FxDSP Documentation

Release 0.1

Hamilton Kibbe

June 29, 2016

Contents

CHAPTER 1

About FxDSP

FxDSP is a DSP library written in C for audio processing and synthesis. The library supports both single- and double-precision calculations, and uses vectorized instructions where possible.

FxDSP contains a set of objects which are building blocks for audio processing and synthesis applications.

1.1 Installation

FxDSP cam be built and installed on most platforms using CMake.

```
# From the top-level FxDSP directory
mkdir build
cd build
cmake ../FxDSP
make && make install
```

On **Mac OSX** FxDSP can be built and installed using the included XCode project.

1.2 Useage

FxDSP is written in an object-oriented style. Each module has an object type and a set of functions that act on that object. Here's an example:

```
#include "FxDSP/CircularBuffer.h"

// Pointers to input and output buffers
float* in;
float* out;

unsigned buffer_length = 1024;

// Create an instance of a CircularBuffer "object"
CircularBuffer* buffer = CircularBufferInit(buffer_length);

// Write 128 samples to the circular buffer
CircularBufferWrite(buffer, in, 128);

// Read 64 samples from the circular buffer
CircularBufferRead(buffer, out, 64);
```

```
// Delete the circular buffer
CircularBufferFree(buffer);
```

The double-precision version is nearly identical, with a "D" added to the end of each object type and function name:

```
#include "FxDSP/CircularBuffer.h"

// Pointers to input and output buffers
double* in;
double* out;

unsigned buffer_length = 1024;

// Create an instance of a CircularBuffer "object"
CircularBufferD* buffer = CircularBufferInitD(buffer_length);

// Write 128 samples to the circular buffer
CircularBufferWriteD(buffer, in, 128);

// Read 64 samples from the circular buffer
CircularBufferReadD(buffer, out, 64);

// Delete the circular buffer
CircularBufferFreeD(buffer);
```

CHAPTER 2

FxDSP API Reference

2.1 `Dsp.h` — DSP Utilities

The DSP module contains basic functions for working with buffers of samples.

2.1.1 Sample Type Conversion

Floating Point to Integer Conversion

Error_t **`FloatBufferToInt16`**(signed short **dest*, **const** float **src*, unsigned *length*)

Convert an array of float samples to 16-bit signed.

Converts array of floating point samples in [-1, 1] to signed 16-bit samples

Return Error code.

Parameters

- `dest` - Signed samples array
- `src` - Floating point samples to convert
- `length` - Number of samples.

Error_t **`DoubleBufferToInt16`**(signed short **dest*, **const** double **src*, unsigned *length*)

Convert an array of double samples to 16-bit signed.

Converts array of double-precision samples in [-1, 1] to signed 16- bit samples

Return Error code.

Parameters

- `dest` - Signed samples array
- `src` - double-precision samples to convert
- `length` - Number of samples.

Integer to Floating Point Conversion

Error_t **`Int16BufferToFloat`** (float **dest*, **const** signed short **src*, unsigned *length*)

Convert an array of 16-bit signed samples to floats.

Converts array of 16-bit integer samples to float samples in [-1,1]

Return Error code.

Parameters

- `dest` - floating point samples array
- `src` - integer samples to convert
- `length` - Number of samples.

Error_t **`Int16BufferToDouble`** (double **dest*, **const** signed short **src*, unsigned *length*)

Convert an array of 16-bit signed samples to double.

Converts array of 16-bit integer samples to double samples in [-1,1]

Return Error code.

Parameters

- `dest` - double-precisionsamples array
- `src` - integer samples to convert
- `length` - Number of samples.

Single- to Double-Precision Conversion

Error_t **`DoubleToFloat`** (float **dest*, **const** double **src*, unsigned *length*)

Convert double-precision samples to floats.

Convert array of double-precision samples to float samples in [-1,1]

Return Error code.

Parameters

- `dest` - single-precision sample destination array
- `src` - double-precision samples to convert
- `length` - Number of samples.

