```
//
// 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)
// Filename:      MzChronogram.cpp
// URL:           http://sv.mazurka.org.uk/src/MzChronogram.cpp
// Documentation: http://sv.mazurka.org.uk/MzChronogram
// Syntax:        ANSI99 C++; vamp plugin
//
// Description:   Display audio signal in two dimensions.
//

#include "MzChronogram.h"

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

#define SENSIZE      2001
#define MZSTEREO     -2
#define MZSTEREODIFF -1


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

///////////////////////////////
//
// MzChronogram::MzChronogram -- class constructor.
//

MzChronogram::MzChronogram(float samplerate) : MazurkaPlugin(samplerate) {
   mz_whichchannel  =  MZSTEREO;
   mz_diffB         =  0;
   mz_lookup        = new float[SENSIZE];
}



///////////////////////////////
//
// MzChronogram::˜MzChronogram -- class destructor.
//

MzChronogram::˜MzChronogram() {
   delete [] mz_lookup;
}


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

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

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

   ParameterList       pdlist;
   ParameterDescriptor pd;

   // first parameter: The number of samples on the vertical axis
   pd.name         = "verticalperiod";
   pd.description  = "Vertical period";
   pd.unit         = "samples";
   pd.minValue     = 1.0;
   pd.maxValue     = 10000;
   pd.defaultValue = 100.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // second parameter: The Frequency of the period on the vertical axis
   pd.name         = "frequency";
   pd.description  = "or Frequency";
   pd.unit         = "Hz";
   pd.minValue     = 0.0;
   pd.maxValue     = 10000.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = false;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   // third parameter: The Chroma for a frequency (base-12 pitch name)
   pd.name         = "chroma";
   pd.description  = "or Chroma";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 12.0;
   pd.defaultValue = 12.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   // names for each quantized step:
   pd.valueNames.push_back("C");
   pd.valueNames.push_back("C#");
   pd.valueNames.push_back("D");
   pd.valueNames.push_back("D#");
   pd.valueNames.push_back("E");
   pd.valueNames.push_back("F");
   pd.valueNames.push_back("F#");
   pd.valueNames.push_back("G");
   pd.valueNames.push_back("G#");
   pd.valueNames.push_back("A");
   pd.valueNames.push_back("A#");
   pd.valueNames.push_back("B");
   pd.valueNames.push_back("");
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fourth parameter: The Octave of a chroma
   pd.name         = "octave";
   pd.description  = "+ Octave";
   pd.unit         = "";
   pd.minValue     = -1.0;
   pd.maxValue     = 9.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pd.valueNames.push_back("-1");
   pd.valueNames.push_back("0");
   pd.valueNames.push_back("1");
   pd.valueNames.push_back("2");
   pd.valueNames.push_back("3");
   pd.valueNames.push_back("4");
   pd.valueNames.push_back("5");
   pd.valueNames.push_back("6");
   pd.valueNames.push_back("7");
   pd.valueNames.push_back("8");
   pd.valueNames.push_back("9");
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fifth parameter: Which channel(s) to display
   pd.name         = "channelview";
   pd.description  = "Channel view";
   pd.unit         = "";
   pd.minValue     = -2.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = -2.0;
   pd.isQuantized  = true;
   pd.quantizeStep = 1.0;
   pd.valueNames.push_back("stereo");
   pd.valueNames.push_back("stereo difference");
   pd.valueNames.push_back("left channel");
   pd.valueNames.push_back("right channel");
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // sixth parameter: Amplitude sensitivity
   pd.name         = "sensitivity";
   pd.description  = "Sensitivity";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = false;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   return pdlist;
}


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

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

size_t MzChronogram::getPreferredStepSize(void) const {
   return getPreferredBlockSize();
}



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

size_t MzChronogram::getPreferredBlockSize(void) const {
```

```
   float output = 0.0;
   float frequency, chroma, octave;
   if (!isParameterAtDefault("chroma")) {
      chroma = getParameterInt("chroma");
      octave = getParameterInt("octave");
      frequency = 440.0 * pow(2.0, ((chroma-9) + 12*(octave-4))/12.0);
      output = getSrate() / frequency;
   } else if (!isParameterAtDefault("frequency")) {
      frequency = getParameter("frequency");
      output = getSrate() / frequency;
   } else {
      output = getParameter("verticalperiod");
   }

   output = std::min(output, getParameterMax("verticalperiod"));
   output = std::max(output, getParameterMin("verticalperiod"));

   return size_t(output + 0.5);
}


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

std::string MzChronogram::getName(void) const
   { return "mzchronogram"; }

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

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

std::string MzChronogram::getDescription(void) const
   { return "Chronogram"; }

int MzChronogram::getPluginVersion(void) const {
   #define P_VER    "200605270"
   #define P_NAME   "MzChronogram"

   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:
//

//////////////////////////////
//
// MzChronogram::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.
//

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



//////////////////////////////
//
// MzChronogram::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.
//

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

   OutputList       odlist;
   OutputDescriptor od;

   // First and only output channel:
   od.name             = "chronogram";
   od.description      = "Chronogram";
   od.unit             = "";
   od.hasFixedBinCount = true;
   if (getParameterInt("channelview") == MZSTEREO) {
```

```
      od.binCount = getBlockSize() * getChannelCount(); // stereo display
   } else {
      od.binCount = getBlockSize();                 // mono display
   }
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 0.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::OneSamplePerStep;
   // od.sampleRate    = 0.0;
   odlist.push_back(od);

   return odlist;
}



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

bool MzChronogram::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;
   }

   // Only one copy of a particular sample should be displayed.
   // If the step size is smaller than the block size, pretend
   // that the block size is the same as the step size.
   setBlockSize(std::min(stepsize, blocksize));
   setStepSize(stepsize);
   setChannelCount(channels);

   mz_whichchannel = getParameterInt("channelview");
   if (mz_whichchannel >= getChannelCount()) {
      mz_whichchannel = getChannelCount() - 1;
   }

   // If stereo (or higher),  channel 1 will be subtracted from channel 0
   // when doing stereo diff display.
   if (getChannelCount() >= 1) {
      mz_diffB = 1;
   } else {
      // monophonic input, so subtract from itself.
      mz_diffB = 0;
   }

   buildLookupTable(mz_lookup, SENSIZE, getParameter("sensitivity"));

   return true;
}



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

