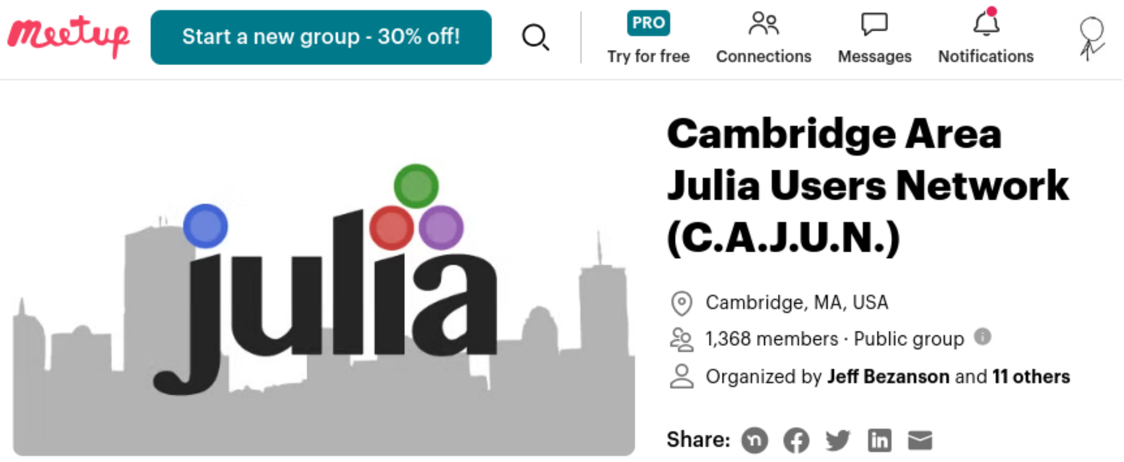


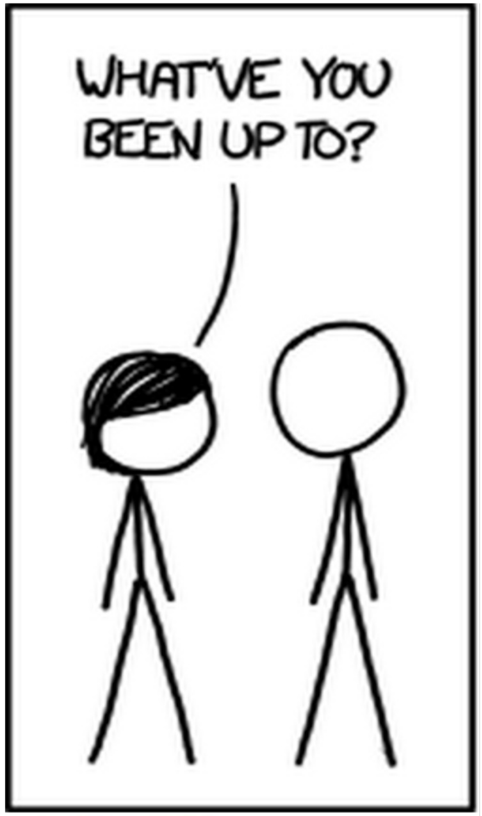
Composable iterator processors applied to digital signal processing

David Hossack

https://github.com/arghhhh/julia-signals-systems

https://www.linkedin.com/in/david-m-hossack/

Please ask questions as we go along…



David Hossack

Integrated Circuit Design Engineer

- Some analog, mostly digital
- A/D and D/A conversion – Sigma-Delta Modulation
- Signal Processing
- Acoustics


Not logged in Talk Contributions Create account Log in

WIKIPEDIA
The Free Encyclopedia

Article Talk Read Edit View history Search Wikipedia

GP5 chip

From Wikipedia, the free encyclopedia

This article is about the GP5 probability processor. For the glycoprotein V, see GP5.

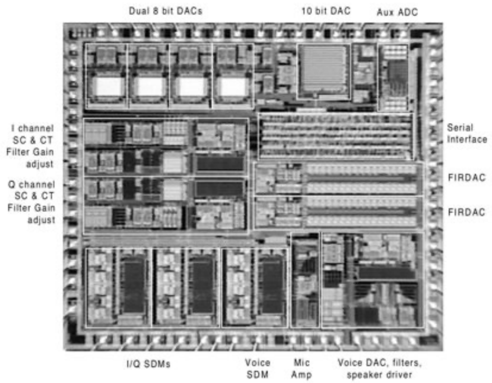
The **GP5** is a co-processor accelerator built to accelerate discrete belief propagation on factor graphs and other large-scale tensor product operations for machine learning. It is related to, and anticipated by a number of years, the Google Tensor Processing Unit

