WIRE FRAMES

reference documentation v1.0 2017

WireFrames Library for Symbolic Sound Kyma 7
Design and Programming by
Cristian Vogel and Gustav Scholda

NEVERENGINE LABS™

SAFETY WARNING

Many WireFrames Classes output control data as audio. These are loud. They are not intended to be listened to. They should be resynthesised with an Oscillator Bank/Filter Bank or used in other ways to control other WireFrame modules - try not to play the prototypes directly as you may damage your ears or speakers. Listen to our examples instead, or study on silent FrameScopes.

This guide was compiled by Cristian Vogel based on technical documentation originally authored by Cristian Vogel and Gustav Scholda.

All artwork by Cristian Vogel
All rights reserved.
Published by NeverEngine Labs November 2017
www.neverenginelabs.com

Many thanks to all the subscribers at the WireFrames Lab who supported and motivated us during the development phase, to Pete Johnston for advice and inspiration. Of course huge thanks to Carla Scaletti and Kurt Hebel of Symbolic Sound for their massive support and for having created the awesome Kyma sound design environment upon which all our work is based.

Table of Contents

What is WireFrames?

WireFrames is a collection of encapsulated Kyma Sounds (**Classes**) and Examples. The Classes have been designed to work together, with shared key concepts, in order to allow a user to explore an advanced method of signal programming in Kyma.

The WireFrames library can be powerful and efficient but it requires the user to learn an alternative approach to Kyma programming. In this approach, we don't work with Capytalk scripting that much (although you can still use Capytalk everywhere in our Classes).

The bulk of WireFrames programming is done by manipulating the values of individual samples directly inside the signal processor - **the Wire**.

To do this, we define a container to hold a fixed length of sample points - **the Frame**.

What is a Frame?

A Frame then, is an audio signal, which has a fixed length counted in samples. Even though they are single samples, each element of the Frame is referred to as a **Track**[1] rather than a sample point. Each sample in the Frame is in a time based relationship with it's previous, current and next value, without affecting other sample values in the Frame - in other words, the values inside the Frame are running along their own virtual Tracks. When the Frame stream is running and correctly synchronised, the Tracks update so fast that their changing values form time varying signals.

Tracks update their values at the current **FrameLength**. The FrameLength also defines how many Tracks there are in a Frame. The FrameLength update speed will always be slower than the current sample rate and generally faster than the control rate (Capytalk and VCS widgets update every millisecond). FrameLength is chosen by the user and/or certain sound design contexts. For example, the Kyma spectrum analysis prototypes set their own FrameLength according to the lowest frequency analysed (you can read the Kyma LiveSpectralAnalysis parameter description for more details on that). For correct synchronisation, FrameLength should match throughout a WireFrames signal flow.

For example, at a frame length of 256, each Track can only be updated every 256 samples. To access Track 1 we must wait for 255 other Tracks to pass first. Therefore, Frame processing introduces a tiny amount of latency. The realtime duration of your chosen Frame length depends on your current sample rate too.

Evaluate `256 samp ms` in Kyma to see how fast that is at your current sample rate.

As it is made of audio samples, a Frame stream will run at the speed of the current sample rate in realtime. Kyma guarantees not to drop any samples along the way therefore the Paca(rana) will always keep Frame streams in sample accurate synchronisation.

[1] In the context of Kyma Spectrum data, the Track may also be described as a Partial, as each Track contains the information that will be resynthesised as single partial in an Oscillator bank, or filter bank for example.

Why work with WireFrames?

If you want to process the Frames generated by Kyma's Spectrum Analysis Classes or FFT, before the resynthesis stage, then WireFrames is pretty much essential.

If you are interested to learn an alternative way of thinking about realtime signal processing, by manipulating samples directly at a low level without using high level scripting.

If you are interested in using audio data itself as a kind of audio programming method.

If you are looking for ways to process large arrays of values in realtime with sample accurate 32 bit precision, and very fast. The best example of this application is when working with 'double wire' streams generated by Kyma's Spectrum analysis prototypes but there may also be applications for big data processing from controllers or other mass data streams.

If you are curious about what kind of music, sound or controller designs might emerge from the WireFrames approach to programming DSP.

SAFETY WARNING

Many of our Classes output frame streams. These are audio signals. They must be resynthesised with an Oscillator Bank/Filter Bank or used in other ways to control other modules - try not to listen to them directly as you may damage your ears or speakers. Listen to our examples instead, or study on silent FrameScopes.

In general, keep your volume low whilst programming in WireFrames until you are confident in what you are doing.

Some other important things to consider when working with WireFrames.

1. Keep your FrameLength in sync across your signal flow design.
2. A live spectrum analysis will define a Frame length , depending on what you have set to be the lowest frequency analysed. Please refer to the Symbolic Sound LiveSpectralAnalysis prototype documentation for more details.
3. Use FrameScopes a lot to understand better how WireFrames works.

About This Guide

This documentation guide, is derived from the inline parameter and Class descriptions in our encapsulated Classes. You will also find them there, when you are working in Kyma. This guide does not cover the many examples of WireFrames programming which you will find in the Library - some of them are internally documented in their VCS or Annotation objects.

Installation

All the prototypes can be added to your Prototype strip. This is quite straight forward
https://www.youtube.com/watch?v=xOvo6awCq3E

Frame Generators

WF RandomWalks

Generates many stochastic walks contained as Tracks in a Frame stream. The current value of each Track in the frame is randomly deviated from its previous value within a maximum distance of !StepSize. The change is smoothly achieved over the duration of !TimeConstant s. This Class can generate thousands of simultaneous walks with great efficiency. The update time constant and step size can be controlled independently for EACH random walk by using WireFrames FrameShaper Classes.

The TimeConstantFrameShaper input introduces the possibility to control individual TimeConstants for each walk. Please read below for a detailed description of what a FrameShaper does. Similarly the StepSizeFrameShaper input introduces the possibility to use individual step sizes for each walk.

By default, cosine interpolation moves a value smoothly between changes at sample rate.

`Absolute`
Optionally output postive values only.

`Seed`
An initial value for the noise source, for a repeatable random walk.

`StartingFrom`
Defines the start value of the RandomWalks when they get initialised by Initialise button.

`FrameLength`
The length of the frame in samples. Effectively a replicator of the random walks. For example, if you have a framelength of 256 the module will output 256 independent randomwalks updated every 256 samples.

`StepSize`
The maximum size of the random walk step.

`FrameScope`
Use the built-in FrameScope to visualise the frame.

`StepSizeFrameShaper`
As in our other NeverEngineLabs WireFrames classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a separate time constant for each 'walking Track' in the frame.

In NeverEngineLabs WireFrames a value of 1 received by a FrameShaper input will shape the frame in a positive direction, and a value of -1 inversely. A value of 0 at this input bypasses any shaping effect.

In the context of the WF RandomWalks, the FrameShaper input has been programmed to scale (or shape) the StepSize parameter for each random walk. Internally this has been set to 4th order scaling.

A value of 1 will scale the StepSize by 16 (16 times higher), a value of -1 scales the StepSize by 1/16.

The power of the FrameShaper concept comes clear when you work with frame synchronised signals, such as a WF LinearRamp. A linear ramp set to the same frequency in samples as the FrameLength specified in the WF RandomWalks (eg. 256 samp) will accelerate the deviation rates of the walks on the higher Tracks for example.

To control all Tracks equally use a WF ConstantFrameShaper and a fixed value, or a hotvalue or use some CapyTalk expressions. As mentioned above, to bypass any FrameShaper affect set a WF ConstantFrameShaper value to zero.

`InitialDeviation`
Defines the maximum deviation of each Track from the StartingFrom value, when they are initialised. When InitialDeviation is set to 0, all random walks will start from the same value set by StartingFrom.

`TimeConstant`
This represents the maximum duration a single stream takes to reach its new value in seconds.

`Initialise`
Initialises the RandomWalks according to startingFrom and initialDeviation

`TimeConstantFrameShaper`
As in our other NeverEngineLabs WireFrames classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a seperate time constant for each Track in the frame.

In NeverEngineLabs WireFrames a value of 1 received by a FrameShaper input will shape the frame in a positive direction, and a value of -1 inversely. A value of 0 at this input bypasses any shaping effect.

In the context of the WF RandomWalks, the FrameShaper input has been programmed to scale (or shape) the TimeConstant parameter for each random walk. Internally this has been set to 4th order scaling.

A value of 1 will scale the TimeConstant by 1/16 (16 times faster), a value of -1 scales the TimeConstant by 16 (16 times slower).

The power of the FrameShaper concept comes clear when you work with frame synchronised signals, such as a WF LinearRamp. A linear ramp set to the same frequency in samples as the FrameLength specified in the WF RandomWalks (eg. 256 samp) will accelerate the deviation rates of the walks on the higher Tracks for example.

To control all Tracks equally use a WF ConstantFrameShaper and a fixed value, or a hotvalue or use some CapyTalk expressions. As mentioned above, to bypass any FrameShaper affect set a WF ConstantFrameShaper value to zero.

`Interpolation`

Optionally use cosine interpolation at sample rate between previous value and next value. Disable this to save some DSP.

`JoinTogether`

Turn this on to let the tracks join together towards their average value.

WF ShapedPolySignal

Its the Multidimensional Mexican Wave of DSP.

This Class generates a new type of signal which we have decided to call a PolySignal.

It begins with a typical discrete signal, a wavetable which can be selected on the fly from a multicycle "necklace" of wavetables. This signal is then expressed through the Frame, like the other FrameShapers in WireFrames Library. Its shape is "plotted" using each Track in the Frame. A low resolution Framelength, say 64 samp, will express the shape using only 64 Tracks.

By allowing each Track to be modulated by an external Frame synchronised signal (such as a CurvilinearRamp) whilst it also reads through the values of the source wavetable , then a linear discrete signal transforms into a spiralling dephased *polysignal*.

This means that each track of the Frame whose static value is set to render the shape of the discrete wavetable through the Frame, can be offset or dephased, so that it begins to express different parts of the discrete signal its reading from, at its fixed Track position in the frame. In order to begin this de-correlation process, we need a base frequency which you can think of as a kind of LFO low frequency rate. This frequency is used by each Track in the Frame to look up its next value from the source wavetable, in other words to read through the wavetable. Basically, each Track is reading the source wavetable in sync, until we start to de-correlate the lookup positions of each Track using this low frequency oscillator, which is being slightly detuned for each Track.

What happens is that the Tracks begin to express the discrete wavetable shape along different timeindex paths, introducing a kind of other dimension to a one dimensional signal. The Frame continues to express the wavetable structure according to the each fixed Track position in the Frame (the X dimension). The Y dimension we can say is the amplitude value of each Track read from the Frame before it updates (at an update frequency of FrameLength samp). And then the PolySignal has this Z-dimension, a kind of distance or depth which can almost appear to be zooming in or out.

Take a look on the FrameScopes, try and understand it with your eyes, and then with your ears.

We used this PolySignal to great effect in our Mode based Physical Modelling synth, `NeverEngine Labs SciPhy`, to create dense micro modulations at different frequencies and phases such as you might find on the surface of a struck gong or cymbal.

The PolySignal may also have some special applications for granular synthesis, as well as complex additive synthesis forms and digital oscillators. It can also be used as SpectralLFO providing many unique phase effects when combined with a FrameMult or FrameModifier module and the Amps leg of spectrum analysis running at the same FrameLength.

AmpsFrameShaper

A frame shaper input for shaping the amplitudes of each track. A FrameShaper when correctly syncronised to the Frame Period of the module it is plugged to, will shape each Track of the Frame differently. What shaping affect this will be, is defined by the context of the WireFrame you are plugging into.

Remember a FrameShaper sample value of 0 here means no shaping, a negative value shapes downwards and a positive value shapes upwards.

BPM

The BPM of the base low frequency for dephasing the Tracks lookup points. Should be kept low for best effects, like an LFO.

MultiCycleWavetable

Select a MultiCycleWavetable here. A valid MultiCycleWavetable contains a 'necklace' of concatenated mono wavetables, each cycle must be exactly 4096 samples in length.
More wavetables available in the "NeverEngine Labs 4096 Volume One" available at our website or generate more with our ROMTools pack.

FrameLength

Set your framelength here. It makes sense to match it with the framelength of the spectral source which you are going to modulate, however it doesn't need to be: If you set it to 16 and use it on a 256 partials source you can modulate in groups of 16. Just make sure that the framelength of the source is an integer multiple to prevent the modulation from drifting. (Integer expected)

PhaseFrameShaper

A frame shaper input for shaping the position of each timeindex of each track, which it uses to get its next value from the discrete signal being used as the source wavetable.

For example:

Track 1 will look up sample 1 from the Wavetable
Track 2 will look up sample 2
Track 3 will look up sample 3 and so on

By shaping the phase you can change these relationships dynamically. So Track 1 might start to lookup sample 3 and Track 2 sample 10 and so on.

`FreqFrameShaper`
A frame shaper input for shaping the frequency offset (from the base LFO BPM) of each track. Remember a FrameShaper Track value of 0 here means no shaping, a negative value shapes downwards and a positive value shapes upwards.

`Reset`
When reset is changed from 0 to any non-zero value all Tracks will reset to their original lookup positions. Depending on the PhaseOffset that might not be the beginning of the cycle. Use it regularly to resynch everything.

`Index`
Selects the Wavetable from the MultiCycleWavetable file. 0 will use the first Wavetable, 16 will use the 15th Wavetable and so on.. In-between values will result in morphed wavetables.

WF SimplePolySignal

This Class generates a new type of signal which we have decided to call a PolySignal.

(This version is the Simplified version with built in FrameShapers. For more advanced version with external FrameShaper inputs see WF ShapedPolySignal)

Take a look on the FrameScopes, try and understand it with your eyes, and then with your ears.

`ShapeFrequencies`
Use this hotvalue to scale a built in FrameShaper that is affecting the Frequencies offsets. For more advanced applications, see the WF ShapedPolySignal Class

`ShapePhases`
Use this hotvalue to scale a built in FrameShaper that is affecting the Phase offsets. For more advanced applications, see the WF ShapedPolySignal Class

`BPM`
The BPM of the base low frequency for dephasing the Tracks lookup points. Should be kept low for best effects, like an LFO.