Error_t **`FloatToDouble`** (double **dest*, **const** float **src*, unsigned *length*)

Convert single-precision samples to doubles.

Convert array of single-precision samples to doubles in [-1,1]

Return Error code.

Parameters

- `dest` - double-precision sample destination array
- `src` - single-precision samples to convert
- `length` - Number of samples.

2.1.2 Basic Buffer Operations

Error_t **FillBuffer** (float **dest*, unsigned *length*, float *value*)

Fill an array with a given value.

Fill the passed array with the value passed in as value. Uses a vectorized implementation if available.

Return Error code.

Parameters

- `dest` - Array to fill
- `length` - Size of array in samples
- `value` - Value to use.

Error_t **ClearBuffer** (float **dest*, unsigned *length*)

Set array to zero.

Fill the passed array with zeros.

Return Error code.

Parameters

- `dest` - Array to fill
- `length` - Size of array in samples

Error_t **CopyBuffer** (float **dest*, **const** float **src*, unsigned *length*)

Copy an array.

copy an array from src to dest

Return Error code.

Parameters

- `dest` - Array to fill
- `src` - source buffer
- `length` - Size of array in samples

Error_t **CopyBufferStride** (float **dest*, unsigned *dest_stride*, **const** float **src*, unsigned *src_stride*, unsigned *length*)

Copy an array with given source and destination strides.

copy an array from src to dest

Return Error code.

Parameters

- `dest` - Array to fill
- `dest_stride` - Destination stride
- `src` - source buffer
- `src_stride` - Sounrce stride
- `length` - Size of array in samples

2.1.3 Vector Min/Max

Min

float **VectorMin** (**const** float **src*, unsigned *length*)
Find the Minimum value in a vector.

Return Miminum value in vector

Parameters

- `src` - Vector to search
- `length` - Vector length in samples

Error_t **VectorMinVI** (float **value*, unsigned **index*, **const** float **src*, unsigned *length*)
Find the index and value of the Maximum value in a vector.

Parameters

- `value` - Minimum value in vector
- `index` - Index of minimum value
- `src` - Vector to search
- `length` - Vector length in samples

Max

float **VectorMax** (**const** float **src*, unsigned *length*)
Find the Maximum value in a vector.

Return Maximum value in vector

Parameters

- `src` - Vector to search
- `length` - Vector length in samples

Error_t **VectorMaxVI** (float **value*, unsigned **index*, **const** float **src*, unsigned *length*)
Find the index and value of the Maximum value in a vector.

Parameters

- `value` - Maximum value in vector
- `index` - Index of maximum value
- `src` - Vector to search
- `length` - Vector length in samples

2.1.4 Vector Absolute Value

Error_t **VectorAbs** (float **dest*, **const** float **in*, unsigned *length*)

Vector Absolute Value.

Calculate absolute value of elements in in1 and write the results to dest:

```
dest[i] = |in[i]| | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write
- `in` - Input buffer
- `length` - Number of samples to negate

2.1.5 Vector Negation

Error_t **VectorNegate** (float **dest*, **const** float **in*, unsigned *length*)

Negate a vector.

Negate every element in in and write the results to dest:

```
dest[i] = -in[i] | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write
- `in` - Input buffer
- `length` - Number of samples to negate

2.1.6 Vector Summation

float **VectorSum** (**const** float **src*, unsigned *length*)

Sum all values in an array.

Return Sum of all values in src

Parameters

- `src` - Data to sum
- `length` - Number of samples to sum

2.1.7 Vector Addition

Error_t **VectorVectorAdd** (float **dest*, **const** float **in1*, **const** float **in2*, unsigned *length*)

Add two buffers.

Add values in in1 to values in in2 element-by-element and write results to dest:

```
dest[i] = in1[i] + in2[i] | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write
- `in1` - First input buffer
- `in2` - Second input buffer
- `length` - Number of samples to add

Error_t **VectorScalarAdd** (float **dest*, **const** float **in1*, **const** float *scalar*, unsigned *length*)

Add scalar to a vector.

add scalar to every element in in1 and write results to dest:

```
dest[i] = in1[i] + scalar | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write.
- `in1` - Input buffer.
- `scalar` - Scalar value.
- `length` - Number of samples to add.