The GP5 chip

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

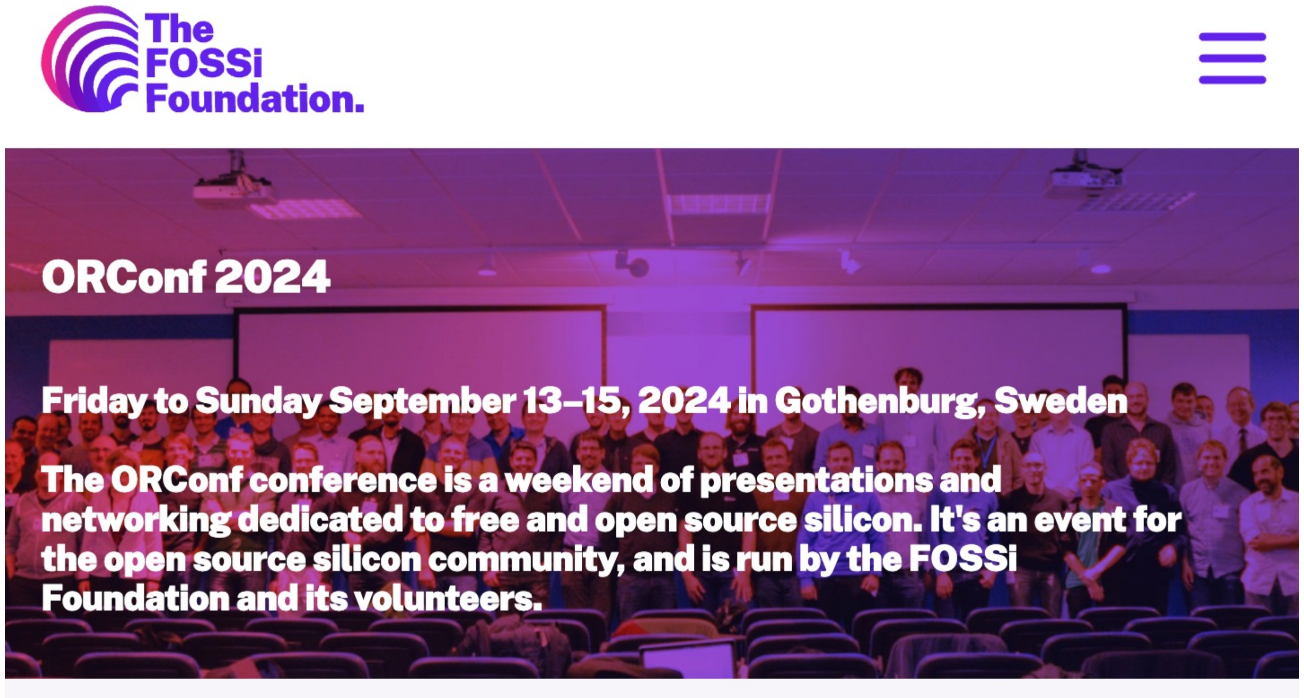
Interaction

Help
About Wikipedia
Community portal



Overview

- Motivation
  - Brief review of Signal Processing applications
  - Brief review of Julia iterators
- Composable Iterator Processors
- Implementation in Julia
- Signal Processing Examples
  - Running Sum – CIC filter
  - First Order Sigma Delta Modulator
- Fixed Point Library

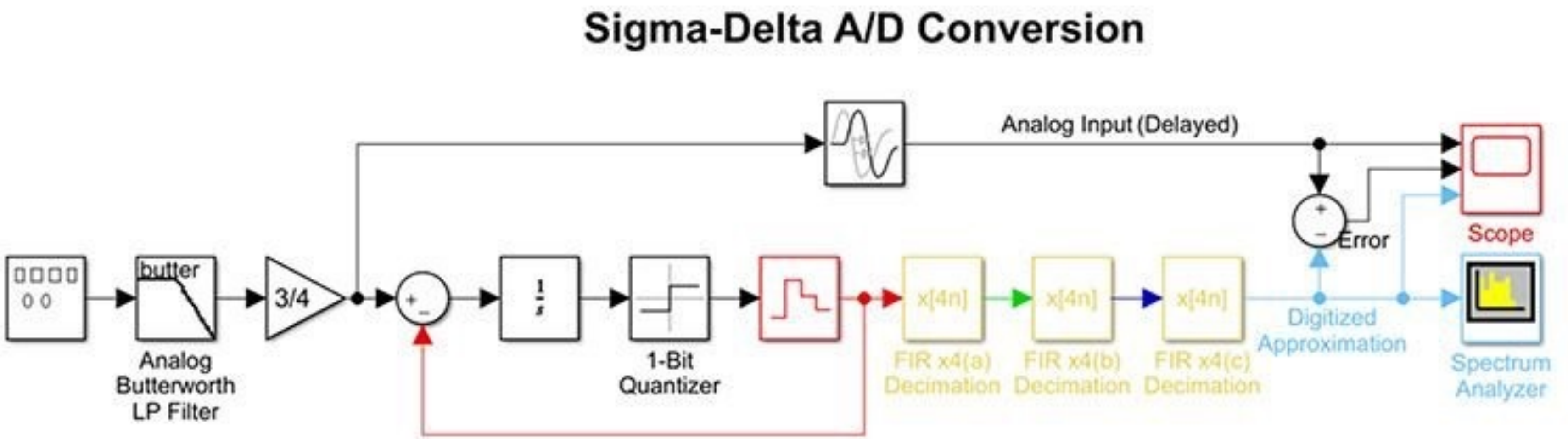


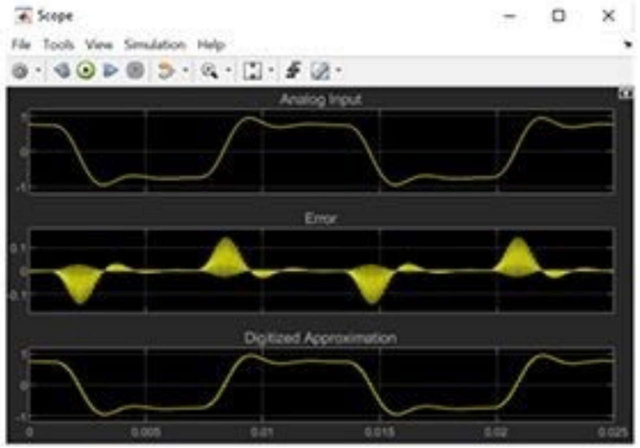
A version of this presentation was presented at the Free and Open Source Silicon Conference

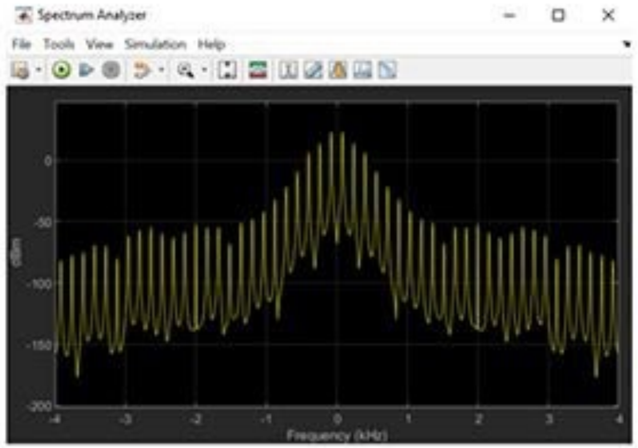
An alternative - Mathworks Simulink

https://www.mathworks.com/products/simulink.html

- Multiple sample rates, including continuous time
- Examples do not separate the DUT from the stimulus and analysis







Digital Signal Processing

For the purposes of this talk:

- **Signal**:
  - Source of values, of any type
    - Input signal, output signal etc
    - Sequence of numbers, fixed point, integer, floating point, complex
    - Sequence of Vector/Tuple – eg pairs of numbers for stereo audio, RF IQ, audio frames
- **Processing**:
  - Converting signals to other signals
    - Not necessarily at a one-to-one sample rate
    - Stateful
    - Causal

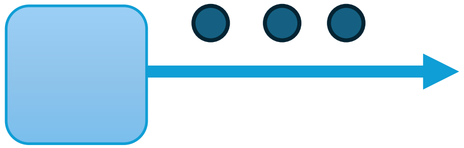
Signals & Processing

Why model DSP for integrated circuits?

