```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Mon Dec 18 20:37:48 PST 2006
// Last Modified: Wed Jan  3 06:09:24 PST 2007
// Filename:      MzSpectralFlux.cpp
// URL:           http://sv.mazurka.org.uk/src/MzSpectralFlux.cpp
// Documentation: http://sv.mazurka.org.uk/MzSpectralFlux
// Syntax:        ANSI99 C++; vamp plugin
//
// Description:   Generate various forms and steps in the process of
//                of calculating spectral flux.
//
// Reference:     http://en.wikipedia.org/wiki/Spectral_flux
//

#include "MzSpectralFlux.h"

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

#include <string>

// Defines used in getPluginVersion():
#define P_VER    "200612280"
#define P_NAME   "MzSpectralFlux"

// Type of spectral flux measurement:
#define SLOPE_ALL           0
#define SLOPE_POSITIVE      1
#define SLOPE_NEGATIVE      2
#define SLOPE_DIFFERENCE    3
#define SLOPE_COMPOSITE     4
#define SLOPE_PRODUCT       5
#define SLOPE_ANGULAR       6
#define SLOPE_COSINE        7

// Type of magnitude spectrum for calculating spectral derivative:
#define SPECTRUM_DFT        0
#define SPECTRUM_LOWDFT     1
#define SPECTRUM_HIDFT      2
#define SPECTRUM_MIDI       3

using namespace std;        // avoid stupid std:: prefixing

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

//////////////////////////////
//
// MzSpectralFlux::MzSpectralFlux -- class constructor.  The values
//   for the mz_* variables are just place holders demonstrating the
//   default value.  These variables will be set in the initialise()
//   function from the user interface.
//

MzSpectralFlux::MzSpectralFlux(float samplerate) :
      MazurkaPlugin(samplerate) {
   mz_slope  = SLOPE_POSITIVE;   // consider positive spectral derivative
   mz_stype  = SPECTRUM_MIDI;    // use MIDI spectrum by default
   mz_pnorm  = 2.0;              // for calculating spectral difference norm
   mz_delta  = 0.45;             // higher value gives more false negatives
   mz_alpha  = 0.90;             // higher values gives few false positives
}



//////////////////////////////
//
// MzSpectralFlux::~MzSpectralFlux -- class destructor.
//

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


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

//////////////////////////////
//
// MzSpectralFlux::getParameterDescriptors -- return a list of
//      the parameters which can control the plugin.
//

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

   ParameterList       pdlist;
   ParameterDescriptor pd;

   // first parameter: Number of samples in the audio window
   pd.name         = "windowsamples";
   pd.description  = "Window Size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 10000;
   pd.defaultValue = 2048.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // second parameter: Step size between analysis windows
   pd.name         = "stepsamples";
   pd.description  = "Step Size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = 441.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // third parameter: Slope limiting for adjusting spectral derivative
   pd.name         = "fluxtype";
   pd.description  = "Flux Type";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 7.0;
   pd.valueNames.push_back("Total Flux");
   pd.valueNames.push_back("Positive Flux");
   pd.valueNames.push_back("Negative Flux");
   pd.valueNames.push_back("Difference Flux");
   pd.valueNames.push_back("Composite Flux");
```

```
   pd.valueNames.push_back("Product Flux");
   pd.valueNames.push_back("Angular Flux");
   pd.valueNames.push_back("Cosine Flux");
   pd.defaultValue = 1.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fourth parameter: Spectral smoothing
   pd.name         = "smooth";
   pd.description  = "Spectral\nSmoothing";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = false;
   // pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fifth parameter: p-Norm Order
   pd.name         = "pnorm";
   pd.description  = "Norm Order";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = +100.0;
   pd.defaultValue = 1.0;
   pd.isQuantized  = false;
   // pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // sixth parameter: Magnitude spectrum type for calculating spectral flux
   pd.name         = "spectrum";
   pd.description  = "Magnitude\nSpectrum";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 3.0;
   pd.valueNames.push_back("DFT");
   pd.valueNames.push_back("Low DFT");
   pd.valueNames.push_back("High DFT");
   pd.valueNames.push_back("MIDI");
   pd.defaultValue = 3.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // seventh parameter: Local mean threshold for peak identification
   pd.name         = "delta";
   pd.description  = "Local Mean\nThreshold";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 100.0;
   pd.defaultValue = 0.45;
   pd.isQuantized  = false;
   // pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // eighth parameter: Threshold function feedback gain
   pd.name         = "alpha";
   pd.description  = "Exponential\nDecay Factor";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 0.999;
   pd.defaultValue = 0.90;
   pd.isQuantized  = false;
   // pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   return pdlist;
}


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

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

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



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

size_t MzSpectralFlux::getPreferredBlockSize(void) const {
   return getParameterInt("windowsamples");
}


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

std::string MzSpectralFlux::getName(void) const
{ return "mzspectralflux"; }

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

std::string MzSpectralFlux::getCopyright(void) const
{ return "2006 Craig Stuart Sapp"; }

std::string MzSpectralFlux::getDescription(void) const
{ return "Spectral Flux"; }

int MzSpectralFlux::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:
//

//////////////////////////////
//
// MzSpectralFlux::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.
