```
//
// Programmer:    Craig Stuart Sapp <craig@ccrma.stanford.edu>
// Creation Date: Fri May 12 23:41:37 PDT 2006
// Last Modified: Fri Jun 23 00:33:33 PDT 2006 (subclassed to MazurkaPlugin)
// Filename:      MzSpectrogramClient.cpp
// URL:           http://sv.mazurka.org.uk/src/MzSpectrogramClient.cpp
// Documentation: http://sv.mazurka.org.uk/MzSpectrogramClient
// Syntax:        ANSI99 C++; vamp 0.9 plugin
//
// Description:   Demonstration of how to create spectral data from time data
//                supplied by the host application.
//


#include "MzSpectrogramClient.h"

#include <math.h>


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

//////////////////////////////
//
// MzSpectrogramClient::MzSpectrogramClient -- class constructor.
//

MzSpectrogramClient::MzSpectrogramClient(float samplerate) :
      MazurkaPlugin(samplerate) {
   mz_signalbuffer = NULL;
   mz_windbuffer   = NULL;
   mz_freqbuffer   = NULL;

   mz_minbin     = 0;
   mz_maxbin     = 0;
}



//////////////////////////////
//
// MzSpectrogramClient::~MzSpectrogramClient -- class destructor.
//

MzSpectrogramClient::~MzSpectrogramClient() {
   delete [] mz_signalbuffer;
   delete [] mz_windbuffer;
   delete [] mz_freqbuffer;
}


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

std::string MzSpectrogramClient::getName(void) const
   { return "mzspectrogramclient"; }

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

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

std::string MzSpectrogramClient::getDescription(void) const
   { return "Client Spectrogram"; }

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

   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.
//

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

MzSpectrogramClient::ParameterList
MzSpectrogramClient::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     = 50000.0;
   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     = 50000.0;
   pd.defaultValue = -1.0;
   pd.isQuantized  = 1;
   pd.quantizeStep = 1.0;
   pdlist.push_back(pd);

   return pdlist;
}



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

```
//

//////////////////////////////
//
// MzSpectrogramClient::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.
//

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



//////////////////////////////
//
// MzSpectrogramClient::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.
//

MzSpectrogramClient::OutputList
MzSpectrogramClient::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;
}



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

#define ISPOWEROFTWO(x) ((x)&&!(((x)-1)&(x)))

bool MzSpectrogramClient::initialise(size_t channels, size_t stepsize,
      size_t blocksize) {

   if (channels < getMinChannelCount() || channels > getMaxChannelCount()) {
      return false;
   }

   // The signal size/transform size are equivalent for this plugin, and
   // must be a power of two in order to use the given FFT algorithm.
   // Give up if the blocksize is not a power of two.
   if (!ISPOWEROFTWO(blocksize)) {
      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_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); }


   delete [] mz_signalbuffer;
   mz_signalbuffer = new double[getBlockSize()];

   // the mz_freqbuffer is twice the length of the input signal because
   // it will store the complex frequency bins which consist of pairs
   // of real and imaginary numbers.
   delete [] mz_freqbuffer;
   mz_freqbuffer = new double[getBlockSize() * 2];

   delete [] mz_windbuffer;
   mz_windbuffer = new double[getBlockSize()];

   // calculate the analysis window which will be applied to the
   // signal before it is transformed.

   return true;
}



//////////////////////////////
//
// MzSpectrogramClient::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 ZEROLOG -120.0

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

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

   // first window the input signal frame
   windowSignal(mz_signalbuffer, mz_windbuffer, inputbufs[0], getBlockSize());

   // then calculate the complex DFT spectrum. (note this fft
   // function will automatically rotate the time buffer 1/2 of
   // a frame to place the center of the windowed signal at index 0).