- Initially – early in the architecture and design process:
  - Determine and confirm what signal processing is necessary
    - filter types and orders, cutoff frequency, etc
    - quantization noise, overflow, numeric word length requirements etc
- Subsequently:
  - Design verification
    - Does the hardware correctly implement the desired design and meet requirements
  - Huge non-recurring-engineering (NRE) charge at tape-out
    - Design absolutely must be correct…

- Essential link in the verification chain
  - Datasheet
    - filter response curves, signal-to-noise ratio curves etc

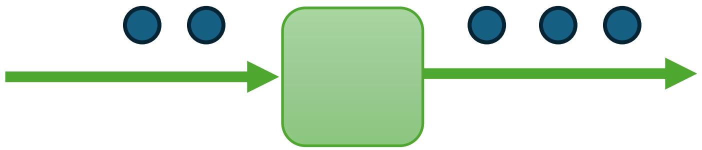
Signal / Sequence



- Source of samples
- Many computer languages have the concept of an **iterator**
  - Define a signal using language's iterator protocol
  - Can repeatedly ask for the next sample
  - Could have bounded length, unknown length, or unbounded length
    - Julia handles this nicely
  - Can serve up samples of any type
- Added arithmetic operations
  - Example: a test signal, with DC offset and noise:

```
0.01 + 0.1 * sinusoid(1000.0,3.0e6) + 0.01 * gaussian_noise()
```

System / Processor



- Casual & Lazy
  - Output is only a function of current and previous input
    - State
  - Produce output on the fly as inputs arrive
    - Can process infinite length signals
- Mathworks/MATLAB encourages potentially non-causal calculation
  - Process entire signals at one time
    - Good for tight inner loops, at the cost of excessive memory bandwidth
    - No concept of an infinite sequence
- Julia has a just in time (JIT) compiler (LLVM)
  - should not be afraid of compiling inner loops

Origins... a long, long time ago...

- UNIX command line
  - `ac 0.9 0.0 1000 3.0e6 | sdm | cic | fir1 | fir2 | head -10000 > y`
  - individually compiled C/C++ programs
  - read from standard input and write to standard output
  - slow

Great flexibility and composability

- C++ library
  - exploiting templates, operator overloading, fixed point arithmetic library

- Now Julia

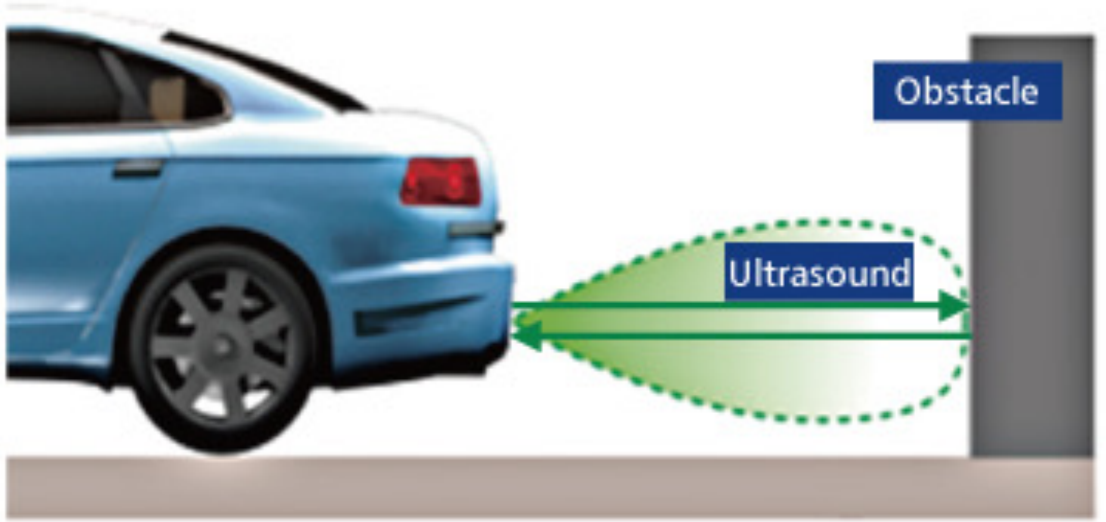
DSP Application

- build up system from components
  - system that drives the Device Under Test (DUT), DUT, postprocessing
- build up many different inputs
- vary parameters:
  - Input: amplitude, frequency, DC level, noise…
  - System: coefficients, gains, arithmetic precision (wordlengths, scalings), tolerances, different implementations…
- measure performance:
  - Output: level (gain), DC, noise, error rate, distortion – harmonic, intermodulation…

Diagram from:
https://corporate.murata.com/more_murata/techmag/metamorphosis17/productsmarket/ultrasonic

Parking Assist Example

- Transmit signal generation
- Transducer (frequency response)
- Acoustics – target echo, ground reflection clutter, multipath…
- Transducer, analog front end, ADC..
- Digital filters, mixing, multirate matched filter, detection, constant false alarm rate (CFAR)…



Characterize parts of the system, whole system etc
Lots of scenarios

Julia Iteration

- length can be:
  - known, finite
  - known, infinite
  - unknown
- eltype
- iterate – first call
  - create initial state
- Iterate – subsequent calls
  - function state->(value,next_state)

Interfaces

A lot of the power and extensibility in Julia comes from a collection of informal interfaces. By extending a few specific methods to work for a custom type, objects of that type not only receive those functionalities, but they are also able to be used in other methods that are written to generically build upon those behaviors.

