IMPLEMENTATIONS OF UNIFORM FREQUENCY-DOMAIN QUANTIZATION AS AUDIO EFFECT

Tim O'Brien

CCRMA
Stanford University
tsob@ccrma.stanford.edu

ABSTRACT

Two new implementations of uniform frequency-domain quantization as a digital audio effect are presented. A SuperCollider function is created which accomplishes such quantization of FFT values and extends built-in phase vocoder functionality. Finally, the same is implemented in C++ in a manner leveraging Faust's architecture compilation capabilities. The Faust-related protocol generalizes for other DSP-related tasks which can be implemented in C++ but not in Faust.[1]

1. INTRODUCTION

This paper explores two different approaches to implementing uniform frequency-domain quantization as a digital audio effect. While frequency-domain quantization is widely used in modern perceptual audio coders [1, 2, 3, 4, 5], a review of the relevant literature as well as existing functionality available in common musical software environments such as SuperCollider, Pd, Max/MSP, and ChucK yields little in terms of frequency-domain quantization for intentional spectral and temporal distortion effects.

While the algorithms and mathematics are solved for the former case, the latter case poses certain challenges and opportunities. Because of the different paradigm – the desirability of nonlinearity and dynamic levels of spectral and temporal distortion in real time, as opposed to the necessity of maximum perceptual transparency and minimum data rates – we make certain choices which diverge from frequency-domain quantization in current perceptual audio coding techniques.

A brief discussion of frequency-domain audio coders follows in section 2. Section 3 discusses the SuperCollider implementation of the audio effect, and section 4 explores the C++ implementation utilizing the Faust compiler's conventions and ability to integrate with various architectures. Finally, section 5 discusses areas of ongoing research and opportunities for future development. The appendices list the relevant code for the SuperCollider and Faust implementations (sections A and B, respectively).

All relevant code may be found on Github at frequantizer[2] and fcpp2appls[3]. A companion website[4] to this paper contains audio examples.

2. AUDIO CODER BACKGROUND

As we seek to model our audio effect after the artifacts possible with audio coders, it is instructive to consider the signal flow in these systems. Bosi and Goldberg[1] provide a comprehensive overview of such coders and the most common elements. Fig. 1 illustrates the simplified encoding process we use. What concerns us most here is the way in which quantization, and thus most of the file size reduction, is implemented.

2.1. Quantization

Quantization refers to the process of mapping an input value (or set of values) of arbitrary precision onto a prescribed, discrete set of values, most often to reduce to memory allocation for said values [6]. Although we are working in the discrete digital domain, our ability to quantize the values of our signal allows us to achieve significant reductions in file size.

While there are many ways in which to quantize numbers (a topic we will revisit below), our current system concerns itself with perhaps the simplest means: uniform quantization. That is, the range of input values is divided into equally-spaced quantized values. There are two flavors of uniform quantization: mid-tread and mid-rise. Mid-tread maps the smallest values exactly to zero, which is desirable for audio signals, but yields one fewer output value than the mid-rise. Mid-rise maps all 2^R equally-spaced values to the range (where R is the bit depth), but cannot exactly reproduce 0. In our implementations below, we use mid-tread uniform quantization.

[1] This paper covers research conducted under the advisement of Julius O. Smith III in the spring of 2013 for Stanford's Music 420B course.

[2] https://github.com/tsob/frequantizer

[3] https://github.com/tsob/fcpp2appls

[4] http://ccrma.stanford.edu/~tsob/420B/

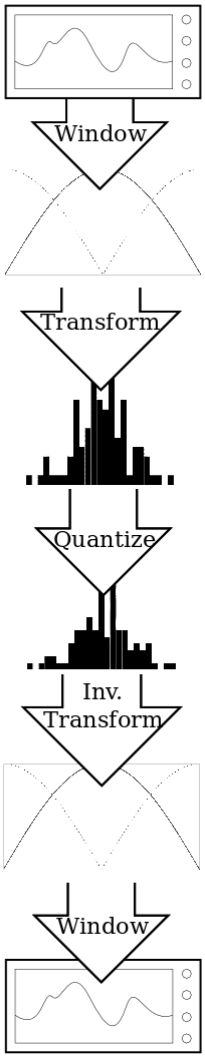


Figure 1: A simplified frequency-domain encoder.

Note that quantization depth is generally given in bits, i.e. powers of two, since such is the most efficient way in which to store the intermediate bit codes digitally. We don't need to store the intermediate bit values, since resynthesis immediately follows the analysis stage, but we retain the bitwise quantization for aesthetic reasons as well as the increased efficiency achievable with bit shifts in the quantization function.

2.2. Time-frequency transform

Many signals are quasi-periodic. That is, they are characterized by a large portion of tonal elements. Thus, as opposed to specifying each time sample in the audio stream, we can much more compactly record the amplitudes of the frequency components. The canonical example of this fact is the pure sine wave. For a pure sine wave of frequency f, each sample is a constantly changing value between -1 and 1. However, excluding the transient starting and stopping portions, we can equivalently describe this signal with its frequency (or more precisely, $\pm f$).

Additionally, as we will discuss below, quantizing the frequency values as opposed to the the time sample values allows us greater flexibility in allocating the noise/distortion in specific parts of the signal. However, the more important observation for our purposes is that quantization in the frequency domain can produce a more varied and, to our ears, desirable distortion. Quantization in the time domain yields essentially noise overlaid on the input signal. However, frequency-based quantization yields nonlinear spectral distortion based on transform sample size as well as quantization bit depth, as well as temporal distortion resulting from the time-smearing of signal components (especially transient components) over each transform block.

Of course, to avoid discontinuities in the signal when transitioning between FFT (or as we'll discuss, MDCT) blocks, it is customary to window and overlap each block by 50%, and correspondingly add the overlapping portions of frequency data to decode. Since this effectively doubles our computation, the modified discrete cosine transform (MDCT) and its inverse are commonly employed [1]. The fast computation of the MDCT via the FFT allows for critical sampling – that is, halving the amount of output we'd get from the normal FFT and allowing us to quantize $N/2$ frequency values per N-sample block [7].

2.3. Psychoacoustic optimizations

Once in the frequency domain, most audio coders employ psychoacoustic techniques. One example is to determine masked elements in a signal, which can be dropped without a perceived lack in audio quality, and can inform how many bits to allocate to each frequency component of the signal at a particular block.

Another technique is known as block switching. There is an inherent trade-off in frequency-based analysis/resynthesis between frequency resolution (larger block sizes) and temporal resolution (smaller block sizes). Instead of choosing one window size for the entire signal, block switching implements multiple block sizes (and relevant transition techniques) to maximize the frequency resolution of stable, tonal portions as well as the temporal resolution of transient attacks [8, 9].

As we are interested in distortion effects instead of transparency, we do not implement any of these perceptually-based approaches. However, we note that such non-uniform quantization, transform and bit allocation techniques may indeed be used to produce even more varied and controllable distortion effects. We plan to explore this path in further work.

3. SUPERCOLLIDER PLUGIN IMPLEMENTATION

SuperCollider[5] [10, 11] offers a powerful platform in which to implement uniform frequency-domain quantization. FFT-based analysis and resynthesis is trivial to implement with the built-in functions. Additionally, there are numerous phase vocoder UGens which modify the FFT output easily and efficiently. One such function is `PV_MagNoise`, which multiplies each frequency bin by noise. A simple example from the documentation is illustrative (Fig. 2).

```
1  (
2  SynthDef("help-magNoise", { arg out=0;
3    var in, chain;
4    in = SinOsc.ar(
5      SinOsc.kr(
6        SinOsc.kr(0.08, 0, 6, 6.2).squared,
7        0, 100, 800
8      )
9    );
10   chain = FFT(LocalBuf(2048), in);
11   chain = PV_MagNoise(chain);
12   Out.ar(out, 0.1 * IFFT(chain).dup);
13 }).play(s);
14 )
```

Figure 2: FFT, PV_MagNoise, IFFT example.

In the above example, the input `in` is a varying FM signal defined in lines 4-9. The `SynthDef` takes a 2048-point FFT and returns the values in `chain` on line 10. These values are modified by `PV_MagNoise` on line 11, and then the values are resynthesized with `IFFT` on line 12.

Of particular convenience are the FFT/IFFT arguments `hop` and `wintype`. The `hop` specifies the amount of overlap in FFT frames, and defaults to 0.5 (50%). The `wintype` can be rectangular, sine (default), or Hann. Not only are the default values sensible for most cases, but as we shall see below, the fact that such windowing and overlap-and-adding are built in saves the user considerable time and effort.

3.1. PV_Quantize

Unfortunately, there is no comparable PV UGen for FFT chain quantization. Attempts to perform the quantization in the SynthDef via the `pvcalc` and `pvcollect` methods proved prohibitively inefficient.

Thus, we have written a simple quantization function, `PV_Quantize`, which acts on the magnitudes and phases returned by the FFT. The main functionality has been added to the `PV_ThirdParty.cpp` file, listed in A.2 below. New code is in lines 33-35, 81-136, and 145. The appropriate SuperCollider class was added to `FFT2.sc`, listed in A.1. See lines 80-86. An initial help file is listed in A.3. Example usage is provided in A.4. Implementation in this manner (as part of the core SuperCollider functionality) requires no alteration of the makefiles, as we tack this function on to the third party PV Ugens. Indeed, it is ready to be pulled into the main distribution; one simply needs to add/change the .cpp, .schelp, and FFT2.sc files and recompile SuperCollider. However, one could easily separate this new functionality into a more modular package as part of the sc3-plugins[6].

For an audio demonstration of the effect using the exact code in A.4, please see `https://ccrma.stanford.edu/~tsob/420B/demo_audio.wav`.

4. LEVERAGING FAUST'S UTILITIES

Faust[7] (Functional AUdio Streams) is a functional audio programming language and compiler which is uniquely suited to many DSP tasks [12]. Not only is the Faust compiler capable of generating optimized C++ code from comparably simple Faust code, but this code can be easily and efficiently compiled to numerous different architectures. These architectures include standalone applications such as Jack-GTK, Jack-QT, CoreAudio-QT; plugins such as VST, LADSPA, LV2; and external objects in audio programming environments such as Pd[13], SuperCollider, and Max/MSP.