```
//
// MzChronogram::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).
//

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

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

   FeatureSet returnFeatures;
   Feature feature;

   if (mz_whichchannel == MZSTEREO ) {
      feature.values.resize(getChannelCount() * getBlockSize());
   } else {
      feature.values.resize(getBlockSize());
   }

   feature.hasTimestamp = false;

   // The Chronogram display has to be turned "upside-down" so that
   // steeper downward slopes indicate flatter notes, and steeper
   // higher slopes indicate sharper notes.

   int chan, samp;
   float sample;
   int i = 0;

   switch (mz_whichchannel) {
      case MZSTEREO:
         for (chan=getChannelCount()-1; chan>=0; chan--) {
            for (samp=getBlockSize()-1; samp>=0; samp--) {
               sample = inputbufs[chan][samp];
               if      (sample < -1.0) { sample = -1.0; }
               else if (sample > +1.0) { sample = +1.0; }
               sample = mz_lookup[int((sample+1)/2*(SENSIZE-1))];
               feature.values[i++] = sample;
            }
         }
         break;

      case MZSTEREODIFF:
         // stereo difference display
         for (samp=getBlockSize()-1; samp>=0; samp--) {
            sample = inputbufs[0][samp] - inputbufs[mz_diffB][samp];
            if      (sample < -2.0) { sample = -2.0; }
            else if (sample > +2.0) { sample = +2.0; }
            sample = mz_lookup[int((sample+2)/4*(SENSIZE-1))];
            feature.values[i++] = sample;
         }
         break;

      default:
         // monophonic display
         for (samp=getBlockSize()-1; samp>=0; samp--) {
            sample = inputbufs[mz_whichchannel][samp];
            if      (sample < -1.0) { sample = -1.0; }
            else if (sample > +1.0) { sample = +1.0; }
            sample = mz_lookup[int((sample+1)/2*(SENSIZE-1))];
            feature.values[i++] = sample;
         }
   }

   returnFeatures[0].push_back(feature);

   return returnFeatures;
}



//////////////////////////////
//
// MzChronogram::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.
//

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



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


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


//////////////////////////////
//
// MzChronogram::buildLookupTable -- Compresses the audio so that smaller
//     amplitudes can be seen as well (or nearly as well) as
//     larger amplitudes.  If the sensitivity is 0.0, then the sound data
//     is unaltered.  If the sensitivity is 1.0, then most posititive
//     amplitudes are mapped to positive max, and negative amplitudes
//     are mapped to negative max.
//


#define MZSIG(x,w)      (1.0/(1.0+exp(-(x)/(w))))
#define MZSINSIG(x,w)   (MZSIG(x,w) + sin((x)*(w)) * MZSIG(1,(w)) - 0.5)
#define MZSCALING(x,w)  (MZSINSIG(x,w)/MZSINSIG(1,w) - 0.04 * sin(M_PI * (x)))

void MzChronogram::buildLookupTable(float* buffer, int size, float sensitivity)
{

   // flip and scale the sensitivity factor
   if      (sensitivity > 1.0) { sensitivity = 1.0; }
   else if (sensitivity < 0.0) { sensitivity = 0.0; }
   double weight = (1.0 - pow(double(sensitivity), 0.125)) * 0.84 + 0.005;

   if (sensitivity == 0.0) {
      for (int i=0; i<size; i++) {
         buffer[i] = float(2.0 * i/(size-1.0) - 1.0);
      }
   } else {
      for (int i=0; i<size; i++) {
         buffer[i] = float(MZSCALING(2.0 * i/(size-1.0) - 1.0, weight));
      }
   }
}
```