disco

Claude Heiland-Allen

2019

Contents

1 disco-designer.c

```
#include <SDL2/SDL.h>
#include <SDL2/SDL_audio.h>
#include <SDL2/SDL_opengl.h>

#ifdef __EMSCRIPTEN__
#include <emscripten.h>
#else
#include <unistd.h>
#endif

#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <tgmath.h>
#include <time.h>

#define OCTAVES 11
#define OCTAVESIZE (1 << (OCTAVES - 1))

#define RHYTHMS 11
#define RHYTHMSIZE (1 << (RHYTHMS - 1))

#define OVERLAP 4

#define OHOP (OCTAVESIZE / OVERLAP)
#define RHOP (RHYTHMSIZE / OVERLAP)
```

```
#define CHANNELS 2

#define CELLSIZE 64
#define FPS 60

static struct
{
  float owindow[OCTAVESIZE]; // octave window
  float rwindow[RHYTHMSIZE]; // rhythm window
  float nbuffer[OCTAVESIZE][CHANNELS]; // source audio block
  float obuffer[OCTAVESIZE][CHANNELS]; // output audio block
  float ebuffer[RHYTHMSIZE][OCTAVES]; // source rhythm history
  float rbuffer[RHYTHMSIZE][OCTAVES]; // output rhythm history
  float abuffer[OHOP][CHANNELS]; // source audio
  float R[OCTAVES][RHYTHMS]; // fingerprint
  bool normalize_carrier;
  int eindex; // source rhythm history index
  int rindex; // output rhythm history index
  int aindex; // source audio index
  int oindex; // output audio index
  int oavailable; // output audio buffer fill state
  int SR; // sample rate
  // temporary data follows
  float haar[2][OCTAVESIZE][CHANNELS];
  double E[OCTAVES];
  float rhaar[2][RHYTHMSIZE];
  // video state
  SDL_Window* window;
  SDL_GLContext context;
  int target_octave;
  int target_rhythm;
  double target_delta;
} state;


static bool read_fingerprint(const char *filename)
{
  FILE *ifile = fopen(filename, "rb");
  if (ifile)
  {
    bool ok = true;
    for (int i = 0; i < OCTAVES; ++i)
    {
      for (int j = 0; j < RHYTHMS; ++j)
      {
        double r = 0;
        ok &= (1 == fscanf(ifile, "%lf ", &r));
        state.R[i][j] = r;
      }
    }
    fclose(ifile);
    return ok;
  }
  else
  {
    return false;
  }
```

```
}

static void normalize_fingerprint(void)
{
  double sr = 0;
  for (int i = 0; i < OCTAVES; ++i)
  {
    double r = state.R[i][0];
    sr += r * r;
  }
  sr /= OCTAVES;
  sr = sqrt(sr);
  for (int i = 0; i < OCTAVES; ++i)
  {
    state.R[i][0] /= sr;
  }
#if 0
  sr = 0;
  for (int i = 0; i < OCTAVES; ++i)
  {
    for (int j = 1; j < RHYTHMS; ++j)
    {
      double r = state.R[i][j];
      sr += r * r;
    }
  }
  sr /= OCTAVES * (RHYTHMS - 1);
  sr = sqrt(sr);
#endif
  for (int i = 0; i < OCTAVES; ++i)
  {
    for (int j = 1; j < RHYTHMS; ++j)
    {
      state.R[i][j] /= sr;
    }
  }
}

static void initialize_windows(void)
{
  for (int i = 0; i < OCTAVESIZE; ++i)
  {
    state.owindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / OCTAVESIZE);
  }
  for (int i = 0; i < RHYTHMSIZE; ++i)
  {
    state.rwindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / RHYTHMSIZE);
  }
}

static void audio1(float output[CHANNELS])
{
  while (state.oavailable <= 0)
  {
    // generate noise
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
```

```
      state.abuffer[state.aindex][channel] = 2 * (rand() / (double) RAND_MAX - 0.5);
    }
    state.aindex++;
    if (state.aindex == OHOP)
    {
      state.aindex = 0;
      for (int i = 0; i < OCTAVESIZE - OHOP; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          state.nbuffer[i][channel] = state.nbuffer[i + OHOP][channel];
        }
      }
      for (int i = OCTAVESIZE - OHOP; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          state.nbuffer[i][channel] = state.abuffer[i - (OCTAVESIZE - OHOP)][channel];
        }
      }

      // window
      int src = 0;
      int dst = 1;
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          state.haar[src][i][channel] = state.nbuffer[i][channel] * state.owindow[i];
        }
      }

      // compute Haar wavelet transform
      for (int length = OCTAVESIZE >> 1; length > 0; length >>= 1)
      {
        for (int i = 0; i < length; ++i)
        {
          for (int channel = 0; channel < CHANNELS; ++channel)
          {
            float a = state.haar[src][2 * i + 0][channel];
            float b = state.haar[src][2 * i + 1][channel];
            float s = (a + b) / 2;
            float d = (a - b) / 2;
            state.haar[dst][         i][channel] = s;
            state.haar[dst][length + i][channel] = d;
          }
        }
        for (int i = 0; i < OCTAVESIZE; ++i)
        {
          for (int channel = 0; channel < CHANNELS; ++channel)
          {
            state.haar[src][i][channel] = state.haar[dst][i][channel];
          }
        }
```

```
    }

    // compute energy per octave
    memset(state.E, 0, sizeof(state.E));
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      state.E[0] += fabsf(state.haar[src][0][channel]); // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      double rms = 0;
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          double a = state.haar[src][length + i][channel];
          rms += a * a;
        }
      }
      rms /= length * CHANNELS;
      rms = sqrt(fmax(rms, 0));
      state.E[octave] = rms;
    }

    // copy to ebuffer
    for (int octave = 0; octave < OCTAVES; ++octave)
    {
      state.ebuffer[state.eindex][octave] = state.E[octave];
    }
    state.eindex++;

    if (state.eindex == RHYTHMSIZE)
    {
      for (int octave = 0; octave < OCTAVES; ++octave)
      {

        // window
        for (int i = 0; i < RHYTHMSIZE; ++i)
        {
          state.rhaar[src][i] = state.ebuffer[i][octave];// * rwindow[i];
        }

        // compute Haar wavelet transform
        for (int length = RHYTHMSIZE >> 1; length > 0; length >>= 1)
        {
          for (int i = 0; i < length; ++i)
          {
              float a = state.rhaar[src][2 * i + 0];
              float b = state.rhaar[src][2 * i + 1];
              float s = (a + b) / 2;
              float d = (a - b) / 2;
              state.rhaar[dst][         i] = s;
              state.rhaar[dst][length + i] = d;
          }
```

```
        for (int i = 0; i < RHYTHMSIZE; ++i)
        {
            state.rhaar[src][i] = state.rhaar[dst][i];
        }
      }

      // normalize carrier
      if (state.normalize_carrier)
      {
        state.rhaar[src][0] = 1;
        for ( int rhythm = 1, length = 1
            ; length <= RHYTHMSIZE >> 1
            ; rhythm += 1, length <<= 1
            )
        {
          double rms = 0;
          for (int i = 0; i < length; ++i)
          {
            double d = state.rhaar[src][length + i];
            rms += d * d;
          }
          rms /= length;
          rms = sqrt(rms);
          if (rms > 0)
          {
            for (int i = 0; i < length; ++i)
            {
              state.rhaar[src][length + i] /= rms;
            }
          }
        }
      }

      // amplify by control data
      state.rhaar[src][0] *= state.R[octave][0];
      for ( int rhythm = 1, length = 1
          ; length <= RHYTHMSIZE >> 1
          ; rhythm += 1, length <<= 1
          )
      {
        for (int i = 0; i < length; ++i)
        {
          state.rhaar[src][length + i] *= state.R[octave][rhythm];
        }
      }

      // compute inverse Haar wavelet transform
      for (int length = 1; length <= RHYTHMSIZE >> 1; length <<= 1)
      {
        for (int i = 0; i < RHYTHMSIZE; ++i)
        {
            state.rhaar[dst][i] = state.rhaar[src][i];
        }