Iteration

There are two methods that are always required:

Required method	Brief description
`iterate(iter)`	Returns either a tuple of the first item and initial state or `nothing` if empty
`iterate(iter, state)`	Returns either a tuple of the next item and next state or `nothing` if no items remain

Julia iterators are ideal for representing signals for Signal Processing

Julia Base.Iterators   Iteration Utilities

- many functions of iterators returning iterators
- many of these are "lazy" versions of existing "eager" transformations

Iterators.cycle(iter, n) is the lazy equivalent of Base.repeat(vector, n),
while Iterators.repeated(iter, n) is the lazy Base.fill(item, n).

Iterators.map(f, iterators...).   Create a lazy mapping.
This is another syntax for writing (f(args...) for args in zip(iterators...)).

Iterators.filter(flt, itr) lazy version of See Base.filter for an eager implementation
Iterators.accumulate(f, itr; [init])   effectively a lazy version of Base.accumulate.

Adding associativity :

associativity properties - all equivalent:

```
Iter1 |>  process1 |>  process2 |>  process3 |>  process4

( ( (Iter1 |> process1) |> process2 ) |> process3 ) |> process4

Iter1 |> ( process1 |> process2 |> process3 |> process4 )

Iter1 |> ( process1 |> ( process2 |> process3 |> process4  ) )

Iter1 |> ( ( process1 |> process2 ) |> ( process3 |> process4 ) )
```

Adding associativity :

- can separate the "source" iterator from the "process" components

```
input = Iter1
system1 = process1 |> process2
system2 = process3 |> process4
system = system1 |> system2

output = input |> system
```

- all of the above are lazy – defining what to do, but not actually doing it
- to actually do the work and get a resulting signal:

```
y = collect(output)
```

Example System: Moving average

- Linear filter, has frequency response, impulse response etc
  - "improve" sensor measurements by averaging out noise
- Better filters – e.g. combine multiple moving average filters
- Specialized hardware implementations
  - https://en.wikipedia.org/wiki/Cascaded_integrator%E2%80%93comb_filter
  - mobile phone will contain many of these
    - audio, accelerometer and other sensors

Example:
Impulse response of cascade of four moving average filters

- Define input:

```
impulse = concatenate( zeros(10), 1.0, sequence(0.0) )
```

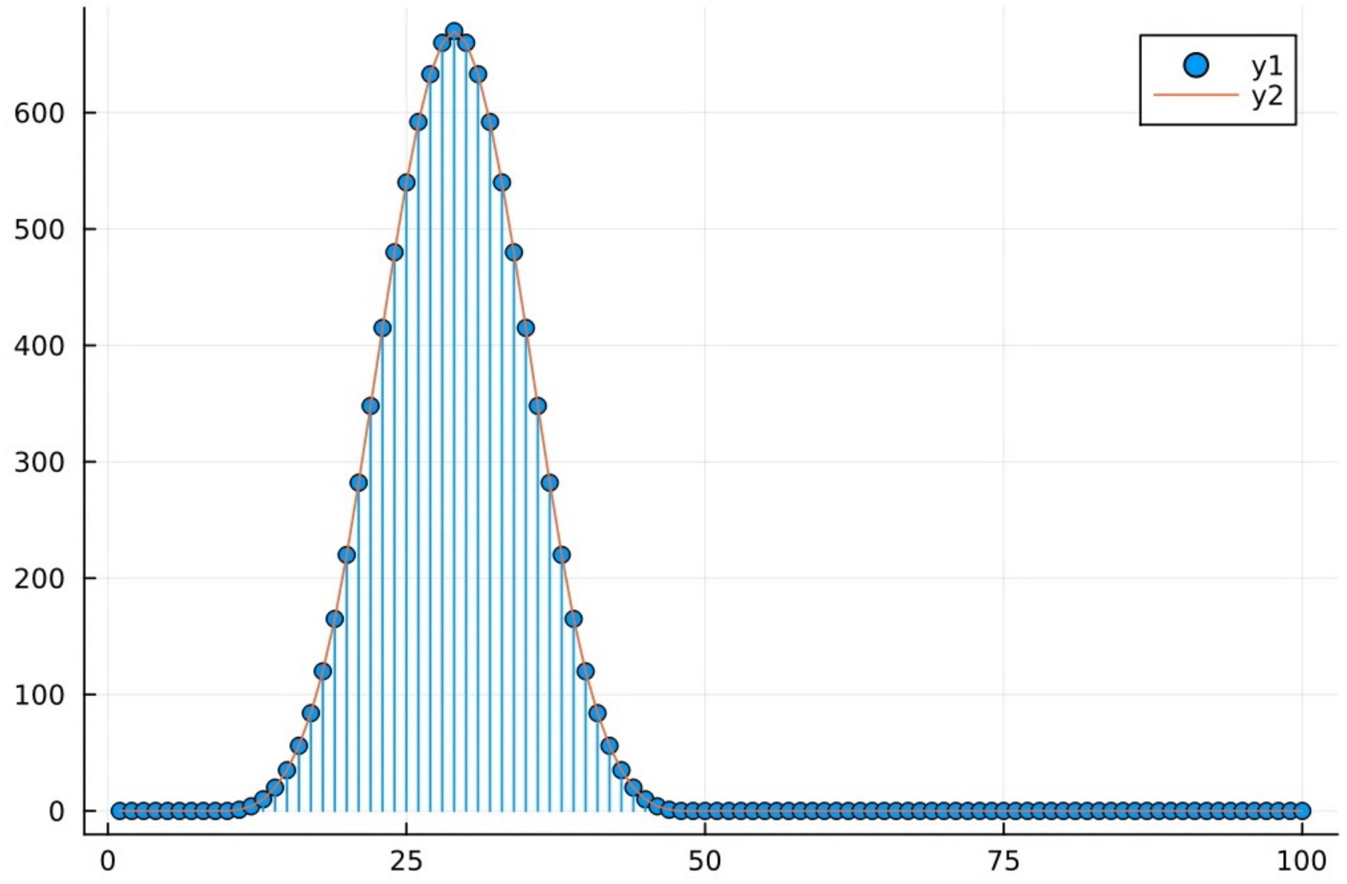
- Define system:

```
sys = fir(ones(10)) |> fir(ones(10)) |> fir(ones(10)) |> fir(ones(10))
```