//

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



//////////////////////////////
//
// MzSpectralFlux::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.
//

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

   OutputList       odlist;
   OutputDescriptor od;

   std::string s;

   int spectrumbincount = calculateSpectrumSize(mz_stype, getBlockSize(),
                                                           getSrate());

   // First output channel: Underlying Spectral Data
   od.name             = "spectrum";
   od.description      = "Basis Spectrum";
   od.unit             = "bin";
   od.hasFixedBinCount = true;
   od.binCount         = spectrumbincount;
   od.hasKnownExtents  = false;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);
   #define OUTPUT_SPECTRUM 0
   od.binNames.clear();

   // Second output channel: Spectrum Derivative
   od.name             = "spectrumderivative";
   od.description      = "Spectrum Derivative";
   od.unit             = "bin";
   od.hasFixedBinCount = true;
   od.binCount         = spectrumbincount;
   od.hasKnownExtents  = false;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);
   #define OUTPUT_DERIVATIVE 1
   od.binNames.clear();

   // Third output channel: Raw Spectral Flux Function
   od.name             = "rawspectralflux";
   od.description      = "Raw Spectral Flux Function";
   od.unit             = "raw";
   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_RAW_FUNCTION 2
   odlist.push_back(od);
   od.binNames.clear();

   // Fourth output channel: Scaled Spectral Flux Function
```

```
   od.name             = "scaledspectralflux";
   od.description      = "Scaled Spectral Flux Function";
   od.unit             = "scaled";
   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_SCALED_FUNCTION 3
   odlist.push_back(od);
   od.binNames.clear();

   // Fifth output channel: Exponential Decay Threshold
   od.name             = "thresholdfunction";
   od.description      = "Exponential Decay Threshold";
   od.unit             = "scaled";
   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_THRESHOLD_FUNCTION 4
   odlist.push_back(od);
   od.binNames.clear();

   // Sixth output channel: Mean Threshold Function
   od.name             = "meanfunction";
   od.description      = "Local Mean Threshold";
   od.unit             = "scaled";
   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_MEAN_FUNCTION 5
   odlist.push_back(od);
   od.binNames.clear();

   // Seventh output channel: Detected Onset Times
   od.name             = "spectralfluxonsets";
   od.description      = "Onset Times";
   od.unit             = "";
   od.hasFixedBinCount = true;
   od.binCount         = 0;
   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_ONSETS 6
   odlist.push_back(od);
   od.binNames.clear();

   return odlist;
}



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

bool MzSpectralFlux::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;
   }

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

   mz_slope  = getParameterInt("fluxtype");
   mz_stype  = getParameterInt("spectrum");
   mz_delta  = getParameterDouble("delta");
   mz_alpha  = getParameterDouble("alpha");
   mz_pnorm  = getParameterDouble("pnorm");
   mz_smooth = 1.0 - getParameterDouble("smooth");

   mz_transformer.setSize(getBlockSize());
   mz_transformer.zeroSignal();
   mz_windower.setSize(getBlockSize());
   mz_windower.makeWindow("Hann");

   mz_rawfunction.resize(0);
   mz_rawtimes.resize(0);

   return true;
}



//////////////////////////////
//
// MzSpectralFlux::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).