2.1.8 Vector Multiplication

Error_t **VectorVectorMultiply** (float **dest*, **const** float **in1*, **const** float **in2*, unsigned *length*)

Multiply two buffers.

Multiply values in in1 by values in in2 element by element and write results to dest:

```
dest[i] = in1[i] * in2[i] | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write
- `in1` - First input buffer
- `in2` - Second input buffer
- `length` - Number of samples to multiply

Error_t **VectorScalarMultiply** (float **dest*, **const** float **in1*, **const** float *scalar*, unsigned *length*)

Multiply buffer by a scalar.

Multiply values in in1 by scalar and write results to dest:

```
dest[i] = in1[i] * scalar | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write.
- `in1` - Input buffer.
- `scalar` - Scalar value.
- `length` - Number of samples to multiply.

2.1.9 Vector Mixing

Error_t **VectorVectorMix** (float **dest*, **const** float **in1*, **const** float **scalar1*, **const** float **in2*, **const** float **scalar2*, unsigned *length*)

Multiply two buffers by a scalar and sum.

Multiply inputs by scalars and write sum to dest:

```
dest[i] = (in1[i] * scalar1) + (in2[i] * scalar2) | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write.
- `in1` - Input buffer.
- `scalar1` - Scalar value.
- `in2` - Input buffer.
- `scalar2` - Scalar value.
- `length` - Number of samples to multiply.

Error_t **VectorVectorSumScale** (float **dest*, **const** float **in1*, **const** float **in2*, **const** float **scalar*, unsigned *length*)

Sum two vectors and multiply result by a scalar.

Sum input vectors and multiply by a scalar, leaves results in dest:

```
dest[i] = (in1[i] + in2[i]) * scalar | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write.
- `in1` - Input buffer.
- `in2` - Input buffer.
- `scalar` - Scalar value.
- `length` - Number of samples to multiply.

2.1.10 Vector Power

Error_t **VectorPower** (float **dest*, **const** float **in*, float *power*, unsigned *length*)

Raise elements of vector to a power.

Raise values in in to power of 'power' and write results to dest:

```
dest[i] = in1[i]^power | i = [0, length)
```

Return Error code.

Parameters

- `dest` - Output array to write.
- `in1` - Input buffer.
- `power` - Power to raise input by.
- `length` - Number of samples to process.

2.1.11 Vector Convolution

Error_t **Convolve** (float **in1*, unsigned *in1_length*, float **in2*, unsigned *in2_length*, float **dest*)

Perform Convolution *.

convolve in1 with in2 and write results to dest

Return Error code.

Parameters

- `in1` - First input to convolve.
- `in1_length` - Length [samples] of in1.
- `in2` - Second input to convolve.
- `in2_length` - Length[samples] of second input.
- `dest` - Output buffer. needs to be of length in1_length + in2_length - 1

2.2 `FFT.h` — Fast Fourier Transforms

The FFT module provides a unified FFT interface for several popular FFT libraries. It also provides a fallback implementation if none of the supported backends are available.

Supported Backends are

- FFTW3
- Apple Accelerate

If none of the supported backends are available, the FFT module will use an implementation based on Takuya Ooura's FFT library.

2.2.1 Real-To-Complex Forward FFT

Error_t **FFT_R2C** (FFTConfig **fft*, **const** float **inBuffer*, float **real*, float **imag*)

Calculate Real to Complex Forward FFT.

Calculates the magnitude of the real forward FFT of the data in inBuffer.

Return Error code, 0 on success.

Parameters

- `fft` - Pointer to the FFT configuration.
- `inBuffer` - Input data. should be the same size as the fft.
- `real` - Allocated buffer where the real part will be written. length should be (fft->length/2).
- `imag` - Allocated buffer where the imaginary part will be written. l length should be (fft->length/2).