        for (int i = 0; i < length; ++i)
        {
            float s = state.rhaar[dst][         i];
            float d = state.rhaar[dst][length + i];
```

```
                float a = s + d;
                float b = s - d;
                state.rhaar[src][2 * i + 0] = a;
                state.rhaar[src][2 * i + 1] = b;
            }
          }

          // window
          for (int i = 0; i < RHYTHMSIZE; ++i)
          {
              state.rhaar[src][i] *= state.rwindow[i];
          }

          // overlap-add
          for (int i = 0; i < RHYTHMSIZE; ++i)
          {
              state.rbuffer[i][octave]
                = i + RHOP < RHYTHMSIZE
                ? state.rbuffer[i + RHOP][octave]
                : 0;
          }
          for (int i = 0; i < RHYTHMSIZE; ++i)
          {
              state.rbuffer[i][octave] += state.rhaar[src][i];
          }

        } // for octave

        // shunt
        for (int i = 0; i < RHYTHMSIZE - RHOP; ++i)
        {
          for (int octave = 0; octave < OCTAVES; ++octave)
          {
            state.ebuffer[i][octave] = state.ebuffer[i + RHOP][octave];
          }
        }
        for (int i = RHYTHMSIZE - RHOP; i < RHYTHMSIZE; ++i)
        {
          for (int octave = 0; octave < OCTAVES; ++octave)
          {
            state.ebuffer[i][octave] = 0;
          }
        }

        state.eindex -= RHOP;
        state.rindex = 0;
      } // if eindex == RHYTHMSIZE

      // copy
      for (int octave = 0; octave < OCTAVES; ++octave)
      {
        state.E[octave] = state.rbuffer[state.rindex][octave];
      }
      state.rindex++;

      // amplify octaves
      for (int channel = 0; channel < CHANNELS; ++channel)
```

```
    {
      state.haar[src][0][channel] *= state.E[0]; // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          state.haar[src][length + i][channel] *= state.E[octave];
        }
      }
    }

    // compute inverse Haar wavelet transform
    for (int length = 1; length <= OCTAVESIZE >> 1; length <<= 1)
    {
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          state.haar[dst][i][channel] = state.haar[src][i][channel];
        }
      }
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float s = state.haar[dst][          i][channel];
          float d = state.haar[dst][length + i][channel];
          float a = s + d;
          float b = s - d;
          state.haar[src][2 * i + 0][channel] = a;
          state.haar[src][2 * i + 1][channel] = b;
        }
      }
    }

    // window
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        state.haar[src][i][channel] *= state.owindow[i];
      }
    }

    // overlap-add
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        state.obuffer[i][channel]
          = i + OHOP < OCTAVESIZE
```

```
            ? state.obuffer[i + OHOP][channel]
            : 0;
        }
      }
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          state.obuffer[i][channel] += state.haar[src][i][channel];
        }
      }

      state.oavailable += OHOP;
      state.oindex = 0;
    } // if aindex == OHOP
  } // while oavailable <= 0

  for (int channel = 0; channel < CHANNELS; ++channel)
  {
    output[channel] = state.obuffer[state.oindex][channel];
  }
  state.oindex++;
  state.oavailable--;
}

static void audio(void *userdata, Uint8 *stream, int len)
{
  (void) userdata;
  float *b = (float *) stream;
  int m = len / sizeof(float) / CHANNELS;
  int k = 0;
  for (int i = 0; i < m; ++i)
  {
    float out[CHANNELS];
    audio1(out);
    for (int j = 0; j < CHANNELS; ++j)
    {
      b[k++] = out[j];
    }
  }
}

static bool interact(void)
{
  bool running = true;
  SDL_Event event;
  while (SDL_PollEvent(&event) != 0)
  {
    switch (event.type)
    {
      case SDL_QUIT:
      {
        running = false;
        break;
      }
    }
  }
```

```
  int x = -1, y = -1;
  Uint32 b = SDL_GetMouseState(&x, &y);
  state.target_octave = x / CELLSIZE + 1;
  state.target_rhythm = RHYTHMS - 1 - y / CELLSIZE;
  state.target_delta = 2 * (0.5 - ((y % CELLSIZE) + 0.5) / CELLSIZE);
  if ( (b & SDL_BUTTON(SDL_BUTTON_LEFT)) &&
       1 <= state.target_octave && state.target_octave < OCTAVES &&
       1 <= state.target_rhythm && state.target_rhythm < RHYTHMS &&
      -1 <= state.target_delta  && state.target_delta <= 1
     )
  {
    double r = state.R[state.target_octave][state.target_rhythm];
    r += state.target_delta / FPS;
    r = fmin(fmax(r, 0), 0.1);
    state.R[state.target_octave][state.target_rhythm] = r;
  }
  return running;
}

static void video1(void)
{
  glClearColor(1, 1, 1, 1);
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_QUADS);
  {
    for (int i = 1; i < OCTAVES; ++i)
    {
      for (int j = 1; j < RHYTHMS; ++j)
      {
        double radius = fmin(fmax(sqrt(state.R[i][j] / 0.1) / 2.0, 1.0 / ↙
            ↳ CELLSIZE), 0.5);
        const int sx[4] = { -1, -1, 1, 1 };
        const int sy[4] = { -1, 1, 1, -1 };
        for (int k = 0; k < 4; ++k)
        {
          glColor4f(0, 0, 0, 1);
          glVertex4f
            ( ((i - 0.5 + sx[k] * radius) / (OCTAVES - 1) - 0.5) * 2.0
            , ((j - 0.5 + sy[k] * radius) / (RHYTHMS - 1) - 0.5) * 2.0
            , 0
            , 1
            );
        }
      }
    }
  }
  glEnd();
  SDL_GL_SwapWindow(state.window);
}

void video(void)
{
  while (interact())
  {
    video1();
  }
}
```

```
void main1(void)
{
  if (interact())
  {
    video1();
  }
}

int main(int argc, char **argv)
{
  printf("disco/designer (Free Art License 1.3+) 2019 Claude Heiland-Allen\n");
  srand(time(0));
  memset(&state, 0, sizeof(state));
  // load initial state
  if (argc > 1)
  {
    if (! read_fingerprint(argv[1]))
    {
      printf("error: could not read fingerprint\n");
      return 1;
    }
    if (0)
      normalize_fingerprint();
  }
  state.normalize_carrier = 1;
  initialize_windows();
  // initialize SDL2
  SDL_Init(SDL_INIT_AUDIO | SDL_INIT_VIDEO);
  SDL_AudioSpec want, have;
  want.freq = 44100;
  want.format = AUDIO_F32;
  want.channels = CHANNELS;
  want.samples = 4096;
  want.callback = audio;
  SDL_AudioDeviceID dev = SDL_OpenAudioDevice(NULL, 0, &want, &have, SDL_AUDIO_ALLOW_ANY_CHANGE);
  if (have.freq > 192000 || have.format != AUDIO_F32 || have.channels != CHANNELS)
  {
    printf("want: %d %d %d %d\n", want.freq, want.format, want.channels, want.samples);
    printf("have: %d %d %d %d\n", have.freq, have.format, have.channels, have.samples);
    printf("error: bad audio parameters\n");
  }
  else
  {
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MAJOR_VERSION, 2);
    SDL_GL_SetAttribute(SDL_GL_CONTEXT_MINOR_VERSION, 1);
    state.window = SDL_CreateWindow("disco/designer", SDL_WINDOWPOS_CENTERED, SDL_WINDOWPOS_CENTERED, (OCTAVES - 1) * CELLSIZE, (RHYTHMS - 1) * CELLSIZE, SDL_WINDOW_OPENGL | SDL_WINDOW_SHOWN);
    if (! state.window)
    {
      printf("error: couldn't create SDL2 window\n");
      return 1;
```

```
    }
    state.context = SDL_GL_CreateContext(state.window);
    if (! state.context)
    {
      printf("error: couldn't create OpenGL context\n");
      return 1;
    }
    state.SR = have.freq;
    // start audio processing
    SDL_PauseAudioDevice(dev, 0);
#ifdef __EMSCRIPTEN__
    emscripten_set_main_loop(main1, 0, 1);
#else
    video();
#endif
  }
  SDL_DestroyWindow(state.window);
  state.window = NULL;
  SDL_Quit();
  return 0;
}
```