//
```

```
MzSpectralFlux::FeatureSet MzSpectralFlux::process(float **inputbufs,
      Vamp::RealTime timestamp) {

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

   int i;
   Feature    feature;
   FeatureSet returnFeatures;

   // calculate the the underlying spectrum data:
   mz_windower.windowNonCausal(mz_transformer, inputbufs[0], getBlockSize());
   mz_transformer.doTransform();

   // generate the variety of spectrum to be used to calculate spectral flux:
   vector<double> workingspectrum;
   createWorkingSpectrum(workingspectrum, mz_transformer, getSrate(),
         mz_stype, mz_smooth);

   // store the size of the spectrum:
   int framesize = (int)(workingspectrum.size());


   ///////////////////////////////////////////////////////////////////////////
   ///// store the plugin’s FIRST output: the raw spectral data ///////////
   ///////////////////////////////////////////////////////////////////////////

   feature.values.resize(framesize);
   for (i=0; i<framesize; i++) {
      feature.values[i] = workingspectrum[i];
   }
   feature.hasTimestamp = false;
   returnFeatures[OUTPUT_SPECTRUM].push_back(feature);


   // Calculate the spectral derivative: the difference between
   // two sequential spectrums.

   vector<double> spectral_derivative;
   spectral_derivative.resize(framesize);

   // if the lastframe has not been initialized, then copy current spectrum
   // (or maybe set to zero if audio starts with an attack??)
   if (lastframe.size() == 0) {
      lastframe.resize(framesize);
      for (i=0; i<framesize; i++) {
         lastframe[i] = workingspectrum[i] / 2.0;
      }
   }

   // selectively remove slopes from the spectral difference vector
   // depending on the type of spectral flux calculation being done:
   switch (mz_slope) {

      case SLOPE_NEGATIVE:   // negative slopes only
         for (i=0; i<framesize; i++) {
            spectral_derivative[i] = workingspectrum[i] - lastframe[i];
            if (spectral_derivative[i] > 0.0) {
               spectral_derivative[i] = 0.0;
            }
         }
      break;

      case SLOPE_PRODUCT:   // slope product rather than difference
         for (i=0; i<framesize; i++) {
            spectral_derivative[i] = workingspectrum[i] * lastframe[i];
         }
      break;

      case SLOPE_ANGULAR:   // angle rather than difference
      case SLOPE_COSINE:    // angle rather than difference
         {
         double asum = 0.0;
         double bsum = 0.0;
         double cval = 0.0;
         for (i=0; i<framesize; i++) {
            asum += workingspectrum[i] * workingspectrum[i];
            bsum += lastframe[i] * lastframe[i];
         }
         cval = sqrt(asum) * sqrt(bsum);
         for (i=0; i<framesize; i++) {
            spectral_derivative[i] = workingspectrum[i] * lastframe[i] / cval;
         }
         }
      break;

      case SLOPE_POSITIVE:    // positive slopes only
         for (i=0; i<framesize; i++) {
            spectral_derivative[i] = workingspectrum[i] - lastframe[i];
            if (spectral_derivative[i] < 0.0) {
               spectral_derivative[i] = 0.0;
            }
         }
      break;

      case SLOPE_ALL:        // no selectivity
      case SLOPE_DIFFERENCE: // mixed selectivity so don’t remove anything
      case SLOPE_COMPOSITE:  // mixed selectivity so don’t remove anything
      default:
         for (i=0; i<framesize; i++) {
            spectral_derivative[i] = workingspectrum[i] - lastframe[i];
         }

   }

   // store the current spectrum so that it can be used next time:
   lastframe = workingspectrum;


   ///////////////////////////////////////////////////////////////////////////
   ///// store the plugin’s SECOND output: spectral derivative  ///////////
   ///////////////////////////////////////////////////////////////////////////

   // to make the data more visible, normalize each frame.
   // maybe consider sigmoiding it also...
   double normval = 0.0;
   for (i=0; i<framesize; i++) {
      if (fabs(spectral_derivative[i]) > normval) {
         normval = fabs(spectral_derivative[i]);
      }
   }
   if (normval == 0.0) {   // avoid any divide by zero problems
      normval = 1.0;
   }

   feature.values.resize(framesize);
```

```
   for (i=0; i<framesize; i++) {
      feature.values[i] = spectral_derivative[i] / normval;
   }
   feature.hasTimestamp = false;
   returnFeatures[OUTPUT_DERIVATIVE].push_back(feature);


   ////////////////////////////////////////////////////////////////////////
   ///// store the plugin's THIRD output: spectral flux value   ///////////
   ////////////////////////////////////////////////////////////////////////

   double fluxvalue;
   fluxvalue = getSpectralFlux(spectral_derivative, mz_slope, mz_pnorm);

   // the spectral flux is the difference between two spectral
   // frames, so it is best placed 1/2 of the way between the
   // center of each of the two spectral frames.  To do this,
   // subtract 1/2 of the hopsize to move to the average location
   // between the start of each frame, then add 1/2 of the block
   // size to center in the average middle time of the two frames.