Error_t **FFT_R2CD** (FFTConfigD **fft*, **const** double **inBuffer*, double **real*, double **imag*)

2.2.2 Complex-To-Real Inverse FFT

Error_t **IFFT_C2R** (FFTConfig **fft*, **const** float **inReal*, **const** float **inImag*, float **out*)

Calculate Complex to Real Inverse FFT.

Calculates the inverse FFT of the data in inBuffer.

Return Error code, 0 on success.

Parameters

- `fft` - Pointer to the FFT configuration.
- `inReal` - Input real part. Length fft->length/2
- `inImag` - Input imaginary part. Length fft->length/2
- `out` - Allocated buffer where the signal will be written. length should be fft->length.

Error_t **IFFT_C2RD** (FFTConfigD **fft*, **const** double **inreal*, **const** double **inImag*, double **out*)

2.2.3 FFT Convolution

Convolution of two real signals using the FFT.

Error_t **FFTConvolve** (FFTConfig **fft*, float **in1*, unsigned *in1_length*, float **in2*, unsigned *in2_length*, float **dest*)

Perform Convolution using FFT*.

convolve in1 with in2 and write results to dest

Return Error code.

Parameters

- `in1` - First input to convolve.
- `in1_length` - Length [samples] of in1.
- `in2` - Second input to convolve.

- `in2_length` - Length[samples] of second input.
- `dest` - Output buffer. needs to be of length in1_length + in2_length - 1

Error_t **FFTConvolveD** (FFTConfigD **fft*, **const** double **in1*, unsigned *in1_length*, **const** double **in2*, unsigned *in2_length*, double **dest*)

2.2.4 FFT Convolution With Pre-Transformed kernel

Convolve a signal using a pre-transformed kernel. This is useful when using FFT convolution for filtering, as FFT(filter_kernel) only needs to be calculated once.

Error_t **FFTFilterConvolve** (FFTConfig **fft*, **const** float **in*, unsigned *in_length*, FFTSplitComplex *fft_ir*, float **dest*)

Perform Convolution using FFT*.

Convolve in1 with IFFT(fft_ir) and write results to dest. This takes an already transformed kernel as the second argument, to be used in an LTI filter, where the FFT of the kernel can be pre- calculated.

Return Error code.

Parameters

- `in1` - First input to convolve.
- `in1_length` - Length [samples] of in1.
- `fft_ir` - Second input to convolve (Already FFT'ed).
- `dest` - Output buffer. needs to be of length in1_length + in2_length - 1

Error_t **FFTFilterConvolveD** (FFTConfigD **fft*, **const** double **in*, unsigned *in_length*, FFTSplitComplexD *fft_ir*, double **dest*)

2.3 `BiquadFilter.h` — Biquad Filters

The BiquadFilter module provides a biquad filter implementation. A biquad filter is a second-order linear Infinite Impulse Response (IIR) filter with two poles and two zeros. Biquad filters are often cascaded together and used in place of individual higher-order filters because they are much less sensitive to quantization of their coefficients.

The basic biquad filter implementation is known as Direct-Form I (or DF-I):

$$y_{[n]} = b_0 x_{[n]} + b_1 x_{[n-1]} + b_2 x_{[n-2]} - a_1 y_{[n-1]} - a_2 y_{[n-2]}$$

FxDSP uses the canonical, or Direct-Form II (DF-II) implementation:

$$w_{[n]} = x_{[n]} - a_1 w_{[n-1]} - a_2 w_{[n-2]}$$

$$y_{[n]} = b_0 w_{[n]} + b_1 w_{[n-1]} + b_2 w_{[n-2]}$$

which uses fewer multiplies, adds and delays to implement an identical filter.

2.3.1 Initialization and Deletion

BiquadFilter ***BiquadFilterInit** (**const** float **bCoeff*, **const** float **aCoeff*)

Create a new BiquadFilter.

Allocates memory and returns an initialized BiquadFilter. Play nice and call BiquadFilterFree on the filter when you're done with it.