2 .gitignore

```
timbre-stamp
markov-analysis
markov-synthesis
rhythm-analysis
rhythm-synthesis
normalize
disco-designer
designer.html
designer.js
designer.js.gz
designer.wasm
designer.wasm.gz
*.dat
*.ogg
*.wav
```

3 index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <meta content="text/html; charset=utf-8" http-equiv="Content-Type">
    <title>disco/designer</title>
    <meta name="description" content="disco/designer" />
    <meta name="keywords" content="music noise" />
    <meta name="generator" content="emscripten" />
    <meta name="author" content="mathr" />
    <style>
      canvas
      {
        width: 704px;
```

```
          height: 832px;
          position: absolute;
          top:0;
          bottom: 0;
          left: 0;
          right: 0;
          border: 0;
          padding: 0;
          margin: auto;
        }
      </style>
    </head>
    <body>
      <canvas id="canvas" oncontextmenu="event.preventDefault()" tabindex="-1"></canvas>
      <script>
        // audio autoplay
        const audioContextList = [];
        (function() {
          self.AudioContext = new Proxy(self.AudioContext, {
            construct(target, args) {
              const result = new target(...args);
              audioContextList.push(result);
              return result;
            }
          });
        })();
        function resumeAudio() {
          audioContextList.forEach(ctx => {
            if (ctx.state !== "running") { ctx.resume(); }
          });
        }
        [ "click", "contextmenu", "auxclick", "dblclick"
        , "mousedown", "mouseup", "pointerup", "touchend"
        , "keydown", "keyup"
        ].forEach(name => document.addEventListener(name, resumeAudio));
        // emscripten
        var Module
              = { preRun: []
                , postRun: []
                , print: function(e) {
                    1<arguments.length&&(e=Array.prototype.slice.call(arguments).join(" "));
                    console.log(e);
                  }
                , printErr: function(e) {
                    1<arguments.length&&(e=Array.prototype.slice.call(arguments).join(" "));
                    console.error(e)
                  }
                , canvas: function() {
                    var e = document.getElementById("canvas");
                    e.addEventListener("webglcontextlost", function(e) {
                      alert("WebGL context lost. You will need to reload the page.");
                      e.preventDefault();
                    });
```

```
                    return e;
                }()
            }
          ;
    </script>
    <script async src="designer.js"></script>
  </body>
</html>
```

4 LICENSE.md

```
# Free Art License 1.3 (FAL 1.3)

## Preamble

The Free Art License grants the right to freely copy, distribute, and
transform creative works without infringing the author's rights.

The Free Art License recognizes and protects these rights. Their
implementation has been reformulated in order to allow everyone to use
creations of the human mind in a creative manner, regardless of their
types and ways of expression.

While the public's access to creations of the human mind usually is
restricted by the implementation of copyright law, it is favoured by the
Free Art License. This license intends to allow the use of a work's
resources; to establish new conditions for creating in order to increase
creation opportunities. The Free Art License grants the right to use a
work, and acknowledges the right holder's and the user's rights and
responsibility.

The invention and development of digital technologies, Internet and Free
Software have changed creation methods: creations of the human mind can
obviously be distributed, exchanged, and transformed. They allow to
produce common works to which everyone can contribute to the benefit of
all.

The main rationale for this Free Art License is to promote and protect
these creations of the human mind according to the principles of
copyleft: freedom to use, copy, distribute, transform, and prohibition
of exclusive appropriation.

## Definitions

*work* either means the initial work, the subsequent works or the
common work as defined hereafter:

*common work* means a work composed of the initial work and all
subsequent contributions to it (originals and copies). The initial
author is the one who, by choosing this license, defines the conditions
under which contributions are made.

*Initial work* means the work created by the initiator of the common
work (as defined above), the copies of which can be modified by whoever
wants to

*Subsequent works* means the contributions made by authors who
```

```
participate in the evolution of the common work by exercising the rights
to reproduce, distribute, and modify that are granted by the license.

*Originals* (sources or resources of the work) means all copies of
either the initial work or any subsequent work mentioning a date and
used by their author(s) as references for any subsequent updates,
interpretations, copies or reproductions.

*Copy* means any reproduction of an original as defined by this
license.

## 1. OBJECT

The aim of this license is to define the conditions under which one can
use this work freely.

## 2. SCOPE

This work is subject to copyright law. Through this license its author
specifies the extent to which you can copy, distribute, and modify it.

### 2.1 FREEDOM TO COPY (OR TO MAKE REPRODUCTIONS)

You have the right to copy this work for yourself, your friends or any
other person, whatever the technique used.

### 2.2 FREEDOM TO DISTRIBUTE, TO PERFORM IN PUBLIC

You have the right to distribute copies of this work; whether modified
or not, whatever the medium and the place, with or without any charge,
provided that you:

- attach this license without any modification to the copies of this work
  or indicate precisely where the license can be found,
- specify to the recipient the names of the author(s) of the originals,
  including yours if you have modified the work,
- specify to the recipient where to access the originals (either initial
  or subsequent).

The authors of the originals may, if they wish to, give you the right
to distribute the originals under the same conditions as the copies.

### 2.3 FREEDOM TO MODIFY

You have the right to modify copies of the originals (whether initial
or subsequent) provided you comply with the following conditions:

- all conditions in article 2.2 above, if you distribute modified
  copies;
- indicate that the work has been modified and, if it is possible, what
  kind of modifications have been made;
- distribute the subsequent work under the same license or any compatible
  license.

The author(s) of the original work may give you the right to modify it
under the same conditions as the copies.
```

```
## 3. RELATED RIGHTS

Activities giving rise to author's rights and related rights shall not
challenge the rights granted by this license.
For example, this is the reason why performances must be subject to the
same license or a compatible license. Similarly, integrating the work in
a database, a compilation or an anthology shall not prevent anyone from
using the work under the same conditions as those defined in this
license.

## 4. INCORPORATION OF THE WORK

Incorporating this work into a larger work that is not subject to the
Free Art License shall not challenge the rights granted by this
license.
If the work can no longer be accessed apart from the larger work in
which it is incorporated, then incorporation shall only be allowed under
the condition that the larger work is subject either to the Free Art
License or a compatible license.

## 5. COMPATIBILITY

A license is compatible with the Free Art License provided:

- it gives the right to copy, distribute, and modify copies of the work
  including for commercial purposes and without any other restrictions
  than those required by the respect of the other compatibility criteria;
- it ensures proper attribution of the work to its authors and access to
  previous versions of the work when possible;
- it recognizes the Free Art License as compatible (reciprocity);
- it requires that changes made to the work be subject to the same
  license or to a license which also meets these compatibility criteria.

## 6. YOUR INTELLECTUAL RIGHTS

This license does not aim at denying your author's rights in your
contribution or any related right. By choosing to contribute to the
development of this common work, you only agree to grant others the same
rights with regard to your contribution as those you were granted by
this license. Conferring these rights does not mean you have to give up
your intellectual rights.

## 7. YOUR RESPONSIBILITIES

The freedom to use the work as defined by the Free Art License (right
to copy, distribute, modify) implies that everyone is responsible for
their own actions.

## 8. DURATION OF THE LICENSE

This license takes effect as of your acceptance of its terms. The act
of copying, distributing, or modifying the work constitutes a tacit
agreement. This license will remain in effect for as long as the
copyright which is attached to the work. If you do not respect the terms
of this license, you automatically lose the rights that it confers.
If the legal status or legislation to which you are subject makes it
impossible for you to respect the terms of this license, you may not
```

```
make use of the rights which it confers.

## 9. VARIOUS VERSIONS OF THE LICENSE

This license may undergo periodic modifications to incorporate
improvements by its authors (instigators of the Copyleft Attitude
movement) by way of new, numbered versions.
You will always have the choice of accepting the terms contained in the
version under which the copy of the work was distributed to you, or
alternatively, to use the provisions of one of the subsequent versions.

## 10. SUB-LICENSING

Sub-licenses are not authorized by this license. Any person wishing to
make use of the rights that it confers will be directly bound to the
authors of the common work.

## 11. LEGAL FRAMEWORK

This license is written with respect to both French law and the Berne
Convention for the Protection of Literary and Artistic Works.

## USER GUIDE

### How to use the Free Art License?

To benefit from the Free Art License, you only need to mention the
following elements on your work:

    [Name of the author, title, date of the work. When applicable,
    names of authors of the common work and, if possible, where to
    find the originals].

    Copyleft: This is a free work, you can copy, distribute, and
    modify it under the terms of the Free Art License
    <http://artlibre.org/licence/lal/en/>

### Why to use the Free Art License?

1. To give the greatest number of people access to your work.
2. To allow it to be distributed freely.
3. To allow it to evolve by allowing its copy, distribution, and
   transformation by others.
4. So that you benefit from the resources of a work when it is under the
   Free Art License: to be able to copy, distribute or transform it
   freely.
5. But also, because the Free Art License offers a legal framework to
   disallow any misappropriation. It is forbidden to take hold of your work
   and bypass the creative process for one's exclusive possession.

### When to use the Free Art License?

Any time you want to benefit and make others benefit from the right to
copy, distribute and transform creative works without any exclusive
appropriation, you should use the Free Art License. You can for example
use it for scientific, artistic or educational projects.
```

```
### What kinds of works can be subject to the Free Art License?

The Free Art License can be applied to digital as well as physical
works.
You can choose to apply the Free Art License on any text, picture,
sound, gesture, or whatever sort of stuff on which you have sufficient
author's rights.

