```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Sat May 13 12:19:13 PDT 2006
// Last Modified: Sat May 20 15:24:51 PDT 2006 (added parameter control)
// Last Modified: Tue Jun 20 19:42:50 PDT 2006 (features and/or bugs added)
// Last Modified: Sun Jul  9 06:42:47 PDT 2006 (automated scaled power slope)
// Last Modified: Sun May  6 01:48:58 PDT 2007 (upgraded to vamp 1.0)
// Filename:      MzPowerCurve.cpp
// URL:           http://sv.mazurka.org.uk/src/MzPowerCurve.cpp
// Documentation: http://sv.mazurka.org.uk/MzPowerCurve
// Syntax:        ANSI99 C++; vamp 0.9 plugin
//
// Description:   Calculate the power of an audio signal as it changes
//                over time.
//

// Defines used in getPluginVersion():
#define P_VER    "200607100"
#define P_NAME   "MzPowerCurve"

#include "MzPowerCurve.h"
#include "MazurkaWindower.h"

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

#define ZEROLOG -120.0

#define FILTER_SYMMETRIC        0
#define FILTER_FORWARD          1
#define FILTER_BACKWARD         2


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

//////////////////////////////
//
// MzPowerCurve::MzPowerCurve -- class constructor.
//

MzPowerCurve::MzPowerCurve(float samplerate) : MazurkaPlugin(samplerate) {
   mz_windowsum = 1.0;
}



//////////////////////////////
//
// MzPowerCurve::~MzPowerCurve -- class destructor.
//

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



////////////////////////////////////////////////////////////
//
// optional polymorphic parameter functions inherited from PluginBase:
//
// Note that the getParameter() and setParameter() polymorphic functions
// are handled in the MazurkaPlugin class.
//

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

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

   ParameterList       pdlist;
   ParameterDescriptor pd;

   // first parameter: The size of the analysis window in milliseconds
   pd.identifier   = "windowsize";
   pd.name         = "Window size";
   pd.unit         = "ms";
   pd.minValue     = 10.0;
   pd.maxValue     = 10000.0;
   pd.defaultValue = 10.0;
   pd.isQuantized  = 0;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   // second parameter: The hop size between windows in milliseconds
   pd.identifier   = "hopsize";
   pd.name         = "Window hop size";
   pd.unit         = "ms";
   pd.minValue     = 1.0;
   pd.maxValue     = 10000.0;
   pd.defaultValue = 10.0;
   pd.isQuantized  = 0;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   // third parameter: Windowing method
   pd.identifier   = "window";
   pd.name         = "Weighting window";
   pd.unit         = "";
   pd.minValue     = 1.0;
   MazurkaWindower::getWindowList(pd.valueNames);
   pd.maxValue     = pd.valueNames.size();
   pd.defaultValue = 1.0;
   pd.isQuantized  = 1;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);
   pd.valueNames.clear();

   // fourth parameter: Factor for exponential smoothing filter
   pd.identifier   = "smoothingfactor";
   pd.name         = "Smoothing\n  (outputs 2-4)";
   pd.unit         = "";
   pd.minValue     = -1.0;
   pd.maxValue     = 1.0;
   pd.defaultValue = 0.2;
   pd.isQuantized  = 0;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   // fifth parameter: Filtering method
   pd.identifier   = "filtermethod";
   pd.name         = "Filter method\n  (outputs 2-4)";
   pd.unit         = "";
   pd.minValue     = 0.0;
```

```
   pd.maxValue     = 2.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = 1;
   pd.quantizeStep = 1.0;
   pd.valueNames.push_back("Symmetric");
   pd.valueNames.push_back("Forward");
   pd.valueNames.push_back("Reverse");
   pdlist.push_back(pd);
   pd.valueNames.clear();

/*

   // sixth parameter: Cut-off threshold for scaled power slope
   pd.identifier   = "cutoffthreshold";
   pd.name         = "Cut-off threshold\n  (output 4 only)";
   pd.unit         = "dB";
   pd.minValue     = -100.0;
   pd.maxValue     = 10.0;
   pd.defaultValue = -40.0;
   pd.isQuantized  = 0;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

   // seventh parameter: The width of the cut-off transition region
   pd.identifier   = "cutoffwidth";
   pd.name         = "Cut-off width\n  (output 4 only)";
   pd.unit         = "dB";
   pd.minValue     = 1.0;
   pd.maxValue     = 100.0;
   pd.defaultValue = 20.0;
   pd.isQuantized  = 0;
   // pd.quantizeStep = 0.0;
   pdlist.push_back(pd);

*/

   return pdlist;
}


//////////////////////////////////////////////////////////////////////////////
//
// optional polymorphic functions inherited from Plugin:
//

//////////////////////////////
//
// MzPowerCurve::getPreferredStepSize --
//

size_t MzPowerCurve::getPreferredStepSize(void) const {
   return size_t(getParameter("hopsize")*getSrate()/1000.0 + 0.5);
}



//////////////////////////////
//
// MzPowerCurve::getPreferredBlockSize --
//

size_t MzPowerCurve::getPreferredBlockSize(void) const {
   return size_t(getParameter("windowsize")*getSrate()/1000.0 + 0.5);
}
```