   // There should also be an compensation for the window size
   // relationship to the hop size (large windows will smear the flux
   // so onsets will become earlier than for shorter windows).

   feature.hasTimestamp = true;
   feature.timestamp = timestamp
      - Vamp::RealTime::fromSeconds(0.5 * getStepSize()/getSrate())
      + Vamp::RealTime::fromSeconds(0.5 * getBlockSize()/getSrate());

   feature.values.resize(0);
   feature.values.push_back(fluxvalue);
   returnFeatures[OUTPUT_RAW_FUNCTION].push_back(feature);

   // also store the spectral flux function for later onset processing
   // in the getRemainingFeatures() function:
   mz_rawfunction.push_back(feature.values[0]);
   mz_rawtimes.push_back(feature.timestamp);

   return returnFeatures;
}



//////////////////////////////
//
// MzSpectralFlux::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.
//

MzSpectralFlux::FeatureSet MzSpectralFlux::getRemainingFeatures(void) {

   Feature    feature;
   FeatureSet returnFeatures;
   int i;


   ////////////////////////////////////////////////////////////////////////
   ///// store the plugin's FOURTH output: scaled SF function /////////////
   ////////////////////////////////////////////////////////////////////////

   // for the SLOPE_PRODUCT, store the log-slope of the stored data in
   // mz_rawfunction:
   vector<double> tempprod;
   tempprod.resize(mz_rawfunction.size());
   tempprod[0] = 0.0;
   if (mz_stype == SLOPE_PRODUCT) {
      for (i=1; i<(int)mz_rawfunction.size(); i++) {
         tempprod[i] = log(mz_rawfunction[i] - mz_rawfunction[i-1]);
      }
      for (i=0; i<(int)mz_rawfunction.size(); i++) {
         mz_rawfunction[i] = tempprod[i];
      }
   }

   // scale the raw spectral flux function so that its mean (average) is 0.0
   // and its standard deviation is 1.0.

   double mean = getMean(mz_rawfunction);
   double sd   = getStandardDeviation(mz_rawfunction, mean);

   vector<double> scaled_function;
   scaled_function.resize(mz_rawfunction.size());

   feature.hasTimestamp = true;
   for (i=0; i<(int)mz_rawfunction.size(); i++) {
      scaled_function[i] = (mz_rawfunction[i] - mean) / sd;
      feature.values.resize(0);
      feature.values.push_back(scaled_function[i]);
      feature.timestamp    = mz_rawtimes[i];
      returnFeatures[OUTPUT_SCALED_FUNCTION].push_back(feature);
   }

   vector<Vamp::RealTime> onset_times;
   vector<double> threshold_function;
   vector<double> mean_function;
   vector<double> onset_levels;

   findOnsets(onset_times, onset_levels, mean_function, threshold_function,
              scaled_function, mz_rawtimes, mz_delta, mz_alpha);

   ////////////////////////////////////////////////////////////////////////
   ///// store the plugin's FIFTH output: threshold function  ////////////
   ////////////////////////////////////////////////////////////////////////

   feature.hasTimestamp = true;
   for (i=0; i<(int)threshold_function.size(); i++) {
      feature.timestamp = mz_rawtimes[i];
      feature.values.clear();
      feature.values.push_back(threshold_function[i]);
      returnFeatures[OUTPUT_THRESHOLD_FUNCTION].push_back(feature);
   }

   ////////////////////////////////////////////////////////////////////////
   ///// store the plugin's SIXTH output: mean function  /////////////////
   ////////////////////////////////////////////////////////////////////////

   feature.hasTimestamp = true;
   for (i=0; i<(int)mean_function.size(); i++) {
      feature.timestamp = mz_rawtimes[i];
      feature.values.clear();
      feature.values.push_back(mean_function[i]);
      returnFeatures[OUTPUT_MEAN_FUNCTION].push_back(feature);
   }

   ////////////////////////////////////////////////////////////////////////
   ///// store the plugin's SEVENTH output: detected onsets //////////////
```

```
   ////////////////////////////////////////////////////////////////////////

   char buffer[1024] = {0};
   feature.values.clear();
   feature.hasTimestamp = true;
   for (i=0; i<(int)onset_times.size(); i++) {
      feature.timestamp = onset_times[i];
      sprintf(buffer, "%6.2lf", ((int)(onset_levels[i] * 100.0 + 0.5))/100.0);
      feature.label = buffer;
      returnFeatures[OUTPUT_ONSETS].push_back(feature);
   }

   return returnFeatures;
}



//////////////////////////////
//
// MzSpectralFlux::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 MzSpectralFlux::reset(void) {
   lastframe.resize(0);
   mz_rawfunction.resize(0);
   mz_rawtimes.resize(0);
}


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

//////////////////////////////
//
// MzSpectralFlux::generateMidiNoteList -- Create a list of pitch names
//   for the specified MIDI key number range.
