```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Sat Jan 13 05:29:01 PST 2007 (copied over from MzNevermore)
// Last Modified: Sun Jan 14 08:13:38 PST 2007
// Filename:      MzSpectralFlatness.cpp
// URL:           http://sv.mazurka.org.uk/src/MzSpectralFlatness.cpp
// Documentation: http://sv.mazurka.org.uk/MzSpectralFlatness
// Syntax:        ANSI99 C++; vamp plugin
//
// Description:   Spectral flatness measurement plugin for vamp.
//

#define P_VER    "200701140"
#define P_NAME   "MzSpectralFlatness"

#include "MzSpectralFlatness.h"
#include <stdio.h>
#include <math.h>

#include <vector>
#include <string>

using namespace std;


///////////////////////////////////////////////////////////////////////////
//
// Vamp Interface Functions
//


///////////////////////////////////////////////////////////////////////////
//
// Vamp Interface Functions
//

//////////////////////////////
//
// MzSpectralFlatness::MzSpectralFlatness -- class constructor.
//

MzSpectralFlatness::MzSpectralFlatness(float samplerate) :
      MazurkaPlugin(samplerate) {
   mz_transformsize =  1024;
   mz_minbin        =  0;
   mz_maxbin        =  511;
   mz_compress      =  0;
}



//////////////////////////////
//
// MzSpectralFlatness::~MzSpectralFlatness -- class destructor.
//

MzSpectralFlatness::~MzSpectralFlatness() {
   // do nothing
}


////////////////////////////////////////////////////////////
//
// parameter functions --
//

//////////////////////////////
//
// MzSpectralFlatness::getParameterDescriptors -- return a list of
//      the parameters which can control the plugin.
//
//
//    "windowsamples"    -- number of samples in audio window
//    "transformsamples" -- number of samples in transform
//    "stepsamples"      -- number of samples between analysis windows
//    "minbin"           -- lowest transform bin to display
//    "maxbin"           -- highest transform bin to display

MzSpectralFlatness::ParameterList
MzSpectralFlatness::getParameterDescriptors(void) const {

   ParameterList       pdlist;
   ParameterDescriptor pd;

   // first parameter: The number of samples in the audio window
   pd.name         = "windowsamples";
   pd.description  = "Window size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 20000.0;
   pd.defaultValue = 512.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // second parameter: The number of samples in the DFT transform
   pd.name         = "transformsamples";
   pd.description  = "Transform size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 100000.0;
   pd.defaultValue = 512.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // third parameter: The step size between analysis windows.
   pd.name         = "stepsamples";
   pd.description  = "Step size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 300000.0;
   pd.defaultValue = 441.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // fourth parameter: The minimum bin number to display.
   // Note: must be less or equal to the maximum bin size.
   // This will be enforced in the initialise() function.
   pd.name         = "minbin";
   pd.description  = "Min spectral bin";
   pd.unit         = "bin";
   pd.minValue     = 0.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
```

```
   // fifth parameter: The minimum bin number to display in terms
   // of frequency.  This will override "minbin" if set to a value
   // other than 0.0;
   pd.name         = "minfreq";
   pd.description  = "         or in Hz:";
   pd.unit         = "Hz";
   pd.minValue     = 0.0;
   pd.maxValue     = getSrate()/2.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = false;
   //pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // sixth parameter: The maximum bin number to display.
   // Note: must be greater or equal to the mininimum bin size,
   // and smaller than the transform size.  This will
   // be enforced in the initialise() function.
   pd.name         = "maxbin";
   pd.description  = "Max spectral bin";
   pd.unit         = "bin";
   pd.minValue     = 0.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = 2048.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // seventh parameter: The maximum bin number to display in
   // terms of frequency.  This will override "maxbin" if set
   // to a value other than 0.0
   pd.name         = "maxfreq";
   pd.description  = "         or in Hz:";
   pd.unit         = "Hz";
   pd.minValue     = 0.0;
   pd.maxValue     = getSrate()/2.0;
   pd.defaultValue = pd.minValue;
   pd.isQuantized  = false;
   // pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

/*
   // eighth parameter: Magnitude range compression.