Unfortunately for our purposes, Faust does not currently support FFT-based functionality (or multirate processes in general). However, one can write C++ code which respects the Faust compiler's conventions and still take advantage of the Faust architecture files for compilation to plugins, externals, etc. Indeed, with a bit of fancy footwork with a "dummy" `.dsp` file, one can easily integrate GUI objects which translate to each architecture.

4.1. Architectures: from general to specific

The Faust compiler generates generalized C++ code when an architecture is not specified. That is, when one compiles `dummy.dsp` with the following command:

```
faust dummy.dsp -o dummy.cpp
```

the output is architecture-independent. Otherwise, one can specify the architecture with the `-a` flag, as in:

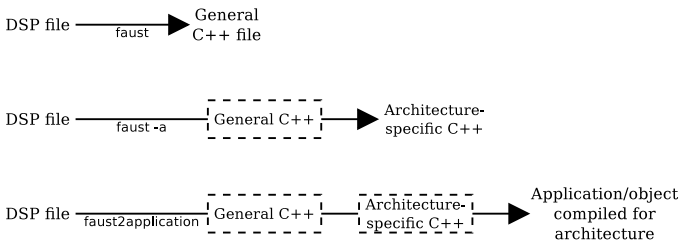
```
faust -a jack-gtk.cpp dummy.dsp -o dummy.cpp
```

In the latter case, the Faust compiler integrates the architecture-independent C++ code with the specified architecture file, which is itself a C++ file which contains placeholders for the relevant architecture-independent C++ code as well as functionality for mapping the general GUI- and DSP-related functionality to the specific architecture. Table 1 contains a list of the current architecture files available in the Faust distribution.

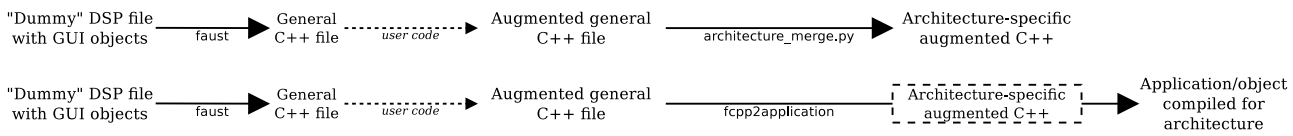
[5] `http://supercollider.sourceforge.net/`

[6] `https://github.com/supercollider/sc3-plugins`

[7] `http://faust.grame.fr/`



(a) Normal faust compilation processes.



(b) Faust-style C++ compilation processes.

Figure 3: Compilation chains.

alchemy-as.cpp	module.cpp
alsa-console.cpp	ms-jack-gtk.cpp
alsa-gtk.cpp	netjack-console.cpp
alsa-qt.cpp	netjack-qt.cpp
bench.cpp	octave.cpp
ca-qt.cpp	oscio-gtk.cpp
csound.cpp	oscio-qt.cpp
dssi.cpp	oss-gtk.cpp
gen-json.cpp	pa-gtk.cpp
ios-coreaudio.cpp	pa-qt.cpp
ios-coreaudio-jack.cpp	plot.cpp
iphone-cocoa.cpp	pure.cpp
jack-console.cpp	puredata.cpp
jack-gtk.cpp	q.cpp
jack-internal.cpp	sndfile.cpp
jack-qt.cpp	snd-rt-gtk.cpp
ladspa.cpp	supercollider.cpp
lv2.cpp	synthfile.cpp
lv2synth.cpp	vst2p4.cpp
matlabplot.cpp	vst.cpp
max-msp64.cpp	vsti-mono.cpp
max-msp.cpp	windowsdll.cpp
minimal.cpp	

Table 1: Current architecture files available in the Faust distribution's architecture directory.

Finally, Faust includes scripts such as `faust2jack`, `faust2pd`, etc. which compile the `.dsp` file for the specified architecture and immediately compile the architecture dependent executable/object. These three examples are illustrated in Fig. 3a.

4.2. Dummy DSP to dummy CPP

In order to set up a Faust-friendly C++ framework in which we may introduce our own C++ code, free of the restrictions of the Faust language, we may generate a "dummy" `.dsp` file and compile it to C++ without specifying an architecture. For the sake of simplicity, while still taking advantage of Faust's built-in GUI support, one can write a file which contains only the GUI object declarations (sliders, check boxes, etc.). Note, however, that the Faust compiler will strip away any code not utilized in the DSP process. Thus, one must include the GUI elements in the `process`, for example multiplying the incoming signal. To get started with our uniform frequency-domain quantization effect, we used the code in Fig. 4.

Section B.1 shows the architecture-independent C++ code generated from our dummy `.dsp` file. The bulk of the code is contained in the `mydsp` class, starting on line 21. This class will encompass all of our functionality.

We should note the `buildUserInterface` function on line 61. This function creates the UI elements we specified in the `.dsp`, and we don't need to make any modifications.

Next, note the `init` function on line 57. This function is, of course, called when our program starts, and thus we'll need to add any initialization tasks related to our added functionality.

The core of the signal processing is specified in the callback `compute` function on line 70. This function is called

```
declare name "Frequency Domain Quantizer";
declare version "0.1";
declare author "Tim O'Brien";

import("music.lib");

onswitch = checkbox("On");
bits = hslider(
    "Quantization bits",
    4, 2, 16, 1.0
    ); //bits for quantization
fft_power = hslider(
    "FFT block size (power of two)",
    60, 6, 18, 1.0
    ); //power of two
smooth(c) = *(1-c) : +~*(c);
level  = hslider("Level (db)",
    0, -96, 4, 0.1
    ) : db2linear : smooth(0.999);

process = _ : *(level) : *(onswitch)
            : *(fft_power) : *(bits);
```

Figure 4: dummy.dsp example

at each control block, and its arguments are the two-dimensional arrays of input and output samples, `input` and `output`, as well as the number of samples in this control block, `count`.

4.3. Dummy CPP to augmented CPP

Section B.2 lists the C++ code which we transformed from `dummy.cpp` so as to perform the uniform frequency-domain quantization effect with a single FFT size. We have added our mid-tread quantization function on lines 30-62, as well as a simple sine windowing function on lines 64-71. Additionally, we use the preprocessor macro on line 28 to calculate powers of two via bit shifts.

We have added the function `FFTInit` (lines 122-141), which is called by the aforementioned `init` function, as well as `FFTDestruct`, which is necessary to free our FFT-related memory allocations. Although Faust generally has no need for a destructor, the dummy C++ output contains no destruction function. However, as we have added on lines 173-176, the class destructor `~mydsp` successfully calls the destructor upon termination of the program.

One peculiarity of our program is the need for the `checksize` function (defined on lines 154-171 and called on line 205). Since our code is general, we have no way of knowing the size of the control blocks. For example, SuperCollider's default is 64 samples, but is adjustable by the user. Indeed, the Jack buffer size could be anywhere from 16 to 4096. We don't have access to this values until it is given to us in the first control block as `count`. Thus, we have to initialize our FFT buffers without knowing for sure whether their sizes are appropriately long. (In our implementation, the FFT buffers must be at least twice the control block size for the overlap-and-add algorithm to work properly.) We have introduced `checksize` as a (perhaps inelegant) workaround which checks these buffer sizes against the control block, exiting the program if necessary.

The `compute` function has been augmented with the necessary protocol to fill the FFT buffer, window and FFT the full FFT buffers, quantize the FFT output, take the IFFT, and queue the resulting samples for output after overlapping and adding with the 50%-offset IFFT output samples.

One of the advantages of this implementation is the high degree of control over the FFT/IFFT implementation and accompanying processing. However, the SuperCollider implementation benefits from the fact that much of this functionality is already written and tested, and future improvements will improve our SynthDef without any intervention by us.

For a program that more closely resembles the SuperCollider SynthDef mentioned in section 3.1, with multiple FFT window sizes, see B.3.

4.4. Augmented CPP to architecture-specific CPP

Once we have our Faust-style C++ code working, integrating it into an architecture is trivial. A cursory examination of the architecture files listed in Table 1 as well as a comparison of Faust compiler output with and without an architecture specified reveals that the integration requires only the copying of the augmented C++ file into the relevant spot in the architecture file. Each architecture file contains the following line:

```
<<includeclass>>
```

If we simply replace this line with the contents of the augmented C++ file, we have our architecture-specific C++ file.

One should note that, currently, the only other placeholder in the architecture files is for vector intrinsics:

```
<<includeIntrinsic>>
```

At this point in our research, we decide not to explore vector optimizations and thus leave this line blank.

We have written a python script, `architecture_merge.py`, which takes the input and architecture C++ files as arguments and performs this simple integration, saving the architecture-specific C++ file in the current working directory. The code is listed in B.4.

4.5. Final architecture compilation

Once we are able to generate the architecture-dependent C++ code, the final process of compilation to the corresponding executable/plugin/etc. should be a straightforward task of modifying the existing `faust2...` scripts. Although our work in this regard is ongoing, we were able to successfully convert `faust2jack` into `fcpp2jack` (see the code listing in B.5). Whereas `faust2jack` compiles the .dsp file

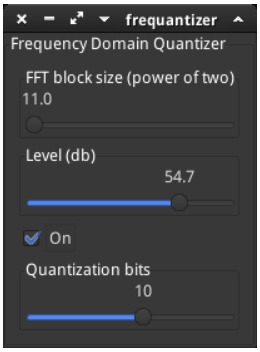


Figure 5: Jack GTK frequantizer GUI

to the jack-gtk-dependent C++ code and then compiles the executable, `fcpp2jack` calls `architecture_merge.py` before compiling to the executable.

Figure 5 shows the GUI from our frequantizer program, compiled with `fcpp2jack`. The sliders and check box are reproduced as expected, and interact with the audio effect appropriately.