- Run the system with the input:

```
y3 = impulse |> sys |> Take(100) |> collect
```

Impulse response of filter



Implementation

julia / base / **operators.jl** ↑ Top

Code | Blame 1528 lines (1184 loc) · 39.6 KB Raw

```julia

# function pipelining

"""
    |>(x, f)

Infix operator which applies function `f
This allows `f(g(x))` to be written `x |
When used with anonymous functions, pare
the definition to get the intended chain

# Examples
```jldoctest
julia> 4 |> inv
0.25

julia> [2, 3, 5] |> sum |> inv
0.1

julia> [0 1; 2 3] .|> (x -> x^2) |> sum
14
```
"""
|>(x, f) = f(x)
```

Pipe operator built into Julia

```
|>(x,f) = f(x)
```

Alternate syntax for a function call

Used for both:

1. composition of processors
2. application of sequence

processor.jl (part 1/2)

```
struct Apply{Iter,Processor}
    in::Iter
    p::Processor
end

struct Compose{Processor1,Processor2} <: abstract_processor
    p1::Processor1
    p2::Processor2
end
```

Processors will be defined in terms of an Apply{} struct instance

Compose{} is a place holder struct, waiting for some input sequence to be applied

processor.jl (part 2/2)

```
(p2::abstract_processor)( p1::abstract_processor ) = Compose( p1, p2 )
(p2::abstract_processor)( it ) = Apply( it, p2 )

(c::Compose)( in ) = c.p2( c.p1( in ) )

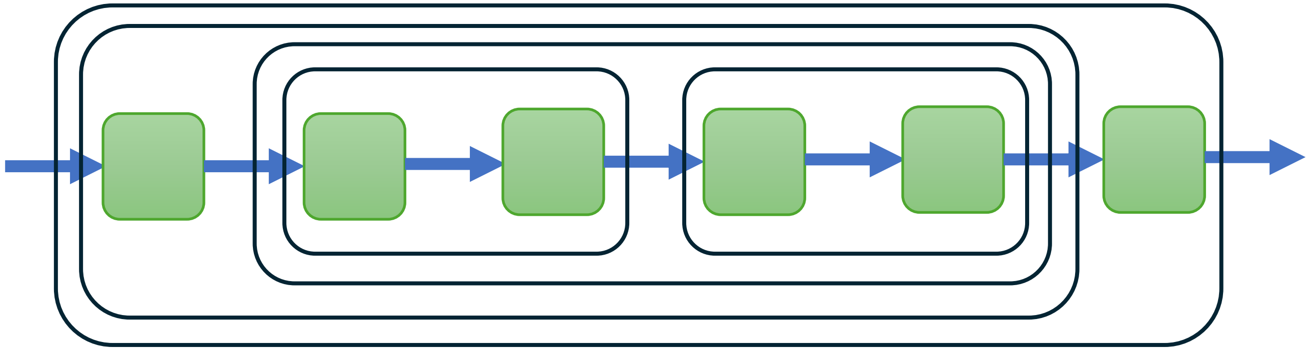
(c::Compose)( p1::abstract_processor ) = Compose( p1, c )
```

Compose is a holder – waiting for a sequence to be applied

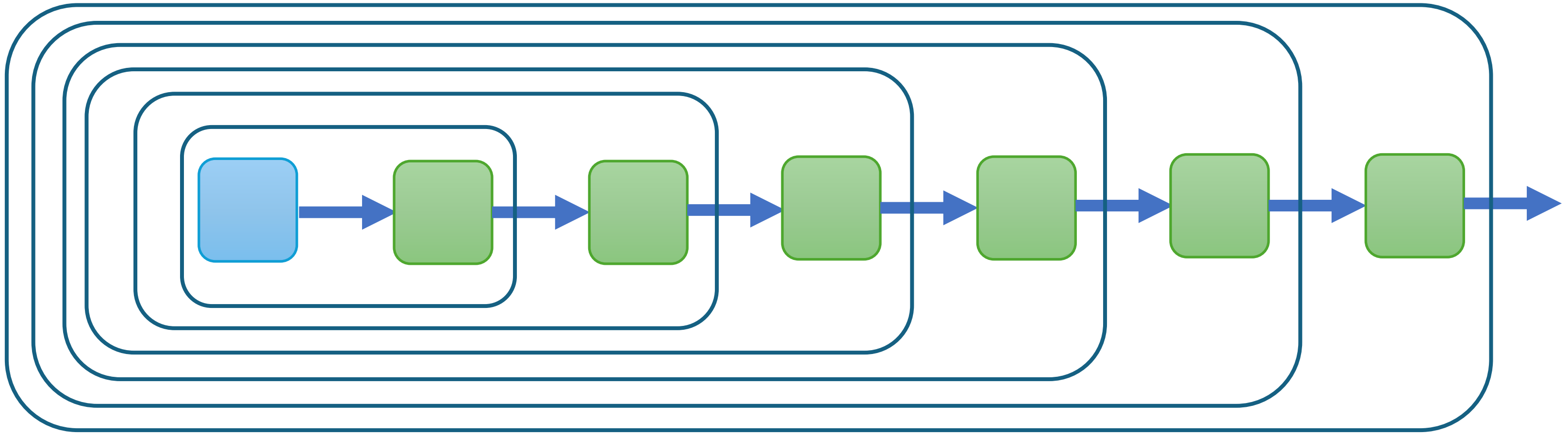
When sequence is applied, the input iterator is applied to p1 and then that iterator is applied to p2

Converting System Definition to Sequence

Define system as nested Compose{} **tree**



Apply input sequence, form new sequence, apply to next system component, recursively. No Compose{} structs left, everything is an iterator



There is effectively no code required to define what it means to apply an input to a Compose{} system

Example using Vectorize

- Building block for FIR filter and much more

```
julia> 100:150 |> Vectorize(4) |> Downsample(4,4-1) |> x->collect(Vector,x)
12-element Vector{Vector}:
[100, 101, 102, 103]
[104, 105, 106, 107]
[108, 109, 110, 111]
[112, 113, 114, 115]
[116, 117, 118, 119]
[120, 121, 122, 123]
[124, 125, 126, 127]
[128, 129, 130, 131]
[132, 133, 134, 135]
[136, 137, 138, 139]
[140, 141, 142, 143]
[144, 145, 146, 147]
```