//

void MzSpectralFlux::generateMidiNoteList(vector<std::string>& alist,
        int minval, int maxval) {

   alist.clear();
   if (maxval < minval) {
      std::swap(maxval, minval);
   }

   int i;
   int octave;
   int pc;
   char buffer[32] = {0};
   for (i=minval; i<=maxval; i++) {
      octave = i / 12;
      pc = i - octave * 12;
      octave = octave - 1;  // Make middle C (60) = C4
      switch (pc) {
         case 0:   sprintf(buffer, "C%d",  octave); break;
         case 1:   sprintf(buffer, "C#%d", octave); break;
         case 2:   sprintf(buffer, "D%d",  octave); break;
         case 3:   sprintf(buffer, "D#%d", octave); break;
         case 4:   sprintf(buffer, "E%d",  octave); break;
         case 5:   sprintf(buffer, "F%d",  octave); break;
         case 6:   sprintf(buffer, "F#%d", octave); break;
         case 7:   sprintf(buffer, "G%d",  octave); break;
         case 8:   sprintf(buffer, "G#%d", octave); break;
         case 9:   sprintf(buffer, "A%d",  octave); break;
         case 10:  sprintf(buffer, "A#%d", octave); break;
         case 11:  sprintf(buffer, "B%d",  octave); break;
         default:  sprintf(buffer, "x%d", i);
      }
      alist.push_back(buffer);
   }
}



//////////////////////////////
//
// MzSpectralFlux::makeFreqMap -- Calculates the bin mapping from
//      a DFT spectrum into a MIDI-like spectum.  When DFT bins are
//      wider than a half-step (MIDI note number), the DFT bin is
//      used as a single MIDI bin.  When the DFT bin is smaller than
//      a half-step, they are grouped together into a single MIDI bin.
//
// As an example, here is the mapping when the DFT transform size is 2048,
// and the samping rate is 44100 Hz:
//
// MIDI bins 0 to 34 map one-to-one with the DFT bins 0 to 34, then each
// of the subsequent MIDI bins contains the following number of DFT bins:
//
// 34:2 35:2 36:2 37:3 38:2 39:3 40:3 41:2 42:4 43:3 44:4 45:3 46:4 47:4
// 48:5 49:5 50:5 51:5 52:6 53:5 54:7 55:6 56:8 57:7 58:8 59:8 60:9 61:10
// 62:10 63:10 64:12 65:11 66:13 67:13 68:15 69:15 70:15 71:17 72:18 73:19
// 74:20 75:21 76:23 77:23 78:26 79:26 80:29 81:30 82:31 83:459
//
// MIDI bin 83 represents MIDI note number 127, and it contains the last 459
// positive frequency bins of the DFT.  MIDI bin 34 is probably representing
// MIDI note number 78 (F-sharp 5).