5. FUTURE WORK

Now that we have laid the groundwork for uniform frequency-domain quantization as an audio effect, there exist several areas ripe for future exploration. From the general perspective of the audio effect, we plan to explore non-uniform quantization of frequency data. We might implement this while still using the uniform mid-tread quantization algorithm. For example, we currently perform the same quantization on every frequency bin; we could allow the user to control the uniform quantization bit depth as some arbitrary function of the frequency bin, thus altering the spectral response of the effect. Additionally, we could borrow a modification mentioned in [1] and scale the input frequency bin data by some nonlinear function immediately before quantization, then multiply the resulting quantized values by the inverse function. In that manner, we may warp the distribution of the quantized values along the range.

Additionally, we view the Faust-related C++ compilation process as worth exploring further. We may attempt to enable vector optimization, and of course will continue to alter more `faust2...` scripts/programs to enable compilation to more architectures. Continued work toward automating the process should help in the development of multirate and FFT-based audio programs in the short term.

A. SUPERCOLLIDER CODE LISTING

For the SuperCollider code listed below, `supercollider/` refers to the main SuperCollider directory (e.g. `/usr/share/supercollider/`).

A.1. supercollider/SCClassLibrary/Common/Audio/FFT2.sc

```
//third party FFT UGens

//sick lincoln remembers complex analysis courses
PV_ConformalMap : PV_ChainUGen
{

  *new { arg buffer, areal = 0.0, aimag = 0.0;
    ^this.multiNew('control', buffer, areal, aimag)
  }
}

//in and kernel are both audio rate changing signals
Convolution : UGen
{
  *ar { arg in, kernel, framesize=512,mul = 1.0, add = 0.0;
    ^this.multiNew('audio', in, kernel, framesize).madd(mul, add);
  }
}

//fixed kernel convolver with fix by nescivi to update the kernel on receipt of a trigger message
Convolution2 : UGen
{
  *ar { arg in, kernel, trigger = 0, framesize=2048,mul = 1.0, add = 0.0;
    ^this.multiNew('audio', in, kernel, trigger, framesize).madd(mul, add);
  }
}
```

```
27
28 //fixed kernel convolver with linear crossfade
29 Convolution2L : UGen
30 {
31   *ar { arg in, kernel, trigger = 0, framesize=2048, crossfade=1, mul = 1.0, add = 0.0;
32     ^this.multiNew('audio', in, kernel, trigger, framesize, crossfade).madd(mul, add);
33   }
34 }
35
36 //fixed kernel stereo convolver with linear crossfade
37 StereoConvolution2L : MultiOutUGen
38 {
39   *ar { arg in, kernelL, kernelR, trigger=0, framesize=2048, crossfade=1, mul = 1.0, add = 0.0;
40     ^this.multiNew('audio', in, kernelL, kernelR, trigger, framesize, crossfade).madd(mul, add);
41   }
42   init { arg ... theInputs;
43     inputs = theInputs;
44     channels = [
45       OutputProxy(rate, this, 0),
46       OutputProxy(rate, this, 1)
47     ];
48     ^channels
49   }
50 }
51
52 //time based convolution by nescivi
53 Convolution3 : UGen
54 {
55   *ar { arg in, kernel, trigger=0, framesize=2048, mul = 1.0, add = 0.0;
56     ^this.multiNew('audio', in, kernel, trigger, framesize).madd(mul, add);
57   }
58   *kr { arg in, kernel, trigger=0, framesize=2048, mul = 1.0, add = 0.0;
59     ^this.multiNew('control', in, kernel, trigger, framesize).madd(mul, add);
60   }
61 }
62
63
64 //jensen andersen inspired FFT feature detector
65 PV_JensenAndersen : PV_ChainUGen
66 {
67   *ar { arg buffer, propsc=0.25, prophfe=0.25, prophfc=0.25, propsf=0.25, threshold=1.0, waittime=0.04;
68     ^this.multiNew('audio', buffer, propsc, prophfe, prophfc, propsf,  threshold, waittime);
69   }
70 }
71
72
73 PV_HainsworthFoote : PV_ChainUGen
74 {
75   *ar { arg buffer, proph=0.0, propf=0.0, threshold=1.0, waittime=0.04;
76     ^this.multiNew('audio', buffer, proph, propf, threshold, waittime);
77   }
78 }
79
80 // Magnitude and phase quantizer
81 PV_Quantize : PV_ChainUGen
82 {
83   *new { arg buffer, bits = 8;
84     ^this.multiNew('control', buffer, bits)
85   }
86 }
87
88 //not FFT but useful for time domain onset detection
89 RunningSum : UGen
90 {
91   *ar { arg in, numsamp=40;
92     ^this.multiNew('audio', in, numsamp);
93   }
94
```

```
  *kr { arg in, numsamp=40;
    ^this.multiNew('control', in, numsamp);
  }

  *rms { arg in, numsamp=40;
    ^(RunningSum.ar(in.squared,numsamp)*(numsamp.reciprocal)).sqrt;
  }
}
```

A.2. supercollider/server/plugins/PV_ThirdParty.cpp

```
/*
  SuperCollider real time audio synthesis system
    Copyright (c) 2002 James McCartney. All rights reserved.
  http://www.audiosynth.com

    This program is free software; you can redistribute it and/or modify
    it under the terms of the GNU General Public License as published by
    the Free Software Foundation; either version 2 of the License, or
    (at your option) any later version.

    This program is distributed in the hope that it will be useful,
    but WITHOUT ANY WARRANTY; without even the implied warranty of
    MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
    GNU General Public License for more details.

    You should have received a copy of the GNU General Public License
    along with this program; if not, write to the Free Software
    Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301  USA
*/

//third party Phase Vocoder UGens

#include "math.h"
#include "FFT_UGens.h"

#define TWOPOW(x) (1 << (x))

extern "C"
{
  void PV_ConformalMap_Ctor(PV_Unit *unit);
  void PV_ConformalMap_next(PV_Unit *unit, int inNumSamples);

  void PV_Quantize_Ctor(PV_Unit *unit);
  void PV_Quantize_next(PV_Unit *unit, int inNumSamples);
  float quant(int bits, float input);
}


void PV_ConformalMap_Ctor(PV_Unit *unit)
{
  SETCALC(PV_ConformalMap_next);
  ZOUT0(0) = ZIN0(0);
}


void PV_ConformalMap_next(PV_Unit *unit, int inNumSamples)
{
  PV_GET_BUF

  SCComplexBuf *p = ToComplexApx(buf);

  float real2 = ZIN0(1);
  float imag2 = ZIN0(2);

  for (int i=0; i<numbins; ++i) {
    float real1 = p->bin[i].real;
```

```
    float imag1 = p->bin[i].imag;

    //apply conformal map z-> z-a/(1-za*) where z is the existing complex number in the bin...
    float numr= real1-real2;
    float numi= imag1-imag2;
    float denomr= 1.f - (real1*real2+imag1*imag2);
    float denomi= (real1*imag2- real2*imag1);

    numr= numr*denomr+numi*denomi;
    numi= numi*denomr-numr*denomi;

    //squared modulus
    denomr= denomr*denomr+denomi*denomi;

    //avoid possible divide by zero
    if(denomr<0.001f) denomr=0.001f;
    denomr=1.f/denomr;

    p->bin[i].real = numr*denomr;
    p->bin[i].imag = numi*denomr;
  }

}

// for PV_Quantize:
float quant(int bits, float input)
{
  unsigned long quantval = 0;
  int sign;
  double output;

  //quantize mid tread
  sign = (input < 0);

  input = fabs(input);
  if (input >= 1.0) {
    quantval = (long)(TWOPOW(bits - 1) - 1);
  } else {
    quantval = (long)(
          (
            (unsigned long)(TWOPOW(bits-1) + (TWOPOW(bits-1) - 1)) * input + 1.0
          ) / 2.0
        );
  }
  if (sign) { quantval |= TWOPOW(bits - 1); }

  //dequantize mid tread
  if (quantval & TWOPOW(bits - 1)) {
    sign = -1;
    quantval &= TWOPOW(bits - 1) - 1;
  } else {
    sign = 1;
  }
  output = sign * 2.0 * ((float)quantval / (TWOPOW(bits) - 1));

  return output;
}

void PV_Quantize_Ctor(PV_Unit *unit)
{
  SETCALC(PV_Quantize_next);
  ZOUT0(0) = ZIN0(0);
}

void PV_Quantize_next(PV_Unit *unit, int inNumSamples)
{
  PV_GET_BUF

```

```
  SCPolarBuf *p = ToPolarApx(buf);

  int bits = ZIN0(1);

  for (int i=0; i<numbins; ++i) {
    float mag = p->bin[i].mag;
    float phase = p->bin[i].phase;
    p->bin[i].mag = quant(bits, mag);
    p->bin[i].phase = quant(bits, phase);
  }

}

#define DefinePVUnit(name) \
  (*ft->fDefineUnit)(#name, sizeof(PV_Unit), (UnitCtorFunc)&name##_Ctor, 0, 0);

//void initPV_ThirdParty(InterfaceTable *it);
void initPV_ThirdParty(InterfaceTable *it)
{
  DefinePVUnit(PV_ConformalMap);
  DefinePVUnit(PV_Quantize);
}
```

A.3. supercollider/HelpSource/Classes/PV_Quantize.schelp

```
class:: PV_Quantize
summary:: Quantize magnitudes and phases.
related:: Classes/FFT, Classes/IFFT
categories:: UGens>FFT

Description::

Magnitudes and phases are quantized uniformly.


classmethods::

method::new

argument::buffer

FFT buffer.

argument::bits

Uniform quantization based on the number of bits.


Examples::

code::

s.boot;


b = Buffer.read(s, Platform.resourceDir +/+ "sounds/a11wlk01.wav");

(
SynthDef("help-quantize", { arg out=0, bits=8;
  var in, chain;
  in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
  chain = FFT(LocalBuf(2048), in);
  chain = PV_Quantize(chain, bits);
  Out.ar(out, 0.1 * IFFT(chain).dup);
}).play(s);
)

```

```
(
SynthDef("help-magNoise2", {  arg out=0, soundBufnum=2, bits=8;
  var in, chain;
  in = PlayBuf.ar(1, soundBufnum, BufRateScale.kr(soundBufnum), loop: 1);
  chain = FFT(LocalBuf(2048), in);
  chain = PV_Quantize(chain, bits);
  Out.ar(out, 0.2 * IFFT(chain).dup);
}).play(s,[\soundBufnum, b]);
)

b.free;

::
```