`MultiCycleWavetable`
Select a MultiCycleWavetable here. A valid MultiCycleWavetable contains a 'necklace' of concatenated mono wavetables, each cycle must be exactly 4096 samples in length.More wavetables available in the "NeverEngine Labs 4096 Volume One" available at our website.

`FrameLength`
Set your framelength here. It makes sense to match it with the framelength of the spectral source which you are going to modulate, however it doesn't need to be: If you set it to 16 and use it on a 256 partials source you can modulate in groups of 16. Just make sure that the

framelength of the source is an integer multiple to prevent the modulation from drifting. (Integer expected)

`Reset`

When reset is changed from 0 to any non-zero value all Tracks will reset to their original lookup positions. Depending on the PhaseOffset that might not be the beginning of the cycle. Use it regularly to resync everything.

`Index`

Selects the Wavetable from the MultiCycleWavetable file. 0 will use the first Wavetable, 15 will use the 16th Wavetable and so on.. In-between values will result in morphed wavetables.

WF SequencedFrame

SequencedFrame allows you to create audio rate Frame streams from Collections of VCS widgets, hot values, Capytalk expressions or fixed values.

The SequencedFrame can process the Collections before they become Frame streams using ShuffleSort, StretchFill and RepeatFill. These transformations are programmed in Smalltalk which means they are executed before being sent to the Pacarana. This means the transformations cannot be changed once the sound has started running. Nevertheless, once the Smalltalk Collection exists in the Frame domain, running at sample rate 'in the wire' , the control data can undergo sample accurate processing (blurring, averaging, shifting, randomising, shaping etc) very efficiently using the rest of Classes in the NeverEngineLabs WireFrames and SPCSR ibraries.

If you have less elements in your Capytalk Array than the FrameLength, which is often the case, you are provided with strategies for filling the Frame using only the data available.

Stretch fills the Frame proportionally with copies of each of the Capytalk (hot) values. For example if you have only 8 VCS widgets you would like to construct into a Frame of 256 elements, the Stretch fill will stretch the value of each widget over 32 samples, effectively stretching the data available out to fill the Frame.

The other method is to repeat each of the Capytalk Array elements in order until the Frame is full. For example if you have only 8 VCS widgets you would like to construct into a Frame of 256 elements, the repeat fill will repeat the Capytalk array 32 times. This means that when you move one of the 8 VCS faders, its movement will be repeated 32 times periodically throughout the frame.

Try these modes on the Framescope to understand better how they work.

Shuffle (with seed) and retrograde (aka reverse) will be peformed after the above distribution strategies, in the order of transformations.

One further interesting feature for the composer, is that each Frame Array can be masked by a boolean sieve. These sieve arrays can have a different length to the Frame Arrays, but the same distribution transformations still apply (Shuffle, stretch etc). Therefore you can mute and unmute values in the frame data using these masks. If you use time varying expressions here, for example { 1 bpm: 20 } { 1 bpm: 30 } { 1 bpm: 60 } you can easily sequence complex dynamic density effects. To bypass the sieves, set both parameter fields to a single value of 1.

Tip: One of the big advantages of converting a Capytalk array to a Frame , is that it wont take much more DSP to process 8 hotvalues than it does to process 4096. Yes a delay of Framelength samp is introduced in order to process every value, but in almost all Kyma workflows (expect for live input processing) this is not a major problem.

Scale
Scale lets you scale, mute and also invert (with negative scale values) the resulting Frames.

DoubleWire
Set to true if you would like to generate paired Frame streams, travelling in parallel on the left and right outputs of Class.

Clip01
Clip the range of each element to lie between 0 and 1 (inclusive), never outputting negative values

SecondaryFrameArray
Enter an array of values seperated by spaces. Enclise arithmetic expressions or units or in curly braces, for example:

!Value1 {1 repeatingRamp: 3 s * !Value2} !Value3 {1 s random}

The resulting Frame derived from the Secodnary Array will be output on the right channel of the Class when in DoubleWire mode and ignored when in single wire.

FrameLength
Frame period should match with other WireFrames modules in your sdesign, for correct functioning of the internal algorithms. Tip: Use the ?FrameLength variable to set them all at once.

SecondarySieveArray
Each Frame Array can be masked by a boolean sieve. These sieve arrays can have a different length to the Frame Arrays, but the transformations same distribution transformations still apply. Therefore you can mute and unmute values in the frame data using these masks. If you use time varying expressions here, for example

{ 1 bpm: 20 } { 1 bpm: 30 } { 1 bpm: 60 }

you can easily sequence complex dynamic density effects.

To bypass the sieves, set both parameter fields to a single value of 1.

FrameScope

When checked, a FrameScope will appear in the VCS showing the Frame data being output by this Class

`SeedForShuffle`

A seed for repeatability of the shuffle sort. If you like the result of a shuffle sort you can stick to it, in other words.

`LogScale`

When working in DoubleWire mode check this box to output the Frequencies in log rather than linear frequency. In most cases, this box should be unchecked; the only time it should be checked is if you want to manipulate the frequency envelopes in pitch space rather than in hertz.

`ShuffleSort`

Shuffle the elements of the array before converting into a Frame. Shuffle randomisation happens before a retrograde.

`PrimaryFrameArray`

Enter an array of values seperated by spaces. Enclise arithmetic expressions or units or in curly braces, for example:

```
!Value1 {1 repeatingRamp: 3 s * !Value2} !Value3 {1 s random}
```

The resulting Frame derived from the Primary Array will be output on the Left channel of the Class when in DoubleWire mode or both channels when in single wire mode.

`StretchFill`

Stretch fills the Frame proportionally with copies of each of the Capytalk (hot) values, effectively stretching the data available out to fill the Frame.

`PrimarySieveArray`

Each Frame Array can be masked by a boolean sieve. These sieve arrays can have a different length to the Frame Arrays, but the transformations same distribution transformations still apply. Therefore you can mute and unmute values in the frame data using these masks. If you use time varying expressions here, for example

```
{ 1 bpm: 20 } { 1 bpm: 30 } { 1 bpm: 60 }
```

you can easily sequence complex dynamic density effects.

To bypass the sieves, set both parameter fields to a single value of 1.

`RepeatFill`

Repeat each of the Capytalk Array elements in order until the Frame is full.

WF SimpleArrayToFrame

This is a very basic low level class translating a Capytalk array into frame data (an WireFrames array of audio samples). Every single sample of the frame corresponds to every single element in the Array. Make sure the size of your array matches the FrameLength.

`Array`
Enter an array here, remember to put expressions in curly braces {!value * 0.5}. Of course you can also use SmallTalk to generate your arrays, for example using the collect: message.

`Scale`
This is an overall scaling factor that applies to the whole frame. [Range -1,1]

`FrameLength`
The FrameLength should match with the Array size, and most of the time also with other WireFrames modules in your design, for correct functioning of the internal algorithms. Tip: Use a script to distribute the FrameLength to all modules at once.

WF FunctionGeneratorAR

Generates ?Framelength amount of Attack and Release functions as a Frame signal. To gate an AR there needs to be a positive value at the TrackGates Input. In other words, a particular AR outputting on a particular Track number, will only trigger when it receives a positive value on its Track at the TrackGates input.

`AttackFrameShaper`
Apply frameshaping to the AttackTime.

`ReleaseFrameShaper`
Apply frameshaping to the ReleaseTime.

`ReleaseTime`
Defines the ReleaseTime.

`AttackTime`
Defines the AttackTime.

`TrackGates`
Any value higher than 0 read from a particular Track number of the incoming Frame stream, will gate an AR function that will run on the same corresponding Track number in the Frame stream generated by WF FunctionGeneratorAR.

`FrameLength`
Should match with the FrameLength of the frame you want to modify using the ARs.

WF RandomGenerator

Generates frames containing random values updating every TimeConstant s.

`Absolute`
If turned on the random values will be positive only.

`Scale`
Scales the output.

`Seed`
Defines the seed of the random generator.

`CenterValue`
Defines the centerValue of the uniform distributed random values.

`TimeConstant`
Defines the time interval the frame gets fille with new random values. If interpolation is checked it defines the speed of the random values changing.

`FrameLength`
For correct synchronisation this integer parameter, the FrameRate value, should match that of the WF Module you are plugging into (and as a general rule of thumb, the Framerate of the entire WireFrames signal flow you are designing)

`TimeConstantFrameShaper`
As in our other NeverEngineLabs WireFrames™ classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a separate time constant for each Track in the frame.

`Interpolation`
Turn on if you wish to smoothly interpolate between the random values.

WF ReadFrames

Use this class to read frames you previously recorded using `WF WriteFrames`.

`ChannelNbr`
If you have used the ChannelNbr option you can choose which channel to use here. Use 0 to bypass this function.

`FrameLock`
Locks the frame in a way that you will always read the full frame before updating the TimeIndex and reading the next. If you unlock it however you can for example add some noise to the TimeIndex to read out tracks from surrounding frames to construct a unique frame on each cycle.

`SRCompensation`
If you have recorded spectra it's always a good idea to check this. If it's some other type of frame based data better uncheck it.

`Filename`
Choose the desired file previously recorded using WF WriteFrames

`TimeIndex`
Controls the position of your readout

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes. You can choose a shorter frameLength than your recorded file is.

WF ReadFrames RAM

Use this class to play back recordings made with WF WriteFrames RAM.

`DoubleWire`
If you play back a double wire recording you need to check this.

`RecordingName`
This should match with the RecordingName in the accompanied WF WriteFrames RAM class.

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes. You can choose a shorter frameLength than your recorded file is.

`TimeIndex`
Controls the position of your readout

`FrameLock`
Locks the frame in a way that you will always read the full frame before updating the TimeIndex and reading the next.

WF SwarmDynamics

The Swarm Dynamics Class belongs to a set of Classes which work together to enable the creation and exploration of virtual multi-agent simulations (aka Artificial Life simulations or simply virtual swarms) in Kyma.

This Class is the core class which handles the physics modelling for each agent (aka drone) in a single swarm system. A swarm can be of any size higher than 4 agents, and should always be an even number (the same ammount of DSP will be used for a swarm of 16 as for a swarm of thousands - this is the power of Frame based programming!)

The InputFrames field contains the attractor points for each agent. The agent will move to that position according to its Acceleration, Velocity and Friction characteristics. You can use an arbitrary number of InputFrames, which means that there can be more than one attractor for each agent.

Using the hotArrays AttractorRange and AttractorScale you can change the range and the power of each attractor from left to right. For example, if you have 3 different attractor signals at the input, you could enter an Array of hotvalues such as !AttractorScale1 !AttractorScale2 !AttractorScale3

You have three FrameShaper inputs available, to sculpt and add difference to the physics of each of the Drones in the Swarm. Read below how to work with this powerful concept.

The output of this Core Class, which handles the dynamics and physics of the Swarm, can resynthesised directly, or used as a FrameShaper itself in other WireFrames contexts.

To explore even more advanced and complex dynamics, use the WF SwarmIntelligence as input which computes nearest neighbour awareness and cohesion/evasion dynamics. It is with these types of feedback mechanisms that true emergent behaviours can begin to emerge.

For further study, here is a good paper by the team at the ICMS in Zurich University of the Arts introducing the conceptual background to swarm simulations in music http://swarms.cc/files/SwarmMusic_GA_2008.pdf

`InitialDeviation`
Defines the initial deviation of the agents.

`Initialise`
Places all agents to random positions again.

`InputFrames`
This should be a mono frame based signal and defines the attractor points for each agent. Use a constant if you want all agents to share the same attractor point. Or use a spectrum to get each agent attracted by the current amplitude or frequency (depending what you are using) of the spectrum. You can combine an arbitrary number of attractors by using multiple inputs.

AttractorRange

AttractorRange is the range of an attractor signal, if an agent is out of range it doesn't get attracted at all.

When more than one Attractor Input Frame is used (you can drag and drop Sounds into the input frames field, like a Mixer) then you can define the power of each input signal individually, from left to right. In other words, this field is a hotArray - and as elsewhere in Kyma, CapyTalk expressions must be put in curly braces.

```
For Example:     {!AttractorRange1 * (1 repeatingTriangle: 5 s) }  {!
AttractorRange2 * (1 repeatingTriangle: 10 s} etc.
```

Seed

Defines the seed of the random generator used to initialise the agents.

AttractorScale

AttractorScale scales the power of the attractors at the InputFrames input, in other words how strongly it attracts agents to move to that position. A negative value turns the attractor into a repellor pushing agents away instead of attracting them.

When more than one Attractor InputFrame is used (you can drag and drop Sounds into the input frames field, like a Mixer) then you can define the power of each input signal individually, from left to right. In other words, this field is a hotArray - and as elsewhere in Kyma, CapyTalk expressions must be put in curly braces.

SwarmID

All classes that belong to one swarm should share the same ID.

AttractorType

AttractorType defines the type of the attractors at the InputFrames input. There are 3 types and you can smoothly go from one to the other. A positive value results in a spring like attractor where the force increases with the distance (if you attach an object to a spring the force increases the more you stretch the spring); a negative value results in a magnetic attractor where the force decreases with distance (similar how a magnet attracts a metal object the force is strongest when close to the magnet); a value of 0 results in a static attractor where the force is constant no matter the distance.

FrameLength

For correct synchronisation this integer parameter, the FrameRate value, should match that of the WF Module you are plugging into (and as a general rule of thumb, the Framerate of the entire WireFrames signal flow you are designing)

Acceleration

Defines the Acceleration for all agents. To introduce individual characteristics for each agent use the according FrameShaper input.

`Velocity`

Defines the Velocity for all agents. To introduce individual characteristics for each agent use the according FrameShaper input.

`Friction`

Defines the Friction for all agents. To introduce individual characteristics for each agent use the according FrameShaper input.

`VelocitiesFrameShaper`
`FrictionsFrameShaper`
`AccelerationsFrameShaper`

As in our other NeverEngineLabs WireFrames™ classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a seperate velocity for each Track in the frame.

WF SwarmIntelligence

This class needs to be used as input to a WF SwarmDynamics class sharing the same SwarmID. It outputs attractors based on calculations on the agents positions letting them interact in an intelligent way.

`ActiveAgents`

Defines the number of active agents. Inactive agents will be sent to position 0 and will not affect the active agents.

`EvasionReactionTime`

Defines the time it takes for an agent to switch from cohesion state to evasion state.