//
// Implementation Reference:
//     http://www.ofai.at/˜simon.dixon/beatbox
//

void MzSpectralFlux::makeFreqMap(vector<int>& mapping,
      int fftsize, float srate) {

   if (fftsize <= 0) {
      // getOutputDescriptors() will call this function
      // before the fftsize is set, so avoid an unintialized
      // fftsize.
      mapping.resize(0);
      return;
   }
   double width  = srate / fftsize;
   double a4freq = 440.0;
   int    a4midi = 69;
   int    mapsize= fftsize/2+1;
   int    xbin   = (int)(2.0/(pow(2.0, 1.0/12.0) - 1.0));
   int    xmidi  = (int)(log(xbin*width/a4freq)/log(2.0)*12 + a4midi + 0.5);
   int    midi;
   int    i;
```

```
   mapping.resize(mapsize);

   for (i=0; i<=xbin; i++) {  // store the one-to-one mappings
      mapping[i] = i;
   }
   for (i=xbin+1; i<mapsize; i++) {
      midi = (int)(log(i*width/a4freq)/log(2.0)*12 + a4midi + 0.5);
      if (midi >  127) {
         midi = 127;
      }
      mapping[i] = xbin + midi - xmidi;
   }
}



//////////////////////////////
//
// MzSpectralFlux::createWorkingSpectrum -- Creates a magnitude
//     spectrum from the input complex DFT spectrum according to
//     the user specified spectrum type.
//

void MzSpectralFlux::createWorkingSpectrum(vector<double>& magspectrum,
      MazurkaTransformer& transformer, double srate, int spectrum_type,
      double smooth) {

   vector<double> tempspec;
   int tsize = (int)transformer.getSize() / 2 + 1;
   tempspec.resize(tsize);
   int i;
   for (i=0; i<tsize; i++) {
      tempspec[i] = transformer.getSpectrumMagnitude(i);
   }

   // smooth the spectrum if requested by the user:
   if (smooth < 1.0) {
      smoothSpectrum(tempspec, smooth);
   }

   int ssize;
   switch (spectrum_type) {
      case SPECTRUM_DFT:
         ssize = transformer.getSize() / 2 + 1;
         magspectrum.resize(ssize);
         for (i=0; i<ssize; i++) {
            magspectrum[i] = tempspec[i];
         }
         break;
      case SPECTRUM_LOWDFT:
         ssize = (transformer.getSize() / 2 + 1) / 2;
         magspectrum.resize(ssize);
         for (i=0; i<ssize; i++) {
            magspectrum[i] = tempspec[i];
         }
         break;
      case SPECTRUM_HIDFT: // check for off-by-one errs here if plugin crashes
         ssize = (transformer.getSize() / 2 + 1) / 2;
         magspectrum.resize(ssize);
         for (i=0; i<ssize; i++) {
            magspectrum[i] = tempspec[i+ssize];
         }
         break;
      case SPECTRUM_MIDI:
      default:
         createMidiSpectrum(magspectrum, tempspec, srate);
   }

}



//////////////////////////////
//
// MzSpectralFlux::createMidiSpectrum -- Maps the non-negative
//     DFT spectrum into a MIDI-like spectrum.  DFT bins which are
//     less than one half-step in size (1 MIDI note) are preserved.
//     DFT bins smaller than a half-step are grouped together into
//     one MidiSpectrum bin.
//

void MzSpectralFlux::createMidiSpectrum(vector<double>& midispectrum,
      vector<double>& magspec, double srate) {

   static vector<int> mapping;

   // build the bin mapping table betwen the positive DFT bins
   // and the MIDI spectrum bins if the size of the map does
   // not match the input spectrum non-zero bin count:
   //
   if ((int)mapping.size() != (int)magspec.size()) {
      makeFreqMap(mapping, (magspec.size() - 1) * 2, srate);
   }

   // calculate the size of the output MIDI spectrum:
   int midispectrumsize = mapping[mapping.size()-1] + 1;
   midispectrum.resize(midispectrumsize);

   // choose the bin grouping method and calculate output spectrum:
   int i;

   for (i=0; i<(int)midispectrum.size(); i++) {
      midispectrum[i] = 0.0;
   }
   for (i=0; i<(int)mapping.size(); i++) {
      midispectrum[mapping[i]] += magspec[i];
   }
}



//////////////////////////////
//
// MzSpectralFlux::calculateMidiSpectrumSize -- Used in getOutputDescriptors().