A.4. Example SC code: quantizer_example.scd

```
(
o = s.options;
o.memSize = 32768; //increase server's memory size for larger FFTs
s.boot;
s.meter;
)


(
SynthDef("myCoder", {arg out=0, bits=8, fftsize=4096, mul=1.0;
  // Applies frequency quantization on input audio -- CAREFUL OF FEEDBACK!
  var in, outSig, chain;
  //in = SinOsc.ar(SinOsc.kr(SinOsc.kr(0.08, 0, 6, 6.2).squared, 0, 100, 800));
  in = SoundIn.ar(0);
  chain = FFT(LocalBuf(fftsize), in);
  chain = PV_Quantize(chain, bits);
  Out.ar(out, IFFT(chain)*mul);
}).add;
)

// Try this out with a long window length -- left channel.
// Notice the temporal artifacts.
a = Synth("myCoder", [\out, 0, \bits, 8, \fftsize, 32768]);
a.set(\mul, 10.0); //increase gain - but be careful
a.set(\bits, 2);   //very pronounced quantization effects now.
a.set(\bits, 6);
a.set(\bits, 10);
a.free;

//try with a smaller window in the right channel.
b = Synth("myCoder", [\out, 1, \bits, 2, \fftsize, 256]);
b.set(\mul, 8.0); //increase gain - but be careful
b.set(\bits, 10);
b.set(\bits, 6);
b.set(\bits, 4);
b.set(\bits, 2);
b.free;
```

B. FAUST-RELATED C++ CODE LISTING

B.1. dummy.cpp

Below is the Faust compiler's output from the `dummy.dsp` file listed in Fig. 4, using `faust -o dummy.cpp dummy.dsp`.

```
//-----------------------------------------------------
// name: "Frequency Domain Quantizer"
// version: "0.1"
// author: "Tim O'Brien"
```

```
//
// Code generated with Faust 0.9.62 (http://faust.grame.fr)
//--------------------------------------------------------
#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

typedef long double quad;
/* link with  */

#include <math.h>

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

class mydsp : public dsp {
  private:
    FAUSTFLOAT     fslider0;
    float          fRec0[2];
    FAUSTFLOAT     fslider1;
    FAUSTFLOAT     fslider2;
    FAUSTFLOAT     fcheckbox0;
  public:
    static void metadata(Meta* m)     {
        m->declare("name", "Frequency Domain Quantizer");
        m->declare("version", "0.1");
        m->declare("author", "Tim O'Brien");
        m->declare("music.lib/name", "Music Library");
        m->declare("music.lib/author", "GRAME");
        m->declare("music.lib/copyright", "GRAME");
        m->declare("music.lib/version", "1.0");
        m->declare("music.lib/license", "LGPL with exception");
        m->declare("math.lib/name", "Math Library");
        m->declare("math.lib/author", "GRAME");
        m->declare("math.lib/copyright", "GRAME");
        m->declare("math.lib/version", "1.0");
        m->declare("math.lib/license", "LGPL with exception");
    }

    virtual int getNumInputs()     { return 1; }
    virtual int getNumOutputs()     { return 1; }
    static void classInit(int samplingFreq) {
    }
    virtual void instanceInit(int samplingFreq) {
        fSamplingFreq = samplingFreq;
        fslider0 = 0.0f;
        for (int i=0; i<2; i++) fRec0[i] = 0;
        fslider1 = 4.0f;
        fslider2 = 6e+01f;
        fcheckbox0 = 0.0;
    }
    virtual void init(int samplingFreq) {
        classInit(samplingFreq);
        instanceInit(samplingFreq);
    }
    virtual void buildUserInterface(UI* interface) {
        interface->openVerticalBox("dummy");
        interface->addHorizontalSlider("FFT block size (power of two)",
                &fslider2, 6e+01f, 6.0f, 18.0f, 1.0f);
        interface->addHorizontalSlider("Level (db)", &fslider0, 0.0f, -96.0f, 4.0f, 0.1f);
        interface->addCheckButton("On", &fcheckbox0);
        interface->addHorizontalSlider("Quantization bits", &fslider1, 4.0f, 2.0f, 16.0f, 1.0f);
        interface->closeBox();
    }
    virtual void compute (int count, FAUSTFLOAT** input, FAUSTFLOAT** output) {
        float     fSlow0 = (0.0010000000000000009f * powf(10,(0.05f * float(fslider0))));
        float     fSlow1 = ((float(fcheckbox0) * float(fslider2)) * float(fslider1));
```

```
        FAUSTFLOAT* input0 = input[0];
        FAUSTFLOAT* output0 = output[0];
        for (int i=0; i<count; i++) {
            fRec0[0] = (fSlow0 + (0.999f * fRec0[1]));
            output0[i] = (FAUSTFLOAT)(fSlow1 * ((float)input0[i] * fRec0[0]));
            // post processing
            fRec0[1] = fRec0[0];
        }
    }
};
```

B.2. frequantizer.onefft.cpp

Below is C++ code augmented from `dummy.cpp` above (section B.1). It implements the uniform frequency-domain quantization effect on one input audio stream with one FFT. As such, the window size slider has no effect.

```
//-----------------------------------------------------------------
// name: "Frequency Domain Quantizer"
// version: "0.1"
// author: "Tim O'Brien"
//
// Code generated with Faust 0.9.58 (http://faust.grame.fr)
// and then modified by Tim O'Brien with FFT-based functionality
//-----------------------------------------------------------------
#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

typedef long double quad;
// remember to link with -lfftw3 -lm

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

#include <iostream>
#include <cstdlib>
#include <math.h>
#include <complex>
#include <fftw3.h>
#include "quantize.h"
#include "window.h"

#define TWOPOW(x) (1 << (x))

float quantize(int bits, float input, float maxval)
{
      unsigned long quantval = 0;
      int sign;
      float output;

      sign = (input < 0);
      input = fabs(input);

      //quantize to bit code
      if (input >= maxval) {
      quantval = (long)(TWOPOW(bits - 1) - 1);
      } else {
      quantval = (long)(
        (
            (unsigned long)(TWOPOW(bits-1) +
            (TWOPOW(bits-1)-1)) * input/maxval + 1.0
        ) / 2.0
      );
      }
      if (sign) { quantval |= TWOPOW(bits - 1); }