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`AgentRange`

Each agent behaves like an attractor. A small range favours small individual flocks. To get a single flock set the range to 1.

`Initialise`

Initialises all calculations. Most useful when triggered at the same time when you initialise the WireFrames SwarmDynamics class.

`BooleanStates`

Any positive value here will turn Boolean States on: The agents switch their behaviour immediately and can be only in one state at the same time. If the value is zero or negative BooleanStates is switched off: The agents get emotional, changing their state fluently between cohesion and evasion. For example: An agent is close enough to another one to make it feel uncomfortable. Instead of switching to evasion behaviour, the agent slightly evades. If the other agent is still too close it will evade even more. The overall effect is a smooth transition from cohesion to evasion resulting in smoother movements of the agents.

`ReactionTimeFrameShaper`

A WF LinearRamp set to the same frequency in samples as the FrameLength specified in the this Class will increase the reaction of the agents on the higher Tracks for example.
To control all Tracks equally use a WF ConstantFrameShaper and a fixed value, or a hotvalue or use some CapyTalk expressions. As mentioned above, to bypass any FrameShaper affect set a WF ConstantFrameShaper value to zero.

`EscapeSpeedFrameShaper`

`CohesionReactionTime`

Defines the time it takes for an agent to switch from evasion state to cohesion state.

`Seed`
Defines the seed of the random generator.

`DistanceAwareness`
If set to zero the agents are not aware of the distance to the nearest neighbour, meaning that they are either comfortable or uncomfortable with it. The higher the DistanceAwareness the more they are aware of it: If another agent is in the uncomfortable zone the evasionReactionTime will be faster the closer the agent is. Similary if the nearest neighbour is distant enough to feel comfortable the cohesionReactionTime will be faster the more distant it is.

`Sensitivity`
Defines the sensitivity to close agents. In other words it defines the threshold at which the agents will change their state.

`EscapeDirection`
Defines the direction the agents will take when escaping (Evasion state).

`Stability`
When Stability is 1 the position of the swarm will more or less stay in the middle (around zero). If you decrease the Stability the swarm will more and more drift around in the space, moving up and down.

`EscapeDistance`
Defines the maximum distance the agents will escape (Evasion state).

`SwarmID`
All classes that belong to one swarm should share the same ID.

`EscapeSpeed`
Defines the speed of the random generator used to generate the escape positions (Evasion State).

WF AgentStates

This class only works properly with an according WF SwarmIntelligence in the same network sharing the same SwarmID. It outputs a frame-based signal for each agent with the value specified in the Cohesion parameter when in Cohesion state and the value specified in the Evasion parameter when in Evasion state. For in-between states (BooleanStates in the WF SwarmIntelligence is off) it interpolates in-between the values specified.

`Cohesion`

Defines the output value when in Cohesion state.

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`SwarmID`

All classes that belong to one swarm should share the same ID.

`Evasion`

Defines the output value when in Evasion state.

WF NearestNeighbourFitnessFunction

This algorithm will compare each element in a Frame to every other element, and compute its nearest neighbour, a neighbour being defined as the value which is closest to its own.

The algorithm then calculates the magnitude, or distance, to that nearest neighbour. The output of the fitness function is a Frame stream of the same Frame Period as the input, with the result of each distance computation sequenced in order.

Fitness functions, especially when combined with feedback, are part of the grounding theory of Cybernetics and dynamic systems. They have applications in artificial intelligence, molecular biology, economics and many more fields of study.

`Absolute`

If absolute is checked the output will be the absolute distance to the nearest neighbour. If unchecked the sign will be considered, meaning that if the nearest neighbour is lower in value the distance is negative, and if it is higher the distance is positive.

`InputFrames`

This is the input frame stream which will be processed through the fitness function algorithm. As with all WireFrames modules, for correctly synchronised results the frame period (aka frame length) should be the same throughout the signal network.

`FrameLength`

The Frame period of the frame stream at the input

WF FractalNoise

This kind of Noise lets you choose a fundamental frequency and arranges a set of other noise generators in octaves above. By changing the Persistence you can define how much of that octaves get mixed in - or mathematically LevelOctave(n) = Persistence^n. The module takes care of normalizing the output so when changing the Persistence the Level stays the same.

`AbsoluteValue`
output only positive values

`NumberOctaves`
Number of replicated noise octaves. DSP usage will increase as this Constant increases.

`FreqLow`
fundamental noise update frequency

`Persistence`
How much each octave gets expressed in the final noise stream. Equivalent to Roughness.

`InitialState`

`Scale`
Output scaling

`Interpolation`
interpolate smoothly between noise values

WF LorenzAttractor

Outputs the x and y positions of a Lorenz Attractor.
Useful when used in combination with a Swarm. But also a very nice complex modulator on its own. Read the values of inidividual points using a WireFrames FrameToGlobalControllers. Alternatively, to access the computed postions of points in the Lorenz function, feed the LorenzAttractor into the 2DVisualiser Class and Tool. This will automatically generates hidden !x00 !x01 !x02 etc) and let you see all the amazing dynamic patterns this algorithm can generate in 2D space
More information on the Lorenz Attractor can be found here: http://paulbourke.net/fractals/lorenz

A

Defines the A coefficient.
You must include a !HotValue in this field, or Kyma will throw an error. If you wish to use a Sound or Capytalk expression, please multiply it by a 'workaround' hotvalue.
`For example:  !constant * (1 repeatingRamp: 30)`

`ScaleX`

Scales the x values.

B

Defines the B coefficient.
You must include a !HotValue in this field, or Kyma will throw an error. If you wish to use a Sound or Capytalk expression, please multiply it by a 'workaround' hotvalue.
For example: `!constant * (1 repeatingRamp: 30)`

`ScaleY`

Scales the y values.

C

Defines the C coefficient.
You must include a !HotValue in this field, or Kyma will throw an error. If you wish to use a Sound or Capytalk expression, please multiply it by a 'workaround' hotvalue.
For example: `!constant * (1 repeatingRamp: 30)`

`SourceX`

The Lorenz Attractor is a 3 dimensional attractor. Choose which axis should be plotted on the x-axis:
0 -> X
1 -> Y
2 -> Z

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`SourceY`
The Lorenz Attractor is a 3 dimensional attractor. Choose which axis should be plotted on the y-axis:
0 -> X
1 -> Y
2 -> Z

`Initialise`
Initialises the differential equations and starts the trajectories from the beginning.

`Velocity`
Defines the velocity of the trajectories.

`OffsetX`
Offset the x values.

`OffsetY`
Offset the y values.

Frame Shapers

WF Morph1DFrameShaper

FrameShapers generally plug into FrameShaper inputs. They provide a neat way to alter many parameters simultaneously, which is often the case when working with Frame data. The shape of the wavetable specified in the FrameShaper, becomes expressed upon the Frame data.

Remember those Pin Art 3D scultpure toys from the 90s, where you push your hand into a matrix of pins, and they take on the shape of your hand? This is a bit like how a FrameShaper acts on Frame data. The wavetable shape selected in the Wavetable field would be equivalent to hand or other object. The Frame data it acts upon would be equivalent to the matrix of pins.

A FrameShaper when correctly syncronised to the Frame Period of the module it is plugged to, will shape each Track of the Frame differently. What shaping affect this will be, is defined by the context of the WireFrame you are plugging into.

For example, in the WF RandomWalks one FrameShaper input alters the time constant of the each random walk. In this case a WF LinearRamp FrameShaper would alter the deviation rates of the walks on the higher Tracks more than the lower ones. A FrameShaper sending a Gaussian wavetable would run the walks in the middle of the Frame at the specified Time Constant, whilst slowing down the rate of the walks at the bottom and at the top. Try it!

TIP:
Any Classes in the WireFrames library that can be affected by a FrameShaper, work in a bidirectional way around 0. What that means is that when the sample value of a FrameShaper is at 0 it won't affect. When it is negative it will shape the parameter downwards, when it is positive it will shape it upwards.

For example:
If you used a WF LinearRamp FrameShaper, set to FrameLength 128 and rising from -1 to 1, to shape the time constants of a WF RandomWalks generating 128 independent random walks with a global time constant of 3 s..... well, the first 64 walks will be updating much slower than the second 64. The first walk (Track 1) will be updating the slowest and the last walk (Track 128) will be updating the fastest. Track 64 in the middle will be updating at exactly the global time constant.

Absolute

Set to true, if you want to output positive values only. Any negative values will become positive. This happens before Clip01 when you use them together.

Morph

Morphs smoothly when moving from one wave to another. In the range on 0 to 1 like the other Morph1D Classes in Kyma.

Bandwidth
Controls the bandwidth or tightness of the output shape. Similar to the idea of bandwidth in EQs, except here the shape is completely arbitrary, not just a gaussian shape as in most EQs. A bandwidth of 1 would stretch the shape over full frameLength and 0.001 would be a very tight version occupying only a fraction of the wavetable. A bandwidth higher than 1 will stretch the shape, you can then use scrub to select a portion of the stretched shape.
You could use bandwidth in combination with offset and scale, to move a narrowed version of the shape through the Frame.

Offset
Offsets the output signal up or down.

Clip01
Clips the source wave to its positive values only. Negative values are set to 0.

Scale
Allows you to scale the output. Negative values inverts the shape, 0 makes it flat and positive scales upwards.
Remember that normally in WireFrames, values of 0 recieved by a FrameShaper Input bypasses any effect of the FrameShaper. Therefore this Scale can act like an attenuator on the overall affect of the FrameShaper.
Example: When using the FrameShaper to perform spectral pitch shifting using Product1 or FormantShifter, a value of 0 does no pitch shifting, a value of -1 shifts downwards, and 1 upwards. Therefore if you are sending a 1-0 linear ramp shape, the higher partials on the highest Tracks of the Frame will not be pitched, but the lower ones will be shifted upwards.

Shapes
A list of shapes to morph through

Fade
Perform a fade in or out on the shape , depending on fade direction

FadeDirection
Fade in or fade on the shape. 1 fades out, 0 fades in, continuously variable.

Shift
Horizontally shift the phase of the output shape. At 0 there is no shift.
Shifting a FrameModifier signal effectively allows you to move its affects through the Frame it is modifying, maybe affecting lower Tracks more than higher ones. You could use it in combination with BandWidth to isolate and scale only a tight region of Tracks in a target Frame.

FrameLength
Defines the length of the frame in samples and should usually match with the other WireFrames classes

FrameScope
Check this if you want to use the internal FrameScope.

WF FrameShaper

A 'necklace' version of the Morph1D FrameShaper, which can read from a multicycle wavetable generated in NeverEngineLabs ROM Tools or other method.

`Absolute`
Set to true, if you want the FrameModifier to output positive values only.

`Scale`
Allows you to scale the output. Negative values inverts the shape, 0 makes it flat and positive scales upwards.

Remember that normally in WireFrames, values of 0 received by a FrameShaper Input bypasses any effect of the FrameShaper. Therefore this Scale can act like an attenuator on the overall affect of the FrameShaper.

Example: When using the FrameShaper to perform spectral pitch shifting using FrameMult or FrameModifier1 or FormantShifter, a value of 0 does no pitch shifting, a value of -1 shifts downwards, and 1 upwards. Therefore if you are sending a 1-0 linear ramp shape, the higher partials on the highest Tracks of the Frame will not be pitched, but the lower ones will be shifted upwards.

`Bandwidth`
Controls the bandwidth or tightness of the output shape. Similar to the idea of bandwidth in EQs, except here the shape is completely arbitrary, not just a gaussian shape as in most EQs.

A bandwidth of 1 would stretch the shape over full frameLength and 0.001 would be a very tight version occupying only a fraction of the wavetable. A bandwidth higher than 1 will stretch the shape, you can then use scrub to select a portion of the stretched shape.

You could use bandwidth in combination with offset and scale, to move a narrowed version of the shape through the Frame.

`Scrub`
If you have a multicycle wavetable (aka "necklace" or "ROMBank") loaded you can scrub through it using this. Also if your bandwidth is higher than 1 you can scrub through the table to select a portion of it.

`FrameLength`
This integer parameter should match that of the WF Module you are plugging into for correct synchronisation (and as a general rule of thumb, the frame period should match throughout an entire WireFrames signal flow)

`ShapeIndex`
Morph smoothly between shapes in a multicycle wave table if Morph is set to true. 0 selects the first wave in the necklace, 1 the second and so on. If Morph is not set to true, then the shape index will snap to the new value, rather than morph smoothly.

FrameScope
Check this if you want to use the internal FrameScope.

Shift
Horizontally shift the phase of the output shape. At 0 there is no shift.

Shifting a FrameModifier signal effectivley allows you to move its affects through the Frame it is modifying, maybe affecting lower Tracks more than higher ones. You could use it in combination with BandWidth to isolate and scale only a tight region of Tracks in a target Frame.

Morph
If you have selected a multicyclewavetable (made with NE Labs WaveGarden or otherwise) then here you select if it morphs smoothly when moving from one wave to another waves with !ShapeIndex. You can uncheck this if you need more DSP efficiency.

Wavetable
Load a standard Kyma 4096 sample wavetable or multicycle wavetable.

TIP: Some built in signals you could just type in without having to browse for a file:

ramp - a positive only linear ramp
fullramp - a full range linear ramp
gaussian - a positive only gaussian distribution
random - a random wavetable

or

Load in a mutliCycleWaveTable such as one created using the NeverEngineLabs ROMTools and morph wavetables from it using the !ShapeIndex parameter.

Offset
Offsets the output signal up or down.

Reset
A non zero value resets the cycle of the FrameModifier wavetable shape. Be aware that you are loosing sync when resetting.

WF ConstantFrameShaper

Plug this to a FrameShaper input when you want to shape all values in the destination frame, to the same value.

In the NeverEngineLabs WireFrames system, FrameShaperInputs perform no shaping at 0
....positive shaping for positive values
....inverse shaping for negative values

In other words, a value of 1 received by a FrameShaper input will shape the frame in a positive direction, and a value of -1 inversely. A value of 0 at a FrameShaper input bypasses any shaping effect.

The WF ConstantFrameShaper sends all Tracks in the Frame the same scaling value, when plugged into a FrameShaper input. That value could be a fixed value, or a hotvalue or use some CapyTalk expressions. As mentioned above, to bypass any FrameShaper affect set a WF ConstantFrameShaper value to zero.