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

std::string MzPowerCurve::getIdentifier(void) const
   { return "mzpowercurve"; }

std::string MzPowerCurve::getName(void) const
   { return "Power Curve"; }

std::string MzPowerCurve::getDescription(void) const
   { return "Power Curve"; }

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

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

int MzPowerCurve::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:
//

//////////////////////////////
//
// MzPowerCurve::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.
//

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



//////////////////////////////
//
// MzPowerCurve::getOutputDescriptors -- return a list describing
//    each of the available outputs for the object.  OutputList
//    is defined in the file vamp-sdk/Plugin.h:
//
// .identifier       == short name of output for computer use.  Must not
//                      contain spaces or punctuation.
```

```
// .name             == 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.
//

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

   OutputList       list;
   OutputDescriptor od;

   // First output channel:
   od.identifier       = "rawpower";
   od.name             = "Raw Power";
   od.unit             = "dB";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 0.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::VariableSampleRate;
   // od.sampleRate    = 0.0;
   list.push_back(od);

   // Second output channel: (smoothed data)
   od.identifier       = "smoothpower";
   od.name             = "Smoothed Power";
   od.unit             = "dB";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 0.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::VariableSampleRate;
   // od.sampleRate    = 0.0;
   list.push_back(od);

   // Third output channel: (smoothed data slope)
   od.identifier       = "smoothpowerslope";
   od.name             = "Smoothed Power Slope";
   od.unit             = "dB slope";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 0.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::VariableSampleRate;
   // od.sampleRate    = 0.0;
   list.push_back(od);

   // Fourth output channel: (smoothed data slope product)
   od.identifier       = "powerslopeproduct";
   od.name             = "Scaled Power Slope";
   od.unit             = "dB slope";
   od.hasFixedBinCount = true;
   od.binCount         = 1;
   od.hasKnownExtents  = false;
   // od.minValue      = 0.0;
   // od.maxValue      = 0.0;
   od.isQuantized      = false;
   // od.quantizeStep  = 1.0;
   od.sampleType       = OutputDescriptor::VariableSampleRate;
   // od.sampleRate    = 0.0;
   list.push_back(od);

   return list;
}



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

bool MzPowerCurve::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_window.makeWindow(getParameterString("window"), getBlockSize());

   if (mz_window.getWindowType() == "Rectangular" ||
       mz_window.getWindowType() == "Unknown") {
```

```
      mz_windowsum = 1.0;
   } else {
      mz_windowsum = mz_window.getWindowSum() / mz_window.getSize();
   }

   switch (getParameterInt("filtermethod")) {
      case FILTER_FORWARD:
         mz_filterforward  = 1;
         mz_filterbackward = 0;
         break;
      case FILTER_BACKWARD:
         mz_filterforward  = 0;
         mz_filterbackward = 1;
         break;
      case FILTER_SYMMETRIC:
      default:
         mz_filterforward  = 1;
         mz_filterbackward = 1;
   }

   rawpower.clear();

   return true;
}



//////////////////////////////
//
// MzPowerCurve::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).
//

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

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

   // calculate the raw power for the given input block of audio.
   // Further processing of the raw power data will be done in
   // the getRemainingFeatures() function.

   int    i;
   double value;
   double sum = 0.0;

   if (mz_window.getWindowType() == "Unknown" ||
       mz_window.getWindowType() == "Rectangular") {
      // do unweighted power calculation
      for (i=0; i<getBlockSize(); i++) {
         value = inputbufs[0][i];
         sum += value * value;
      }
   } else {
      // do weighted power calculation
      for (i=0; i<getBlockSize(); i++) {
         value = inputbufs[0][i];
         sum += value * value * mz_window[i];
      }
   }

   float power;
   if (sum <= 0.0) {
      power = ZEROLOG;
   } else {
      power = 10.0 * log10(sum/getBlockSize()/mz_windowsum);
   }

   FeatureSet returnFeatures;
   Feature    feature;

   // center the location of the power measurement at the
   // middle of the analysis region rather than at the beginning.
   feature.hasTimestamp = true;
   feature.timestamp    = timestamp +
               Vamp::RealTime::fromSeconds(0.5 * getBlockSize()/getSrate());

   feature.values.push_back(power);