```

```
52        //dequantize back to float
53        if (quantval & TWOPOW(bits - 1)) {
54        sign = -1;
55        quantval &= TWOPOW(bits - 1) - 1;
56        } else {
57        sign = 1;
58        }
59        output = sign * 2.0 * maxval * ((float)quantval / (TWOPOW(bits) - 1));
60
61        return output;
62  }
63
64  void sinewindow(double* inputdata, int length)
65  {
66        int i;
67        for (i = 0; i < length; ++i)
68        {
69        inputdata[i] = inputdata[i] * sin(M_PI*(i+0.5)/(double)length);
70        }
71  }
72
73  class mydsp : public dsp {
74    private:
75      FAUSTFLOAT     fslider0;   //Level (dB)
76      float          fRec0[2];
77      FAUSTFLOAT     fslider1;   //Quantization bits
78      FAUSTFLOAT     fslider2;   //FFT block size (power of two)
79      FAUSTFLOAT     fcheckbox0; //mute button
80
81      // My variables
82      fftw_complex *fftout; //output fft array
83      double *in, *out; //input & output sample arrays - real-valued signals
84      fftw_plan pf, pb;
85      int n = 32768; //fft window size - samples
86      int nc = (n/2) + 1; //buffer size for complex fft bins (using fftw r2c)
87      int halfn = n/2; //50% overlap
88      double *insig, *outsig, *halfframe;
89      int i, j, samp=0, fftcountin=0, fftcountout=0;
90      int bits = 8;
91
92    public:
93      static void metadata(Meta* m)     {
94          m->declare("name", "Frequency Domain Quantizer");
95          m->declare("version", "0.1");
96          m->declare("author", "Tim O'Brien");
97          m->declare("music.lib/name", "Music Library");
98          m->declare("music.lib/author", "GRAME");
99          m->declare("music.lib/copyright", "GRAME");
100         m->declare("music.lib/version", "1.0");
101         m->declare("music.lib/license", "LGPL with exception");
102         m->declare("math.lib/name", "Math Library");
103         m->declare("math.lib/author", "GRAME");
104         m->declare("math.lib/copyright", "GRAME");
105         m->declare("math.lib/version", "1.0");
106         m->declare("math.lib/license", "LGPL with exception");
107     }
108
109     virtual int getNumInputs()      { return 1; }
110     virtual int getNumOutputs()     { return 1; }
111     static void classInit(int samplingFreq) {
112     }
113     virtual void instanceInit(int samplingFreq) {
114         fSamplingFreq = samplingFreq;
115         fslider0      = 0.0f;   //Level (dB)
116         for (int i=0; i<2; i++) fRec0[i] = 0;
117         fslider1      = 4.0f;   //Quantization bits
118         fslider2      = 6e+01f; //FFT block size (power of two)
119         fcheckbox0    = 0.0;    //mute button
```

```
120     }
121
122     // FFT init function
123     virtual void FFTInit( void ) {
124       //data arrays for use with fftw
125       in     = (double*)       fftw_malloc( sizeof( double ) * n);
126       out    = (double*)       fftw_malloc( sizeof( double ) * n);
127       fftout = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc);
128       //forward & backward transform plans
129       pf = fftw_plan_dft_r2c_1d(n, in, fftout, FFTW_MEASURE);
130       pb = fftw_plan_dft_c2r_1d(n, fftout, out, FFTW_MEASURE);
131       //overlap-add frame...
132       halfframe = (double*) fftw_malloc( sizeof(double) * halfn );
133       for (i = 0; i < halfn; ++i) halfframe[i] = 0.0;
134       //buffers for queueing input and output
135       insig  = (double*) fftw_malloc( sizeof(double) * n);
136       outsig = (double*) fftw_malloc( sizeof(double) * n);
137       for (i = 0; i < n; ++i) {
138         insig[i]  = 0.0;
139         outsig[i] = 0.0;
140       }
141     }
142     // FFT destructor function
143     virtual void FFTDestruct( void ) {
144       printf("Called destructor function...\n");
145       fftw_destroy_plan(pf);
146       fftw_destroy_plan(pb);
147       fftw_free(in);
148       fftw_free(out);
149       fftw_free(fftout);
150       fftw_free(insig);
151       fftw_free(outsig);
152       fftw_free(halfframe);
153     }
154     virtual void checksize( int count ) {
155         //Hack: check that smallest fft buffer size is >= COUNT
156         //      AND n must also be an integer multiple of count...
157         if (2 * count > n) {
158           printf("FFT window size must be at least twice the"
159              " control block size.\n");
160           printf("FFT window: %d, control block size: %d.\n",
161              n, count);
162           exit(1);
163         }
164         else if (n % count) {
165           printf("FFT window size must be an integer multiple of the"
166              " control block size.\n");
167           printf("FFT window: %d, control block size: %d.\n",
168              n, count);
169           exit(1);
170         }
171     }
172
173     ~mydsp() {
174       printf("Calling destructor\n");
175       FFTDestruct();
176     }
177   virtual void init(int samplingFreq) {
178         classInit(samplingFreq);
179         instanceInit(samplingFreq);
180         FFTInit(); //one fft for now
181     }
182     virtual void buildUserInterface(UI* interface) {
183         interface->openVerticalBox("Frequency Domain Quantizer");
184         interface->addHorizontalSlider(
185             "FFT block size (power of two)",
186             &fslider2, 6e+01f, 6.0f, 18.0f, 1.0f
187         );
```

```
188         interface->addHorizontalSlider(
189             "Level (db)",
190             &fslider0, 0.0f, -60.0f, 40.0f, 0.1f
191         );
192         interface->addCheckButton("On", &fcheckbox0);
193         interface->addHorizontalSlider(
194             "Quantization bits",
195             &fslider1, 4.0f, 2.0f, 16.0f, 1.0f
196         );
197         interface->closeBox();
198     }
199     virtual void compute (
200                         int count,
201                         FAUSTFLOAT** input,
202                         FAUSTFLOAT** output
203                        )
204     {
205         checksize(count); //check ctrl block & FFT buffer sizes behave
206 
207         //linear amplitude factor from dB volume input variable
208         float fSlow0 = (
209             0.0010000000000000009f * powf(10,(0.05f * fslider0))
210         );
211 
212         //mute (really "on") button
213         float fSlow1 = fcheckbox0;
214 
215         // incoming audio for this ctrl block
216         FAUSTFLOAT* input0  = input[0];
217 
218         // outgoing audio for this ctrl block
219         FAUSTFLOAT* output0 = output[0];
220 
221 
222         //---------------
223         for (i = 0; i < count; i++) insig[i+fftcountin] = input0[i];
224 
225         fftcountin += count;
226 
227         if (fftcountin == n) //perform the fft/quantization/ifft)
228         {
229               for (i = 0; i < n; i++) in[i] = insig[i];
230               sinewindow(in, n); //window input
231               fftw_execute(pf);  //forward transform
232 
233               // quantize fft values
234               bits = (int) fslider1; //bit depth from GUI
235               for(i=0; i<nc; i++){
236                   fftout[i][0] = quantize(bits, fftout[i][0], (int) n/2.0);
237                   fftout[i][1] = quantize(bits, fftout[i][1], (int) n/2.0);
238               }
239 
240               fftw_execute(pb); // backward transform
241 
242               // scale by n after inverse tranform
243               for(i=0; i<n; i++) out[i] /= n;
244 
245               sinewindow(out, n); //window output
246 
247               //overlap-and-add
248               for (i = 0; i < halfn; i++){
249                   outsig[fftcountout+i] = fmin(
250                       1.0, fmax(-1.0, out[i] + halfframe[i])
251               );
252             insig[i]     = insig[i + halfn];
253             halfframe[i] = out[i + halfn];
254               }
255 
```

```
                fftcountin  -= halfn;
                fftcountout += halfn;
        }

        if (fftcountout>0){
                for (i = 0; i < count; ++i){
                    output0[i] = outsig[i];
                }
                for (i = 0; i<(fftcountout-count); i++){
                    outsig[i] = outsig[i + count];
                }
                fftcountout -= count;
        }
        else {
                for (i = 0; i < count; ++i){
                    output0[i] = 0.0;
                }
        }

        for (int i=0; i<count; i++) {
            fRec0[0] = (fSlow0 + (0.999f * fRec0[1]));
            output0[i] = (FAUSTFLOAT)(fSlow1 * ((float)output0[i] * fRec0[0]));
            // post processing
            fRec0[1] = fRec0[0];
        }

    }
};
```

