```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Sun Jun 18 00:23:13 PDT 2006
// Last Modified: Sat Jun 24 01:28:37 PDT 2006
// Filename:      MzHarmonicSpectrum.cpp
// URL:           http://sv.mazurka.org.uk/src/MzHarmonicSpectrum.cpp
// Documentation: http://sv.mazurka.org.uk/MzHarmonicSpectrum
// Syntax:        ANSI99 C++; vamp plugin
//
// Description:   Display a harmonic spectrum
//

#include "MzHarmonicSpectrum.h"
#include <stdio.h>
#include <string>
#include <math.h>

#define METHOD_MAGNITUDE_PRODUCT   1
#define METHOD_MAGNITUDE_SUMMATION 2
#define METHOD_COMPLEX_SUMMATION   3

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

///////////////////////////////
//
// MzHarmonicSpectrum::MzHarmonicSpectrum -- class constructor.
//

MzHarmonicSpectrum::MzHarmonicSpectrum(float samplerate) :
      MazurkaPlugin(samplerate) {
   mz_harmonics     = 5;
   mz_transformsize = 16384;
   mz_minbin        = 0;
   mz_maxbin        = 511;
   mz_compress      = 0;
}



///////////////////////////////
//
// MzHarmonicSpectrum::~MzHarmonicSpectrum -- class destructor.
//

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


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

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

MzHarmonicSpectrum::ParameterList
MzHarmonicSpectrum::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     = 10000;
   pd.defaultValue = 1500.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // second parameter: The 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 = 512.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // third parameter: The number of harmonics to consider
   pd.name         = "harmonics";
   pd.description  = "Harmonics";
   pd.unit         = "";
   pd.minValue     = 2.0;
   pd.maxValue     = 20.0;
   pd.defaultValue = 5.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // fourth parameter: The minimum pitch to consider
   pd.name         = "minpitch";
   pd.description  = "Min pitch";
   pd.unit         = "MIDI data";
   pd.minValue     = 0.0;
   pd.maxValue     = 127.0;
   generateMidiNoteList(pd.valueNames, 0, 127);
   pd.defaultValue = 36.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fifth parameter: The maximum pitch to consider
   pd.name         = "maxpitch";
   pd.description  = "Max pitch";
   pd.unit         = "MIDI data";
   pd.minValue     = 0.0;
   pd.maxValue     = 127.0;
   generateMidiNoteList(pd.valueNames, 0, 127);
   pd.defaultValue = 84.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // sixth parameter: The method for harmonic correlation
   pd.name         = "method";
```

```
   pd.description  = "Method";
   pd.unit         = "";
   pd.minValue     = 1.0;
   pd.maxValue     = 3.0;
   pd.valueNames.push_back("Magnitude Product");
   pd.valueNames.push_back("Magnitude Summation");
   pd.valueNames.push_back("Complex Summation");
   pd.defaultValue = 1.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

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

   return pdlist;
}


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

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

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



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

size_t MzHarmonicSpectrum::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 MzHarmonicSpectrum::getName(void) const
   { return "mzharmonicspectrum"; }

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

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

std::string MzHarmonicSpectrum::getDescription(void) const
   { return "Harmonic Spectrogram"; }

int MzHarmonicSpectrum::getPluginVersion(void) const {
   #define P_VER    "200606190"
   #define P_NAME   "MzHarmonicSpectrum"

   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:
//

//////////////////////////////
//
// MzHarmonicSpectrum::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.
//

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



//////////////////////////////
//
// MzHarmonicSpectrum::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.
//

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

   OutputList       odlist;
   OutputDescriptor od;

   std::string s;
   char buffer[1024] = {0};
   int val;

   // First output channel: harmonic spectrogram
   od.name             = "spectrogram";
   od.description      = "Spectrogram";
   od.unit             = "bin";
   od.hasFixedBinCount = true;
   od.binCount         = mz_maxbin - mz_minbin + 1;
   for (int i=mz_minbin; i<=mz_maxbin; i++) {
      val = int((i+0.5) * getSrate() / mz_transformsize + 0.5);
      sprintf(buffer, "%d:%d", i, val);
      s = buffer;
      od.binNames.push_back(s);
   }
   if (mz_compress) {
      od.hasKnownExtents  = true;
      od.minValue         = 0.0;
      od.maxValue         = 1.0;
   } else {
      od.hasKnownExtents  = false;
   }
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);
   od.binNames.clear();

   // Second output channel: Spectral Power
   od.name             = "spectralpower";
   od.description      = "Spectral power";
   od.unit             = "";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;   // could set to true.