   // also store the power measurement for later processing in
   // getRemainingFeatures():
   rawpower.push_back(power);

   returnFeatures[0].push_back(feature);

   return returnFeatures;
}



//////////////////////////////
//
// MzPowerCurve::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.
//

MzPowerCurve::FeatureSet MzPowerCurve::getRemainingFeatures(void) {

   int i;
   double filterk = getParameter("smoothingfactor");
   double oneminusk = 1.0 - filterk;
   int size = rawpower.size();
   std::vector<double> smoothpower(size, true);

   // Difference equation for smoothing: y[n] = k * x[n] + (1-k) * y[n-1]

   // do reverse smoothing: time symmetric filtering to remove filter delays
   if (mz_filterbackward && mz_filterforward) {

      // reverse filtering first
      smoothpower[size-1] = rawpower[size-1];
```

```
      for (i=size-2; i>=0; i--) {
         smoothpower[i] = filterk*rawpower[i] + oneminusk*smoothpower[i+1];
      }

      // then forward filtering
      for (i=1; i<size; i++) {
         smoothpower[i] = filterk*smoothpower[i] + oneminusk*smoothpower[i-1];
      }
   } else if (mz_filterbackward) {
      smoothpower[size-1] = rawpower[size-1];
      for (i=size-2; i>=0; i--) {
         smoothpower[i] = filterk * rawpower[i] + oneminusk * smoothpower[i+1];
      }
   } else if (mz_filterforward) {
      // do forward smoothing:
      smoothpower[0] = rawpower[0];
      for (i=1; i<size; i++) {
         smoothpower[i] = filterk * rawpower[i] + oneminusk * smoothpower[i-1];
      }
   } else {
      smoothpower = rawpower;
   }


   FeatureSet returnFeatures;
   Feature    feature;
   feature.hasTimestamp = true;

   // process output features #2: smoothed power data

   double timeinsec;
   for (i=0; i<size; i++) {
      timeinsec = (getBlockSize()*0.5 + i * getStepSize())/getSrate();
      feature.timestamp = Vamp::RealTime::fromSeconds(timeinsec);
      feature.values.clear();
      feature.values.push_back(float(smoothpower[i]));
      returnFeatures[1].push_back(feature);
   }

   // process output features #3 here: smoothed power slope

   std::vector<double> smoothslope(size-1, true);
   for (i=0; i<size-1; i++) {
      smoothslope[i] = smoothpower[i+1] - smoothpower[i];
      // adding additional 1/2 of the block size to center the peaks
      // at attack points
      timeinsec = (getBlockSize()*0.5 + (i+0.5)*getStepSize())/getSrate();
      feature.timestamp = Vamp::RealTime::fromSeconds(timeinsec);
      feature.values.clear();
      feature.values.push_back(float(smoothslope[i]));
      returnFeatures[2].push_back(feature);
   }

   // process output features #4 here: scaled smoothed power slope

   double mean = getMean(smoothpower);
   double sd   = getStandardDeviation(smoothpower);

   std::vector<double> productslope(size-1, true);
   double cutoff = mean - 1.5 * sd;
   double width  = sd / 2.0;
   double scaling;
   for (i=0; i<size-1; i++) {
      scaling = (smoothpower[i] - cutoff)/width;
      scaling = 1.0 / (1.0 + pow(2.718281828, -scaling)); //sigmoid function
      productslope[i] = smoothslope[i] * scaling;
      // adding additional 1/2 of the block size to center the peaks
      // at attack points
      timeinsec = (getBlockSize()*0.5 + (2*i+1)*getStepSize())/(2.0*getSrate());
      feature.timestamp = Vamp::RealTime::fromSeconds(timeinsec);
      feature.values.clear();
      feature.values.push_back(float(productslope[i]));
      returnFeatures[3].push_back(feature);
   }

   return returnFeatures;
}



//////////////////////////////
//
// MzPowerCurve::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 MzPowerCurve::reset(void) {
   rawpower.clear();
}



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

//////////////////////////////
//
// MzPowerCurve::getMean --
//

double MzPowerCurve::getMean(std::vector<double>& data) {
   double sum;
   int i;

   sum = 0.0;
   for (i=0; i<(int)data.size(); i++) {
      sum += data[i];
   }
   return (sum / data.size());
}



//////////////////////////////
//
// MzPowerCurve::getStandardDeviation --
//

double MzPowerCurve::getStandardDeviation(std::vector<double>& data) {

   double mean = getMean(data);
   double sum  = 0.0;
```

```
   double value;
   int i;

   for (i=0; i<(int)data.size(); i++) {
      value = data[i] - mean;
      sum += value * value;
   }

   return sqrt(sum / data.size());
}
```