When double wire mode is selected, two separate constants will be output, the primary value on the left channel of the dual signal output and the secondary value on the right.

When in single wire mode, the primary value is output on the left only.

`PrimaryValue`

When in single wire mode, the primary value is output on the left only.

`SecondaryValue`

When double wire mode is selected, two separate constants will be output, the primary value on the left channel of the dual signal output and the secondary value on the right.

`DoubleWire`

Set to true if you are working with paired Frame streams, travelling independently on the left and right outpus of Class. The most common examples of this type of signal is a Spectrum Frame stream. A Kyma Spectrum Frame (and NeverEngine Labs SPC Frame) will use the dual streams of information running on the 'double wire' to sonify (or resynthesise) the Frame data using an Oscillator bank or one of the other Kyma prototypes that resynthesise from Spectrum Frames.

Double wire spectrum data consists of one frame stream on the left channel carrying the Amplitude values that will specify the volume of each partial, and likewise one stream on the right channel with data that specifies the Frequencies for each partial.

WF CurvilinearRamp

Frame synchronised ramp function with exponential curvature.

One specific use case would be a spectral tilt EQ on a Spectrum Frame stream. Use the curved ramp on the Amps stream. Set it to falling half range (1-0). This will allow lower partials to pass through unaffected whilst reducing the amplitudes of higher partials. Introducing some curvature allows you to explore different loudness curves and their psychoacoustic effects.

Another example: A falling full range ramp used as a PitchShifting or FormantShifting Frame shaper would shift the pitch of the lower Tracks upwards and the higher Tracks downwards.

`AllowCurvature`

Set to true of you want to work with exponential smooth curves using the Curve parameter. Set to false if you are using the ramp as a FrameSync and don't need this feature, to save a bit of DSP.

`Offset`

Offset (vertical shift) the ramp upwards or downwards

`Curve`

If AllowCurvature is set to true, then this parameter defines the type of curvature on the ramp. Positive values create convex exponentiation curves and negative values create concave ones.

`Rising`

True: A rising ramp (eg. -1 -> 1)
False: A falling ramp (eg. 1 -> -1)

`Flip`

a value of 1 here will reverse your current ramp design

`Scale`

Allows you to scale the output. -1 inverts the ramp, 0 makes it flat.

Remember that normally in WireFrames, values of 0 received by a FrameShaper Input bypasses the effect of the FrameShaper. Therefore this Scale can act like an attenuator on the overall affect of the FrameShaper.

Example: When using the FrameShaper to perform spectral pitch shifting using FrameMult or FrameModifier1 or FormantShifter, a value of 0 does no pitch shifting, a value of -1 shifts downwards, and 1 upwards. Therefore if you are sending a 1-0 linear ramp shape, the higher partials on the highest Tracks of the Frame will not be pitched, but the lower ones will be shifted upwards.

Example: When using the FrameShaper to attenuate the level of each amp Track in a spectrum frame stream independently using a FrameLevels module, a 1-0 linear ramp shaper would attenuate the higher partials (a kind of spectral tilt eq once resynthesised)

`FrameLength`

For correct synchronisation this integer parameter, the FrameRate value, should match that of the WF Module you are plugging into (and as a general rule of thumb, the Framerate of the entire WireFrames signal flow you are designing)

Of course feel free to experiment with powerOfTwo division or multiplications when modifiying another Frame Stream. For example to stretch the affect of your Ramp over 16 frames of a frame stream, set the CurviLinear Ramp Framelength to be 16 times that of the frame stream you are modulating.

For example, to repeat the ramps' influence 32 times through a spectrum of 512 partials for example, a frequency ripple affecting clusters of 32 partials at a time, set the CurviLinear

Ramp FrameLength to be 512/16 (ie. 32) and multiply the spectrum frequencies by this ramp - thats a pretty cool effect, and very DSP efficient.

`Width`
Alter the bandwidth of the ramp - a value of 1 is full width of the Frame.

`FrameScope`
Optionally visualise the Curvilinear ramp with a built in FrameScope

`FullRamp`
True: A full ramp is generated in the range (-1, 1)
False: A half ramp is generated in the range (0,1)

WF LinearRamp

Generates a linear ramp at the current FrameLength. Can be scaled and offset, and (smoothly) flipped without ever losing frame synchronisation.

`Flip`
When at 0 will generate a rising full ramp from -1 to 1.
When at 1 will generate the opposite.
This value is continuously variable, so you can morph smoothly from a falling to rising ramp.

`Scale`
Allows you to scale the output. Negative values inverts the shape, 0 makes it flat and positive scales upwards.

Remember that normally in WireFrames, values of 0 recieved by a FrameShaper Input bypasses any effect of the FrameShaper. Therefore this Scale can act like an attenuator on the overall affect of the FrameShaper.

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes

`FrameScope`
Check this if you want to use the internal FrameScope.

`Offset`
Offsets the output signal up or down.

Frame Processing

WF FrameModifier

Most of the WireFrames classes contain a FrameShaper input. However if you want to shape frames at arbitrary points of the signal flow, you can use the FrameModifier.

When used to shape a Frame, it would apply the same logic at the FrameShaper input, except the Order changes any scaling to have an exponential character.

For example:
Input = 0 -> no shaping
Input = 1 -> multiply by 16 (when order = 4)
Input = -1 -> multiply by 1/16 (when order = 4)

The order is an Integer which changes the multiplication coefficient:
Order = 1 -> 1/2 to 2
Order = 2 -> 1/4 to 4
Order = 3 -> 1/8 to 8
Order = 4 -> 1/16 to 16

For those of you mathematically inclined, here's the formula:
Output = InputFrames * ((2 ** Order) ** InputFrameShaper)

`InputFrames`
A Frame stream, ideally running at the same Frame Length as the FrameShaper being combined with it.

`Order`
The order is an Integer which changes the multiplication coefficient:
Order = 1 -> 1/2 to 2
Order = 2 -> 1/4 to 4
Order = 3 -> 1/8 to 8
Order = 4 -> 1/16 to 16

`InputFrameShaper`
Ideally suited to plug a FrameShaper module in sync at the same FrameLength as the InputFrames. It can be any signal though.

WF FrameMult

Combines the input Frame stream with the signal at the FrameShaper input by scaling (multiplying) one with other. Use it to affect the value of each Track in the Frame stream using another Frame signal, such as a WF FrameShaper. Ideally the FrameShaper and the InputFrames should be in sync using the same FrameLength.

`FrameShaper`
Ideally suited to plug a FrameShaper module in sync at the same FrameLenght as the InputFrames

`InvertShaper`
If true, will invert the FrameShaper input, subtracting it from 1

`InputFrames`
A Frame stream, ideally running at the same Frame Length as the FrameShaper being combined with it

WF FrameStretch

This class lets you stretch or compress the values inside the frame (the best way to observe and grasp this is using a FrameScope). Using a constant for the Stretch input stretches/ compresses the whole frame, however you can also use other frame-based signals to only process parts of the Frame.

`FrameLength`
Defines the FrameLength.

`Stretch`
Positive values stretch the frame while negative values compress the frame. You can also use other frame-based signals here to only process parts of the Frame. A ramp would for example gradually stretch the frame the higher the track.

`InputFrames`
The input should be a frame-based signal.

WF FrameShift

Use this Class to "shift" a Frame signal horizontally with wrap around, within the scope defined by FrameLength.

`Shift`
Ammount to shift through the Frame

`Slide`
Optionally, choose to slide smoothly through shift changes

`FrameScope`
Optionally show a FrameScope visualising the FrameShift

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms, but you may of course experiment with powerOfTwo divisions or dephased FrameLength, which could give you some interesting results in certain contexts.

WF FrameFlex

Stretches and compresses the incoming frame along the x-axis, with curvature, like a kind spring flex compression and expansion. Use for novel frame stretching and other types of remapping or shaping when working with Spectrum data or stretching and manipulating other Frame data in interesting ways.

`ClipTo01`
Optionally clip the outgoing processed frame to the 01 range.

`InputFrames`
The input should be a frame based signal (Single)

`Flex`
0 will perform no flex, positive will expand and negative will compress

`Invert`
a value higher than 0.5 here will flip the resulting flex frame data around 0.

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes

`RemoveMetaData`
Removes the last 4 tracks from a spectrum because they are sometimes used for metadata which results in odd results when doing formantShifting for example.

`FrameScope`
Optional see a Framescope visualising the Flex

WF Blur
Every TimeConstant duration, it will take a snapshot of the current frame. Within the very same time it will interpolate from the previous snapshot to the current snapshot.

`DoubleWire`
Set to true if you are working with paired Frame streams, travelling independently on the left and right outpus of Class. The most common examples of this type of signal is a Spectrum Frame stream.

A Kyma Spectrum Frame (and NeverEngine Labs SPC Frame) will use the dual streams of information running on the 'double wire' to sonify (or resynthesise) the Frame data using an Oscillator bank or one of the other Kyma prototypes that resynthesise from Spectrum Frames.

Double wire spectrum data consists of one frame stream on the left channel carrying the Amplitude values that will specify the volume of each partial, and likewise one stream on the right channel with data that specifies the Frequencies for each partial.

`Reset`
Reset the blurring algorithm. This is useful if you want to get rid of some offset introduced by shaping the timeconstant.

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`Freeze`
If set to true the interpolation is bypassed and the class works like a sampleAndHold outputting freezed frames at an interval specified by timeconstant.

`TimeConstant`
Sets the timeconstant for the blurring.

`InputFrames`
The input should be a frame based signal.

`TimeConstantFrameShaper`
As in our other NeverEngineLabs WireFrames classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a separate delay scale for each Track in the frame.

In NeverEngineLabs WireFrames a value of 1 received by a FrameShaper input will shape the frame in a positive direction, and a value of -1 inversely. A value of 0 at this input bypasses any shaping effect.

A sample value of 1 will scale the time constant on a Track by 4 while a value of -1 scales it by 1/4, a value of 0 will bypass the scaling.

`Offset`
Use a frame based control signal here to offset the blurring for each track individually.

WF TrackDelays

This class lets you delay each partial (track) individually. Delay sets the maximum delay time while the DelayScaleFrames input scales the delay time for each track: If you use a ramp matching the FrameLength the first track will not be delayed and the last track will be delayed by the time you've specified in Delay.

TIP:
A way to think of the whole process in a visual way is this: Imagine your Tracks (or spectrum partials) stacked on top of each other and a cursor progressing through them horizontally. Similar to what you are used to in a DAW. Without delay the cursor is a straight line so all the tracks readout their values from the same point in time. The DelayScalesFrameShaper input defines the line shape of the cursor. If you use a WF LinearRamp to shape the DelayScales your cursor will be tilted at an angle so it essentially plays back the lowest track without delay and gradually increases delay as the tracks get higher.

You could use any type of frame shaper (try WF RandomWalks or anything you want, as long as its in synch!), to delay Tracks in all sorts of distributions.

`Delay`
Defines the maximum delay time.

`FrameLength`
For correct synchronisation this integer parameter, the FrameRate value, should match that of the WF Module you are plugging into (and as a general rule of thumb, the Framerate of the entire WireFrames signal flow you are designing)

`DelayScaleFrames ( DelayScalesFrameShaper )`
Scales the delay time for each track.

`InputFrames`
This should be a mono or stereo (if DoubleWire is checked) frame signal.

`DoubleWire`
Check this box if you are going to use two individual frame based signals on the left and right channel. You can think of this as being a stereo input (although essentially it doesn't relate to the number of audio channels, that's why it is called DoubleWire). The most useful application is definitely when using spectra and you want to equally delay amps & freqs.

WF Smooth

Smooths each track independently using a first order low pass filter. The amount of smoothing is set by the frame signal at the Smooth input. To use a constant smooth value for all tracks use a ConstantFrameShaper. Optionally you can check unsmooth to get the highpass filtered tracks. The smoothed and the unsmoothed sum to the original signal again.

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes

`Smooth`
Defines the amount of smoothing

`UnSmooth`
If checked the output will be the unsmoothed (highpass filtered) tracks

`InputFrames`
The input should be a frame based signal (Single)

WF LogicalAND

Convenience class to perform a logical AND operation. The output is only true if all inputs are true, else it is false.

Truth table:

A	B	Output
0	0	0
0	1	0
1	0	0
1	1	1

`Inputs`
The input should be a logical value (either 0 or 1). Use as many inputs as you like. The output will be Input1 && Input2 && Input3 &&...

WF LogicalXOR

Convenience class to perform a logical XOR operation. The output is only true if InputA and InputB are different, else it is false.

Truth table:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

`InputA`
The input should be a logical value (either 0 or 1).

`InputB`
The input should be a logical value (either 0 or 1).

WF LogicalOR

Convenience class to perform a logical OR operation. The output is true if any of the inputs are true, else it is false.

Truth table:

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	1

`Inputs`
The input should be a logical value (either 0 or 1). Use as many inputs as you like. The output will be Input1 || Input2 || Input3 ||...

WF LogicalNOT

Convenience class to perform a logical NOT operation. The output is true if the input is false and vice versa.

`Input`
The input should be a logical value (either 0 or 1).

WF ClippingLimiter

You can think of the WireFrames Limiters as being equivalent to world limits in a virtual space, such as a video game or a particle simulation. We could say that the value held by one Track in a Frame, is equivalent to the position of an element in the game world, or particle system, updated every frame period. Using this analogy, lets look at the three types of world limiting.

Clipping Limiter: when an element is moving (being incremented) and it reaches its world limit, its output value through the limiter, will remain clipped at the same value as the threshold it has reached. Its output value will stay there until its input value returns under the limit.

Mirror Limiter: when the moving element reaches its world limit, its output value through the limiter reflects its travel beyond the limit. Effectively the element appears to bounce off the limit, and move in the opposite direction for as long as its input value continues to increment past the threshold. When its input value begins to decrement, this path is reflected too, until it reaches the limit from the other side where it simply passes back through.

Wrapping Limiter: when the moving element reaches one of its world limits (for example the ceiling), its output value through the limiter wraps around and emerges at the opposite limit (for example the floor). Analogous to PacMan wrapping disappearing from one end of the maze and appearing at the other.

`InputFrames`
InputFrames: This should be a mono frame signal

`Floor`
Threshold which defines the lower limit. Values below this threshold will not pass through.

`Ceiling`
Threshold which defines the upper limit. Values above this threshold will not pass through.

