```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Sun Jun 11 21:04:49 PDT 2006
// Last Modified: Fri Jun 23 01:45:07 PDT 2006 (subclassed to MazurkaPlugin)
// Filename:      MzSpectrogramFFTW.cpp
// URL:           http://sv.mazurka.org.uk/src/MzSpectrogramFFTW.cpp
// Documentation: http://sv.mazurka.org.uk/MzSpectrogramFFTW
// Syntax:        ANSI99 C++; vamp 0.9 plugin
//
// Description:   Demonstration of how to create spectral data from time data
//                supplied by the host application using the FFTW library
//                for Fourier Transforms.
//


#include "MzSpectrogramFFTW.h"

#include <math.h>


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

///////////////////////////////
//
// MzSpectrogramFFTW::MzSpectrogramFFTW -- class constructor.
//

MzSpectrogramFFTW::MzSpectrogramFFTW(float samplerate) :
      MazurkaPlugin(samplerate) {
   mz_minbin    = 0;
   mz_maxbin    = 0;
   mz_wind_buff = NULL;
}



///////////////////////////////
//
// MzSpectrogramFFTW::~MzSpectrogramFFTW -- class destructor.
//

MzSpectrogramFFTW::~MzSpectrogramFFTW() {
   delete [] mz_wind_buff;
}


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

std::string MzSpectrogramFFTW::getName(void) const
   { return "mzspectrogramfftw"; }

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

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

std::string MzSpectrogramFFTW::getDescription(void) const
   { return "FFTW Spectrogram"; }

int MzSpectrogramFFTW::getPluginVersion(void) const {
   #define P_VER    "200606260"
   #define P_NAME   "MzSpectrogramFFTW"

   const char *v = "@@VampPluginID@" P_NAME "@" P_VER "@" __DATE__ "@@";
   if (v[0] != '@') { std::cerr << v << std::endl; return 0; }
   return atol(P_VER);
}



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

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

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

   ParameterList       pdlist;
   ParameterDescriptor pd;

   // first parameter: The minimum spectral bin to display
   pd.name         = "minbin";
   pd.description  = "Minimum\nfrequency\nbin";
   pd.unit         = "";
   pd.minValue     = 0.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = 0.0;
   pd.isQuantized  = 1;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   // second parameter: The maximum spectral bin to display
   pd.name         = "maxbin";
   pd.description  = "Maximum\nfrequency\nbin";
   pd.unit         = "";
   pd.minValue     = -1.0;
   pd.maxValue     = 30000.0;
   pd.defaultValue = -1.0;
   pd.isQuantized  = 1;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   return pdlist;
}


////////////////////////////////////////////////////////////
//
// required polymorphic functions inherited from Plugin:
//

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

```
// MzSpectrogramFFTW::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.
//

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



//////////////////////////////
//
// MzSpectrogramFFTW::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.
//

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

   OutputList       list;
   OutputDescriptor od;

   // First and only output channel:
   od.name             = "magnitude";
   od.description      = "Magnitude Spectrum";
   od.unit             = "decibels";
   od.hasFixedBinCount = true;
   od.binCount         = mz_maxbin - mz_minbin + 1;
   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;
   list.push_back(od);


   return list;
}



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

bool MzSpectrogramFFTW::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);
   setBlockSize(blocksize);
   setStepSize(stepsize);

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

   if (mz_minbin >= getBlockSize()/2) { mz_minbin = getBlockSize()/2-1; }
   if (mz_maxbin >= getBlockSize()/2) { mz_maxbin = getBlockSize()/2-1; }
   if (mz_maxbin <  0)                { mz_maxbin = getBlockSize()/2-1; }
   if (mz_maxbin <  mz_minbin)        { std::swap(mz_minbin, mz_maxbin); }

   // The signal size/transform size are equivalent for this
   // plugin but the FFTW can handle any size transform.
   // If the size of the transform is a multiple of small
   // prime numbers the FFT will be used, otherwise it will
   // be slow (when block size=1021 for example).

   mz_transformer.setSize(getBlockSize());
   delete [] mz_wind_buff;
   mz_wind_buff = new double[getBlockSize()];
   makeHannWindow(mz_wind_buff, getBlockSize());
```

```
   return true;
}



///////////////////////////////
//
// MzSpectrogramFFTW::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 ABSSQUARE(x, y) ((x)*(x) + (y)*(y))
#define ZEROLOG         -120.0

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

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

   // first window the input signal frame
   windowSignal(mz_transformer, mz_wind_buff, inputbufs[0]);

   // then calculate the complex DFT spectrum.
   mz_transformer.doTransform();

   // return the spectral magnitude frame to the host application:

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

   float magnitude;
   for (int i=mz_minbin; i<=mz_maxbin; i++) {
      magnitude = (float)mz_transformer.getSpectrumMagnitudeDb(i);
      feature.values.push_back(magnitude);
   }

   returnFeatures[0].push_back(feature);

   return returnFeatures;
}



///////////////////////////////
//
// MzSpectrogramFFTW::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.
//

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



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


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


///////////////////////////////
//
// MzSpectrogramFFTW::makeHannWindow -- create a raised cosine (Hann)
//     window.
//

void MzSpectrogramFFTW::makeHannWindow(double* output, int blocksize) {
   for (int i=0; i<blocksize; i++) {
      output[i] = 0.5 - 0.5 * cos(2.0 * M_PI * i/blocksize);
   }
}



///////////////////////////////
//
// MzSpectrogramFFTW::windowSignal -- multiply the time signal
//     by the analysis window to prepare for transformation.
//

void MzSpectrogramFFTW::windowSignal(MazurkaTransformer& transformer,
      double* window, float* input) {
   int blocksize = transformer.getSize();
   for (int i=0; i<blocksize; i++) {
      transformer.signalNonCausal(i) = window[i] * double(input[i]);
   }
}
```