   // od.minValue      = 0.0;
   // od.maxValue      = 1.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);

   // Third output channel: Maximum value as central frequency of max bin.
   od.name             = "rawpitch";
   od.description      = "HS raw pitch estimate";
   od.unit             = "Hz";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;   // could set to true.
   // od.minValue      = 0.0;
   // od.maxValue      = 1.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);
   od.binNames.clear();

   // output channel: refined pitch estimate
   // to be added

   return odlist;
}



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

bool MzHarmonicSpectrum::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);

   if (getBlockSize() > mz_transformsize) {
      setBlockSize(mz_transformsize);
   }

   mz_method       = getParameterInt("method");
   mz_harmonics    = getParameterInt("harmonics");
   mz_compress     = getParameterInt("compress");

   double minfreq, maxfreq, a440interval;

   a440interval = getParameter("minpitch") - 69.0;
   minfreq = 440.0 * pow(2.0, a440interval / 12.0);
   mz_minbin = int(minfreq * mz_transformsize / getSrate());

   a440interval = getParameter("maxpitch") - 69.0;
   maxfreq = 440.0 * pow(2.0, a440interval / 12.0);
   mz_maxbin = int(maxfreq * mz_transformsize / getSrate() + 0.999);

   if (mz_minbin > mz_maxbin) {
      std::swap(mz_minbin, mz_maxbin);
   }

   if (mz_maxbin >= mz_transformsize) {
      std::cerr << "MzHarmonicSpectrum::initialize: maxbin size problem"
                << std::endl;
      std::cerr << "MzHarmonicSpectrum::initialize: maxbin = "
                << mz_maxbin << std::endl;
      std::cerr << "MzHarmonicSpectrum::initialize: transformsize = "
                << mz_transformsize << std::endl;
      return false;
   }

   if (mz_minbin < 0) {
      std::cerr << "MzHarmonicSpectrum::initialize: minbin size problem"
                << std::endl;
      std::cerr << "MzHarmonicSpectrum::initialize: minbin = "
                << mz_minbin << std::endl;
      return false;
   }

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

   return true;
}



//////////////////////////////
//
// MzHarmonicSpectrum::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))))
#define NONPEAKFACTOR 0.2

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

   if (getStepSize() <= 0) {
      std::cerr << "ERROR: MzHarmonicSpectrum::process: "
                << "MzHarmonicSpectrum 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;
   feature.values.resize(bincount);

   int spectrumsize = mz_transformsize / 2;
   std::vector<double> magnitudespectrum(spectrumsize);
   std::vector<mz_complex> complexspectrum(spectrumsize);
   std::vector<int>    harmoniccount(bincount);

   int i, j;
   for (i=0; i<bincount; i++) {
      harmoniccount[i] = 0;
   }

   int topbin = mz_maxbin * mz_harmonics;
   if (topbin >= spectrumsize) {
      topbin = spectrumsize - 1;
   }


   int index;
   std::vector<int> maxpeak(spectrumsize);
   mz_complex complexsum;
   mz_complex&cs = complexsum;
   int maxvaluebin = 0;
   double spectralpower = 0.0;

   switch (mz_method) {

      case METHOD_MAGNITUDE_SUMMATION:

         for (i=0; i<spectrumsize; i++) {
            magnitudespectrum[i] = mz_transformer.getSpectrumMagnitude(i);
            if (i > topbin) {
               // won't need the rest of the magnitude spectrum
               break;
            }
         }
```

```
         for (i=mz_minbin; i<=mz_maxbin; i++) {
            feature.values[i - mz_minbin] = 0.0;
            for (j=1; j<=mz_harmonics; j++) {
               index = i*j;
               if (index > spectrumsize) {
                  break;
               }
               feature.values[i - mz_minbin] += magnitudespectrum[index];
               harmoniccount[i - mz_minbin]++;
            }
         }

         // convert the harmonic spectrum to db
         for (i=0; i<bincount; i++) {
            if (feature.values[i] <= 0.0) {
               feature.values[i] = -120.0;
            } else {
               spectralpower += feature.values[i] / harmoniccount[i];
               feature.values[i] = 20.0
                     * log10(feature.values[i] / harmoniccount[i]);