### Historical background of this license:

It is the result of observing, using and creating digital technologies,
free software, the Internet and art. It arose from the Copyleft
Attitude meetings which took place in Paris in 2000. For the first
time, these meetings brought together members of the Free Software
community, artists, and members of the art world. The goal was to adapt
the principles of Copyleft and free software to all sorts of creations.

------------------------------------------------------------------------

<http://www.artlibre.org>
Copyleft Attitude, 2007.

You can make reproductions and distribute this license verbatim
(without any changes).

Translation: Jonathan Clarke, Benjamin Jean, Griselda Jung, Fanny Mourguet,
Antoine Pitrou. Thanks to <http://framalang.org>
```

5 Makefile

```
EXES := timbre-stamp rhythm-analysis rhythm-synthesis markov-analysis markov-synthesis normalize

all: $(EXES)

clean:
	-rm $(EXES)

%: %.c
	gcc -O3 -march=native $< -o $@ -Wall -Wextra -pedantic -g `pkg-config --cflags --libs gsl` -lsndfile -lm

disco-designer: disco-designer.c
	gcc -O3 -march=native $< -o $@ -Wall -Wextra -pedantic -g `sdl2-config --cflags --libs` -lGL -lm

designer.html: disco-designer.c
	emcc -O3 -Os -o designer.html disco-designer.c --closure 1 -s USE_SDL=2 -s LEGACY_GL_EMULATION=1 -s GL_FFP_ONLY=1
	gzip -9 -k -f designer.js
	gzip -9 -k -f designer.wasm
```

6 markov-analysis.c

```
// note: uses stack allocated arrays, may need `ulimit -s unlimited`

#include <math.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

#include <gsl/gsl_linalg.h>
#include <gsl/gsl_rng.h>
#include <gsl/gsl_randist.h>

#define OCTAVES 11
#define OCTAVESIZE (1 << (OCTAVES - 1))

#define OVERLAP 16

#define OHOP (OCTAVESIZE / OVERLAP)

#define SOMSIZE 16

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
      ( stderr
      , "usage:\n"
        "  %s in.wav out.dat\n"
      , argv[0]
      );
    return 1;
  }
  SF_INFO info;
  memset(&info, 0, sizeof(info));
  fprintf(stderr, "reading %s\n", argv[1]);
  SNDFILE *ifile = sf_open(argv[1], SFM_READ, &info);
  if (! ifile) abort();
  const int CHANNELS = info.channels;

  float window[OCTAVESIZE];
  memset(window, 0, sizeof(window));
  for (int i = 0; i < OCTAVESIZE; ++i)
  {
    window[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / OCTAVESIZE);
  }

  int Ecount = 0;
  double Emean[OCTAVES];
  memset(Emean, 0, sizeof(Emean));
  double Ecov[OCTAVES][OCTAVES];
  memset(Ecov, 0, sizeof(Ecov));

  double imap[SOMSIZE][SOMSIZE][OCTAVES];
  memset(imap, 0, sizeof(imap));

  gsl_rng_env_setup();
  gsl_rng *PRNG = gsl_rng_alloc(gsl_rng_default);
  int step = 0;
```

```
  int lasti = 0, lastj = 0;
  double chain[SOMSIZE][SOMSIZE][SOMSIZE][SOMSIZE];
  memset(chain, 0, sizeof(chain));
  double total[SOMSIZE][SOMSIZE];
  memset(total, 0, sizeof(total));

  for (int pass = 0; pass < 4; ++pass)
  {
    fprintf(stderr, "pass %d/4\n", pass + 1);
    sf_seek(ifile, 0, SEEK_SET);
    float ibuffer[OCTAVESIZE][CHANNELS];
    memset(ibuffer, 0, sizeof(ibuffer));

    if (pass == 2)
    {
      // Cholesky decomposition of covariance matrix
      for (int i = 0; i < OCTAVES; ++i)
      {
        for (int j = 0; j < OCTAVES; ++j)
        {
          Ecov[i][j] /= Ecount;
          fprintf(stderr, "%g ", Ecov[i][j]);
        }
        fprintf(stderr, "\n");
      }
      gsl_matrix_view cov = gsl_matrix_view_array(&Ecov[0][0], OCTAVES, OCTAVES);
      gsl_linalg_cholesky_decomp1(&cov.matrix);
      // randomize SOM
      for (int i = 0; i < SOMSIZE; ++i)
      {
        for (int j = 0; j < SOMSIZE; ++j)
        {
          // generate independent Gaussian vectors
          double d[OCTAVES];
          for (int k = 0; k < OCTAVES; ++k)
          {
            d[k] = gsl_ran_gaussian(PRNG, 1);
          }
          gsl_vector_view x = gsl_vector_view_array(d, OCTAVES);
          // correlate them to the correct covariance matrix
          gsl_blas_dtrmv(CblasLower, CblasNoTrans, CblasNonUnit, &cov.matrix, &x.vector);
          for (int k = 0; k < OCTAVES; ++k)
          {
            imap[i][j][k] = Emean[k] / Ecount + d[k];
          }
        }
      }
      step = 0;
    }

    int reading = OVERLAP;
    while (reading > 0)
    {

      // read input
```

```
for (int i = 0; i < OCTAVESIZE - OHOP; ++i)
{
  for (int channel = 0; channel < CHANNELS; ++channel)
  {
    ibuffer[i][channel] = ibuffer[i + OHOP][channel];
  }
}
for (int i = OCTAVESIZE - OHOP; i < OCTAVESIZE; ++i)
{
  for (int channel = 0; channel < CHANNELS; ++channel)
  {
    ibuffer[i][channel] = 0;
  }
}
if (reading == OVERLAP)
{
  int r = sf_readf_float(ifile, &ibuffer[OCTAVESIZE - OHOP][0], OHOP);
  if (r != OHOP)
  {
    reading--;
  }
}
else
{
  reading--;
}

// window
float haar[2][OCTAVESIZE][CHANNELS];
memset(haar, 0, sizeof(haar));
int src = 0;
int dst = 1;
for (int i = 0; i < OCTAVESIZE; ++i)
{
  for (int channel = 0; channel < CHANNELS; ++channel)
  {
    haar[src][i][channel] = ibuffer[i][channel] * window[i];
  }
}

// compute Haar wavelet transform
for (int length = OCTAVESIZE >> 1; length > 0; length >>= 1)
{
  for (int i = 0; i < length; ++i)
  {
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      float a = haar[src][2 * i + 0][channel];
      float b = haar[src][2 * i + 1][channel];
      float s = (a + b) / 2;
      float d = (a - b) / 2;
      haar[dst][         i][channel] = s;
      haar[dst][length + i][channel] = d;
    }
  }
  for (int i = 0; i < OCTAVESIZE; ++i)
  {
```

```
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = haar[dst][i][channel];
        }
      }
    }

    // compute energy per octave
    double E[OCTAVES];
    memset(E, 0, sizeof(E));
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      E[0] += fabsf(haar[src][0][channel]); // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      double rms = 0;
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          double a = haar[src][length + i][channel];
          rms += a * a;
        }
      }
      rms /= length * CHANNELS;
      rms = sqrt(fmax(rms, 0));
      E[octave] = rms;
    }

    if (pass == 0)
    {
      // compute mean
      for (int octave = 0; octave < OCTAVES; ++octave)
      {
        Emean[octave] += E[octave];
      }
      Ecount += 1;
    }
    else if (pass == 1)
    {
      // compute covariance
      for (int octave1 = 0; octave1 < OCTAVES; ++octave1)
      {
        for (int octave2 = 0; octave2 < OCTAVES; ++octave2)
        {
          Ecov[octave1][octave2]
            += (E[octave1] - Emean[octave1] / Ecount)
             * (E[octave2] - Emean[octave2] / Ecount);
        }
      }
    }
    else if (pass == 2)
    {
```

```
  // compute SOM

  // find best match
  int mini = 0;
  int minj = 0;
  double mins = 1.0 / 0.0;
  for (int i = 0; i < SOMSIZE; ++i)
  {
    for (int j = 0; j < SOMSIZE; ++j)
    {
      double s = 0;
      for (int k = 0; k < OCTAVES; ++k)
      {
        double d = E[k] - imap[i][j][k];
        s += d * d;
      }
      if (s < mins)
      {
        mins = s;
        mini = i;
        minj = j;
      }
    }
  }

