benchmark for some operations.

SIMD Libraries

see SIMD

Fast atan2f

- mazzo.li
  - Speeding up atan2f by 50x (2021-08-16)
  - vectorized-atan2 (2021-08-16)
    - gist source vectorized-atan2f.cpp
- Morlocks' Fun with AVX Blog
  - FUN with AVX (2013)
  - Handling vector loop left-overs
  - Branch-Free Blend() '(a < b) ? x : y'
  - Fast Arc Tangent polynomial approximation
- Hacker News
  - https://webbindustries.com/hackernews/story/28209097
  - https://news.ycombinator.com/item?id=28209097
- dsprelated
  - Find Fast Atan2 Approximation
  - https://www.embedded.com/performing-efficient-arctangent-approximation/

Fast Inverse Square Root

- https://www.linkedin.com/pulse/fast-inverse-square-root-still-armin-kassemi-langroodi/
- https://en.wikipedia.org/wiki/Fast_inverse_square_root
- Matthew Robertson: A Brief History of InvSqrt, 2012
  - https://cs.uwaterloo.ca/~m32rober/rsqrt.pdf
- YouTube (DE): in depth mathematical derivation of the algorithm
  - https://www.youtube.com/watch?v=vcPGkmz_Wtk
- YouTube (EN):
  - https://www.youtube.com/watch?v=p8u_k2LIZyo

General

- Random ASCII, Bruce Dawson's tech blog
- compare float
- Crunching Numbers with AVX
- DirectXMath

    - https://docs.microsoft.com/en-us/windows/win32/dxmath/directxmath-portal
    - https://github.com/microsoft/DirectXMath
- Paul Mineiro's fastapprox
    - with exp, log, pow, sind, cos, tan, erf, .. - but no atan[2]
    - https://github.com/romeric/fastapprox
    - https://github.com/pmineiro/fastapprox
- Cephes
    - http://gruntthepeon.free.fr/ssemath/
    - Benchmark code
        - https://twiki.cern.ch/twiki/pub/LCG/VICephes/vecfunBench.tgz
- Misc
    - OpenLibm
        - https://github.com/JuliaMath/openlibm/blob/master/src/e_atan2.c
    - Sleef (Public Domain)
        - SLEEF SIMD Math Function Library (BG/Q Port)
    - https://codereview.stackexchange.com/questions/174617/avx-assembly-for-fast-atan2-approximation
    - https://github.com/JishinMaster/simd_utils