   pd.name         = "compress";
   pd.description  = "Compress range";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = 1.0;
   pd.valueNames.push_back("no");
   pd.valueNames.push_back("yes");
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();
*/

   // ninth parameter: Signal windowing method
   pd.name         = "windowtype";
   pd.description  = "Window type";
   pd.unit         = "";
   MazurkaWindower::getWindowList(pd.valueNames);
   pd.minValue     = 1.0;
   pd.maxValue     = pd.valueNames.size();
   pd.defaultValue = 2.0;                   // probably the Hann window
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // tenth parameter: Smoothing gain
   pd.name         = "smooth";
   pd.description  = "Smoothing";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 0.999;
   pd.defaultValue = 0.95;
   pd.isQuantized  = false;
   //pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   return pdlist;
}


////////////////////////////////////////////////////////////
//
// optional polymorphic functions inherited from PluginBase:
//

//////////////////////////////
//
// MzSpectralFlatness::getPreferredStepSize -- overrides the
//     default value of 0 (no preference) returned in the
//     inherited plugin class.
//

size_t MzSpectralFlatness::getPreferredStepSize(void) const {
   return getParameterInt("stepsamples");
}



//////////////////////////////
//
// MzSpectralFlatness::getPreferredBlockSize -- overrides the
//     default value of 0 (no preference) returned in the
//     inherited plugin class.
//

size_t MzSpectralFlatness::getPreferredBlockSize(void) const {

   int transformsize = getParameterInt("transformsamples");
   int blocksize     = getParameterInt("windowsamples");

   if (blocksize > transformsize) {
      blocksize = transformsize;
   }

   return blocksize;
}


////////////////////////////////////////////////////////////
//
// required polymorphic functions inherited from PluginBase:
//

std::string MzSpectralFlatness::getName(void) const
   { return "mzspectralflatness"; }
```

```
std::string MzSpectralFlatness::getMaker(void) const
   { return "The Mazurka Project"; }

std::string MzSpectralFlatness::getCopyright(void) const
   { return "2007 Craig Stuart Sapp"; }

std::string MzSpectralFlatness::getDescription(void) const
   { return "Spectral Flatness"; }

int MzSpectralFlatness::getPluginVersion(void) const {
   const char *v = "@@VampPluginID@" P_NAME "@" P_VER "@" __DATE__ "@@";
   if (v[0] != '@') { std::cerr << v << std::endl; return 0; }
   return atol(P_VER);
}


////////////////////////////////////////////////////////////
//
// required polymorphic functions inherited from Plugin:
//

//////////////////////////////
//
// MzSpectralFlatness::getInputDomain -- the host application needs
//    to know if it should send either:
//
// TimeDomain      == Time samples from the audio waveform.
// FrequencyDomain == Spectral frequency frames which will arrive
//                    in an array of interleaved real, imaginary
//                    values for the complex spectrum (both positive
//                    and negative frequencies). Zero Hz being the
//                    first frequency sample and negative frequencies
//                    at the far end of the array as is usually done.
//                    Note that frequency data is transmitted from
//                    the host application as floats.  The data will
//                    be transmitted via the process() function which
//                    is defined further below.
//

MzSpectralFlatness::InputDomain MzSpectralFlatness::getInputDomain(void) const {
   return TimeDomain;
}



//////////////////////////////
//
// MzSpectralFlatness::getOutputDescriptors -- return a list describing
//    each of the available outputs for the object.  OutputList
//    is defined in the file vamp-sdk/Plugin.h:
//
// .name              == short name of output for computer use.  Must not
//                       contain spaces or punctuation.
// .description       == long name of output for human use.
// .unit              == the units or basic meaning of the data in the
//                       specified output.
// .hasFixedBinCount == true if each output feature (sample) has the
//                       same dimension.
// .binCount          == when hasFixedBinCount is true, then this is the
//                       number of values in each output feature.
//                       binCount=0 if timestamps are the only features,
//                       and they have no labels.