Vectorize outputs a vector of the previous n inputs

Uses:

- FIR Filter
- Frequency Domain Processing

Making an FIR filter from parts

```
fir( coeffs ) = (
          Vectorize( length(coeffs) )
       |> DotProduct( reverse(coeffs) )
)

decimator( n, coeffs ) = (
          Vectorize( length(coeffs) )
       |> Downsample(n)
       |> DotProduct( reverse(coeffs) )
)
```

fir() is a function returning a system

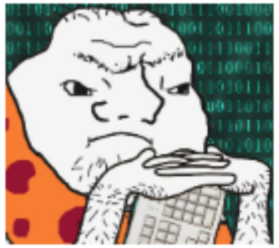
- DotProduct is stateless
- Downsample at the end can be moved up for efficiency
- Equivalent to a polyphase decimator

Simple Example Integrator

Processor Example: Integrator

- Produces a running sum of all the inputs so far
- Works for ints or floats (or any other type with arithmetic ops)
- Could overflow – but this isn't necessarily a problem
  - (and not relevant to this discussion)

Writing the body of a System/Processor is just writing a Julia Iterator – no new protocols to define or learn



The Grug Brained Developer

A layman's guide to thinking like the self-aware smol brained

https://grugbrain.dev/

Integrator – full code (1/3)

Define the iterator that is called Apply{I,Integrator}

```
struct Integrator <: abstract_processor
      # include any parameters, but not state here:
      gain
end
```

"gain" is only included in this definition to illustrate how parameters can be used

```
# functions dependent on the type only
Base.IteratorEltype(::Type{Apply{I,Integrator}}) where {I} = Base.IteratorEltype(I)
Base.IteratorSize(  ::Type{Apply{I,Integrator}}) where {I} = Base.IteratorSize(  I)

# functions dependent on the instance:
Base.eltype( a::Apply{I,Integrator}) where {I} = Base.eltype(a.in)
Base.length( a::Apply{I,Integrator}) where {I} = Base.length(a.in)
Base.size(   a::Apply{I,Integrator}) where {I} = Base.size(  a.in)
```

Integrator – full code (2/3)

```
# first call to iterate:
function Base.iterate(it::Apply{I,Integrator} ) where {I}
       # get the input:
       t = Base.iterate(it.in )
       t  === nothing && return nothing
       x,input_state = t

       # initialize processor state:
       integrator_state = zero(x)

       # calculate next processor state and current output
       integrator_state = x * it.p.gain + integrator_state
       yout = integrator_state

       # return output and the combined state for next time
       return yout, (input_state,integrator_state)
end
```

Define how the first sample is created – including forming the initial state

Integrator – full code (3/3)

```
# subsequent calls to iterate:
function Base.iterate(it::Apply{I,Integrator}, state ) where {I}
       # separate the combined state into
       # the input iterator state and the processor state:
       input_state,integrator_state = state

       # get the input:
       t = Base.iterate(it.in, input_state )
       t === nothing && return nothing
       x,input_state = t

       # calculate next processor state and current output
       integrator_state = x * it.p.gain + integrator_state
       yout = integrator_state

       # return output and the combined state for next time
       return yout, (input_state,integrator_state)
end
```

Define how subsequent outputs are generated – as a function of the current state and the input

Iterator Example Comment

- This example made
  one call to the input iterate function
  for each output

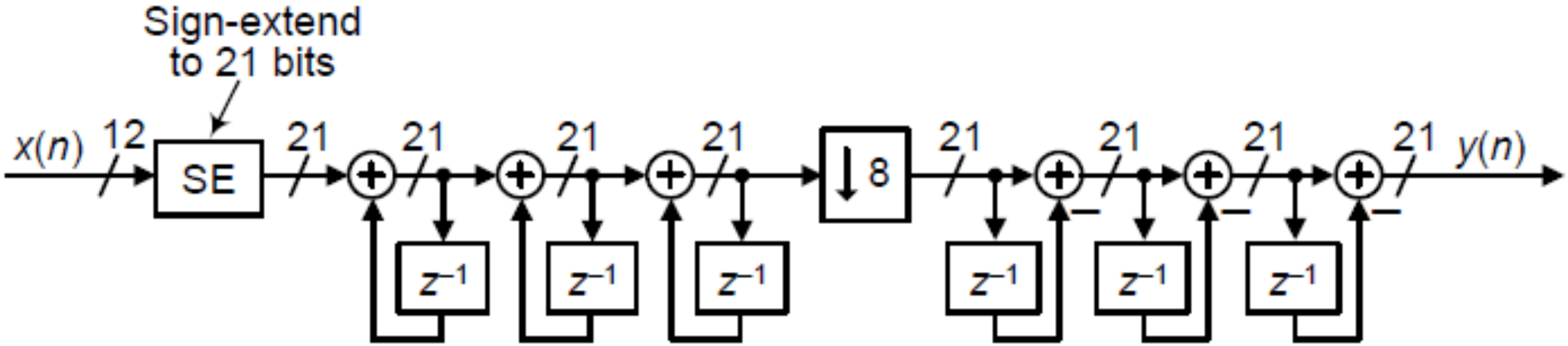
- The input iterator could easily be called zero or many times,
  allowing straight forward implementation of
  sample rate changing processes

Example: CIC and FIR filters

Cascaded Integrator-Comb (CIC) Filters

- Higher order running sum filter
- Decimating (down-sampling)

3rd Order



From: A Beginner's Guide To Cascaded Integrator-Comb (CIC) Filters
Rick Lyons, March 26, 2020
https://www.dsprelated.com/showarticle/1337.php

Impulse response of 4th order CIC

```
impulse = concatenate( zeros(Int64,10), 1, sequence(0) )
y = ( impulse
        |> Take(100)
        |> Integrator(1)
        |> Integrator(1)
        |> Integrator(1)
        |> Integrator(1)
        |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
        |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
        |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
        |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
        |> collect
)
```