  // update weights
  for (int i = 0; i < SOMSIZE; ++i)
  {
    for (int j = 0; j < SOMSIZE; ++j)
    {
      double di = (i - mini) * 1.0 / SOMSIZE;
      double dj = (j - minj) * 1.0 / SOMSIZE;
      // FIXME magic numbers to control learning rate
      double kernel = 0.1 * exp(- (di * di + dj * dj) * (step + 1.0) / 100000);
      for (int k = 0; k < OCTAVES; ++k)
      {
        imap[i][j][k] += kernel * (E[k] - imap[i][j][k]);
      }
    }
  }
  step++;
}
else if (pass == 3)
{
  // compute Markov chain

  // find best match
  int mini = 0;
  int minj = 0;
  double mins = 1.0 / 0.0;
  for (int i = 0; i < SOMSIZE; ++i)
  {
    for (int j = 0; j < SOMSIZE; ++j)
    {
      double s = 0;
      for (int k = 0; k < OCTAVES; ++k)
```

```
            {
              double d = E[k] - imap[i][j][k];
              s += d * d;
            }
            if (s < mins)
            {
              mins = s;
              mini = i;
              minj = j;
            }
          }
        }
        chain[lasti][lastj][mini][minj] += 1;
        total[lasti][lastj] += 1;
        lasti = mini;
        lastj = minj;
      }
    } // while reading
  } // for pass
  sf_close(ifile);

  // output
  fprintf(stderr, "writing %s\n", argv[2]);
  FILE *ofile = fopen(argv[2], "wb");
  for (int i = 0; i < SOMSIZE; ++i)
  for (int j = 0; j < SOMSIZE; ++j)
  for (int k = 0; k < OCTAVES; ++k)
    fprintf(ofile, "%.18e\n", imap[i][j][k]);
  for (int i = 0; i < SOMSIZE; ++i)
  for (int j = 0; j < SOMSIZE; ++j)
  for (int ii = 0; ii < SOMSIZE; ++ii)
  for (int jj = 0; jj < SOMSIZE; ++jj)
    fprintf(ofile, "%.18e\n", chain[i][j][ii][jj] / total[i][j]);
  fclose(ofile);
  fprintf(stderr, "total frames %d\n", Ecount);

  return 0;
}
```

7 markov-synthesis.c

```
// note: uses stack allocated arrays, may need `ulimit -s unlimited`

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

// FIXME these defines must match markov-analysis.c

#define OCTAVES 11
#define OCTAVESIZE (1 << (OCTAVES - 1))

#define OVERLAP 16
```

```
#define OHOP (OCTAVESIZE / OVERLAP)

#define SOMSIZE 16

// FIXME should match analyzed source audio
#define SR 44100

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
      ( stderr
      , "usage:\n"
        "  %s in.dat out.wav\n"
      , argv[0]
      );
    return 1;
  }

  fprintf(stderr, "reading %s\n", argv[1]);
  FILE *dfile = fopen(argv[1], "rb");
  double imap[SOMSIZE][SOMSIZE][OCTAVES];
  int lasti = 0;
  int lastj = 0;
  int mins = 1.0 / 0.0;
  for (int i = 0; i < SOMSIZE; ++i)
  for (int j = 0; j < SOMSIZE; ++j)
  {
    double s = 0;
    for (int k = 0; k < OCTAVES; ++k)
    {
      double r = 0;
      fscanf(dfile, "%lf ", &r);
      imap[i][j][k] = r;
      s += r * r;
    }
    if (s < mins)
    {
      mins = s;
      lasti = i;
      lastj = j;
    }
  }
  double chain[SOMSIZE][SOMSIZE][SOMSIZE][SOMSIZE];
  for (int i = 0; i < SOMSIZE; ++i)
  for (int j = 0; j < SOMSIZE; ++j)
  for (int ii = 0; ii < SOMSIZE; ++ii)
  for (int jj = 0; jj < SOMSIZE; ++jj)
  {
    double r = 0;
    fscanf(dfile, "%lf ", &r);
    chain[i][j][ii][jj] = r;
  }
  fclose(dfile);

  const int CHANNELS = 2;
```

```c
  fprintf(stderr, "writing %s\n", argv[2]);
  SF_INFO outfo =
    { 0
    , SR
    , CHANNELS
    , SF_FORMAT_WAV | SF_FORMAT_FLOAT
    , 0
    , 0
    };
  SNDFILE *ofile = sf_open(argv[2], SFM_WRITE, &outfo);
  if (! ofile) abort();

  float nbuffer[OCTAVESIZE][CHANNELS];
  memset(nbuffer, 0, sizeof(nbuffer));

  float obuffer[OCTAVESIZE][CHANNELS];
  memset(obuffer, 0, sizeof(obuffer));

  float owindow[OCTAVESIZE];
  memset(owindow, 0, sizeof(owindow));
  for (int i = 0; i < OCTAVESIZE; ++i)
  {
    owindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / OCTAVESIZE);
  }

  float abuffer[OHOP][CHANNELS];
  memset(abuffer, 0, sizeof(abuffer));

  // FIXME 10mins should be enough for testing?
  for (int duration = 0; duration < 10 * 60 * SR; duration += OHOP)
  {
    // generate noise
    for (int i = 0; i < OCTAVESIZE - OHOP; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        nbuffer[i][channel] = nbuffer[i + OHOP][channel];
      }
    }
    for (int i = OCTAVESIZE - OHOP; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        nbuffer[i][channel] = rand() / (double) RAND_MAX - 0.5;
      }
    }

    // window
    float haar[2][OCTAVESIZE][CHANNELS];
    memset(haar, 0, sizeof(haar));
    int src = 0;
    int dst = 1;
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] = nbuffer[i][channel] * owindow[i];
```

```
    }
  }

  // compute Haar wavelet transform
  for (int length = OCTAVESIZE >> 1; length > 0; length >>= 1)
  {
    for (int i = 0; i < length; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        float a = haar[src][2 * i + 0][channel];
        float b = haar[src][2 * i + 1][channel];
        float s = (a + b) / 2;
        float d = (a - b) / 2;
        haar[dst][         i][channel] = s;
        haar[dst][length + i][channel] = d;
      }
    }
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] = haar[dst][i][channel];
      }
    }
  }

  // Markov chain
  double E[OCTAVES];
  int nexti = lasti;
  int nextj = lastj;
  double t = rand() / (double) RAND_MAX;
  for (int i = 0; i < SOMSIZE; ++i)
  {
    for (int j = 0; j < SOMSIZE; ++j)
    {
      t -= chain[lasti][lastj][i][j];
      if (t <= 0)
      {
        nexti = i;
        nextj = j;
        break;
      }
    }
    if (t <= 0)
      break;
  }
  for (int k = 0; k < OCTAVES; ++k)
  {
    E[k] = imap[nexti][nextj][k];
  }
  lasti = nexti;
  lastj = nextj;

  // amplify octaves
  for (int channel = 0; channel < CHANNELS; ++channel)
  {
```

```
      haar[src][0][channel] *= E[0]; // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          haar[src][length + i][channel] *= E[octave];
        }
      }
    }

    // compute inverse Haar wavelet transform
    for (int length = 1; length <= OCTAVESIZE >> 1; length <<= 1)
    {
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[dst][i][channel] = haar[src][i][channel];
        }
      }
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float s = haar[dst][         i][channel];
          float d = haar[dst][length + i][channel];
          float a = s + d;
          float b = s - d;
          haar[src][2 * i + 0][channel] = a;
          haar[src][2 * i + 1][channel] = b;
        }
      }
    }

    // window
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] *= owindow[i];
      }
    }

    // overlap-add
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        obuffer[i][channel] = i + OHOP < OCTAVESIZE ? obuffer[i + OHOP][channel]
            : 0;
      }
```

```
    }
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        obuffer[i][channel] += haar[src][i][channel];
      }
    }

    // output
    sf_writef_float(ofile, &obuffer[0][0], OHOP);
  } // for duration

  sf_close(ofile);
  return 0;
}
```

8 normalize.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
      ( stderr
      , "usage:\n"
        "  %s in.wav out.wav\n"
      , argv[0]
      );
    return 1;
  }
  SF_INFO info;
  memset(&info, 0, sizeof(info));
  fprintf(stderr, "reading %s\n", argv[1]);
  SNDFILE *ifile = sf_open(argv[1], SFM_READ, &info);
  if (! ifile)
  {
    return 1;
  }
  float *data = malloc(info.frames * info.channels * sizeof(*data));
  sf_readf_float(ifile, data, info.frames);
  sf_close(ifile);
  double peak = 0.0;
  for (size_t i = 0; i < (size_t) info.frames * info.channels; ++i)
  {
    peak = fmax(peak, data[i]);
  }
  fprintf(stderr, "peak %g\n", peak);
  float gain = 1 / peak;
  for (size_t i = 0; i < (size_t) info.frames * info.channels; ++i)
```

```
  {
    data[i] *= gain;
  }
  SF_INFO outfo = { 0, info.samplerate, info.channels, SF_FORMAT_WAV | ↙
      ↳ SF_FORMAT_FLOAT, 0, 0 };
  SNDFILE *ofile = sf_open(argv[2], SFM_WRITE, &outfo);
  sf_writef_float(ofile, data, info.frames);
  sf_close(ofile);
  return 0;
}
```