Return An initialized BiquadFilter

Parameters

- `bCoeff` - Numerator coefficients [b0, b1, b2]
- `aCoeff` - Denominator coefficients [a1, a2]

Error_t **`BiquadFilterFree`** (BiquadFilter **filter*)

Free memory associated with a BiquadFilter.

release all memory allocated by BiquadFilterInit for the supplied filter.

Return Error code, 0 on success

Parameters

- `filter` - BiquadFilter to free.

2.3.2 Resetting

Error_t **`BiquadFilterFlush`** (BiquadFilter **filter*)

Flush filter state buffers.

Return Error code, 0 on success

Parameters

- `filter` - BiquadFilter to flush.

2.3.3 Setting Parameters

Error_t **`BiquadFilterUpdateKernel`** (BiquadFilter **filter*, **const** float **bCoeff*, **const** float **aCoeff*)

Update the filter kernel for a given filter.

Parameters

- `filter` - The filter to update
- `bCoeff` - Numerator coefficients [b0, b1, b2]
- `aCoeff` - Denominator coefficients [a1, a2]

2.3.4 Processing Audio

Error_t **`BiquadFilterProcess`** (BiquadFilter **filter*, float **outBuffer*, **const** float **inBuffer*, unsigned *n_samples*)

Filter a buffer of samples.

Uses a DF-II biquad implementation to filter input samples

Return Error code, 0 on success

Parameters

- `filter` - The BiquadFilter to use.
- `outBuffer` - The buffer to write the output to.
- `inBuffer` - The buffer to filter.

- `n_samples` - The number of samples to filter.

float **BiquadFilterTick** (BiquadFilter **filter*, float *in_sample*)

Filter a single samples.

Uses a DF-II biquad implementation to filter input sample

Return Filtered sample.

Parameters

- `filter` - The BiquadFilter to use.
- `in_sample` - The sample to process.

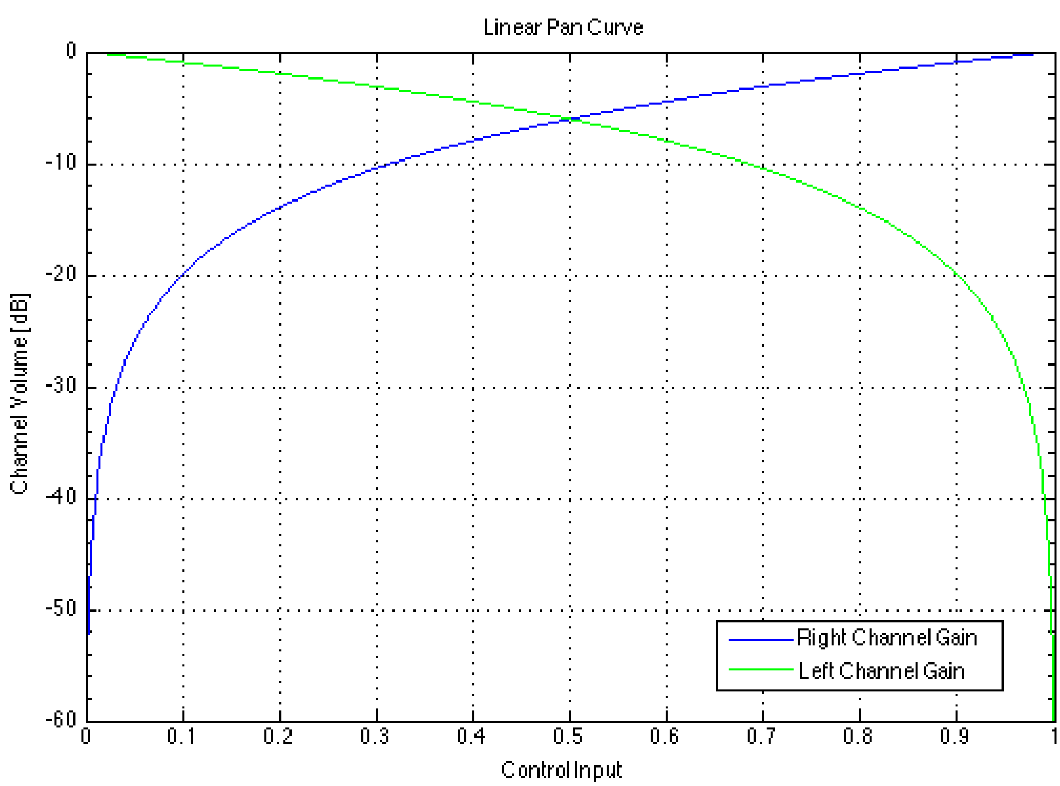
2.4 `FIRFilter.h` — Finite Impulse Response Filters