            }
            if (feature.values[i] > feature.values[maxvaluebin]) {
               maxvaluebin = i;
            }
         }

         break;

      case METHOD_COMPLEX_SUMMATION:

         for (i=0; i<spectrumsize; i++) {
            complexspectrum[i] = mz_transformer.getSpectrum(i);
            if (i > topbin) {
               // won't need the rest of the magnitude spectrum
               break;
            }
         }

         for (i=mz_minbin; i<=mz_maxbin; i++) {
            complexsum.re = 0.0;
            complexsum.im = 0.0;
            for (j=1; j<=mz_harmonics; j++) {
               index = i*j;
               if (index > spectrumsize) {
                  break;
               }
               complexsum.re +=  complexspectrum[index].re;
               complexsum.im +=  complexspectrum[index].im;
               harmoniccount[i - mz_minbin]++;
            }
            feature.values[i - mz_minbin] = sqrt(cs.re*cs.re + cs.im*cs.im);
         }

         // convert the harmonic spectrum to db
         for (i=0; i<bincount; i++) {
            if (feature.values[i] <= 0.0) {
               feature.values[i] = -120.0;
            } else {
               spectralpower += feature.values[i] / harmoniccount[i];
               feature.values[i] = 20.0
                     * log10(feature.values[i] / harmoniccount[i]);
            }
            if (feature.values[i] > feature.values[maxvaluebin]) {
               maxvaluebin = i;
            }
         }

         break;

      case METHOD_MAGNITUDE_PRODUCT:
      default:

         for (i=0; i<spectrumsize; i++) {
            magnitudespectrum[i] = mz_transformer.getSpectrumMagnitude(i);
            if (i > topbin) {
               // won't need the rest of the magnitude spectrum
               break;
            }
         }

         for (i=mz_minbin; i<=mz_maxbin; i++) {
            feature.values[i - mz_minbin] = 1.0;
            for (j=1; j<=mz_harmonics; j++) {
               index = i*j;
               if (index > spectrumsize) {
                  break;
               }
               feature.values[i - mz_minbin] *= magnitudespectrum[index];
               harmoniccount[i - mz_minbin]++;
            }
         }

         // convert the harmonic spectrum to db
         for (i=0; i<bincount; i++) {
            if (feature.values[i] <= 0.0) {
               feature.values[i] = -120.0;
            } else {
               spectralpower += pow(feature.values[i], 1.0/harmoniccount[i]);
               feature.values[i] = 20.0 / harmoniccount[i]
                                   * log10(feature.values[i]);
            }
            if (feature.values[i] > feature.values[maxvaluebin]) {
               maxvaluebin = i;
            }
         }

   }

   double cen;
   if (mz_compress) {
      for (i=0; i<bincount; i++) {
         cen = -40.0 * i / bincount;
         feature.values[i] =
               sigmoidscale(feature.values[i], cen, 60);
      }
   }

   returnFeatures[0].push_back(feature);


   // process the second output from the plugin:

   feature.hasTimestamp = false;
   feature.values.clear();
   feature.values.push_back(spectralpower / (mz_maxbin - mz_minbin + 1));

   returnFeatures[1].push_back(feature);
```

```
   // process the third output from the plugin:

   float pitchestimate = float(maxvaluebin * getSrate() / mz_transformsize);
   feature.hasTimestamp = false;
   feature.values.clear();
   feature.values.push_back(pitchestimate);

   returnFeatures[2].push_back(feature);

   return returnFeatures;
}



//////////////////////////////
//
// MzHarmonicSpectrum::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.
//

MzHarmonicSpectrum::FeatureSet MzHarmonicSpectrum::getRemainingFeatures(void) {
   // no remaining features, so return a dummy feature
   return FeatureSet();
}



//////////////////////////////
//
// MzHarmonicSpectrum::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 MzHarmonicSpectrum::reset(void) {
   // no actions necessary to reset this plugin
}


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


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

void MzHarmonicSpectrum::generateMidiNoteList(std::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);
   }
}
```