9 README.md

```
# disco

Audio fingerprint discrimination and resynthesis via Haar wavelets.

- <https://mathr.co.uk/disco>
- <https://code.mathr.co.uk/disco>
- <https://mathr.co.uk/disco/designer>

## dependencies

gcc, make, libsndfile, libgsl

for disco/designer native version: libsdl2, opengl

for disco/designer web version: emscripten

## build

    make                    # native versions
    make disco-designer     # native version
    make designer.html      # web version

## examples

Timbre stamp white noise with energy per octave of a control input:

    ./timbre-stamp input.wav output.wav

Compute energy per octave (audio) per octave (rhythm) fingerprint:

    ./rhythm-analysis input.wav output.dat

Resynthesize from fingerprint by stamping on white noise:

    ./rhythm-synthesis input.dat output.wav

Compute energy per octave (audio) Markov chain via self-organizing map:

    ./markov-analysis input.wav output.dat

Resynthesize from Markov chain by stamping on white noise:

    ./markov-synthesis input.dat output.wav
```

```
Normalize an audio file to peak value 1.0:

    ./normalize input.wav output.wav

Run disco/designer interactive demo native version:

    ./disco-designer

Run disco/designer interactive demo web version:

    python -m SimpleHTTPServer 8080 &    # wasm needs http(s) server
    sensible-browser http://localhost:8080/index.html

## bugs

Uses large stack-allocated arrays: if it crashes, try running this first:

    ulimit -s unlimited

## legal

Copyright (C) 2019 Claude Heiland-Allen <mailto:claude@mathr.co.uk>

Copyleft: This is a free work, you can copy, distribute, and
modify it under the terms of the Free Art License
<http://artlibre.org/licence/lal/en/>
```

10 rhythm-analysis.c

```c
// note: uses stack allocated arrays, may need `ulimit -s unlimited`

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

#define OCTAVES 11
#define OCTAVESIZE (1 << (OCTAVES - 1))

#define RHYTHMS 13
#define RHYTHMSIZE (1 << (RHYTHMS - 1))

#define OVERLAP 4

#define OHOP (OCTAVESIZE / OVERLAP)
#define RHOP (RHYTHMSIZE / OVERLAP)

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
      ( stderr
      , "usage:\n"
        "  %s in.wav out.dat\n"
```

```
        , argv[0]
        );
      return 1;
    }
    SF_INFO info;
    memset(&info, 0, sizeof(info));
    fprintf(stderr, "reading %s\n", argv[1]);
    SNDFILE *ifile = sf_open(argv[1], SFM_READ, &info);
    if (! ifile) abort();
    const int CHANNELS = info.channels;

    float ibuffer[OCTAVESIZE][CHANNELS];
    memset(ibuffer, 0, sizeof(ibuffer));

    float window[OCTAVESIZE];
    memset(window, 0, sizeof(window));
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      window[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / OCTAVESIZE);
    }

    float rwindow[RHYTHMSIZE];
    memset(rwindow, 0, sizeof(rwindow));
    for (int i = 0; i < RHYTHMSIZE; ++i)
    {
#if 0
      // FIXME needs two passes (for DC removal)
      rwindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / RHYTHMSIZE);
#else
      // FIXME use one pass without windowing for now
      rwindow[i] = 1;
#endif
    }

    int eindex = 0;
    float ebuffer[RHYTHMSIZE][OCTAVES];
    memset(ebuffer, 0, sizeof(ebuffer));

    float R[OCTAVES][RHYTHMS];
    memset(R, 0, sizeof(R));
    int R_count = 0;

    int reading = OVERLAP;
    while (reading > 0)
    {

      // read input
      for (int i = 0; i < OCTAVESIZE - OHOP; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          ibuffer[i][channel] = ibuffer[i + OHOP][channel];
        }
      }
      for (int i = OCTAVESIZE - OHOP; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
```

```
      {
        ibuffer[i][channel] = 0;
      }
    }
    if (reading == OVERLAP)
    {
      int r = sf_readf_float(ifile, &ibuffer[OCTAVESIZE - OHOP][0], OHOP);
      if (r != OHOP)
      {
        reading--;
      }
    }
    else
    {
      reading--;
    }

    // window
    float haar[2][OCTAVESIZE][CHANNELS];
    memset(haar, 0, sizeof(haar));
    int src = 0;
    int dst = 1;
    for (int i = 0; i < OCTAVESIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] = ibuffer[i][channel] * window[i];
      }
    }

    // compute Haar wavelet transform
    for (int length = OCTAVESIZE >> 1; length > 0; length >>= 1)
    {
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float a = haar[src][2 * i + 0][channel];
          float b = haar[src][2 * i + 1][channel];
          float s = (a + b) / 2;
          float d = (a - b) / 2;
          haar[dst][         i][channel] = s;
          haar[dst][length + i][channel] = d;
        }
      }
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = haar[dst][i][channel];
        }
      }
    }

    // compute energy per octave
    double E[OCTAVES];
    memset(E, 0, sizeof(E));
```

```
for (int channel = 0; channel < CHANNELS; ++channel)
{
  E[0] += fabsf(haar[src][0][channel]); // DC
}
for ( int octave = 1, length = 1
    ; length <= OCTAVESIZE >> 1
    ; octave += 1, length <<= 1
    )
{
  double rms = 0;
  for (int channel = 0; channel < CHANNELS; ++channel)
  {
    for (int i = 0; i < length; ++i)
    {
      double a = haar[src][length + i][channel];
      rms += a * a;
    }
  }
  rms /= length * CHANNELS;
  rms = sqrt(fmax(rms, 0));
  E[octave] = rms;
}

// copy to ebuffer
for (int octave = 0; octave < OCTAVES; ++octave)
{
  ebuffer[eindex][octave] = E[octave];
}
eindex++;

if (eindex == RHYTHMSIZE)
{

  for (int octave = 0; octave < OCTAVES; ++octave)
  {

    float rhaar[2][RHYTHMSIZE][CHANNELS];
    memset(rhaar, 0, sizeof(rhaar));

    // window
    for (int i = 0; i < RHYTHMSIZE; ++i)
    {
      rhaar[src][i][0] = ebuffer[i][octave] * rwindow[i];
    }

    // compute Haar wavelet transform
    for (int length = RHYTHMSIZE >> 1; length > 0; length >>= 1)
    {
      for (int i = 0; i < length; ++i)
      {
          float a = rhaar[src][2 * i + 0][0];
          float b = rhaar[src][2 * i + 1][0];
          float s = (a + b) / 2;
          float d = (a - b) / 2;
          rhaar[dst][         i][0] = s;
          rhaar[dst][length + i][0] = d;
      }
```

```
          for (int i = 0; i < RHYTHMSIZE; ++i)
          {
            rhaar[src][i][0] = rhaar[dst][i][0];
          }
        }

        // compute energy per octave
        R[octave][0] += fabsf(rhaar[src][0][0]); // DC
        for ( int rhythm = 1, length = 1
            ; length <= RHYTHMSIZE >> 1
            ; rhythm += 1, length <<= 1
            )
        {
          double rms = 0;
          for (int i = 0; i < length; ++i)
          {
            double a = rhaar[src][length + i][0];
            rms += a * a;
          }
          rms /= length;
          rms = sqrt(fmax(rms, 0));
          R[octave][rhythm] += rms;
        }

      } // for octave
      R_count++;
      eindex -= RHOP;

      // shunt
      for (int i = 0; i < RHYTHMSIZE - RHOP; ++i)
      {
        for (int octave = 0; octave < OCTAVES; ++octave)
        {
          ebuffer[i][octave] = ebuffer[i + RHOP][octave];
        }
      }
      for (int i = RHYTHMSIZE - RHOP; i < RHYTHMSIZE; ++i)
      {
        for (int octave = 0; octave < OCTAVES; ++octave)
        {
          ebuffer[i][octave] = 0;
        }
      }

    } // if (eindex == RHYTHMSIZE)
  } // while (reading > 0)

  sf_close(ifile);

  // output fingerprint
  fprintf(stderr, "writing %s\n", argv[2]);
  FILE *ofile = fopen(argv[2], "wb");
  for (int i = 0; i < OCTAVES; ++i)
  {
    for (int j = 0; j < RHYTHMS; ++j)
    {
      fprintf(ofile, "%.18e\n", R[i][j] / R_count);
```

```
    }
    fprintf(ofile, "\n");
  }
  fclose(ofile);
  fprintf(stderr, "total frames %d\n", R_count);

  return 0;
}
```