//

int MzSpectralFlux::calculateMidiSpectrumSize(int transformsize, double srate) {
   if (transformsize <= 1) {
      // getOutputDescriptors() will call this function before
      // the transform size is initialized, so give some dummy
      // data when that happens.
      return 1000;
   } else {
      vector<int> mapping;
      makeFreqMap(mapping, transformsize, srate);
      return mapping[mapping.size()-1] + 1;
   }
}
```

```
//////////////////////////////
//
// MzSpectralFlux::getStandardDeviation -- calculates the standard deviation
//     of a set of numbers.
//

double MzSpectralFlux::getStandardDeviation(vector<double>& sequence,
      double mean) {
   if ((int)sequence.size() == 0) {
      return 1.0;
   }
   double sum = 0.0;
   double value;
   int i;
   for (i=0; i<(int)sequence.size(); i++) {
      value = sequence[i] - mean;
      sum += value * value;
   }

   return sqrt(sum / sequence.size());
}



//////////////////////////////
//
// MzSpectralFlux::getMean -- calculates the average of the input values.
//

double MzSpectralFlux::getMean(vector<double>& sequence, int mmin, int mmax) {
   if ((int)sequence.size() == 0) {
      return 0.0;
   }
   if (mmin < 0) {
      mmin = 0;
   }
   if (mmax < 0) {
      mmax = (int)sequence.size()-1;
   }

   double sum = 0.0;
   for (int i=mmin; i<=mmax; i++) {
      sum += sequence[i];
   }
   return sum / (mmax - mmin + 1);
}



//////////////////////////////
//
// MzSpectralFlux::findOnsets -- identify onset peaks in the scaled
//   spectral flux function according to the three criteria found
//   in section 2.6 of (Dixon 2006):
//     (1) f[n] >= local maximum
//     (2) f[n] >= local mean + delta
//     (3) f[n] >= g[n], where g[n] = max(f[n], a g[n-1] + (1-a) f[n])
//                 "g[n]" == threshold function.
//

void MzSpectralFlux::findOnsets(vector<Vamp::RealTime>& onset_times,
   vector<double>& onset_levels, vector<double>& mean_function,
   vector<double>& threshold_function, vector<double>& scaled_function,
   vector<Vamp::RealTime>& functiontimes, double delta, double alpha) {
   int    i;
   int    length      = (int)scaled_function.size();
   int    width       = 3;
   int    backwidth   = 3 * width;
   double localmeanthreshold;

   vector<double>& tf = threshold_function;
   vector<double>& sf = scaled_function;
   double&          a = alpha;

   onset_times.clear();
   onset_levels.clear();
   mean_function.resize(length);
   threshold_function.resize(length);
   threshold_function[0] = scaled_function[0];

   for (i=1; i<length; i++) {
      threshold_function[i] = std::max(sf[i], a*tf[i-1] + (1-a)*sf[i]);
   }

   for (i=0; i<length; i++) {

      // Additive method which is scaling sensitive (i.e., misses quiet
      // attacks). delta = 0.35 is the recommended value for this test.
      localmeanthreshold = getMean(sf,i-backwidth,i+width)+delta;

      // Muliplicative method using delta about 10%...  This test is
      // overly sensitive in quiet regions of the audio, so a combination
      // of the Additive and Multiplicative methods might be best.
      // localmeanthreshold = getMean(sf,i-backwidth,i+width)*(1.0+delta/100.0);

      mean_function[i] = localmeanthreshold;

      if (sf[i] < localmeanthreshold) {
         continue;
      }
   /* Additive method which is scaling sensitive (i.e., misses quiet attacks)
    * (delta = 0.35 is a recommended value for this test).
    * if (sf[i] < getMean(sf, i-backwidth, i+width) + delta)) {
    *    continue;
    * }
    */
      if (sf[i] < tf[i]) {
         continue;
      }
      if (!localmaximum(sf, i, i-width, i+width)) {
         continue;
      }

      // an onset detection has been triggered so sore the time of it:
      onset_times.push_back(functiontimes[i]);
      onset_levels.push_back(sf[i]);
   }

}



//////////////////////////////
//
// MzSpectralFlux::localmaximum -- returns true if the specified value
//     is the largest (or ties for the largest) in the given region.
//
```

```
int MzSpectralFlux::localmaximum(vector<double>& data, int target, int minimum,
   int maximum) {

   if (minimum < 0) {
      minimum = 0;
   }
   if (maximum >= (int)data.size()) {
      maximum = (int)data.size() - 1;
   }

   double maxval = data[minimum];
   for (int i=minimum+1; i<=maximum; i++) {
      maxval = std::max(maxval, data[i]);
   }

   return (maxval <= data[target]);
}



//////////////////////////////
//
// MzSpectralFlux::calculateSpectrumSize -- count how many bins
//    are present in the underlying spectrum data frames.  This depends
//    on what type of spectrum is being used.