B.3. frequantizer.cpp

Below is C++ code augmented from `dummy.cpp` above (section B.1). It implements the uniform frequency-domain quantization effect on one input audio stream with multiple FFT sizes. The window size slider crossfades between the different FFT outputs.

```
//-----------------------------------------------------------------
// name: "Frequency Domain Quantizer"
// version: "0.1"
// author: "Tim O'Brien"
//
// Code generated with Faust 0.9.58 (http://faust.grame.fr)
// and then modified by Tim O'Brien with FFT-based functionality
//-----------------------------------------------------------------
#ifndef FAUSTFLOAT
#define FAUSTFLOAT float
#endif

typedef long double quad;
// remember to link with -lfftw3 -lm

#ifndef FAUSTCLASS
#define FAUSTCLASS mydsp
#endif

#include <iostream>
#include <cstdlib>
#include <math.h>
#include <complex>
#include <fftw3.h>
#include "quantize.h"
#include "window.h"

#define TWOPOW(x) (1 << (x))

float quantize(int bits, float input, float maxval)
{
```

```
32        unsigned long quantval = 0;
33        int sign;
34        float output;
35
36        sign = (input < 0);
37        input = fabs(input);
38
39        //quantize to bit code
40        if (input >= maxval) {
41            quantval = (long)(TWOPOW(bits - 1) - 1);
42        } else {
43            quantval = (long)(
44                  (
45                      (unsigned long)(TWOPOW(bits-1) +
46                      (TWOPOW(bits-1)-1)) * input/maxval + 1.0
47                  ) / 2.0
48            );
49        }
50        if (sign) { quantval |= TWOPOW(bits - 1); }
51
52        //dequantize back to float
53        if (quantval & TWOPOW(bits - 1)) {
54            sign = -1;
55            quantval &= TWOPOW(bits - 1) - 1;
56        } else {
57            sign = 1;
58        }
59        output = sign * 2.0 * maxval * ((float)quantval / (TWOPOW(bits) - 1));
60
61        return output;
62  }
63
64  void sinewindow(double* inputdata, int length)
65  {
66        int i;
67        for (i = 0; i < length; ++i)
68        {
69            inputdata[i] = inputdata[i] * sin(M_PI*(i+0.5)/(double)length);
70        }
71  }
72
73  class mydsp : public dsp {
74    private:
75          FAUSTFLOAT          fslider0;   //Level (dB)
76          float                  fRec0[2];
77          FAUSTFLOAT          fslider1;   //Quantization bits
78          FAUSTFLOAT          fslider2;   //FFT block size (power of two)
79          FAUSTFLOAT          fcheckbox0; //mute button
80
81          // My variables
82          // ------------
83          //output fft arrays
84          fftw_complex *fftout0, *fftout1, *fftout2, *fftout3, *fftout4, *fftout5;
85          //input & output sample arrays - real-valued signals
86          double *in0,  *in1,  *in2,  *in3,  *in4,  *in5,
87                 *out0, *out1, *out2, *out3, *out4, *out5;
88          // plans for fftw
89          fftw_plan pf0, pf1, pf2, pf3, pf4, pf5,
90                    pb0, pb1, pb2, pb3, pb4, pb5;
91          int numwindows = 6; //number of different FFT window sizes to use
92          int smalln = 2048;  //smalles FFT window size. Goes up by powers of 2.
93          int *n; //array for fft window sizes
94          int *nc; //array for complex fft window sizes
95          int *halfn; //for 50% overlap
96          double *insig0,  *insig1,  *insig2,  *insig3,  *insig4,  *insig5,
97                 *outsig0, *outsig1, *outsig2, *outsig3, *outsig4, *outsig5,
98                 *halfframe0, *halfframe1, *halfframe2, *halfframe3, *halfframe4,
99                 *halfframe5;
```

```
        double *fftamount;
        int *fftcountin, *fftcountout;
        int i, j;
        int bits;

    public:
        static void metadata(Meta* m)         {
                m->declare("name", "Frequency Domain Quantizer");
                m->declare("version", "0.1");
                m->declare("author", "Tim O'Brien");
                m->declare("music.lib/name", "Music Library");
                m->declare("music.lib/author", "GRAME");
                m->declare("music.lib/copyright", "GRAME");
                m->declare("music.lib/version", "1.0");
                m->declare("music.lib/license", "LGPL with exception");
                m->declare("math.lib/name", "Math Library");
                m->declare("math.lib/author", "GRAME");
                m->declare("math.lib/copyright", "GRAME");
                m->declare("math.lib/version", "1.0");
                m->declare("math.lib/license", "LGPL with exception");
        }

        virtual int getNumInputs()         { return 1; }
        virtual int getNumOutputs()         { return 1; }
        static void classInit(int samplingFreq) {
        }
        virtual void instanceInit(int samplingFreq) {
                fSamplingFreq = samplingFreq;
                fslider0      = 0.0f;   //Level (dB)
                for (int i=0; i<2; i++) fRec0[i] = 0;
                fslider1      = 4.0f;   //Quantization bits
                fslider2      = 6e+01f; //FFT block size (power of two)
                fcheckbox0    = 0.0;    //mute button
        }

        virtual void FFTInit( void ) {
                //FFT window sizes
                n = (int*) malloc( sizeof( int ) * numwindows);
                for (i = 0; i < numwindows; i++) n[i] = ( smalln << i );
                //buffer sizes for complex fft bins (using fftw r2c)
                nc = (int*) malloc( sizeof( int ) * numwindows);
                for (i = 0; i < numwindows; i++) nc[i] = (n[i]/2) + 1;
                //for 50% overlap of windows
                halfn = (int*) malloc( sizeof( int ) * numwindows);
                for (i = 0; i < numwindows; i++) halfn[i] = n[i]/2;

                //data arrays for use with fftw
                in0 = (double*) fftw_malloc( sizeof( double ) * n[0]);
                in1 = (double*) fftw_malloc( sizeof( double ) * n[1]);
                in2 = (double*) fftw_malloc( sizeof( double ) * n[2]);
                in3 = (double*) fftw_malloc( sizeof( double ) * n[3]);
                in4 = (double*) fftw_malloc( sizeof( double ) * n[4]);
                in5 = (double*) fftw_malloc( sizeof( double ) * n[5]);
                out0 = (double*) fftw_malloc( sizeof( double ) * n[0]);
                out1 = (double*) fftw_malloc( sizeof( double ) * n[1]);
                out2 = (double*) fftw_malloc( sizeof( double ) * n[2]);
                out3 = (double*) fftw_malloc( sizeof( double ) * n[3]);
                out4 = (double*) fftw_malloc( sizeof( double ) * n[4]);
                out5 = (double*) fftw_malloc( sizeof( double ) * n[5]);
                fftout0 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[0]);
                fftout1 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[1]);
                fftout2 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[2]);
                fftout3 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[3]);
                fftout4 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[4]);
                fftout5 = (fftw_complex*) fftw_malloc(sizeof(fftw_complex) * nc[5]);
                //forward & backward transform plans
                pf0 = fftw_plan_dft_r2c_1d(n[0], in0, fftout0, FFTW_MEASURE);
                pf1 = fftw_plan_dft_r2c_1d(n[1], in1, fftout1, FFTW_MEASURE);
```

```
pf2 = fftw_plan_dft_r2c_1d(n[2], in2, fftout2, FFTW_MEASURE);
pf3 = fftw_plan_dft_r2c_1d(n[3], in3, fftout3, FFTW_MEASURE);
pf4 = fftw_plan_dft_r2c_1d(n[4], in4, fftout4, FFTW_MEASURE);
pf5 = fftw_plan_dft_r2c_1d(n[5], in5, fftout5, FFTW_MEASURE);
pb0 = fftw_plan_dft_c2r_1d(n[0], fftout0, out0, FFTW_MEASURE);
pb1 = fftw_plan_dft_c2r_1d(n[1], fftout1, out1, FFTW_MEASURE);
pb2 = fftw_plan_dft_c2r_1d(n[2], fftout2, out2, FFTW_MEASURE);
pb3 = fftw_plan_dft_c2r_1d(n[3], fftout3, out3, FFTW_MEASURE);
pb4 = fftw_plan_dft_c2r_1d(n[4], fftout4, out4, FFTW_MEASURE);
pb5 = fftw_plan_dft_c2r_1d(n[5], fftout5, out5, FFTW_MEASURE);
// for keeping track of incomin/outgoing samples...
fftcountin = (int*) malloc( sizeof( int ) * numwindows);
fftcountout = (int*) malloc( sizeof( int ) * numwindows);
// contribution of each fft to output signal...
fftamount = (double*) malloc( sizeof( double ) * numwindows);
for (i = 0; i < numwindows; i++){
  fftcountin[i] = 0.0;
  fftcountout[i] = 0.0;
  fftamount[i] = 0.0;
}
//overlap-add frame...
halfframe0 = (double*) fftw_malloc(sizeof(double) * halfn[0]);
halfframe1 = (double*) fftw_malloc(sizeof(double) * halfn[1]);
halfframe2 = (double*) fftw_malloc(sizeof(double) * halfn[2]);
halfframe3 = (double*) fftw_malloc(sizeof(double) * halfn[3]);
halfframe4 = (double*) fftw_malloc(sizeof(double) * halfn[4]);
halfframe5 = (double*) fftw_malloc(sizeof(double) * halfn[5]);
for (i = 0; i < halfn[0]; ++i) halfframe0[i] = 0.0;
for (i = 0; i < halfn[1]; ++i) halfframe1[i] = 0.0;
for (i = 0; i < halfn[2]; ++i) halfframe2[i] = 0.0;
for (i = 0; i < halfn[3]; ++i) halfframe3[i] = 0.0;
for (i = 0; i < halfn[4]; ++i) halfframe4[i] = 0.0;
for (i = 0; i < halfn[5]; ++i) halfframe5[i] = 0.0;
//buffers for queueing input and output
insig0 = (double*) fftw_malloc( sizeof(double) * n[0]);
insig1 = (double*) fftw_malloc( sizeof(double) * n[1]);
insig2 = (double*) fftw_malloc( sizeof(double) * n[2]);
insig3 = (double*) fftw_malloc( sizeof(double) * n[3]);
insig4 = (double*) fftw_malloc( sizeof(double) * n[4]);
insig5 = (double*) fftw_malloc( sizeof(double) * n[5]);
outsig0 = (double*) fftw_malloc( sizeof(double) * n[0]);
outsig1 = (double*) fftw_malloc( sizeof(double) * n[1]);
outsig2 = (double*) fftw_malloc( sizeof(double) * n[2]);
outsig3 = (double*) fftw_malloc( sizeof(double) * n[3]);
outsig4 = (double*) fftw_malloc( sizeof(double) * n[4]);
outsig5 = (double*) fftw_malloc( sizeof(double) * n[5]);
for (i = 0; i < n[0]; ++i) {
  insig0[i]  = 0.0;
  outsig0[i] = 0.0;
}
for (i = 0; i < n[1]; ++i) {
  insig1[i]  = 0.0;
  outsig1[i] = 0.0;
}
for (i = 0; i < n[2]; ++i) {
  insig2[i]  = 0.0;
  outsig2[i] = 0.0;
}
for (i = 0; i < n[3]; ++i) {
  insig3[i]  = 0.0;
  outsig3[i] = 0.0;
}
for (i = 0; i < n[4]; ++i) {
  insig4[i]  = 0.0;
  outsig4[i] = 0.0;
}
for (i = 0; i < n[5]; ++i) {
  insig5[i]  = 0.0;
```

```
                outsig5[i] = 0.0;
              }
        }
        // FFT destructor function
        virtual void FFTDestruct( void ) {
          printf("Cleaning up...\n");
          fftw_destroy_plan(pf0);
          fftw_destroy_plan(pb0);
          fftw_free(in0);
          fftw_free(in1);
          fftw_free(in2);
          fftw_free(in3);
          fftw_free(in4);
          fftw_free(in5);
          fftw_free(out0);
          fftw_free(out1);
          fftw_free(out2);
          fftw_free(out3);
          fftw_free(out4);
          fftw_free(out5);
          fftw_free(fftout0);
          fftw_free(fftout1);
          fftw_free(fftout2);
          fftw_free(fftout3);
          fftw_free(fftout4);
          fftw_free(fftout5);
          fftw_free(insig0);
          fftw_free(insig1);
          fftw_free(insig2);
          fftw_free(insig3);
          fftw_free(insig4);
          fftw_free(insig5);
          fftw_free(outsig0);
          fftw_free(outsig1);
          fftw_free(outsig2);
          fftw_free(outsig3);
          fftw_free(outsig4);
          fftw_free(outsig5);
          fftw_free(halfframe0);
          fftw_free(halfframe1);
          fftw_free(halfframe2);
          fftw_free(halfframe3);
          fftw_free(halfframe4);
          fftw_free(halfframe5);
          free(n);
          free(nc);
          free(halfn);
          free(fftcountin);
          free(fftcountout);
          free(fftamount);
        }
        virtual void checksize( int count ) {
            for (i = 0; i < numwindows; i++)
            {
                //Hack: check that smallest fft buffer size is >= COUNT
                //      AND n vals must also be an integer multiple of count...
                if (2 * count > n[i]) {
                  printf("FFT window size must be at least twice the"
                         " control block size.\n");
                  printf("FFT window: %d, control block size: %d.\n",
                         n[i], count);
                  exit(1);
                }
                else if (n[i] % count) {
                  printf("FFT window size must be an integer multiple of the"
                         " control block size.\n");
                  printf("FFT window: %d, control block size: %d.\n",
                         n[i], count);
```

```
304                   exit(1);
305                 }
306             }
307         }
308 
309         ~mydsp() {
310           FFTDestruct();
311         }
312   virtual void init(int samplingFreq) {
313                 classInit(samplingFreq);
314                 instanceInit(samplingFreq);
315                 FFTInit(); //one fft for now
316         }
317         virtual void buildUserInterface(UI* interface) {
318                 interface->openVerticalBox("Frequency Domain Quantizer");
319                 interface->addHorizontalSlider(
320                     "FFT block size (power of two)",
321                     &fslider2, 13.0f, 11.0f, 18.0f, 0.1f
322                 );
323                 interface->addHorizontalSlider(
324                     "Level (db)",
325                     &fslider0, 0.0f, -60.0f, 90.0f, 0.1f
326                 );
327                 interface->addCheckButton("On", &fcheckbox0);
328                 interface->addHorizontalSlider(
329                     "Quantization bits",
330                     &fslider1, 4.0f, 2.0f, 16.0f, 1.0f
331                 );
332                 interface->closeBox();
333         }
334         virtual void compute (
335                         int count,
336                         FAUSTFLOAT** input,
337                         FAUSTFLOAT** output
338                        )
339         {
340                 checksize(count); //check ctrl block & FFT buffer sizes behave
341 
342                 //linear amplitude factor from dB volume input variable
343                 float fSlow0 = (
344                     0.0010000000000000009f * powf(10,(0.05f * fslider0))
345                 );
346 
347                 //mute (really "on") button
348                 float fSlow1 = fcheckbox0;
349 
350                 // incoming audio for this ctrl block
351                 FAUSTFLOAT* input0  = input[0];
352 
353                 // outgoing audio for this ctrl block
354                 FAUSTFLOAT* output0 = output[0];
355 
356 
357                 //Queue up this control block of input samples for the various
358                 //FFTs...
359                 for (i = 0; i < count; i++){
360                       insig0[i+fftcountin[0]] = input0[i];
361                       insig1[i+fftcountin[1]] = input0[i];
362                       insig2[i+fftcountin[2]] = input0[i];
363                       insig3[i+fftcountin[3]] = input0[i];
364                       insig4[i+fftcountin[4]] = input0[i];
365                       insig5[i+fftcountin[5]] = input0[i];
366                 }
367                 for (i = 0; i < numwindows; i++) fftcountin[i] += count;
368 
369                 //check/perform the fft/quantization/iffts for the various
370                 //fft sizes...
371 
```

```
372
373                 if (fftcountin[0] == n[0])
374                 {
375                       for (i = 0; i < n[0]; i++) in0[i] = insig0[i];
376                       sinewindow(in0, n[0]); //window input
377                       fftw_execute(pf0);  //forward transform
378
379                       // quantize fft values
380                       bits = (int) fslider1; //bit depth from GUI
381                       for(i=0; i<nc[0]; i++){
382                         fftout0[i][0] = quantize(bits, fftout0[i][0], (int) n[0]/2.0);
383                         fftout0[i][1] = quantize(bits, fftout0[i][1], (int) n[0]/2.0);
384                       }
385
386                       fftw_execute(pb0); // backward transform
387
388                       // scale by n after inverse tranform
389                       for(i=0; i<n[0]; i++) out0[i] /= n[0];
390
391                       sinewindow(out0, n[0]); //window output
392
393                       //overlap-and-add
394                       for (i = 0; i < halfn[0]; i++){
395                         outsig0[fftcountout[0]+i] = fmin(
396                             1.0, fmax(-1.0, out0[i] + halfframe0[i])
397                         );
398                         insig0[i]     = insig0[i + halfn[0]];
399                         halfframe0[i] = out0[i + halfn[0]];
400                       }
401
402                       fftcountin[0]  -= halfn[0];
403                       fftcountout[0] += halfn[0];
404                 }
405
406                 if (fftcountin[1] == n[1])
407                 {
408                       for (i = 0; i < n[1]; i++) in1[i] = insig1[i];
409                       sinewindow(in1, n[1]); //window input
410                       fftw_execute(pf1);  //forward transform
411
412                       // quantize fft values
413                       bits = (int) fslider1; //bit depth from GUI
414                       for(i=0; i<nc[1]; i++){
415                         fftout1[i][0] = quantize(bits, fftout1[i][0], (int) n[1]/2.0);
416                         fftout1[i][1] = quantize(bits, fftout1[i][1], (int) n[1]/2.0);
417                       }
418
419                       fftw_execute(pb1); // backward transform
420
421                       // scale by n after inverse tranform
422                       for(i=0; i<n[1]; i++) out1[i] /= n[1];
423
424                       sinewindow(out1, n[1]); //window output
425
426                       //overlap-and-add
427                       for (i = 0; i < halfn[1]; i++){
428                         outsig1[fftcountout[1]+i] = fmin(
429                             1.0, fmax(-1.0, out1[i] + halfframe1[i])
430                         );
431                         insig1[i]     = insig1[i + halfn[1]];
432                         halfframe1[i] = out1[i + halfn[1]];
433                       }
434
435                       fftcountin[1]  -= halfn[1];
436                       fftcountout[1] += halfn[1];
437                 }
438
439                 if (fftcountin[2] == n[2]) //perform the fft/quantization/ifft
```

```
440                 {
441                       for (i = 0; i < n[2]; i++) in2[i] = insig2[i];
442                       sinewindow(in2, n[2]); //window input
443                       fftw_execute(pf2);  //forward transform
444
445                       // quantize fft values
446                       bits = (int) fslider1; //bit depth from GUI
447                       for(i=0; i<nc[2]; i++){
448                         fftout2[i][0] = quantize(bits, fftout2[i][0], (int) n[2]/2.0);
449                         fftout2[i][1] = quantize(bits, fftout2[i][1], (int) n[2]/2.0);
450                       }
451
452                       fftw_execute(pb2); // backward transform
453
454                       // scale by n after inverse tranform
455                       for(i=0; i<n[2]; i++) out2[i] /= n[2];
456
457                       sinewindow(out2, n[2]); //window output
458
459                       //overlap-and-add
460                       for (i = 0; i < halfn[2]; i++){
461                         outsig2[fftcountout[2]+i] = fmin(
462                             1.0, fmax(-1.0, out2[i] + halfframe2[i])
463                         );
464                         insig2[i]     = insig2[i + halfn[2]];
465                         halfframe2[i] = out2[i + halfn[2]];
466                       }
467
468                       fftcountin[2]  -= halfn[2];
469                       fftcountout[2] += halfn[2];
470                 }
471
472                 if (fftcountin[3] == n[3]) //perform the fft/quantization/ifft
473                 {
474                       for (i = 0; i < n[3]; i++) in3[i] = insig3[i];
475                       sinewindow(in3, n[3]); //window input
476                       fftw_execute(pf3);  //forward transform
477
478                       // quantize fft values
479                       bits = (int) fslider1; //bit depth from GUI
480                       for(i=0; i<nc[3]; i++){
481                         fftout3[i][0] = quantize(bits, fftout3[i][0], (int) n[3]/2.0);
482                         fftout3[i][1] = quantize(bits, fftout3[i][1], (int) n[3]/2.0);
483                       }
484
485                       fftw_execute(pb3); // backward transform
486
487                       // scale by n after inverse tranform
488                       for(i=0; i<n[3]; i++) out3[i] /= n[3];
489
490                       sinewindow(out3, n[3]); //window output
491
492                       //overlap-and-add
493                       for (i = 0; i < halfn[3]; i++){
494                         outsig3[fftcountout[3]+i] = fmin(
495                             1.0, fmax(-1.0, out3[i] + halfframe3[i])
496                         );
497                         insig3[i]     = insig3[i + halfn[3]];
498                         halfframe3[i] = out3[i + halfn[3]];
499                       }
500
501                       fftcountin[3]  -= halfn[3];
502                       fftcountout[3] += halfn[3];
503                 }
504
505                 if (fftcountin[4] == n[4]) //perform the fft/quantization/ifft
506                 {
507                       for (i = 0; i < n[4]; i++) in4[i] = insig4[i];
```

```
508                     sinewindow(in4, n[4]); //window input
509                     fftw_execute(pf4);  //forward transform
510 
511                     // quantize fft values
512                     bits = (int) fslider1; //bit depth from GUI
513                     for(i=0; i<nc[4]; i++){
514                       fftout4[i][0] = quantize(bits, fftout4[i][0], (int) n[4]/2.0);
515                       fftout4[i][1] = quantize(bits, fftout4[i][1], (int) n[4]/2.0);
516                     }
517 
518                     fftw_execute(pb4); // backward transform
519 
520                     // scale by n after inverse tranform
521                     for(i=0; i<n[4]; i++) out4[i] /= n[4];
522 
523                     sinewindow(out4, n[4]); //window output
524 
525                     //overlap-and-add
526                     for (i = 0; i < halfn[4]; i++){
527                       outsig4[fftcountout[4]+i] = fmin(
528                           1.0, fmax(-1.0, out4[i] + halfframe4[i])
529                       );
530                       insig4[i]     = insig4[i + halfn[4]];
531                       halfframe4[i] = out4[i + halfn[4]];
532                     }
533 
534                     fftcountin[4]  -= halfn[4];
535                     fftcountout[4] += halfn[4];
536                   }
537 
538                   if (fftcountin[5] == n[5]) //perform the fft/quantization/ifft
539                   {
540                     for (i = 0; i < n[5]; i++) in5[i] = insig5[i];
541                     sinewindow(in5, n[5]); //window input
542                     fftw_execute(pf5);  //forward transform
543 
544                     // quantize fft values
545                     bits = (int) fslider1; //bit depth from GUI
546                     for(i=0; i<nc[5]; i++){
547                       fftout5[i][0] = quantize(bits, fftout5[i][0], (int) n[5]/2.0);
548                       fftout5[i][1] = quantize(bits, fftout5[i][1], (int) n[5]/2.0);
549                     }
550 
551                     fftw_execute(pb5); // backward transform
552 
553                     // scale by n after inverse tranform
554                     for(i=0; i<n[5]; i++) out5[i] /= n[5];
555 
556                     sinewindow(out5, n[5]); //window output
557 
558                     //overlap-and-add
559                     for (i = 0; i < halfn[5]; i++){
560                       outsig5[fftcountout[5]+i] = fmin(
561                           1.0, fmax(-1.0, out5[i] + halfframe5[i])
562                       );
563                       insig5[i]     = insig5[i + halfn[5]];
564                       halfframe5[i] = out5[i + halfn[5]];
565                     }
566 
567                     fftcountin[5]  -= halfn[5];
568                     fftcountout[5] += halfn[5];
569                   }
570 
571 
572                   //--------------
573                   // combine into count-segmented audio streams for each fft
574                   for (i = 0; i < count; ++i) output0[i] = (FAUSTFLOAT) 0.0;
575                   for (i = 0; i < numwindows; i++) //for crossfading
```

```
                {
                    fftamount[i] = max(1.0 - fabs(11.0+i-fslider2),0.0);
                }
                if (fftcountout[0]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig0[i] * fftamount[0]);
                      }
                      for (i = 0; i<(fftcountout[0]-count); i++){
                          outsig0[i] = outsig0[i + count];
                      }
                      fftcountout[0] -= count;
                }
                if (fftcountout[1]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig1[i] * fftamount[1]);
                      }
                      for (i = 0; i<(fftcountout[1]-count); i++){
                          outsig1[i] = outsig1[i + count];
                      }
                      fftcountout[1] -= count;
                }
                if (fftcountout[2]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig2[i] * fftamount[2]);
                      }
                      for (i = 0; i<(fftcountout[2]-count); i++){
                          outsig2[i] = outsig2[i + count];
                      }
                      fftcountout[2] -= count;
                }
                if (fftcountout[3]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig3[i] * fftamount[3]);
                      }
                      for (i = 0; i<(fftcountout[3]-count); i++){
                          outsig3[i] = outsig3[i + count];
                      }
                      fftcountout[3] -= count;
                }
                if (fftcountout[4]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig4[i] * fftamount[4]);
                      }
                      for (i = 0; i<(fftcountout[4]-count); i++){
                          outsig4[i] = outsig4[i + count];
                      }
                      fftcountout[4] -= count;
                }
                if (fftcountout[5]>0){
                      for (i = 0; i < count; ++i){
                          output0[i] += (FAUSTFLOAT)(outsig5[i] * fftamount[5]);
                      }
                      for (i = 0; i<(fftcountout[5]-count); i++){
                          outsig5[i] = outsig5[i + count];
                      }
                      fftcountout[5] -= count;
                }

                //-----------------------------------------------------------------
                //Volume and mute
                for (int i=0; i<count; i++) {
                        output0[i] = (FAUSTFLOAT)(fSlow1 * ((float)output0[i] * fSlow0));
                }
        }
};
```