// .binNames          == optional description of each bin in a feature.
// .hasKnownExtent    == true if there is a fixed minimum and maximum
//                       value for the range of the output.
// .minValue          == range minimum if hasKnownExtent is true.
// .maxValue          == range maximum if hasKnownExtent is true.
// .isQuantized       == true if the data values are quantized.  Ignored
//                       if binCount is set to zero.
// .quantizeStep      == if isQuantized, then the size of the quantization,
//                       such as 1.0 for integers.
// .sampleType        == Enumeration with three possibilities:
//   OD::OneSamplePerStep   -- output feature will be aligned with
//                              the beginning time of the input block data.
//   OD::FixedSampleRate    -- results are evenly spaced according to
//                              .sampleRate (see below).
//   OD::VariableSampleRate -- output features have individual timestamps.
// .sampleRate        == samples per second spacing of output features when
//                       sampleType is set toFixedSampleRate.
//                       Ignored if sampleType is set to OneSamplePerStep
//                       since the start time of the input block will be used.
//                       Usually set the sampleRate to 0.0 if VariableSampleRate
//                       is used; otherwise, see vamp-sdk/Plugin.h for what
//                       positive sampleRates would mean.
//

MzSpectralFlatness::OutputList
MzSpectralFlatness::getOutputDescriptors(void) const {

   OutputList       odlist;
   OutputDescriptor od;


   // First output channel: The raw spectral flatness values
   od.name              = "rawflatness";
   od.description       = "Spectral Flatness Function";
   od.unit              = "";
   od.hasFixedBinCount = true;
   od.binCount          = 1;
   od.hasKnownExtents  = false;
   // od.minValue       = 0.0;
   // od.maxValue       = 1.0;
   od.isQuantized       = false;
   // od.quantizeStep  = 1.0;
   od.sampleType        = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate     = 0.0;
   #define OUTPUT_FLATNESS_CURVE 0
   odlist.push_back(od);
   od.binNames.clear();

   // Second output channel: The smoothed spectral flatness values
   od.name              = "smoothedflatness";
   od.description       = "Smoothed Spectral Flatness Function";
   od.unit              = "";
   od.hasFixedBinCount = true;
   od.binCount          = 1;
   od.hasKnownExtents  = false;
   // od.minValue       = 0.0;
   // od.maxValue       = 1.0;
   od.isQuantized       = false;
   // od.quantizeStep  = 1.0;
   od.sampleType        = OutputDescriptor::VariableSampleRate;
   // od.sampleRate     = 0.0;
   #define OUTPUT_FLATNESS_SMOOTH 1
   odlist.push_back(od);
   od.binNames.clear();

   // Third output channel: The geometric mean of the audio signal
   od.name              = "geometric mean";
```

```
   od.description      = "Geometric Mean";
   od.unit             = "";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 1.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   #define OUTPUT_GEOMETRIC_MEAN 2
   odlist.push_back(od);
   od.binNames.clear();

   // Fourth output channel: The arithmeticmean of the audio signal
   od.name             = "arithmeticmean";
   od.description      = "Arithmetic Mean";
   od.unit             = "";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 1.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   #define OUTPUT_ARITHMETIC_MEAN 3
   odlist.push_back(od);
   od.binNames.clear();

   return odlist;
}



//////////////////////////////
//
// MzSpectralFlatness::initialise -- this function is called once
//     before the first call to process().