WF WrappingLimiter

Wrapping Limiter: when the moving element reaches one of its world limits (for example the ceiling), its output value through the limiter wraps around and emerges at the opposite limit (for example the floor). Analogous to PacMan wrapping disappearing from one end of the maze and appearing at the other.

`InputFrames`

InputFrames: This should be a mono frame signal

`Ceiling`

Threshold which defines the upper limit. Values above this threshold will not pass through.

`Floor`

Threshold which defines the lower limit. Values below this threshold will not pass through.

WF MirrorLimiter

Mirror Limiter: when the moving element reaches its world limit, its output value through the limiter reflects its travel beyond the limit. Effectively the element appears to bounce off the limit, and move in the opposite direction for as long as its input value continues to increment past the threshold. When its input value begins to decrement, this path is reflected too, until it reaches the limit from the other side where it simply passes back through.

`InputFrames`

InputFrames: This should be a mono frame signal

`Ceiling`

Threshold which defines the upper limit. Values above this threshold will not pass through.

`Floor`

Threshold which defines the lower limit. Values below this threshold will not pass through.

WF FrameMemory

This Class that lets you store a certain number of frames into RAM and recall them later.

Optionally, you can morph between saved states when you recall them.

`DoubleWire`
Set to true if you are working with paired Frame streams, travelling independently on the left and right outpus of Class. The most common examples of this type of signal is a Spectrum Frame stream.

`NbrFrames`
Defines the number of frames to store.

`FrameLength`
Defines the FrameLength.

`RecallIndex`
Defines the memory location that should be at the output.

`InputFrames`
The input should be a frame-based signal.

`Store`
Trigger to store the current frame at the input at the location defines by StoreIndex.

`MemoryName`
Choose a unique name for each instance of this class.

`StoreIndex`
Defines the memory location that should be used to store the frame at the input.

`Morph`
When checked in-between values of RecallIndex will morph between the stored frames. Uncheck to save some DSP.

WF Thresholds

Takes Frames at the input and compares against a signal at the Thresholds input. The signal at the Thresholds input can be any audio rate signal, but if it is also a FrameShaper or other Frame stream running at the same framelength period, you will perform syncronised Track thresholding. In the default BooleanStates mode, if the sample value of a Track at the Frame Input is below the equivalent value at the Thresholds input, then its value is set to zero (false). If it is above the comparison sample at the threshold input, its value is set to one (true).

`BooleanStates`

In the default BooleanStates mode, if the sample value of a Track at the Frame Input is below the equivalent value at the Thresholds input, then its value is set to zero (false). If it is above the comparison sample at the threshold input, its value is set to one (true). Set this to false, to pass thru the input Tracks' value when above the threshold, rather than set to 1.

`InputFrames`

Frame stream input to be thresholded

`Thresholds`

Frame stream input where each Track in the Frame stream will be interpreted as a threshold

`FrameLength`

Set the Frame length of the input frames. For syncronised thresholds, it should also be the framelength of the Thresholds input frames (when its a frame stream)

`FrameScope`

Optionally display a scope of the result

WF BiDiSmooth

Slew rate limiting for all Tracks in a Frame. AttackTime controls how quickly the output reacts to increases in Frame Tracks values and ReleaseTime controls how quickly it responds to decreases in the Frame Tracks values.

`AttackTime`

AttackTime controls how quickly the output reacts to increases in Frame Tracks values

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`InputFrames`

The input should be a frame based signal (Single)

`DecayTime`

ReleaseTime controls how quickly it responds to decreases in the Frame Tracks values.

WF Smooth

Smooths each track independently using a first order low pass filter. The amount of smoothing is set by the frame signal at the Smooth input. To use a constant smooth value for all tracks use a ConstantFrameShaper. Optionally you can check unsmooth to get the highpass filtered tracks. The smoothed and the unsmoothed sum to the original signal again.

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`Smooth`

Defines the amount of smoothing

`UnSmooth`

If checked the output will be the unsmoothed (highpass filtered) tracks

`InputFrames`

The input should be a frame based signal (Single)

WF SampleAndHold

Like the CapyTalk expression it will hold the input when the gate is switched on. The gate needs to be switched off in order to trigger again. Every track works independently, giving you an independent sampleAndHold function for each.

`DoubleWire`

Check this if you want to process a stereo input of framed data at once (like a Spectrum).

`InputFrames`

The input should be a frame based signal (Single / Double if checked)

`InputGates`

Positive values turn the gate on and the input will be held until the gate is turned off and on again (then it will hold the current value again).

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

Scopes and Visualisers

WF FrameScope

A scope optimised for working with Frame data. To see a steady image of the frame on the scope, you should try to make sure that the frame period set at FrameLength matches the incoming frame period.

`DoubleWire`
Set to true if you are working with paired Frame streams, travelling independently on the left and right outpus of Class. The most common examples of this type of signal is a Spectrum Frame stream.

A Kyma Spectrum Frame (and NeverEngine Labs SPC Frame) will use the dual streams of information running on the 'double wire' to sonify (or resynthesise) the Frame data using an Oscillator bank or one of the other Kyma prototypes that resynthesise from Spectrum Frames.

Double wire spectrum data consists of one frame stream on the left channel carrying the Amplitude values that will specify the volume of each partial, and likewise one stream on the right channel with data that specifies the Frequencies for each partial.

`InputFrames`
Input Frame stream.

`Silent`
Set to true to mute the output

`FrameLength`
The Frame period of the frame stream at the input

`Name`
How to give the Scope a unique name. If its one word just type it in this parameter field.

If you want spaces and punctuation you need to put it in curlies and single quotes. eg.
{ 'Frame Scope Mojo' }

This parameter field will accept algorithmic constructions in SmallTalk such as:

{'FrameScope'&?VoiceNumber}

TIP: If you want to do this after a replicator, it doesn't seem to work unless another Sound within the replication is also reading the ?VoiceNumber variable.

WF FFTScope

A spectrum analyser which uses the oscilloscope to display the magnitudes of a stereo or mono signal.

`Emphasis`

Tilt the analysis to put more emphasis on the higher frequency bands

`Reduction`

Frame data Reduction.

`Shift`

Shift the FFT frame around the scope

`Smooth`

Smooth the analysis

`Stereo`

Builds the stereo version of the processor when checked

`FFTsize`

Select an FFT size from 256 up to 16384 .. different FFTSizes will sound quite different in quality.

`Level`

Through level of the incoming audio

`Gain`

Gain up the signal on the scope

WF FrameVisualiser

Use this Class to visualise any SINGLE WIRE Frame with beautiful graphics using our custom Kyma Tool - the NeverEngine Labs FrameVisualiser. You can find it in the WireFrames LIbrary folder.

In our experience its better to run the Tool first then launch the Sound that sends to the Tool via this Class.

TIP:
This Class works by creating up to 64 invisible global controllers which transmit to the SmallTalk Tool. They are labelled as !x01 !x02 etc and !y01 !y02 etc. A special invisble global event called !FrameLength128 also sends to the tool, with the current FrameLength value. You can also use those generated events elsewhere in your Signal Flow by referring to the name.

`Enable`
Enable the generation of global controllers that will be broadcast to the NEL FrameVisualiser Tool

`FrameLength`
The Frame period of the incoming stream. The current version of the tool can only visuliase up to the first 128 Tracks of a frame.

`InputFrames`
Incoming single wire Frame stream. Keep all FrameLengths matched so everything stays correctly in sync.

`Silent`
When not set, then the Frame data will pass thru as well as generate OSC events.

WF Visualiser2D

Use this Class to visualise any 2D Frame such as the Lorenz Attractor or maybe a 2D Swarm System, with beautiful graphics using our custom Kyma Tool - the NeverEngine Labs 2D Visualiser. You can find it in the WireFrames LIbrary folder.

TIP:
This Class works by creating up to 64 invisible global controllers which transmit to the SmallTalk Tool. They are labelled as !x01 !x02 etc and !y_01 !y02 etc. A special invisble global event called !frameLength64 also sends to the tool, with the current FrameLength value. You can also use those generated events elsewhere in your Signal Flow by referring to the name.

`InputFrames`
Incoming double wire Frame stream, left channel becomes the X coordinates and right channel becomes the Y coordinates. Keep all FrameLengths matched so everything stays correctly in sync.

`Enable`
Enable the generation of global controllers.

`FrameLength`
The Frame period of the incoming stream. The Tool currently only reads the first 64 Tracks from a frame.

`Silent`
When not set, then the Frame data will pass thru as well as generate OSC events.

Track Selection

WF TrackMasking

Use a Frame stream (or any other audio input) as a mask on another frame stream with optional fade up and fade down times.

When a Track in the MaskingShaper input stream is above the threshold value, it will cause the same Track number to be muted out of the InputFrames stream. Optionally, you can define how long it takes to smoothly fade out the Track that is being muted, and how long it takes to smoothly fade it back in once the mask signal has dropped below the threshold.

When working with Spectrums and masking frequency information, you might want to block DC on the resulting Frequency using a Kyma SpectrumModifier or WF BlockDC

`FadeDownTime`

Define how long it takes to smoothly fade down the information on the Track that is being muted.
If you don't need fades then set to 0 to save on DSP.

`InputFrames`

Single channel of Frame data which will get its Tracks muted and unmuted according to the MaskShaper stream

`FadeUpTime`

Define how long it takes to smoothly fade up the information on the Track that is being muted.
If you don't need fades then set to 0 to save on DSP.

`MaskingShaper`

Single channel of Frame data which will mute and unmute Tracks in the InputFrames stream when a Track in the MaskShaper passes the threshold

`FrameLength`

FrameLength - try to match Frame Lengths of the Input Frames and Mask Shaper to get the correct behaviour.

`Threshold`

A Track from the MaskingShaper whose value is above this threshold will mute out the same TrackNumber in the Input Frames

`FrameScope`

Generate a Framescope to see the behaviour of the masking

WF TrackSelector

The output of this module is the value of the selected track only. Similar to CapyTalk arrays the track count starts at 0. If you are not going to change the track number, for example using a compile-time constant like ?VoiceNumber, you can uncheck Dynamic to save some DSP. The module handles Single and DoubleWire frame streams automatically.

`Dynamic`

If you are not going to change the track number, for example using a compile-time constant like ?VoiceNumber, you can uncheck Dynamic to save some DSP.

`InputFrames`

Input Frame stream.

`Track`

Similar to CapyTalk arrays the track count starts at 0. If you think about the frame as an array, then the module is the equivalent of:
Track of: #(element1 element2 ...)

`FrameLength`

Defines the FrameLength and should match the incoming FrameLength for correct functioning of the internal algorithms.

WF TrackFilters

Filters out Tracks from the start or the end of a Frame, sort of like a high and low cut off filter for TrackNumbers (rather than the values of each track). Optionally use Slope to allow some 'resonance' or 'radiance' of Tracks around the cut off points.

The thresholds are normalised to FrameLength, so 0.5 represents the TrackNumber at 50% of the FrameLength. If Slope is set high, the 'radiance' effect will wrap around from the high cutoff and reveal Track information from the start and ends of Frame. Observe on a Framescope to see what I mean.

`DoubleWire`

Set to true if you are working with paired Frame streams, travelling in parallel on the left and right outputs/inputs. The most common examples of this type of signal is a Spectrum Frame stream.

A Kyma Spectrum Frame (and NeverEngine Labs SPC Frame) will use the dual streams of information running on the 'double wire' to sonify (or resynthesise) the Frame data using an Oscillator bank or one of the other Kyma prototypes that resynthesise from Spectrum Frames.

Double wire spectrum data consists of one Frame stream of Tracks on the left channel carrying the Amplitude values that will specify the volume of each partial, and likewise one on the right channel with Tracks that specifies the Frequencies for each partial.

`Slope`

use Slope to allow some 'resonance' or 'radiance' of Tracks around the cut off points.

`ThresholdHigh`
High threshold is inverse.... so a value of 1 here lets all high Tracks through and a value of 0 lets none through.... Use (1-!ThresholdHigh) to get the same kind of behaviours as ! ThresholdLow

`FrameLength`
The Frame period of the frame stream at the input

`ThresholdLow`
The thresholds are normalised to FrameLength, so 0.5 represents the TrackNumber at 50% of the FrameLength.

`Input`
Input Frames to be filtered

WF FreezeTracks

This Class freezes individual Tracks in one Frame stream using the boolean logic value of the equivalent Tracks derived from another Frame stream passing a threshold.

When using this Class a Spectrum context, we recommend that you use a WF BlockDC before a resynthesis , as zeros in the Frequency information can cause undesirable low frequency information.

`DoubleWire`
Set to true if you are working with paired Frame streams, travelling independently on the left and right outpus of the InputFrames and/or TrackFreezeShaper input. The most common examples of this type of signal is a Spectrum Frame stream.

`InputFrames`
Input Frame stream from which Tacks will be be frozen

`Reset`
Resets the frozen Tracks and reads directly from the InputFrames when above 0.

`FrameLength`
The Frame period of the frame stream at the input and ideally the Frame stream at the Shaper Input

`Threshold`
Tracks from the InputFrames will be frozen according to whether the values in the Tracks of the TrackFreezeShaper have gone above this Threshold.

`Freeze`
Freezes the entire modified Frame result when above zero.

`TrackFreezeShaper`
FrameShapers generally plug into FrameShaper inputs. They provide a neat way to alter many parameters simultaneously, which is often the case when working with Frame data. The shape of the wavetable specified in the FrameShaper, becomes expressed upon the Frame data.

In the case of WF TrackFreeze it is the input at the TrackFreezeShaper which will cause Tracks from the InputFrames to freeze, according to whether the values in the Tracks of the TrackFreezeShaper have passed the Threshold.

`FrozenTracksOnly`
Any value greater than 0 will output the frozen Tracks only and set unfrozen tracks to 0

WF RadianceGate

Use the Radiance Gate to reduce density in the Frame stream. It is a sieve filter which allows you to select one Track and then choose how many Tracks to allow through on either side of that Track, using a width control.

Width of 1 lets only the Center Track through, a width of 3 lets the Center Track plus each track on either side through and so on.

Unselected tracks will receive a 0 value at the output, and selected tracks will pass through at input values.

Its automatically working with single wire or double wire streams

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`CenterTrack`
A sieve filter which allows you to select one Track and then choose how many Tracks to allow through on either side of that Track, using a width control.