11 rhythm-synthesis.c

```
// note: uses stack allocated arrays, may need `ulimit -s unlimited`

#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

// FIXME these defines must match rhythm-analysis.c

#define OCTAVES 11
#define OCTAVESIZE (1 << (OCTAVES - 1))

#define RHYTHMS 13
#define RHYTHMSIZE (1 << (RHYTHMS - 1))

#define OVERLAP 4

#define OHOP (OCTAVESIZE / OVERLAP)
#define RHOP (RHYTHMSIZE / OVERLAP)

// FIXME should match analyzed source audio
#define SR 44100

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
      ( stderr
      , "usage:\n"
        "  %s in.dat out.wav\n"
      , argv[0]
      );
    return 1;
  }
  const int normalize_control = 0;
  const int normalize_carrier = 1;

  float R[OCTAVES][RHYTHMS];
  memset(R, 0, sizeof(R));
  fprintf(stderr, "reading %s\n", argv[1]);
  FILE *ifile = fopen(argv[1], "rb");
  double sr = 0;
  for (int i = 0; i < OCTAVES; ++i)
```

```
  {
    for (int j = 0; j < RHYTHMS; ++j)
    {
      double r = 0;
      fscanf(ifile, "%lf ", &r);
      R[i][j] = r;
      if (j > 0)
        sr += r * r;
    }
  }
  sr /= OCTAVES * (RHYTHMS - 1);
  sr = sqrt(sr);
  if (normalize_control)
  for (int i = 0; i < OCTAVES; ++i)
  {
    for (int j = 1; j < RHYTHMS; ++j)
    {
      R[i][j] /= sr;
    }
  }
  fclose(ifile);

  fprintf(stderr, "writing %s\n", argv[2]);
  const int CHANNELS = 2;
  SF_INFO outfo =
    { 0
    , SR
    , CHANNELS
    , SF_FORMAT_WAV | SF_FORMAT_FLOAT
    , 0
    , 0
    };
  SNDFILE *ofile = sf_open(argv[2], SFM_WRITE, &outfo);
  if (! ofile) abort();

  float nbuffer[OCTAVESIZE][CHANNELS];
  memset(nbuffer, 0, sizeof(nbuffer));

  float obuffer[OCTAVESIZE][CHANNELS];
  memset(obuffer, 0, sizeof(obuffer));

  float owindow[OCTAVESIZE];
  memset(owindow, 0, sizeof(owindow));
  for (int i = 0; i < OCTAVESIZE; ++i)
  {
    owindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / OCTAVESIZE);
  }

  float rwindow[RHYTHMSIZE];
  memset(rwindow, 0, sizeof(rwindow));
  for (int i = 0; i < RHYTHMSIZE; ++i)
  {
    rwindow[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / RHYTHMSIZE);
  }

  int eindex = 0;
  float ebuffer[RHYTHMSIZE][OCTAVES];
```

```
  memset(ebuffer, 0, sizeof(ebuffer));

  int rindex = 0;
  float rbuffer[RHYTHMSIZE][OCTAVES];
  memset(rbuffer, 0, sizeof(rbuffer));

  int aindex = 0;
  float abuffer[OHOP][CHANNELS];
  memset(abuffer, 0, sizeof(abuffer));

  // FIXME 10mins should be enough for testing?
  for (int duration = 0; duration < 10 * 60 * SR; ++duration)
  {
    // generate noise
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      abuffer[aindex][channel] = rand() / (double) RAND_MAX - 0.5;
    }
    aindex++;
    if (aindex == OHOP)
    {
      aindex = 0;
      for (int i = 0; i < OCTAVESIZE - OHOP; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          nbuffer[i][channel] = nbuffer[i + OHOP][channel];
        }
      }
      for (int i = OCTAVESIZE - OHOP; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          nbuffer[i][channel] = abuffer[i - (OCTAVESIZE - OHOP)][channel];
        }
      }

      // window
      float haar[2][OCTAVESIZE][CHANNELS];
      memset(haar, 0, sizeof(haar));
      int src = 0;
      int dst = 1;
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = nbuffer[i][channel] * owindow[i];
        }
      }

      // compute Haar wavelet transform
      for (int length = OCTAVESIZE >> 1; length > 0; length >>= 1)
      {
        for (int i = 0; i < length; ++i)
        {
          for (int channel = 0; channel < CHANNELS; ++channel)
          {
```

```
          float a = haar[src][2 * i + 0][channel];
          float b = haar[src][2 * i + 1][channel];
          float s = (a + b) / 2;
          float d = (a - b) / 2;
          haar[dst][         i][channel] = s;
          haar[dst][length + i][channel] = d;
        }
      }
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = haar[dst][i][channel];
        }
      }
    }

    // compute energy per octave
    double E[OCTAVES];
    memset(E, 0, sizeof(E));
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      E[0] += fabsf(haar[src][0][channel]); // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      double rms = 0;
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          double a = haar[src][length + i][channel];
          rms += a * a;
        }
      }
      rms /= length * CHANNELS;
      rms = sqrt(fmax(rms, 0));
      E[octave] = rms;
    }

    // copy to ebuffer
    for (int octave = 0; octave < OCTAVES; ++octave)
    {
      ebuffer[eindex][octave] = E[octave];
    }
    eindex++;

    if (eindex == RHYTHMSIZE)
    {
      for (int octave = 0; octave < OCTAVES; ++octave)
      {

        float rhaar[2][RHYTHMSIZE];
        memset(rhaar, 0, sizeof(rhaar));
```

```
          // window
          for (int i = 0; i < RHYTHMSIZE; ++i)
          {
            rhaar[src][i] = ebuffer[i][octave] * 1;//rwindow[i];
          }

          // compute Haar wavelet transform
          for (int length = RHYTHMSIZE >> 1; length > 0; length >>= 1)
          {
            for (int i = 0; i < length; ++i)
            {
                float a = rhaar[src][2 * i + 0];
                float b = rhaar[src][2 * i + 1];
                float s = (a + b) / 2;
                float d = (a - b) / 2;
                rhaar[dst][         i] = s;
                rhaar[dst][length + i] = d;
            }
            for (int i = 0; i < RHYTHMSIZE; ++i)
            {
                rhaar[src][i] = rhaar[dst][i];
            }
          }

          // normalize carrier
          if (normalize_carrier)
          {
            rhaar[src][0] = 1;
            for ( int rhythm = 1, length = 1
                ; length <= RHYTHMSIZE >> 1
                ; rhythm += 1, length <<= 1
                )
            {
              double rms = 0;
              for (int i = 0; i < length; ++i)
              {
                double d = rhaar[src][length + i];
                rms += d * d;
              }
              rms /= length;
              rms = sqrt(rms);
              for (int i = 0; i < length; ++i)
              {
                rhaar[src][length + i] /= rms;
              }
            }
          }

          // amplify by control data
          rhaar[src][0] *= R[octave][0];
          for ( int rhythm = 1, length = 1
              ; length <= RHYTHMSIZE >> 1
              ; rhythm += 1, length <<= 1
              )
          {
            for (int i = 0; i < length; ++i)
```

```
        {
          rhaar[src][length + i] *= R[octave][rhythm];
        }
      }

      // compute inverse Haar wavelet transform
      for (int length = 1; length <= RHYTHMSIZE >> 1; length <<= 1)
      {
        for (int i = 0; i < RHYTHMSIZE; ++i)
        {
            rhaar[dst][i] = rhaar[src][i];
        }
        for (int i = 0; i < length; ++i)
        {
            float s = rhaar[dst][          i];
            float d = rhaar[dst][length + i];
            float a = s + d;
            float b = s - d;
            rhaar[src][2 * i + 0] = a;
            rhaar[src][2 * i + 1] = b;
        }
      }

      // window
      for (int i = 0; i < RHYTHMSIZE; ++i)
      {
          rhaar[src][i] *= rwindow[i];
      }

      // overlap-add
      for (int i = 0; i < RHYTHMSIZE; ++i)
      {
          rbuffer[i][octave]
            = i + RHOP < RHYTHMSIZE
            ? rbuffer[i + RHOP][octave]
            : 0;
      }
      for (int i = 0; i < RHYTHMSIZE; ++i)
      {
          rbuffer[i][octave] += rhaar[src][i];
      }

    } // for octave

    // shunt
    for (int i = 0; i < RHYTHMSIZE - RHOP; ++i)
    {
      for (int octave = 0; octave < OCTAVES; ++octave)
      {
        ebuffer[i][octave] = ebuffer[i + RHOP][octave];
      }
    }
    for (int i = RHYTHMSIZE - RHOP; i < RHYTHMSIZE; ++i)
    {
      for (int octave = 0; octave < OCTAVES; ++octave)
      {
        ebuffer[i][octave] = 0;
```

```
        }
      }

      eindex -= RHOP;
      rindex = 0;
    } // if eindex == RHYTHMSIZE

    // copy
    for (int octave = 0; octave < OCTAVES; ++octave)
    {
      E[octave] = rbuffer[rindex][octave];
    }
    rindex++;

    // amplify octaves
    for (int channel = 0; channel < CHANNELS; ++channel)
    {
      haar[src][0][channel] *= E[0]; // DC
    }
    for ( int octave = 1, length = 1
        ; length <= OCTAVESIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          haar[src][length + i][channel] *= E[octave];
        }
      }
    }

    // compute inverse Haar wavelet transform
    for (int length = 1; length <= OCTAVESIZE >> 1; length <<= 1)
    {
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[dst][i][channel] = haar[src][i][channel];
        }
      }
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float s = haar[dst][         i][channel];
          float d = haar[dst][length + i][channel];
          float a = s + d;
          float b = s - d;
          haar[src][2 * i + 0][channel] = a;
          haar[src][2 * i + 1][channel] = b;
        }
      }
    }
```

```
      // window
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] *= owindow[i];
        }
      }

      // overlap-add
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          obuffer[i][channel]
            = i + OHOP < OCTAVESIZE
            ? obuffer[i + OHOP][channel]
            : 0;
        }
      }
      for (int i = 0; i < OCTAVESIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          obuffer[i][channel] += haar[src][i][channel];
        }
      }

      // output
      sf_writef_float(ofile, &obuffer[0][0], OHOP);

    } // if aindex == OHOP
  } // for duration

  sf_close(ofile);
  return 0;
}
```