2.5 `PanLaw.h` — Pan Laws

Pan Laws are volume curves used when panning a signal in order to maintain a constant perceived loudness from hard left to hard right. Pan laws generally provide a reduction in volume to both channels when panned center, to eliminate a peak in volume when both channels are playing back at the same volume.

2.5.1 Linear Pan Law

A linear Pan law is the simplest pan law, and has each channel at -6dB when panned center.



Error_t **linear_pan** (float *control*, float **l_gain*, float **r_gain*)

Calculate linear pan channel gains.

Use a linear curve to mix between two sources. Results in a peak at the center.

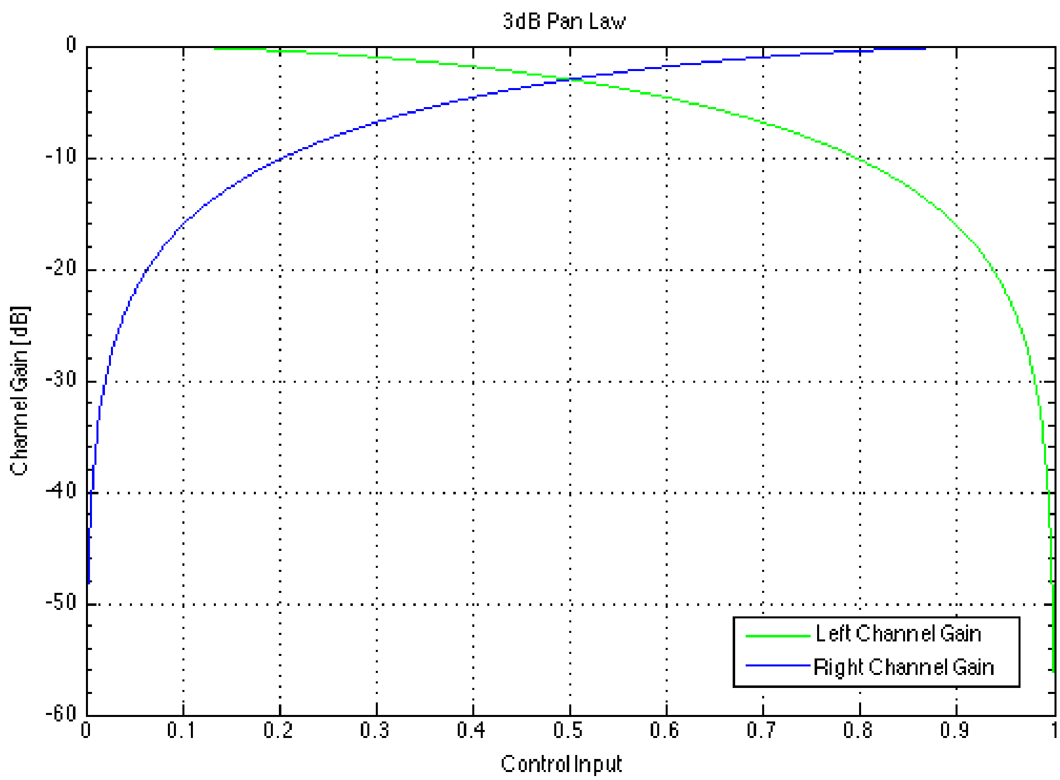
Return Error code.