   // Rotate the signal so the first element in the array is in the
   // middle of the array (or slightly higher for even sizes).
   // This code only works for even sizes (or size-1).  But that is
   // OK because the initialise() function requires the size to
   // be a power of two.
   int i;
   int halfsize   = getBlockSize()/2;
   for (i=0; i<halfsize; i++) {
      std::swap(mz_signalbuffer[i], mz_signalbuffer[halfsize+i]);
   }

   // Calculate the complex DFT spectrum.
   fft(getBlockSize(), mz_signalbuffer, NULL, mz_freqbuffer,
         mz_freqbuffer + getBlockSize());

   // return the spectral frame to the host application

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

   double* real = mz_freqbuffer;
   double* imag = mz_freqbuffer + getBlockSize()/2;
   float magnitude;  // temporary holding space for magnitude value

   for (i=mz_minbin; i<=mz_maxbin; i++) {
      magnitude = real[i] * real[i] + imag[i] * imag[i];

      // convert to decibels:
      if (magnitude <= 0) { magnitude = ZEROLOG; }
      else                { magnitude = 10.0 * log10(magnitude); }

      feature.values.push_back(magnitude);
   }

   // Append new frame of data onto the output channel
   // specified in the function getOutputDescriptors():
   returnFeatures[0].push_back(feature);

   return returnFeatures;
}



//////////////////////////////
//
// MzSpectrogramClient::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.
//

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



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


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


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

void MzSpectrogramClient::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);
   }
}



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

void MzSpectrogramClient::windowSignal(double* output, double* window,
      float* input, int blocksize) {
   for (int i=0; i<blocksize; i++) {
      output[i] = window[i] * double(input[i]);
   }
}




//////////////////////////////
//
//  MzSpectrogramClient::fft -- calculate the Fast Fourier Transform.
//     Modified from the vamp plugin sdk fft() function in
//     host/vamp-simple-host.cpp which was in turn adapted from the
//     FFT implementation of Don Cross:
//  http://www.mathsci.appstate.edu/~wmcb/FFT/Code/fft.p
//  http://cs.marlboro.edu/term/fall01/computation/fourier/fft_c_code/TEMP/FOURIERD.
C
//
//     Note that this fft is about 4 times slower than the
//     FFTW (http://www.fftw.org) implementation of the FFT, so if you
//     want speed, you should use FFTW to calculate the DFT as is done
//     in the Sonic Visualiser host application.
//

void MzSpectrogramClient::fft(int n, double *ri, double *ii, double *ro,
      double *io) {
    if (!ri || !ro || !io) return;

    int bits;
    int i, j, k, m;
    int blockSize, blockEnd;
    double tr, ti;

    // Twiddle the time input to move center of window to index 0.
    if (n & (n-1)) return;
    double angle = 2.0 * M_PI;

    for (i = 0; ; ++i) {
        if (n & (1 << i)) {
            bits = i;
            break;
        }
    }

    static int tableSize = 0;
    static int *table = 0;

    if (tableSize != n) {
        delete[] table;
        table = new int[n];
        for (i = 0; i < n; ++i) {
            m = i;
            for (j = k = 0; j < bits; ++j) {
                k = (k << 1) | (m & 1);
                m >>= 1;
            }
            table[i] = k;
        }
        tableSize = n;
    }

    if (ii) {
        for (i = 0; i < n; ++i) {
            ro[table[i]] = ri[i];
            io[table[i]] = ii[i];
        }
    } else {
        for (i = 0; i < n; ++i) {
            ro[table[i]] = ri[i];
            io[table[i]] = 0.0;
        }
    }

    blockEnd = 1;

    for (blockSize = 2; blockSize <= n; blockSize <<= 1) {
        double delta = angle / (double)blockSize;
        double sm2 = -sin(-2 * delta);
        double sm1 = -sin(-delta);
        double cm2 = cos(-2 * delta);
        double cm1 = cos(-delta);
        double w = 2 * cm1;
        double ar[3], ai[3];

        for (i = 0; i < n; i += blockSize) {
            ar[2] = cm2;
            ar[1] = cm1;
            ai[2] = sm2;
            ai[1] = sm1;
            for (j = i, m = 0; m < blockEnd; j++, m++) {
                ar[0] = w * ar[1] - ar[2];
```

```
                ar[2] = ar[1];
                ar[1] = ar[0];
                ai[0] = w * ai[1] - ai[2];
                ai[2] = ai[1];
                ai[1] = ai[0];
                k = j + blockEnd;
                tr = ar[0] * ro[k] - ai[0] * io[k];
                ti = ar[0] * io[k] + ai[0] * ro[k];
                ro[k] = ro[j] - tr;
                io[k] = io[j] - ti;
                ro[j] += tr;
                io[j] += ti;
            }
        }

        blockEnd = blockSize;
    }

}
```