4^{th} Order

This is equivalent to the earlier example – cascade of four moving average filters

Add Downsample() to make Decimator

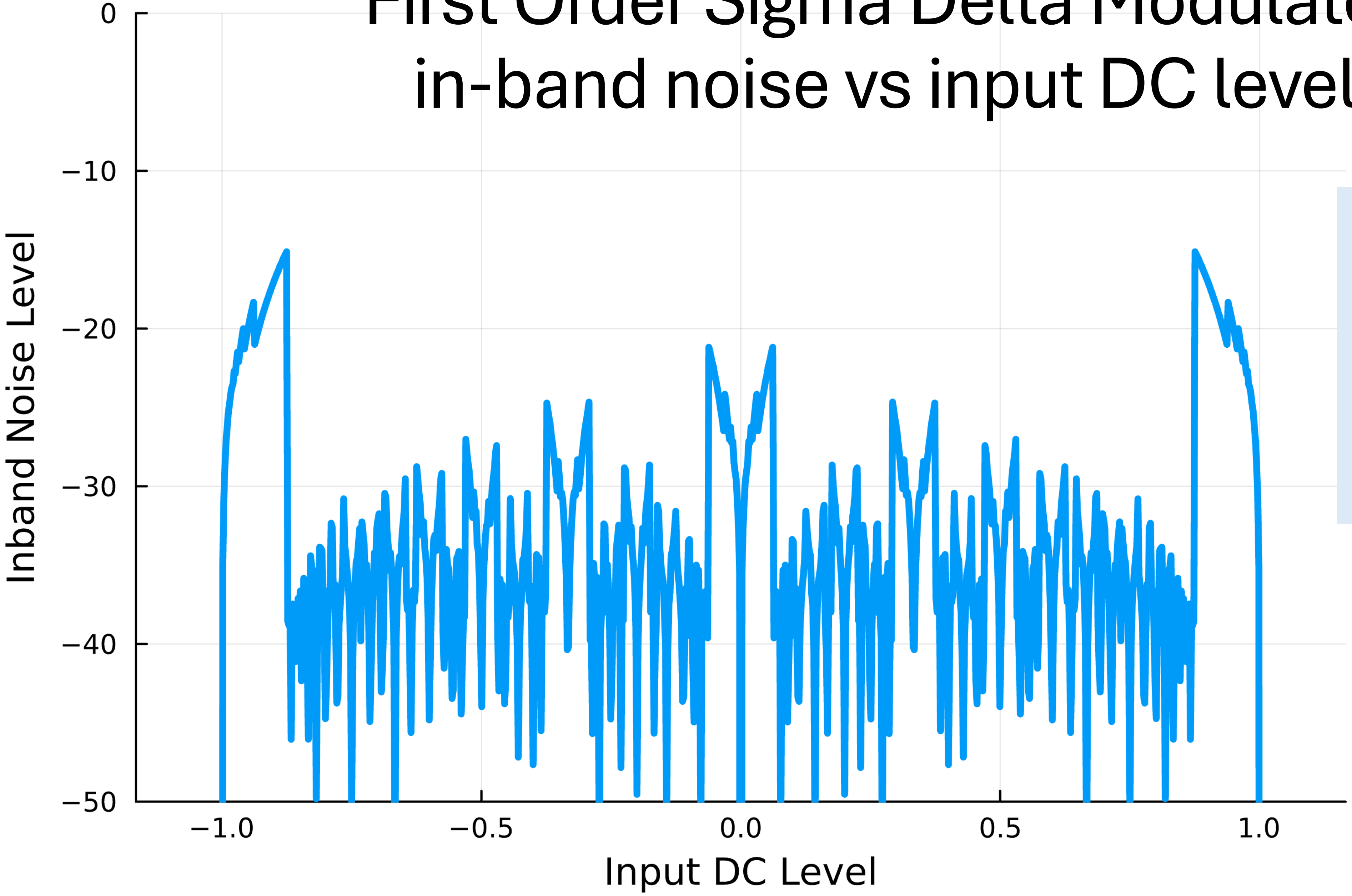
```
y1 = ( impulse
     |> Take(100)
     |> Integrator(1)
     |> Integrator(1)
     |> Integrator(1)
     |> Integrator(1)
     |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
     |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
     |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
     |> fir( [1,0,0,0,0,0,0,0,0,0,-1])
     |> Downsample(10)
     |> collect
)
```

```
y2 = ( impulse
     |> Take(100)
     |> Integrator(1)
     |> Integrator(1)
     |> Integrator(1)
     |> Integrator(1)
     |> Downsample(10)
     |> fir( [1,-1] )
     |> fir( [1,-1] )
     |> fir( [1,-1] )
     |> fir( [1,-1] )
     |> collect
)
```

These produce the same results, but the second one corresponds to a lot less hardware