B.4. architecture_mergy.py

```
#!/usr/bin/env python
# encoding: utf-8

#############################################################################
# architecture_merge.py
#
# python script to convert a .cpp file generated (at least initially) by faust
# (compiled without an architecture specified) and merge it with the relevant
# architecture file.
#############################################################################

from optparse import OptionParser
import re #regex
import os

def main():
    argParser = OptionParser(
        usage="usage: %prog [options] myfile.cpp path/to/architectureFile.cpp"
    )

    (options, args) = argParser.parse_args()

    # Open input file and architecture file, error if something is awry
    if len(args) == 2:
        print("Your C++ file:     {}".format(args[0]))
        try:
            myfile = open(args[0],"r")
        except IOError as e:
            print "I/O error({0}): {1}".format(e.errno, e.strerror)
            exit(-1)
        if (os.path.splitext(os.path.basename(args[0]))[1]!=".cpp"):
            print("{} is not a .cpp file".format(os.path.basename(args[0])))
            exit(-1)
        print("Architecture file: {}".format(args[1]))
        try:
            archfile = open(args[1],"r")
        except IOError as e:
            print "I/O error({0}): {1}".format(e.errno, e.strerror)
            exit(-1)
        if (os.path.splitext(os.path.basename(args[1]))[1]!=".cpp"):
            print("{} is not a .cpp file".format(os.path.basename(args[1])))
            exit(-1)
    else:
        argParser.print_help()
        exit(-1)

    # open output file
    try:
        outputfile = open(
                os.path.splitext(os.path.basename(args[0]))[0] + "_" +\
                os.path.splitext(os.path.basename(args[1]))[0] + ".cpp",
                "w")
    except IOError as e:
        print "Output file I/O error({0}): {1}".format(e.errno, e.strerror)
        exit(-1)

    # Right now, we don't do anything with the vector instrinsics.
    # Look for the <<includeIntrinsic>> tag and erase it.
    # Then go to the <<includeclass>> tag, erase it, and paste in the whole
    # input .cpp file.

    intrinsic = re.compile('<<includeIntrinsic>>')
    dspclass = re.compile('<<includeclass>>')

    for line in archfile:
        if ((intrinsic.findall(line)==[]) and (dspclass.findall(line)==[])):
```

```
            #add line from architecture file to output file
            outputfile.write(line)
        elif ((intrinsic.findall(line)==[]) and (dspclass.findall(line)!=[])):
            #add entire input .cpp file here
            for cppline in myfile:
                outputfile.write(cppline)

    myfile.close()
    archfile.close()
    outputfile.close()



if __name__ == "__main__":
    main()
```