//

bool MzSpectralFlatness::initialise(size_t channels, size_t stepsize,
      size_t blocksize) {

   if (channels < getMinChannelCount() || channels > getMaxChannelCount()) {
      return false;
   }

   // step size and block size should never be zero
   if (stepsize <= 0 || blocksize <= 0) {
      return false;
   }

   setChannelCount(channels);
   setStepSize(stepsize);
   setBlockSize(blocksize);

   mz_compress      = getParameterInt("compress");
   mz_transformsize = getParameterInt("transformsamples");
   mz_minbin        = getParameterInt("minbin");
   mz_maxbin        = getParameterInt("maxbin");
   mz_smooth        = getParameterDouble("smooth");
   if (getParameter("minfreq") > 0.0) {
      // rounding down to the lower integer value
      mz_minbin = int(getParameter("minfreq") / (getSrate()/mz_transformsize));
   }
   if (getParameter("maxfreq") > 0.0) {
      // rounding up to the next higher integer value
      mz_maxbin = int(getParameter("maxfreq") /
                      (getSrate()/mz_transformsize) + 0.999);
   }

   if (mz_maxbin >= mz_transformsize) { mz_maxbin = mz_transformsize / 2 - 1; }
   if (mz_minbin >= mz_transformsize) { mz_minbin = mz_transformsize / 2 - 1; }
   if (mz_minbin > mz_maxbin)         { std::swap(mz_minbin, mz_maxbin); }
   if (mz_minbin < 0)                 { mz_minbin = 0; }
   if (mz_maxbin < 0)                 { mz_maxbin = 0; }

   mz_transformer.setSize(mz_transformsize);
   mz_windower.setSize(getBlockSize());
   mz_windower.makeWindow(getParameterString("windowtype"));

   // std::cerr << "MzSpectralFlatness::initialize : window is set to "
   //           << getParameterString("windowtype") << std::endl;


   flatness_curve.clear();
   flatness_times.clear();

   return true;
}



//////////////////////////////
//
// MzSpectralFlatness::process -- This function is called sequentially on the
//    input data, block by block.  After the sequence of blocks has been
//    processed with process(), the function getRemainingFeatures() will
//    be called.
//
// Here is a reference chart for the Feature struct:
//
// .hasTimestamp   == If the OutputDescriptor.sampleType is set to
//                    VariableSampleRate, then this should be "true".
// .timestamp      == The time at which the feature occurs in the time stream.
// .values         == The float values for the feature.  Should match
//                    OD::binCount.
// .label          == Text associated with the feature (for time instants).
//

// #define sigmoidscale(x,c,w)  (1.0/(1.0+exp(-((x)-(c))/((w)/8.0))))

MzSpectralFlatness::FeatureSet MzSpectralFlatness::process(AUDIODATA inputbufs,
      Vamp::RealTime timestamp) {

   if (getStepSize() <= 0) {
      std::cerr << "ERROR: MzSpectralFlatness::process: "
                << "MzSpectralFlatness has not been initialized"
                << std::endl;
      return FeatureSet();
   }

   FeatureSet returnFeatures;
   Feature feature;

   feature.hasTimestamp = false;
```

```
   mz_windower.windowNonCausal(mz_transformer, inputbufs[0], getBlockSize());
   mz_transformer.doTransform();

   int bincount = mz_maxbin - mz_minbin + 1;

   vector<double> magnitude;
   magnitude.resize(bincount);

   int i;
   for (i=0; i<bincount; i++) {
      magnitude[i] = mz_transformer.getSpectrumMagnitude(i + mz_minbin);
   }

   // double sflat = getSpectralFlatness(magnitude);
   double sflat;
   double arithmeticmean = getArithmeticMean(magnitude);
   double geometricmean  = getGeometricMean(magnitude);
   if (arithmeticmean == 0.0) {
      sflat = 0.0;
   } else {
      sflat = geometricmean / arithmeticmean;
   }

   feature.hasTimestamp = false;
   feature.values.clear();
   feature.values.push_back(sflat);
   returnFeatures[OUTPUT_FLATNESS_CURVE].push_back(feature);

   feature.hasTimestamp = false;
   feature.values.clear();
   feature.values.push_back(geometricmean);
   returnFeatures[OUTPUT_GEOMETRIC_MEAN].push_back(feature);

   feature.hasTimestamp = false;
   feature.values.clear();
   feature.values.push_back(arithmeticmean);
   returnFeatures[OUTPUT_ARITHMETIC_MEAN].push_back(feature);

   // store value for smoothing later in getRemainingFeatures
   flatness_curve.push_back(sflat);
   flatness_times.push_back(timestamp);


   return returnFeatures;
}



//////////////////////////////
//
// MzSpectralFlatness::getRemainingFeatures -- This function is called
//    after the last call to process() on the input data stream has
//    been completed.  Features which are non-causal can be calculated
//    at this point.  See the comment above the process() function
//    for the format of output Features.