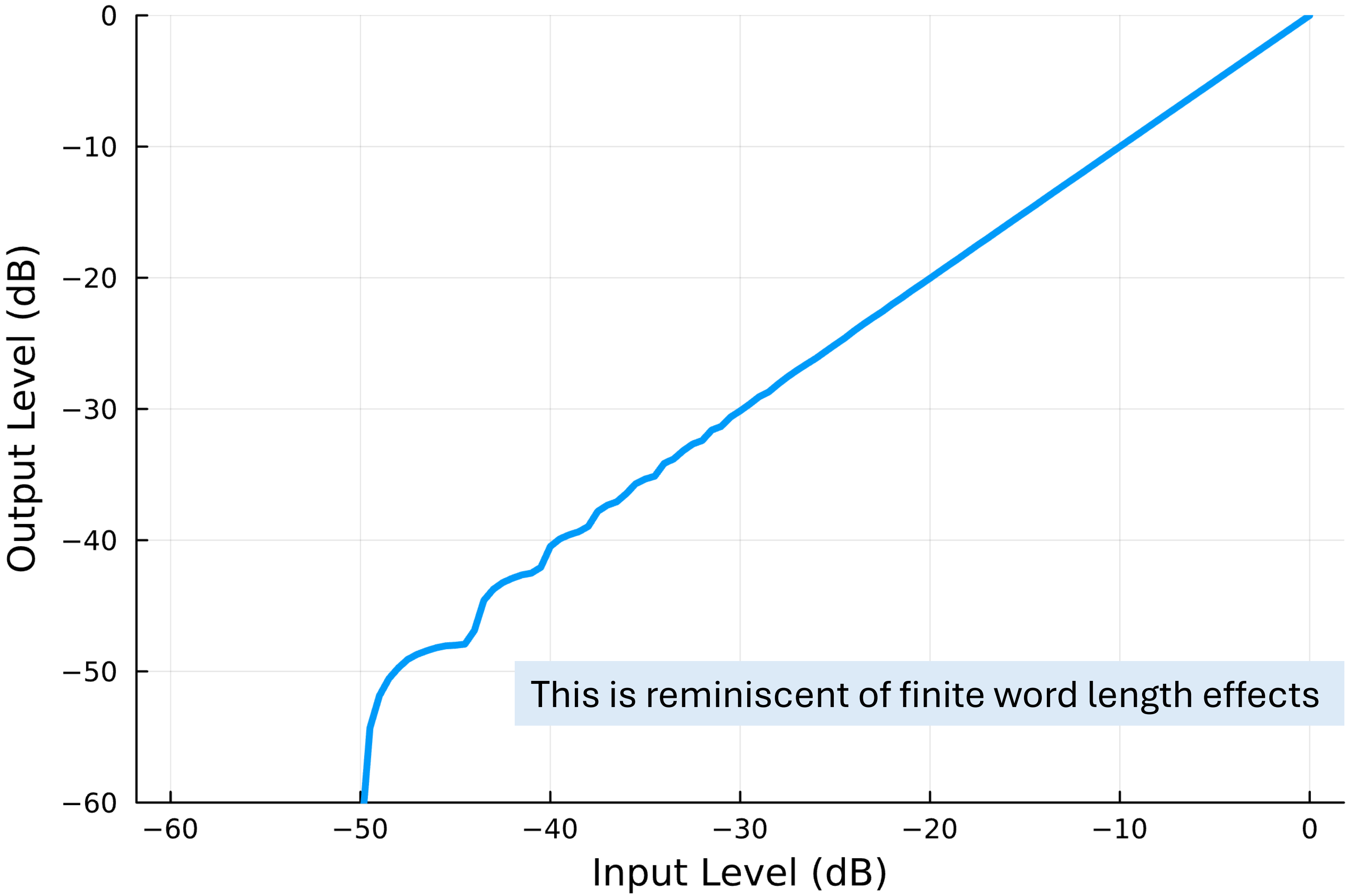
Example:
First Order Sigma Delta Modulator

First Order Sigma Delta Modulator
in-band noise vs input DC level



Sigma-delta modulation is a kind of pulse density modulation

First Order Sigma Delta Modulator output AC level vs input AC level



Evaluating effect of finite word-length arithmetic

Arithmetic Types – Fixed Point

- Longer wordlengths – more silicon area (cost, power), longer time
- Arithmetically correct
  - No overflow "wrap around" behavior by default
  - Adding two 8 bit numbers requires a 9 bit result

2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	unsigned value	ufix<6,4> value
b_1	b_0	b_{-1}	b_{-2}	b_{-3}	b_{-4}		
0	0	0	0	0	0	0	0
0	0	0	0	0	1	1	0.0625
0	0	1	0	0	1	9	0.5625
0	1	0	0	0	0	16	1
1	1	1	1	1	1	63	3.9375

https://schaumont.dyn.wpi.edu/ece4703b21/lecture6.html

- Keep track of wordlengths
  - For associativity, track bounds rather than number of bits
    - also eliminates the distiction between signed and unsigned
- Keep track of the fixed exponent (shift)

Julia equivalent, using fixed point library:

uFixPt(6,-4) returns the *type*

FixPt{ -4, Bint{0,63} }

Julia makes it easy to combine this with other libraries

Some other applications

- Characterizing existing digital designs
  - Build model using Verilator, wrap with Julia interface
  - Wrap to accept and process input
- Tokenization
  - File is sequence of characters of unknown length
  - Transform to sequence of tokens
- VCD file parsing
  - Value Change Dump (VCD) file format from Verilog simulators
  - Parsing to filtered list of signal change events as (time,value) tuples

Many applications beyond signal processing

Github repository, now and future
https://github.com/arghhhh/julia-signals-systems

Now:

- Basic examples as shown in this presentation
- Sequences: arithmetic
- Processors: upsample and downsample rate changes
- Signal Performance Metrics
  - Measure DC, AC levels, bandlimited noise using weighted least squares
- MIT License

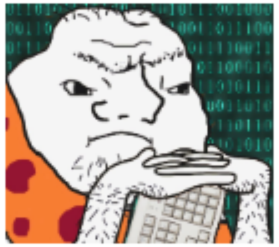
It is early days for this open source repo.
Do not expect to use it as a library.
Use it by looking at and copying it.

Github repository, now and future
https://github.com/arghhhh/julia-signals-systems

- **Future:**
  - Add more common signals and systems examples
    - sources
    - filters, mixers, encoders, decoders, modulators, demodulators of various kinds
    - Examples using Verilator
  - Fixed-Point arithmetic library
  - More performance analysis
    - Signal level, bandlimited noise measurements, SNR, THD, SINAD etc
    - (foundations are already in the repository)
  - Transfer Function stuff – filter design and eval

All of this has already been done – but not in open source environment

Summary



The Grug Brained Developer

A layman's guide to thinking like the self-aware smol brained

https://grugbrain.dev/

- Modelling
  - Signals using iteration protocol
    - Combinations of sequences, adding, multiplying, concatenation
  - Systems
    - Very simple framework – a few lines of code
    - Reduces to iterators with output samples computed as needed
    - Allows state
    - Allows rate changes
  - Modular and easily composed

Thanks to all the Julia developers who have made this all so easy to do

Reusing Parts

- Frequency domain processing of overlapping blocks
  - eg Windowed FFT with 50% overlap
- Sliding FFT

```
partition( n ) = (
        Vectorize( n )
        |> Downsample(n,n-1)
)

partition_with_overlap(n) = (
        Vectorize( n )
        |> Downsample(div(n,2),n-1
)
)
```