B.5. fcpp2jack

```
#!/bin/bash

OSCDEFS=""

# Adapted from faust2jack:
#####################################################################
#                                                                   #
#               Compiles Faust programs to jack-gtk                 #
#               (c) Grame, 2009-2011                                #
#                                                                   #
#####################################################################
# Modification by Tim O'Brien 2013

#-----------------------------------------------------------------
# Set Faust include path

if [ -f $FAUST_LIB_PATH/music.lib ]
then
  FAUSTLIB=$FAUST_LIB_PATH
elif [ -f /usr/local/lib/faust/music.lib ]
then
  FAUSTLIB=/usr/local/lib/faust/
elif [ -f /usr/lib/faust/music.lib ]
then
  FAUSTLIB=/usr/lib/faust/
else
  error "$0: Cannot find Faust library dir (usually /usr/local/lib/faust)"
fi


#-----------------------------------------------------------------
# Check darwin specifics
#
if [[ $(uname) == Darwin ]]; then
    MARCH=""
else
    MARCH="-march=native"
fi

#-----------------------------------------------------------------
# Default compilation flags for gcc and icc :
#
MYGCCFLAGS="-O3 $MARCH -mfpmath=sse -msse -msse2 -msse3 -ffast-math -ftree-vectorize"
MYICCFLAGS="-O3 -xHost -ftz -fno-alias -fp-model fast=2"


#-----------------------------------------------------------------
# Analyze command arguments :
# faust options                 -> OPTIONS
```

```
# if -omp : -openmp or -fopenmp -> OPENMP
# existing *.cpp files          -> FILES
#

# PHASE 1 : Look for -icc option to force use of intel icc (actually icpc)
# without having to configure CXX and CXXFLAGS
CXX=g++
CXXFLAGS=$MYGCCFLAGS
for p in $@; do
    if [ "$p" = -icc ]; then
        CXX=icpc
        CXXFLAGS=$MYICCFLAGS
    fi
done


# PHASE 2 : dispatch command arguments
for p in $@; do
    if [ "$p" = -omp ]; then
        if [[ $CXX == "icpc" ]]; then
            OMP="-openmp"
        else
            OMP="-fopenmp"
        fi
    fi

    if [ "$p" = -icc ]; then
        ignore=" "
    elif [ $p = "-osc" ]; then
        OSCDEFS="-DOSCCTRL -L$FAUSTLIB -lOSCFaust -loscpack"
    elif [ $p = "-httpd" ]; then
        HTTPDEFS="-DHTTPCTRL -L$FAUSTLIB -lHTTPDFaust -lmicrohttpd"
    elif [ $p = "-arch32" ]; then
        PROCARCH="-m32 -L/usr/lib32"
    elif [ $p = "-arch64" ]; then
        PROCARCH="-m64"
    elif [ ${p:0:1} = "-" ]; then
        OPTIONS="$OPTIONS $p"
    elif [[ -e "$p" ]]; then
        FILES="$FILES $p"
    else
        OPTIONS="$OPTIONS $p"
    fi
done


#-------------------------------------------------------------------
# compile the *.cpp files for JACK-GTK on linux
#
SCRIPT_DIR=${0%/*}
ARCH_DIR=$SCRIPT_DIR/architecture_files
for f in $FILES; do

        # compile faust to c++
        ABS_INPUT_PATH=$(dirname "$f")
        INPUT_FILE=${f##*/}
        INPUT="$ABS_INPUT_PATH/$INPUT_FILE"
        MYDSP=${INPUT_FILE%%.*} #base name of input .cpp file
        ./$SCRIPT_DIR/architecture_merge.py $INPUT $ARCH_DIR/jack-gtk.cpp

        # compile c++ to binary
        (
                $CXX $CXXFLAGS $OMP "${MYDSP}_jack-gtk.cpp" `pkg-config --cflags --libs jack gtk+-2.0`
                        $PROCARCH $OSCDEFS $HTTPDEFS -o "${MYDSP}" $OPTIONS
        ) > /dev/null

        # collect binary file name for FaustWorks
        BINARIES="$BINARIES${f%.dsp};"
```

```
done


echo $BINARIES
```

C. REFERENCES

[1] M. Bosi and R. E. Goldberg, *Introduction to Digital Audio Coding and Standards*. Norwell, MA, USA: Kluwer Academic Publishers, 2003.

[2] G. Bosi, Marina; Davidson, "High-quality, low-rate audio transform coding for transmission and multimedia applications," in *Audio Engineering Society Convention 93*, 10 1992.

[3] G. Brandenburg, Karlheinz; Stoll, "ISO/MPEG-1 audio: A generic standard for coding of high-quality digital audio," *J. Audio Eng. Soc*, vol. 42, no. 10, pp. 780–792, 1994.

[4] M. Bosi, K. Brandenburg, S. Quackenbush, L. Fielder, K. Akagiri, H. Fuchs, and M. Dietz, "ISO/IEC MPEG-2 advanced audio coding," *J. Audio Eng. Soc*, vol. 45, no. 10, pp. 789–814, 1997.

[5] ISO/IEC, "ISO/IEC 13818-3:1998 - information technology – generic coding of moving pictures and associated audio information – part 3: Audio," 1998.

[6] D. Bellan, A. Brandolini, and A. Gandelli, "Quantization theory-a deterministic approach," *Instrumentation and Measurement, IEEE Transactions on*, vol. 48, no. 1, pp. 18–25, 1999.

[7] V. Britanak and K. Rao, "An efficient implementation of the forward and inverse mdct in mpeg audio coding," *Signal Processing Letters, IEEE*, vol. 8, no. 2, pp. 48–51, 2001.

[8] B. Edler, "Coding of audio signals with overlapping block transform and adaptive window functions," *Frequenz*, vol. 43, no. 9, pp. 252–256, 1989.

[9] J. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies, and M. B. Sandler, "A tutorial on onset detection in music signals," *Speech and Audio Processing, IEEE Transactions on*, vol. 13, no. 5, pp. 1035–1047, Sept. 2005.

[10] J. McCartney, "Rethinking the computer music language: Supercollider," *Computer Music Journal*, vol. 26, no. 4, pp. 61–68, 2002.

[11] S. Wilson, D. Cottle, and N. Collins, *The SuperCollider Book*. The MIT Press, 2011.

[12] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of the International Computer Music Conference (ICMA)*, (Gothenburg, Sweden), pp. 542–547, 2002.

[13] G. Albert, "Interfacing pure data with faust," *LINUX AUDIO*, p. 24, 2007.