//

MzSpectralFlatness::FeatureSet MzSpectralFlatness::getRemainingFeatures(void) {

   FeatureSet returnFeatures;
   Feature feature;

   feature.hasTimestamp = true;
   smoothSequence(flatness_curve, mz_smooth);
   int i;
   int size = (int)flatness_curve.size();
   for (i=0; i<size; i++) {
      feature.values.clear();
      feature.timestamp = flatness_times[i];
      feature.values.push_back(flatness_curve[i]);
      returnFeatures[OUTPUT_FLATNESS_SMOOTH].push_back(feature);
   }

   return returnFeatures;
}



//////////////////////////////
//
// MzSpectralFlatness::reset -- This function may be called after data processing
//    has been started with the process() function.  It will be called when
//    processing has been interrupted for some reason and the processing
//    sequence needs to be restarted (and current analysis output thrown out).
//    After this function is called, process() will start at the beginning
//    of the input selection as if initialise() had just been called.
//    Note, however, that initialise() will NOT be called before processing
//    is restarted after a reset().
//

void MzSpectralFlatness::reset(void) {
   flatness_curve.clear();
   flatness_times.clear();
}


///////////////////////////////////////////////////////////////////////////
//
// Non-Interface Functions
//

//////////////////////////////
//
// MzSpectralFlatness::getSpectralFlatness --
//

double MzSpectralFlatness::getSpectralFlatness(vector<double>& sequence) {
   double arithmeticmean = getArithmeticMean(sequence);
   if (arithmeticmean == 0.0) {
      return 0.0;
   }
   double geometricmean  = getGeometricMean(sequence);
   return geometricmean / arithmeticmean;
}




//////////////////////////////
//
// MzSpectralFlatness::getGeometricMean -- Ignore zero bins.
//

double MzSpectralFlatness::getGeometricMean(vector<double>& sequence) {
   int i;
   int size = (int)sequence.size();
   int count = 0;
   for (i=0; i<size; i++) {
```

```
      if (sequence[i] != 0.0) {
         count++;
      }
   }

   if (count == 0) {
      return 0.0;
   }

   double power = 1.0 / count;

   double product = 1.0;
   for (i=0; i<size; i++) {
      if (sequence[i] == 0.0) {
         continue;
      }
      product *= pow(sequence[i], power);
   }

   return product;
}



//////////////////////////////
//
// MzSpectralFlatness::getArithmeticMean -- Ignore zero bins.
//

double MzSpectralFlatness::getArithmeticMean(vector<double>& sequence) {
   int i;
   int size = (int)sequence.size();
   int count = 0;
   for (i=0; i<size; i++) {
      if (sequence[i] != 0.0) {
         count++;
      }
   }

   if (count == 0) {
      return 0.0;
   }

   double sum = 0.0;
   for (i=0; i<size; i++) {
      sum += sequence[i];
   }

   return sum / count;
}



//////////////////////////////
//
// MzSpectralFlatness::smoothSequence -- smooth the sequence with a
//    symmetric exponential smoothing filter (applied in the forward
//    and reverse directions with the specified input gain.
//
//    Difference equation for smoothing: y[n] = k * x[n] + (1-k) * y[n-1]
//

void MzSpectralFlatness::smoothSequence(vector<double>& sequence, double gain) {
   double oneminusgain = 1.0 - gain;
   int i;
   int ssize = sequence.size();

   // reverse filtering first
   for (i=ssize-2; i>=0; i--) {
      sequence[i] = gain*sequence[i] + oneminusgain*sequence[i+1];
   }

   // then forward filtering
   for (i=1; i<ssize; i++) {
      sequence[i] = gain*sequence[i] + oneminusgain*sequence[i-1];
   }

}
```