`Width`
An integer value that defines how many Tracks immediately around the CenterTrack will pass through.

`InputFrames`
InputFrames: This should be a mono frame signal

WF Holt-Winters Smooth

Also known as double exponential smoothing, the Holt-Winters Smooth takes into account the possibility of a series exhibiting some form of trend.

`FrameLength`

Defines the length of the frame in samples and should usually match with the other WireFrames classes

`Smooth`

A frame-based signal containing the amounts of smoothing.

`SmoothTrend`

A frame-based signal containing the amounts of smoothing the trend.

`InputFrames`

The input should be a frame based signal (Single)

Frames Conversion

WF FrameToOSC

Will extract each Track from a Frame to a seperate OSC sending widget.
Internally it uses a stripped down variation of NEL SendOSC to acheive this. If you need more features like string rewriting, please check out our NEL OSC Tools Classes.

Known issues:

* if the interpreter throws a ' cannot perform message' type error, this is because the VCS must have at least one hot value which is NOT generated by FramesToOSC. Just add something like !Scale in the Scale paramter field to get around that error.

* if you are programming and changing a lot the messages, Kyma starts to try to memorise all messages sent and heard. It can begin to overload the memory and this seems to make a situation sometimes where all Sounds take a very long time to compile. Try using the built in Kyma OSC Helper tool to make the Pacarana forget all the messages it has ever sent or received if you run into this issue.

`AddLeadingZeroes`
Set to true to automatically pad the Track Number digits used for labelling with a leading zero so that the controls automatically sort in alphabetical and numerical order in the VCS.

`Scale`
A useful array that allows the dynamic scaling of each Kyma signal *before* it gets remapped through the input output matric and transmitted as the generated OSC message.

In this field, you must enclose Capytalk expressions within curly braces, for example:
{!Scale * (1 bpm: 120) }

You can use it as a way of fading outgoing OSC signals in or modulating them with an LFO or with pulses. Or mute them.

`Enable`
Enable the OSC widgets and outputs

`SendToIP`
Here you put in the address of the device you wish to send to.

The Paca(rana) must be connected to the same network as this IP with a Cat5 ethernet cable from the ETHERNET port on the back. Make sure you haven't plugged it into the EXPANSION sockets by mistake!

TIP: This IP is not the same as the one you get from the Kyma DSP Status info.

In Mac OS, the correct IP to use is the one you will see in System Preferences/Network. The service will be an ethernet network and should say CONNECTED in bold, and will usually have a status of self-assigned IP.

Read more about ports, IP and Kyma at http://kyma.symbolicsound.com/qa/1895/how-do-i-change-the-sending-port-for-osc

`FrameLength`
The Frame period of the incoming stream.

`SendToPort`
This is the Port number to send the signals to. Software or a device with the IP specified in the SendToIP paramter fiel, with this port number, will receive the Kyma signals through the OSCMessages you have defined.

`InputFrames`
Incoming single wire Frame stream. Keep all FrameLengths matched so everything stays correctly in sync.

`Silent`
When not set, then the Frame data will pass thru as well as generate OSC events.

`MaxTracks`
If you only want to read only the first few tracks from a Frame then you can specify that number here and generate less events and widgets.

`Smooth`
Convenient smooth on all the outgoing signals with a duration in seconds.
Set to 0 to bypass smoothing.

`Osc_stem`
Only one message stem can be assigned for all Tracks read by this instance of FramesToOSC. Each track will generate a widget automatically sending OSC with this stem, its Track Number and optionally a Suffix as an address.

Examples:

```
/randomWalk
/spectrumAmps/beatSample
```

`StealthMode`
When you have finished routing and are happy with the OSC configuration, go into stealth mode to hide all OSC controlling generated widgets from the VCS.

They will continue to broadcast their values to the RewrittenEventNames you have specified AND transparently to the OSC you have specified. Even though they are hidden, you can still refer elsewhere in your sound design to the signals being sent out over OSC by using the widget names directly, if you wish.

For example, map something like `!osc_VirtualObjectSize.x` to an object in a virtual world, a filter cut off and a reverb decay at the same time, whilst keeping your VCS clutterfree.

TIP: In stealth mode you can't have a completely empty VCS. You will need to have at least one widget for background OSC to keep streaming.

`OutputMax`
Remap the -1,1 range of each Track's extracted value to some other range, This is the max (ceiling) and can be any integer.

`Suffix`
Optionally add a suffix. I put this feature in to allow you to harness Kyma's built in recognition of .x .y and .z to automatically construct aggregate 2D or 2D faders in the VCS. The suffix will also be added to the OSC messages being broadcast.

`OutputMin`
Remap the -1,1 range of each Track's extracted value to some other range, This is the minimum (floor) and can be any integer.

`PollingTrigger`
Messages are only sent when this trigger changes from 0 to positive. Use it to reduce data over the network if necessary... 25 ms tick is usually a pretty reliable polling rate. Or use it as a way of muting or pausing the outputting of the message specified in this instance of FrameToOSC.

WF FrameToOscillator

This Class will take a Frame Input ideally with a Framelength of 4096 and create a wavetable oscillator from it, that you can control with `!Frequency` and `!Envelope` like other oscillators in Kyma.

Tip:
With this oscillator and some WireFrames programming, it is possible to create Xenakis stochastic synthesis oscillator which changes its waveform every cycle and then some.....

`BypassReduction`
Bypass the Frame reducer effect. If bypassed, any !Reduction widget will not be available.

`Frequency`
The frequency can be specified in units of pitch or frequency.

`DutyCycle`
the duty cycle of the waveform

`InputFrames`
Ideally should be a single wire Frame Stream with a FrameLength of 4096

`Envelope`
This is an attenuator on the output of the Oscillator. Enter 1 (or 0 dB) for the full amplitude. For a time-varying amplitude, paste in a Sound (such as AR, ADSR, or FunctionGenerator) or an Event Value (such as !Volume) in this field. It is advisable to smooth the EventValue (for example !Volume smoothed) or to use a full sample rate Sound pasted into the Envelope field. For full sample rate amplitude envelopes, erase the 'L' that automatically appears after the name of the pasted Sound.

`Reduction`
Amount of Frame data Reduction. 0 bypasses any reduction and 1 reduces to the maximum, holding only the last Track value in the Frame. When reduction is active, a linear interpolation fills in the gaps where Tracks have been reduced out of the Frame. See WF Reducer to learn more about how this works.

`FrameScope`
Choose to see the waveform being used on a built in FrameScope

WF FrameToGlobalControllers

The WF FTGC Class will extract values from a Frame stream and convert them into any number of EventValues (also known as Global Controllers).

You can generate less EventValues than there are Tracks in the Frame. In other words you can extract only some of the Tracks from the full FrameLength. You can use offset to move around in the Frame in that case.

Alternatively you can explore Euclidean distributions when extracting less Tracks than there are Tracks in the Frame. This means you could extract or example 4 out of a Framelength of 32 and the extracted Tracks would be spread out evenly.

As EventValues can be scaled beyond -1,1 range , you can also use a ScalingArray to range each extracted EventValue into any number. This could be useful for OSC mapping.

EventValues will be updated on every trigger received in the Trigger field or every millisecond if the Trigger is set to 1, and Gated is set to True.

`Enable`

Leave this set to true. Used for debugging only.

`NumberOfGlobalControllers`

Defines how many EventValues (aka GlobalControllers) to generate. If you request less than the FrameLength , then the corresponding Tracks will be read from Track 1. Use offset to move through the Frame in this case.

`Offset`

If you request less than the FrameLength , then the corresponding Tracks will be read from Track 1. Use offset to move through the Frame .

`EuclideanMode`

When set to true, then the NumberOfGlobalControllers defines how many controllers will be evenly distributed throughout the FrameLength. Use this to explore a lower density of EventValues but spread out more widely rather than just counting from the first Track of the Frame.

`ScalingArray`

Can be a single value or hotvalue, or an Array of hotvalues or values which correspond to a scaling factor that mulitplies its particular generated EventValue. All expressions must be in curly braces

```
Example:  {!mute00 smooth: 2 s} {!Mute01 smooth: 1 s}
```

`FrameLength`

Defines the FrameLength and should match the incoming FrameLength for correct functioning of the internal algorithms.

ShowInVCS
If ShowInVCS is checked, the GeneratedEvents will be displayed in the VCS.

Gated
Check this box to treat the Trigger parameter as a gate: the GeneratedEvents will be continuously set to the value of the Track they are tracking, whenever Trigger is positive.

Silent
If the Silent box is not checked, the audio output of this Sound downsampled sample and hold of the Frame at its input, donwsampled by a factor related to the NumberOfGlobalControllers and also scaled by the ScalingArray. Basically, the through of this Class is NOT the same as the input. If you want an identical through, try setting Silent to true and use a BypassUsingSoundSelector instead

GeneratedEventStem
Choose a stem. Each generated Event will start with this and be numbered for example:

```
!gc001 !gc002 !gc003

!osc_001 !osc__002 !osc__003 etc
```

Trigger
When Trigger becomes greater than zero, the GeneratedEvent will be set (and track if Gated is true) the value of their corresponding Track in the Frame.

InitialWidgetType
 This option lets you decide what type of Widgets will be created the FIRST time this instance of WF FTGC is run

Its handy so as you don't fill up the VCS with many widgets.

Each widget will remember its display type after the first time it is created, so you will have to change the type by hand in the VCS after that.

1 - SmallFader (only numbers)
2 - Panpot
3 - Gate
4 - Fader

InputFrames
Input Frame stream. This Class only works with single wire streams.

WF SimpleFrameToGlobalControllers

The SimpleFrameToGlobalControllers alogrithmically generates a set of Capytalk EventValues from incoming audio signals. It is ideally suited for extracting values from a sequence of data organised into Frames. In fact, a Frame is a realtime sample by sample representation of the data structure known in computer science as an Array.

In Kyma WireFrames, a Frame Array contains a sequence of 32bit Integer values. The sequence is of an arbitrary but fixed length. The values are are sample values, streamed sequentially one after another. This is in effect, a sample rate audio stream which means that the time it takes for a full Frame to travel 'along the wire' is related to the clock rate of your Kyma system (for example 48000 samples per second).

The SimpleFrameToGlobalControllers Sound extracts the sample by sample values from Frame data and converts them into Capytalk HotValues that can be used inside Kyma Sounds (see also WF FrameToOSC) . It is ideally suited to extracting Spectral information from the Frame Arrays generated by the family of Spectral analysis prototypes in Kyma.

It will output a number of generated EventValues (each representing one element of the array). Number specifies the Number of GlobalControllers while FrameLength specifies the length of the array. Use 0 FrameLength if the Number and the FrameLength match. The Index of the element in the array gets suffixed to the GeneratedEvent, depending on the number it will be suffix, suffix2 or suffix3.

Things to be consider when using this module:

A GlobalController, also known as a Hot Value (a red name with an exclamation mark at the front), runs at Capytalk rate. Capytalk is updated every millisecond, not every sample.

TIP: You can choose to hide the generated values from the VCS, which might be useful when generating a large count of algorithimic controllers. Then you can simply read the global controllers elsewhere in your Kyma program by referring to them by name, or as a complete collection.
For example, your stem here is !walk_ and you have asked for 256 copies then !walk_001 up to !walk_256 will be generated.

You could collect them like this if you needed to

```
(1 to: 256) collect: [ :i | {!walk_ suffx3: i} ]
```

for example to step through them in the parameter field of a StepSequencer.

The module will only read the left channel of any input, so if the input is Spectral data, it will only read the Amps part.

If you choose to deselect 'silent', the Class will also pass through the input Array, so you can extract only some of the data as Global controllers without altering what happens next in the signal path design.

`Input`
The input should be an array (like a spectrum or a swarm). Input and output are mono.

`FrameLength`
Specifies the length of the frame. Use 0 if the FrameLength matches the Number of generated Events. (Integer expected)

`Offset`
Specifies the Offset of the index. (Integer expected)

`GeneratedEvent`
Enter an EventValue name (including the exlamation point prefix) for the generated EventValue.

`ShowInVCS`
If ShowInVCS is checked, the GeneratedEvent will be displayed in the VCS.

`Silent`
Check the Silent box to mute the output.

`Number`
Specifies the number of generated EventValues.

WF WriteFrames

Use this class if you want to write frames to disk as wav file. The resulting files look like ordinary wav files but they are not. That's why we automatically use the suffix _WF?FrameLength.?ChannelNbr.wav (for example someFrames_WF256.1.wav). This way you can distinguish them from your "normal" wav samples. To play them back use WF ReadFrames.
ChannelNbr
Optionally you can append a channel number to the filename (for example someFrames_WF256.1.wav). Use 0 to ignore this function.

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes

`DelayCompensation`
If you are recording from a LiveSpectralAnalysis Sound you can check this to avoid recording silence at the beginning (the LiveSpectralAnalysis introduces some latency).

`Gate`
Gates the recording so you can also concatenate arbitrary frames in real-time.

`DoubleWire`
Check this to record a DoubleWire frame (stereo). Useful for recording spectra.

`InputFrames`
The input should be a frame based signal (Single / Double if checked)

`Filename`
Choose a filename and location to save your file.

* Known issue: Currently Kyma will always try to save as an aiff type of file. The workaround for now is to not use the disk icon to load a filename, just type the desired filename directly in to the field. The file will be stored in Kyma 7 Folder/Disk Tracks/...

WF WriteFrames RAM

Use this class if you want to write frames to RAM to read them out at some other point using WF ReadFrames RAM. Usage is pretty much the same as a MemoryWriter, it just makes sure everything stays in sync internally.

`CaptureDuration`
The length of time to record the Input. Enter 0 s if you want to record Input for its full duration.

`RecordingName`
Enter a name for the sample that you are recording into the wavetable memory. Use this same name in the playback Sounds, so they can find the sample in the wavetable memory.

`Cyclic`
When Cyclic is selected, the MemoryWriter does a "looping" recording. In other words, it records for the specified CaptureDuration; then, if Trigger is still positive, it wraps around to the beginning of the recording and continues recording the Input, overwriting what it had previously recorded there.

`Silent`
Click here if you would like to record the Input silently, without also monitoring it at the same time.

`DoubleWire`
Check this to record a DoubleWire frame (stereo). Useful for recording spectra.

