```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Tue May  9 05:25:27 PDT 2006
// Last Modified: Sat May 20 05:41:31 PDT 2006 (added parameters)
// Last Modified: Thu May 25 22:27:53 PDT 2006 (added stereo diff & sensitivity)
// Last Modified: Thu Jul 20 06:54:55 PDT 2006 (added log/linear vertical scale)
// Filename:      MzNevermore.cpp
// URL:           http://sv.mazurka.org.uk/src/MzNevermore.cpp
// Documentation: http://sv.mazurka.org.uk/MzNevermore
// Syntax:        ANSI99 C++; vamp plugin
//
// Description:   Display audio signal in two dimensions.
//

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

#define DB_MIN -120

#define S_LINEAR 0
#define S_LOG    1

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

////////////////////////////
//
// MzNevermore::MzNevermore -- class constructor.
//

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



////////////////////////////
//
// MzNevermore::~MzNevermore -- class destructor.
//

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


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

////////////////////////////
//
// MzNevermore::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

MzNevermore::ParameterList MzNevermore::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 number of samples in the Fourier transform
   // Note: must be equal or greater than the window size.  This will
   // be enforced in the initialise() function.
   pd.name         = "transformsamples";
   pd.description  = "Transform size";
   pd.unit         = "samples";
   pd.minValue     = 2.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = 2048.0;
   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     = 30000.0;
   pd.defaultValue = 512.0;
   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: Vertical scaling type
pd.name         = "scale";
pd.description  = "Frequency scale";
pd.unit         = "";
   pd.valueNames.push_back("Hertz");
   pd.valueNames.push_back("Interval");
   pd.minValue     = 0.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   return pdlist;
}


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

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

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



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

size_t MzNevermore::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 MzNevermore::getName(void) const
   { return "mznevermore"; }

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

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

```
std::string MzNevermore::getDescription(void) const
   { return "Nevermore Spectrogram"; }

int MzNevermore::getPluginVersion(void) const {
   #define P_VER    "200606200"
   #define P_NAME   "MzNevermore"

   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:
//

////////////////////////////
//
// MzNevermore::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.
//

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



////////////////////////////
//
// MzNevermore::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.
//

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

   OutputList       odlist;
   OutputDescriptor od;

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

   // First and only output channel:
   od.name             = "spectrogram";
   od.description      = "Spectrogram";
   od.unit             = "bin";
   od.hasFixedBinCount = true;
   od.binCount         = mz_maxbin - mz_minbin + 1;

   if (getParameterInt("scale") == S_LINEAR) {
      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);
      }
   } else {

      int ii;
      double loghz;
      double hz;
      double minhz = mz_minbin * getSrate() / mz_transformsize;
      double maxhz = mz_maxbin * getSrate() / mz_transformsize;

      if (minhz < 1.0) { minhz = 1.0; }
      if (maxhz < 1.0) { maxhz = 1.0; }

      double minhzlog = log10(minhz) / log10(2.0);
      double maxhzlog = log10(maxhz) / log10(2.0);
      double logdiff = maxhzlog - minhzlog;

      for (int i=0; i<=(int)od.binCount; i++) {
         loghz = (double)i/(od.binCount-1.0) * logdiff + minhzlog;
         hz = pow(2.0, loghz);
         int hzint = int(hz + 0.5);
         ii = int(hz * mz_transformsize / getSrate());

         sprintf(buffer, "%d:%d", ii, hzint);
```

```
         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();


   return odlist;
}



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

bool MzNevermore::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_scale    = getParameterInt("scale");

   mz_transformsize = getParameterInt("transformsamples");
   if (mz_transformsize < getBlockSize()) {
      std::cerr << "MzNevermore::initialize: transform size problem"
                << std::endl;
      std::cerr << "MzNevermore::initialize: transformsize = "
                << mz_transformsize << std::endl;
      std::cerr << "MzNevermore::initialize: blocksize = "
                 << getBlockSize() << std::endl;
      return false;
   }

   mz_minbin     = getParameterInt("minbin");
   mz_maxbin     = getParameterInt("maxbin");

   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 << "MzNevermore::initialize : window is set to "
          << getParameterString("windowtype") << std::endl;


   return true;
}



////////////////////////////
//
// MzNevermore::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))))

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

   if (getStepSize() <= 0) {
      std::cerr << "ERROR: MzNevermore::process: "
                << "MzNevermore 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 i;
   double ii;
   if (mz_scale == S_LINEAR) {
      for (i=0; i<bincount; i++) {
         feature.values[i] = mz_transformer.getSpectrumMagnitudeDb(i);
      }
   } else { // logarithmic scaling
      std::vector<double> dbs;
      dbs.resize(bincount);

      for (i=0; i<bincount; i++) {
         dbs[i] = mz_transformer.getSpectrumMagnitudeDb(i);
         if (dbs[i] < DB_MIN) {
            dbs[i] = DB_MIN;
         }
      }

      double minhz = mz_minbin * getSrate() / mz_transformsize;
      double maxhz = mz_maxbin * getSrate() / mz_transformsize;
      if (minhz < 1.0) { minhz = 1.0; }
      if (maxhz < 1.0) { maxhz = 1.0; }
      double gincr = pow(maxhz / minhz, 1.0 / bincount);
      double hz;
      for (i=0; i<bincount; i++) {
         hz = minhz * pow(gincr, i);
         ii = hz * mz_transformsize / getSrate();
         if (ii > bincount -1) { ii = bincount - 1; }
         else if (ii < 0     ) { ii = 0           ; }

         feature.values[i] = dbs[int(ii+0.5)];
      }

   }

   if (mz_compress) {
      for (i=0; i<bincount; i++) {
         feature.values[i] = sigmoidscale(feature.values[i], -20, 80);
      }
   }

   returnFeatures[0].push_back(feature);

   return returnFeatures;
}



////////////////////////////
//
// MzNevermore::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.
//

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



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


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

// no non-interface functions
```