Parameters

- `control` - Pan amount. Must be between 0 (hard left) and 1 (hard right)
- `l_gain` - Left channel gain.
- `r_gain` - Right channel gain.

2.5.2 3dB Equal Power Pan

A circular pan law with each channel at -3dB when panned center.



Error_t **equal_power_3dB_pan** (float *control*, float **l_gain*, float **r_gain*)

Calculate 3dB Equal-Power pan channel gains.

Use an equal-power curve to mix between two sources. Results in both channels being down 3dB at the center.

Return Error code.

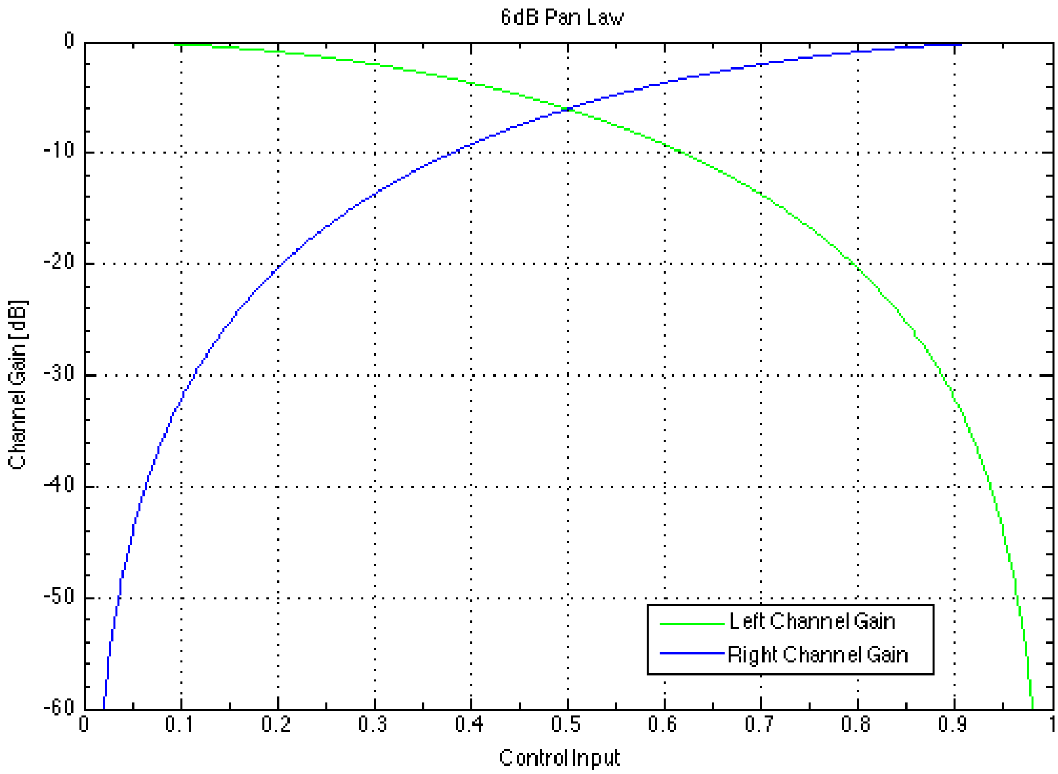
Parameters

- `control` - Pan amount. Must be between 0 (hard left) and 1 (hard right)
- `l_gain` - Left channel gain.

- `r_gain` - Right channel gain.

2.5.3 6dB Equal Power Pan

A circular pan law with each channel at -6dB when panned center.



Error_t **equal_power_6dB_pan** (float *control*, float **l_gain*, float **r_gain*)

Calculate 6dB Equal-Power pan channel gains.

Use an equal-power curve to mix between two sources. Results in both channels being down 6dB at the center.

Return Error code.

Parameters

- `control` - Pan amount. Must be between 0 (hard left) and 1 (hard right)
- `l_gain` - Left channel gain.
- `r_gain` - Right channel gain.

Python Module Index

b

B

C

D

E

F

I

L

V