`SimultRead`
Check this box if this recording will be used in another Sound that will start at exactly the same time as this Sound and will use the recording at the same rate (for example, a Sample using this recording that starts at the same time whose Trigger is 1 and whose Frequency is default).

`FrameLength`
Defines the length of the frame in samples and should usually match with the other WireFrames classes

`Trigger`
When the Trigger becomes nonzero, the recording is triggered.

`Global`

`InputFrames`
The input should be a frame based signal (Single / Double if checked)

Convenience Classes

WF Placeholder

Use this as a placeholder with no effect and zero DSP impact. Its useful when you dont want to use a FrameShaper input. You will still need to put something there, due to the way Kyma handles inputs to Classes.

This is a blank to plug in such situations, to help keep the visual a bit more clear in the Signal Flow diagram.

WF DoubleWireSpectrum

Convenience Class which wraps a syntheticSpectrumFromSounds prototype to construct a Kyma spectrum from two Frame inputs. Optionally you can choose to drop any subsonic information below 20hz. Also, if you are generating synthetic amplitudes, the sum of all the amps can often clip, so you can also choose the Limiting option which will scale down the amps frame so it cannot clip when summed.

`Scale`
Scale the amplitudes

`Amps`
A frame stream to be considered as Spectrum amplitudes information

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`Frequencies`
A frame stream to be considered as Spectrum Frequencies information

`BlockDC`
drop any subsonic information below 20hz

`Limiting`
The Limiting option will scale down the amps frame so it cannot clip when summed. Don't set this option if you are working with amplitudes from a spectrum analysis as these are already normalised not to clip.

`CeilingFrequency`
Hot value that will define the high frequency limit above which any Tracks will be muted.

WF CompensatedLevel

Use this Class before resynthesis to compensate the Amplitude envelopes using an equal loudness psychoacoustic relationship to the Frequency envelopes (see Capytalk reference for compensatedLevel).

Automatic DC blocking (less than 20 hz) is also built in. Makeup gain of 12db when compensating to maximum is also built in, so the overall loudness doesn't drop.

When working with synthetically created DoubleWire spectra you can turn on Limiting to make sure the total sum of the Amp envelopes will never clip a sinewave based resynthesis bank. Use limiting with post gain to find a balance depending on your material.

`Compensation`
To bypass the frequency related Compensation set to 0. Anywhere in between 0 and 1 gives a blend between minimum and maximum frequency based compensation.

`PostGain`
Post gain adds gain to the amplitude envelopes at the output

`FrameLength`
The FrameLength should match with the incoming spectrum for correct functioning of the internal algorithms. Tip: Use a script to distribute ?FrameLength to all modules at once.

`Spectrum`
Shoud be a DoubleWire spectrum for correct frequency related compensation.

`Limiting`
Use Limiting to make sure the total sum of the Amp envelopes will never clip the resynthesis bank

WF Level

Convenient Class which scales the Amps of a 'double wire' input stream of spectrum data. Use this Class before resynthesising with an oscillator bank (or other resynthesis module) to scale only the amps leg, and not the frequency information.

`Input`
Expects frames representing a spectrum (aka 'double wire') stream

`Level`
Scale the amps of the incoming spectrum data before resynthesis

WF 01ToFullRange

Convenience Class that takes an input that may have been previously processed with RangeTo01 and rescales its range back to full (-1,1) range. Equivalent to *2-1 in CapyTalk

`Input`

Signal to be rescaled to -1,1. Can be Double or Single Wire

WF FeedbackLoopOutput

The Kyma built-in FeedbackLoopOutput prototype has an undocumented internal limit to its delay time. This convenience Class behaves in the same way but is not limited in delay time. Use it in conjunction with the normal Kyma FeedbackLoopInput prototype.

`Connection`

This is the name of the delay line and must be the same as that specified in the corresponding FeedbackLoopInput.

`Delay`

Specify a delay time here (in samp)

WF ClipTo01

Convenience Class which clips the incoming audio signal to the 0 to 1 range, discarding all negative values.

`Input`

Single wire frame stream

`Scale`

Set the ceiling after clipping

WF RangeTo01

Convenience Class that takes its input (which can be double wire) and shifts its range to fit between 0 and 1. Useful for logical operations and thresholding.

`Input`

Input to be rescaled to 01 range

WF toTrigger1ms

Convenience Class converts any audio input into a 1 ms wide trig making a single sample trigger, like that from a threshold, more compatible with CapyTalk triggering.

`Input`

Input sound, ideally a 1 samp trigger. If another sample wide trigger comes in before 1 ms has elapsed since the last one, it will extend the current 1 ms trigger on the outpu for another millisecond.

WF AsLogicValue

Same as the CapyTalk expression but working at samplerate. If the input is 0 or less the output will be false (0), if it is above 0 the output will be true (1).

`Input`

If the input is 0 or less the output will be false (0), if it is above 0 the output will be true (1).

WF BlockDC

Convenience Class that will remove very low frequency information from Spectrum frames before resynthesis. Useful to avoid exploding Filterbanks or delay lines when Frequency information might be set to zero or very low by other WireFrames modules in a Spectral design.

`DoubleWireSpectrum`

Ideally put this module directly before the resynthesis bank to clip the frequency envelope at 20hz

WF RemoveSpectrumMetadata

Removes the last 4 tracks from a spectrum because they are sometimes used for metadata which results in odd results when doing formantShifting for example.

`FrameLength`

Frame Length of Input Frame stream

`InputFrames`

Double Wire or Single Wire Spectrum Stream at input

WF DelayByFrames

The output is a delayed version of the input keeping the frame in sync.

Some modules in WireFrames system introduce delays of one frame necessary to perform calculations on a frame that has just passed. In this case, you can use this module as a compensating delay to synchronise across the network if you need to.

It can also be used as a way of holding a frame for a given amount of Frames before splicing it back into the stream, "shifting" any frame horizontally at runtime using a hot value.

`DoubleWire`

If you would like to delay a dual Frame stream coming from a spectrum analysis (i.e. with Amps and Frequencies) set this to true. If you want to delay only one Frame stream set this to false to save DSP

`FramesToDelay`

Integer number of Frames to delay (max. 4096)

`InputFrames`

InputFrames: This should be a mono frame signal unless you have DoubleWire set to true, then the Class expects a dual spectrum input (amps and freqs)

`FrameLength`

Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms, but you may of course experiment with powerOfTwo divisions or dephased FrameLength, which could give you some interesting results in certain contexts.

Frame Mathematics

WF Average

Outputs the average of all the values within one frame. Because it needs to read all values before calculating the average the output is delayed by 1 Frame.

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`InputFrames`
InputFrames: This should be a mono frame signal.

WF Minimum

Outputs the minimum of all the values within one frame. Because it needs to read all values before calculating the minimum the output is delayed by 1 Frame.

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`InputFrames`
InputFrames: This should be a mono frame signal.

WF Maximum

Outputs the maximum of all the values within one frame. Because it needs to read all values before calculating the maximum the output is delayed by 1 Frame.

`FrameLength`
Frame Period: Should match incoming Frame period for correct functioning of the internal algorithms.

`InputFrames`
InputFrames: This should be a mono frame signal.

WF Derivative

This is a low level class outputting the derivative of each track. Basically the derivative tells you the changes of the current frame compared to the previous one. To get the second derivative (or the third, and so on..) you could daisychain those modules.

`DoubleWire`
Check this if the input is DoubleWire.

`InputFrames`
The input should be a frame based signal. Make sure the frameLength matches. If the input is a DoubleWire signal (e.g. a Spectrum) you should check the DoubleWire box.

`Log`
Check this if you want to convert to log space before calculating the derivative.

`FrameLength`
Defines the FrameLength.

WF Cartesian2Polar

This Class converts from Polar to Cartesian.

Ideal use case is to put it right after an FFT module in order to process the Magnitude and/or Phase with WireFrames modules set to a FrameLength of FFTLength/2.

Input: L: Real/x, R: Imaginary
Output: L: Magnitude, R: Phase

`Input`
Left channel input should be real, right channel input should be imaginary.

WF Polar2Cartesian

This Class converts polar to cartesian.

Ideal use case, is to put it right before an inverseFFT

Input: L: Magnitude, R: Phase
Output: L: Real/x, R: Imaginary/y

`Input`
The left channel input should be magnitude, the right channel should be phase.

WF countTriggersMod

This is a sample rate trigger counter that counts at audio rate, and outputs an audio rate quantised signal representing that accumulator and modulo.

The left audio output of this Class normalised to the number of the counter divided by Mod (e.g. counter number 5 with mod: 10 results in an output of 5/10).

The right audio output is a 1 ms positive trigger when the counter resets to 0.

TIP: To use the reset output to trigger a Capytalk expression, paste the sound and substitute the 'L' with an 'R' to read from the the right leg output.

TIP: To extract the actual integer counter in a SoundToGlobalController for example, multiply the output by the value in the maxSteps constant. For convenience, I have included an option to generate a counter as a STGC.

`GeneratedIndex`
Give a name for a conveniently generated rounded integer counter. If you leave this field empty, no EventValue will be generated.

`Mod`
The module starts counting at 0 and wraps around when the value reaches Mod. (Integer expected)

`Reset`
Send a logical trigger to reset the Counter. Converted to a 1 samp trigger internally.

`Input`
Any audio input will be converted to 1 samp triggers internally

`Silent`
Mute the audio rate outputs and generate only the STGC.

`MaxSteps`
This is the maxium steps that will be counted. It is can be any number but needs to be a fixed integer higher than the highest !Mod value.

WF CountTriggersModMirror

This is a trigger counter that counts at sample rate, and outputs an audio rate quantised signal representing the accumulator and modulo. This version counts with Mirror Limiting. When the step counter reaches the ceiling, its output value through the limiter reflects its travel beyond the limit. Effectively the counter appears to bounce off the limit, and count down in the opposite direction for as long as the input value continues to increment past the ceiling threshold. When the Modulo value is reached the Counter will reset to zero. When the counter reaches the Floor threshold, and the internal counter has still not reached the Modulo limit, the counter will reflect upwards again. This algorithm can rapidly create internally oscillating index counters for more complex modulo counting.

The left audio output of this Class normalised to the number of the counter divided by Mod (e.g. counter number 5 with mod: 10 results in an output of 5/10).

The right audio output is a 1 ms positive trigger when the counter resets to 0.

TIP: To use the reset output to trigger a Capytalk expression, paste the sound and substitute the 'L' with an 'R' to read from the the right leg output.

TIP:To extract the actual integer counter in a SoundToGlobalController for example, multiply the output by the value in the maxSteps constant. For convenience, I have included an option to generate a counter as a STGC.

`MaxSteps`
This is the maxium steps that will be counted. It is can be any number but needs to be a fixed integer higher than the highest !Mod value.

`Ceiling`
Integer for the ceiling value which can be less than Modulo

`Mod`
The module starts counting at 0 and wraps around when the value reaches Mod. (Integer expected)

`Floor`
Integer for the floor value, an offset of the lowest counting number

`Reset`
Send a logical trigger to reset the Counter. Converted to a 1 samp trigger internally.

`GeneratedIndex`
Give a name for a conveniently generated rounded integer counter. If you leave this field empty, no EventValue will be generated.

`Silent`
in case you want to mute the audio output and use only the generated index

`Input`
Any audio input will be converted to 1 samp triggers internally

WF RoundTo

The input values will get rounded to the values you provide in the RoundTo parameter. The elements in the RoundTo array should be in ascending order. If you want to use hotValues make sure to put them in curly braces -> `{!Val1}`.

`InputFrames`

The input usually is a frame-based signal, however in this case it could be any signal.

`RoundTo`

Enter the values which you want to round to in ascending order. If you want to use hotValues make sure to put them in curly braces -> `{!Val1}`.

Made With WireFrames

These examples of encapsulated Classes have been designed using lower level WireFrames Classes. They are more like conventional audio processors, and will output normal audio, rather than Frame streams.

WF BarkProcessor

Uses FFT analysis divided into up to 24 Bark bands.

Cut offs in hz
100 200 300 400 510 630 770 920 1080 1270 1480 1720 2000 2320 2700 3150 3700 4400 5300 6400 7700 9500 12000 15500

Optionally creates GlobalControllers with a prefix name you can specify, which will visualise the energy at each bark band. This is a psychoacoustic measurement.

"So what is this Bark frequency scale? First of all, the Bark scale was named in honour of acoustician Heinrich Barkhausen. On this scale, the audible spectrum runs from 0 to 24, each Bark being a sort of psychologically equal division. Lots of things in psychoacoustics work on the Bark scale rather than the scale of Hertz." - johndcook.com

Class is only mono, you could use stereoizer to create a stereo version, but take care to rename generated events carefully so as not to have a naming conflict. All generated event names must be unique!

`FadeInTime`
Smoothing on the band change when the Levels fader moves upwards

`Levels`
An array defining the levels of each bark band.

`FadeOutTime`
Smoothing on the band change when the Levels fader moves downwards

`NumberOfGeneratedEvents`
How many generated events to create. If you only want a few you can for example set this value to 8 and the first 8 bands will become global controllers.

`FFTScope`
generate a built in FFT scope

`ReadTrigger`
A positive value here will update the generated events. Useful if you are wanting to reduce data for OSC or grab a freeze from the spectrum.

`FFTsize`
Suggest 1024 but feel free to experiment. Values can be 256, 512, 1024, 2048, 4096. Gives more fine detail on the bands.

`ScaleReadout`
A value to boost, attenuate or invert the generated global controllers

`Gain`
Output gain

`ShowInVCS`
show or hide the generated events

`GeneratedEventsPrefix`
Generated events will be named with this prefix followed by an underscore and the number of the bark band.

`SmoothReadOut`
A 'fall time' style lag processor for the global controllers. Attack times remain fast, release times can be extended by this parameter

`HFWeighting`
If set to true, the lower dbfs threshold of the higher bands (above band 19) will be decreased, resulting in an amplification of the energy up there (which is naturally quite small)

`Input`
Mono sound source

WF FFTBlur

Basic FFT based spectral blurring effect made with WF Blur / WF PolarToCartesian / WF CartesianToPolar.

For a far more advanced and sonically improved version please see our FFT Suite Product available at www.neverenginelabs.com.

`AdjustOffsets`
At 0 the blurs all start at the same time. Explore different distribution from -1 to 1.