//

int MzSpectralFlux::calculateSpectrumSize(int spectrumType, int
      blocksize, double srate) {

   // give dummy data if uninitialized variables are passed into the function:
   if (blocksize <= 1) {
      return 1000;
   }
   if (srate <= 1.0) {
      return 1000;
   }

   switch (spectrumType) {
      case SPECTRUM_MIDI:
         return calculateMidiSpectrumSize(blocksize, srate);
      break;

      case SPECTRUM_LOWDFT:
         return (blocksize / 2 + 1) / 2;
      break;

      case SPECTRUM_HIDFT:
         return (blocksize / 2 + 1) / 2;
      break;

      case SPECTRUM_DFT:
      default:
         return blocksize / 2 + 1;
   }


}



//////////////////////////////
//
// MzSpectralFlux::getSpectralFlux -- do the actual calcualtion of the
//    flux value from the spectral difference vector.
//
// The Norm calculation is (in latex format):
//    \left| x \right|_p \equiv \left( \sum_i \left|x_i\right|^p \right)^{1/p}
//

double MzSpectralFlux::getSpectralFlux(vector<double>& spectral_derivative,
      int fluxtype, double pnormorder) {

   int framesize = (int)spectral_derivative.size();
   int i;
   double safepnormorder = pnormorder == 0.0 ? 1.0 : pnormorder;

   switch (fluxtype) {
   case SLOPE_COMPOSITE:
      {
      double positive = 0.0;
      double negative = 0.0;
      double total    = 0.0;
      double value;
      for (i=0; i<framesize; i++) {
         if (spectral_derivative[i] == 0.0) {
            continue;  // no need to waste time caculating a power of zero
         }
         value = pow(fabs(spectral_derivative[i]), pnormorder);
         total += value;
         if (spectral_derivative[i] > 0) {
            positive += value;
         } else {
            negative += value;
         }
      }
      positive = pow(positive, 1.0/safepnormorder);
      negative = pow(negative, 1.0/safepnormorder);
      total    = pow(total,    1.0/safepnormorder);

      double denominator = fabs(total - positive);
      if (denominator < 0.001) {
         denominator = 0.01;
      }
      value = (positive - negative)/denominator;
      if (value < 0.0) {
         value = 0.0;
      }
      return value;
      }
   break;

   case SLOPE_DIFFERENCE:
      {
      double positive = 0.0;
      double negative = 0.0;
      double value;
      for (i=0; i<framesize; i++) {
         if (spectral_derivative[i] == 0.0) {
            continue;  // no need to waste time caculating a power of zero
         }
         value = pow(fabs(spectral_derivative[i]), pnormorder);
         if (spectral_derivative[i] > 0) {
            positive += value;
         } else {
            negative += value;
         }
      }
      positive = pow(positive, 1.0/safepnormorder);
      negative = pow(negative, 1.0/safepnormorder);
```

```
      value = positive - negative;
      if (value < 0.0) {   // supress peak detection in negative regions
         value = 0.0;
      }

      return value;
      }
   break;

   case SLOPE_ANGULAR:
      {
      double sum = 0.0;
      for (i=0; i<framesize; i++) {
         sum += spectral_derivative[i];
      }
      return acos(sum);
      }
   break;

   case SLOPE_COSINE:
      {
      double sum = 0.0;
      for (i=0; i<framesize; i++) {
         sum += spectral_derivative[i];
      }
      return -sum;
      }
   break;

   default:
      {
      double sum = 0.0;
      for (i=0; i<framesize; i++) {
         if (spectral_derivative[i] == 0.0) {
            continue;  // no need to waste time caculating a power of zero
         }
         sum += pow(fabs(spectral_derivative[i]), pnormorder);
      }
      return pow(sum, 1.0/safepnormorder);
      }
   }

   return 0.0;   // shouldn’t get to this line

}



//////////////////////////////
//
// MzSpectralFlux::smoothSpectrum -- 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 MzSpectralFlux::smoothSpectrum(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];
   }

}
```