12 timbre-stamp.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <sndfile.h>

#define OCTAVES 11
#define BLOCKSIZE (1 << (OCTAVES - 1))
#define OVERLAP 4
#define HOP (BLOCKSIZE / OVERLAP)

int main(int argc, char **argv)
{
  if (argc != 3)
  {
    fprintf
```

```
      ( stderr
      , "usage:\n"
        "  %s in.wav out.wav\n"
      , argv[0]
      );
    return 1;
  }
  SF_INFO info;
  memset(&info, 0, sizeof(info));
  fprintf(stderr, "reading %s\n", argv[1]);
  SNDFILE *ifile = sf_open(argv[1], SFM_READ, &info);
  if (! ifile) abort();
  fprintf(stderr, "writing %s\n", argv[2]);
  SF_INFO outfo =
    { 0
    , info.samplerate
    , info.channels
    , SF_FORMAT_WAV | SF_FORMAT_FLOAT
    , 0
    , 0
    };
  SNDFILE *ofile = sf_open(argv[2], SFM_WRITE, &outfo);
  if (! ofile) abort();
  const int CHANNELS = info.channels;

  float ibuffer[BLOCKSIZE][CHANNELS];
  memset(ibuffer, 0, sizeof(ibuffer));
  float obuffer[BLOCKSIZE][CHANNELS];
  memset(obuffer, 0, sizeof(obuffer));
  float nbuffer[BLOCKSIZE][CHANNELS];
  memset(nbuffer, 0, sizeof(nbuffer));
  float window[BLOCKSIZE];
  memset(window, 0, sizeof(window));
  for (int i = 0; i < BLOCKSIZE; ++i)
  {
    window[i] = 0.5 - 0.5 * cos(i * 6.283185307179586 / BLOCKSIZE);
  }

  int reading = OVERLAP;
  while (reading > 0)
  {

    // read input
    for (int i = 0; i < BLOCKSIZE - HOP; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        ibuffer[i][channel] = ibuffer[i + HOP][channel];
        nbuffer[i][channel] = nbuffer[i + HOP][channel];
      }
    }
    for (int i = BLOCKSIZE - HOP; i < BLOCKSIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        // pre-fill with zero in case of end of input
        ibuffer[i][channel] = 0;
```

```
          nbuffer[i][channel] = rand() / (double) RAND_MAX - 0.5;
        }
      }
      if (reading == OVERLAP)
      {
        int r = sf_readf_float(ifile, &ibuffer[BLOCKSIZE - HOP][0], HOP);
        if (r != HOP)
        {
          reading--;
        }
      }
      else
      {
        reading--;
      }

      // window
      float haar[2][BLOCKSIZE][CHANNELS];
      memset(haar, 0, sizeof(haar));
      int src = 0;
      int dst = 1;
      for (int i = 0; i < BLOCKSIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = ibuffer[i][channel] * window[i];
        }
      }

      // compute Haar wavelet transform
      for (int length = BLOCKSIZE >> 1; length > 0; length >>= 1)
      {
        for (int i = 0; i < length; ++i)
        {
          for (int channel = 0; channel < CHANNELS; ++channel)
          {
            float a = haar[src][2 * i + 0][channel];
            float b = haar[src][2 * i + 1][channel];
            float s = (a + b) / 2;
            float d = (a - b) / 2;
            haar[dst][         i][channel] = s;
            haar[dst][length + i][channel] = d;
          }
        }
        for (int i = 0; i < BLOCKSIZE; ++i)
        {
          for (int channel = 0; channel < CHANNELS; ++channel)
          {
            haar[src][i][channel] = haar[dst][i][channel];
          }
        }
      }

      // compute energy per octave
      double E[OCTAVES];
      memset(E, 0, sizeof(E));
      for (int channel = 0; channel < CHANNELS; ++channel)
```

```
    {
      E[0] += fabsf(haar[src][0][channel]); // DC
    }
    for ( int octave = 1, length = 1
        ; length <= BLOCKSIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      double rms = 0;
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          double a = haar[src][length + i][channel];
          rms += a * a;
        }
      }
      rms /= length * CHANNELS;
      rms = sqrt(fmax(rms, 0));
      E[octave] = rms;
    }

    // window
    for (int i = 0; i < BLOCKSIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] = nbuffer[i][channel] * window[i];
      }
    }

    // compute Haar wavelet transform
    for (int length = BLOCKSIZE >> 1; length > 0; length >>= 1)
    {
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float a = haar[src][2 * i + 0][channel];
          float b = haar[src][2 * i + 1][channel];
          float s = (a + b) / 2;
          float d = (a - b) / 2;
          haar[dst][         i][channel] = s;
          haar[dst][length + i][channel] = d;
        }
      }
      for (int i = 0; i < BLOCKSIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[src][i][channel] = haar[dst][i][channel];
        }
      }
    }

    // amplify octaves
    for (int channel = 0; channel < CHANNELS; ++channel)
```

```
    {
      haar[src][0][channel] *= E[0];
    }
    for ( int octave = 1, length = 1
        ; length <= BLOCKSIZE >> 1
        ; octave += 1, length <<= 1
        )
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        for (int i = 0; i < length; ++i)
        {
          haar[src][length + i][channel] *= E[octave];
        }
      }
    }

    // compute inverse Haar wavelet transform
    for (int length = 1; length <= BLOCKSIZE >> 1; length <<= 1)
    {
      for (int i = 0; i < BLOCKSIZE; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          haar[dst][i][channel] = haar[src][i][channel];
        }
      }
      for (int i = 0; i < length; ++i)
      {
        for (int channel = 0; channel < CHANNELS; ++channel)
        {
          float s = haar[dst][         i][channel];
          float d = haar[dst][length + i][channel];
          float a = s + d;
          float b = s - d;
          haar[src][2 * i + 0][channel] = a;
          haar[src][2 * i + 1][channel] = b;
        }
      }
    }

    // window
    for (int i = 0; i < BLOCKSIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        haar[src][i][channel] *= window[i];
      }
    }

    // overlap-add
    for (int i = 0; i < BLOCKSIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        obuffer[i][channel]
          = i + HOP < BLOCKSIZE
```

```
          ? obuffer[i + HOP][channel]
          : 0;
      }
    }
    for (int i = 0; i < BLOCKSIZE; ++i)
    {
      for (int channel = 0; channel < CHANNELS; ++channel)
      {
        obuffer[i][channel] += haar[src][i][channel];
      }
    }

    // output
    sf_writef_float(ofile, &obuffer[0][0], HOP);

  }
  sf_close(ifile);
  sf_close(ofile);
  return 0;
}
```