`Reset`
Reset the blurring algorithm. This is useful if you want to get rid of some offset introduced by shaping the timeconstant.

`AdjustTc`
At 0 the TimeConstant is shared amongst all Magnitudes. Explore different distribution ratios across the Magnitudes from -1 to 1.

`FFtSize`
FFT size will be rounded internally to the nearest power of two. Try 8192 for long blurred textures.

`Seed`
unique value for the random phase

`Freeze`
If set to true the interpolation is bypassed and the class works like a sampleAndHold outputting freezed frames at an interval specified by timeconstant.

`TimeConstant`
Sets the timeconstant for the blurring.

`Input`
Mono input.

WF ParticleCloud

Particle cloud with multiple stochastic paths (or Random Walks) for each particle.

The influence of one set of random walks acts on the duration of each wavetable particle, therefore its pitch. If StochasticLevels is set to true, another independent set of random walks acts on the level of each particle.

The source audio from which the particles are created can be of any duration. Be sure to extend the range of your ParticleDuration widget when using a longer duration sample.

A multichannel output version of ParticleCloud is also available.

`Density`
Small Density values result in a sparse texture; large Density values generate a dense texture. This controls the average number of new grains starting up at any given point in time.

`Seed`
An initial value for the noise source, for a repeatable random walk.

`FrameScope`
Select if you would like to see a FrameScope visualising the random walks.

`Spread`
Stereo dispersion

`InitialDeviation`
Defines the maximum deviation of each Track from the StartingFrom value, when they are initialised. When InitialDeviation is set to 0, all random walks will start from the same value set by StartingFrom.

`StochasticLevels`
When set to true, the level of each particle will be affected by an secondary set of random walks. Turn off to save on DSP.

`Initialise`
Initialises the RandomWalks according to startingFrom and initialDeviation

`StochasticPaths`
Specifies how many independent random walks will be generated. This can be more or less than the number of maxParticles.

`Level`
Attenuates the level of each particle before summing. Adjust to avoid internal clipping.

`StochasticStepSize`

The size of each step, smaller step sizes mean less dramatic changes in value.

`MaxParticles`
The maximum particles in the cloud. Tested with up to 128 at 48K on a Paca.

`StochasticTimeConstant`
how often the random walks update

`ParticleDuration`
A base duration for each particle

`Wavetable`
This is a wavetable or sample that represents the particle. It can be stereo or mono.

`ParticleJitter`
Increase this (you can go past one) to introduce the effects of the random walks on each particle.

`Reverse`
Play the particle wavetable in reverse

WF MultiChannelParticleCloud

Multichannel output version of WF ParticleCloud.

`Outputs`
The cloud will be distributed over these outputs
Enter an array list eg: 1 2 3 4 5 will spread over the first five outputs.
1 3 5 7 will spread only onto odd outputs

WF Compander

An experimental compressor/expander stereo dynamic processor.
Features exponential gain characteristics and bi directional amplitude following.

`Order`

The exponential order is an Integer which changes the multiplication coefficient:
Order = 1 -> 1/2 to 2
Order = 2 -> 1/4 to 4
Order = 3 -> 1/8 to 8
Order = 4 -> 1/16 to 16

`ReleaseTime`

If the scaled Value is greater than the current output, the ReleaseTime controls the length of time it takes for the difference between the output and the input to be reduced to about 37% of the current difference.

`Scale`

Attenuate or boost the amplitude follower signal

`Floor`

Floor value

`AttackTime`

AttackTime controls the length of time it takes for the difference between the output and the input to be reduced to about 37% of the current difference when the scaled Value is less than the current output.

`CompensatingDelay`

Delay the source input using negative values here allowing pre-expanding and compression. Using positive values offsets the amplitude follower, allowing for delayed reaction on the audio.

`Direction`

Negative values does downwards compression positive value upward expansion from floor.

`VUMeter`

Give a hotvalue name for a generated VUMeter needle tracking the amplitude modulating curve

`Input`

Mono or stereo input

`MakeUpGain`

Post processing makeup gain

WF StochasticOscillator

Dual oscillators whose waveforms are changing every cycle with each sample point in each wavetable following the output of 4096 random walks (times 2 that makes 8192 stochastic signals updated every cycle!)

Features built in highpass filters to reduce DC offset from the stochastic wavetables and an interpolated breakpoint reduction algorithm.

Stochastic synthesis is heavily inspired by the pioneering work of Iannis Xenakis.

`FrameScope`
set to true and you will see the stochastic wavetable in a framescope

`TimeConstantFrameShaper`
As in our other NeverEngineLabs WireFrames™ classes, this input opens the possibility to use another sample rate signal (such as WF FrameModifier, Kyma Oscillators, Kyma SpectrumFromArray, Noise etc) to have a seperate time constant for each Track in the frame.

`AmpLow`
Scale the output

`Reduction`
Perform a reduction of the break points that definte the internal structure of each periodic waveform.

`Detune`
Detuning factor between the two core oscillators. Can be higher than 1 and lower than 0.

`StereoWidth`
Mid Side stereo width. Each oscillator is output hardpanned so use the stereoWidth parameter to attenuate the Side information and narrow the stereo image.

`InitialDeviation`
Defines the maximum deviation of each Track from the StartingFrom value, when they are initialised. When InitialDeviation is set to 0, all random walks will start from the same value set by StartingFrom.

`TimeConstant`
This represents the maximum duration a single stream takes to reach its new value in seconds. Any TimeConstantFrameShaper present will scale according to this master value.

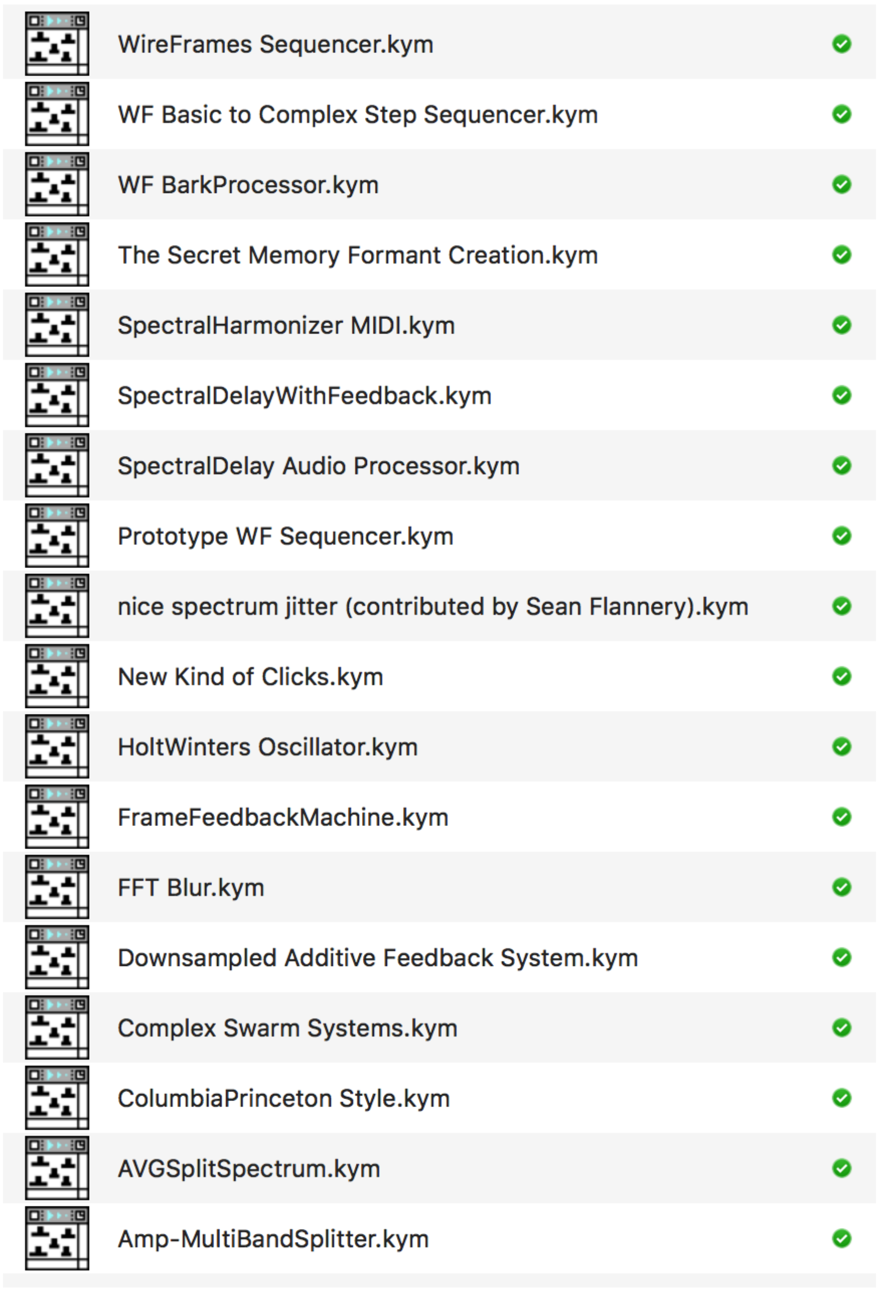
`LogFreq`
Overall Frequency of the oscillator pair

`Initialise`
Initialises the RandomWalks according to startingFrom and initialDeviation

`JoinTogether`
Turn this on to let the tracks join together towards their average value.

More Made With WireFrames examples

In the Made With WireFrames folder you will find even more experimental examples of WF programming.

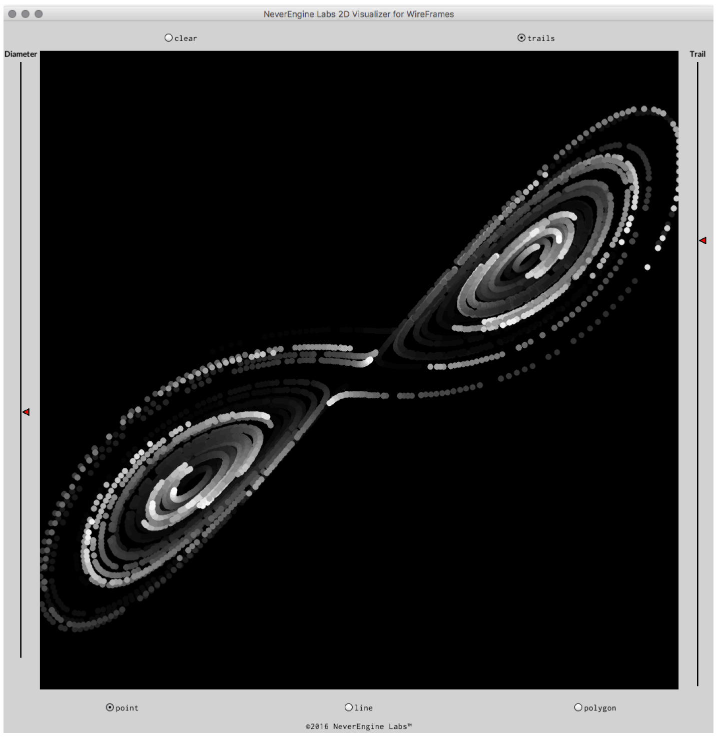


WireFrames Sequencer.kym
WF Basic to Complex Step Sequencer.kym
WF BarkProcessor.kym
The Secret Memory Formant Creation.kym
SpectralHarmonizer MIDI.kym
SpectralDelayWithFeedback.kym
SpectralDelay Audio Processor.kym
Prototype WF Sequencer.kym
nice spectrum jitter (contributed by Sean Flannery).kym
New Kind of Clicks.kym
HoltWinters Oscillator.kym
FrameFeedbackMachine.kym
FFT Blur.kym
Downsampled Additive Feedback System.kym
Complex Swarm Systems.kym
ColumbiaPrinceton Style.kym
AVGSplitSpectrum.kym
Amp-MultiBandSplitter.kym

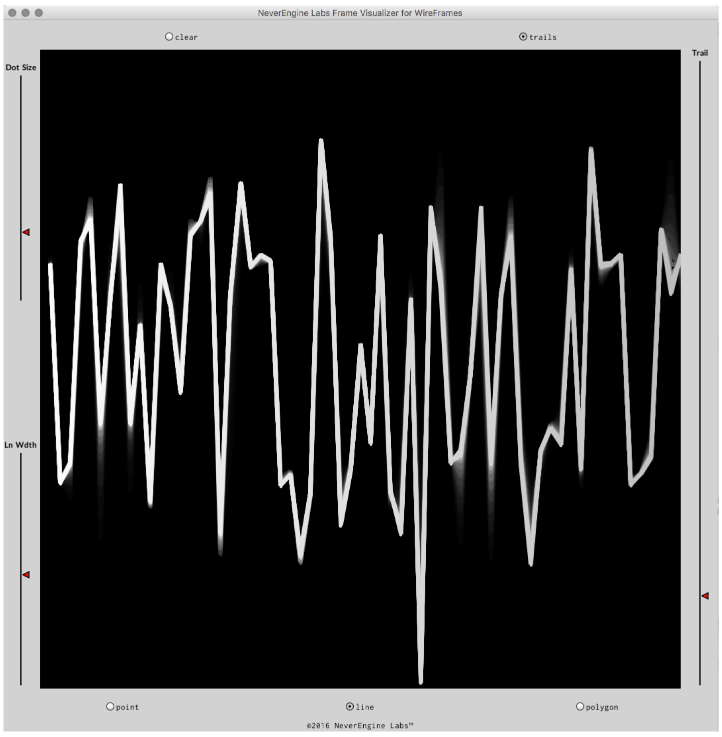
Tools

NeverEngine Labs Frame Visualiser
NeverEngine Labs 2D Visualiser

These two SmallTalk tools connect with the WF FrameVisualiser and WF FrameVisualiser2D Classes. They will let you visualise Frames with some nice graphics. Please note that they are limited to plotting only the first 64 Tracks of any Frame input.



Here is the WF Lorenz double wire output plotted in the NEL 2D visualiser



Here 64 Random Walks are plotted in the NEL Frame Visualiser tool

You can also read our ChangeLog , which documents when we updated the development folder during the WireFrames Lab.

https://www.dropbox.com/s/zw2tircpy7vv4wn/ChangeLog.txt

Many thanks, and we hope you enjoy.

Cristian Vogel and Gustav Scholda