```
/****************************************************************************
*
* NAME: smbPitchShift.cpp
* VERSION: 1.2
* HOME URL: http://blogs.zynaptiq.com/bernsee
* KNOWN BUGS: none
*
* SYNOPSIS: Routine for doing pitch shifting while maintaining
* duration using the Short Time Fourier Transform.
*
* DESCRIPTION: The routine takes a pitchShift factor value which is between 0.5
* (one octave down) and 2. (one octave up). A value of exactly 1 does not change
* the pitch. numSampsToProcess tells the routine how many samples in indata[0...
* numSampsToProcess-1] should be pitch shifted and moved to outdata[0 ...
* numSampsToProcess-1]. The two buffers can be identical (ie. it can process the
* data in-place). fftFrameSize defines the FFT frame size used for the
* processing. Typical values are 1024, 2048 and 4096. It may be any value <=
* MAX_FRAME_LENGTH but it MUST be a power of 2. osamp is the STFT
* oversampling factor which also determines the overlap between adjacent STFT
* frames. It should at least be 4 for moderate scaling ratios. A value of 32 is
* recommended for best quality. sampleRate takes the sample rate for the signal
* in unit Hz, ie. 44100 for 44.1 kHz audio. The data passed to the routine in
* indata[] should be in the range [-1.0, 1.0), which is also the output range
* for the data, make sure you scale the data accordingly (for 16bit signed integers
* you would have to divide (and multiply) by 32768).
*
* COPYRIGHT 1999-2015 Stephan M. Bernsee <s.bernsee [AT] zynaptiq [DOT] com>
*
*                                               The Wide Open License (WOL)
*
* Permission to use, copy, modify, distribute and sell this software and its
* documentation for any purpose is hereby granted without fee, provided that
* the above copyright notice and this license appear in all source copies.
* THIS SOFTWARE IS PROVIDED "AS IS" WITHOUT EXPRESS OR IMPLIED WARRANTY OF
* ANY KIND. See http://www.dspguru.com/wol.htm for more information.
*
*****************************************************************************/

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

#define M_PI 3.14159265358979323846
#define MAX_FRAME_LENGTH 8192

void smbFft(float *fftBuffer, long fftFrameSize, long sign);
double smbAtan2(double x, double y);


// -----------------------------------------------------------------------------------------------------------------


void smbPitchShift(float pitchShift, long numSampsToProcess, long fftFrameSize, long osamp, float sampleRate, float *indata, float *outdata)
/*
	Routine smbPitchShift(). See top of file for explanation
	Purpose: doing pitch shifting while maintaining duration using the Short
	Time Fourier Transform.
	Author: (c)1999-2015 Stephan M. Bernsee <s.bernsee [AT] zynaptiq [DOT] com>
*/
{

	static float gInFIFO[MAX_FRAME_LENGTH];
```

```
	static float gOutFIFO[MAX_FRAME_LENGTH];
	static float gFFTworksp[2*MAX_FRAME_LENGTH];
	static float gLastPhase[MAX_FRAME_LENGTH/2+1];
	static float gSumPhase[MAX_FRAME_LENGTH/2+1];
	static float gOutputAccum[2*MAX_FRAME_LENGTH];
	static float gAnaFreq[MAX_FRAME_LENGTH];
	static float gAnaMagn[MAX_FRAME_LENGTH];
	static float gSynFreq[MAX_FRAME_LENGTH];
	static float gSynMagn[MAX_FRAME_LENGTH];
	static long gRover = false, gInit = false;
	double magn, phase, tmp, window, real, imag;
	double freqPerBin, expct;
	long i,k, qpd, index, inFifoLatency, stepSize, fftFrameSize2;

	/* set up some handy variables */
	fftFrameSize2 = fftFrameSize/2;
	stepSize = fftFrameSize/osamp;
	freqPerBin = sampleRate/(double)fftFrameSize;
	expct = 2.*M_PI*(double)stepSize/(double)fftFrameSize;
	inFifoLatency = fftFrameSize-stepSize;
	if (gRover == false) gRover = inFifoLatency;

	/* initialize our static arrays */
	if (gInit == false) {
		memset(gInFIFO, 0, MAX_FRAME_LENGTH*sizeof(float));
		memset(gOutFIFO, 0, MAX_FRAME_LENGTH*sizeof(float));
		memset(gFFTworksp, 0, 2*MAX_FRAME_LENGTH*sizeof(float));
		memset(gLastPhase, 0, (MAX_FRAME_LENGTH/2+1)*sizeof(float));
		memset(gSumPhase, 0, (MAX_FRAME_LENGTH/2+1)*sizeof(float));
		memset(gOutputAccum, 0, 2*MAX_FRAME_LENGTH*sizeof(float));
		memset(gAnaFreq, 0, MAX_FRAME_LENGTH*sizeof(float));
		memset(gAnaMagn, 0, MAX_FRAME_LENGTH*sizeof(float));
		gInit = true;
	}

	/* main processing loop */
	for (i = 0; i < numSampsToProcess; i++){

		/* As long as we have not yet collected enough data just read in */
		gInFIFO[gRover] = indata[i];
		outdata[i] = gOutFIFO[gRover-inFifoLatency];
		gRover++;

		/* now we have enough data for processing */
		if (gRover >= fftFrameSize) {
			gRover = inFifoLatency;

			/* do windowing and re,im interleave */
			for (k = 0; k < fftFrameSize;k++) {
				window = -.5*cos(2.*M_PI*(double)k/(double)fftFrameSize)+.5;
				gFFTworksp[2*k] = gInFIFO[k] * window;
				gFFTworksp[2*k+1] = 0.;
			}


			/* ***************** ANALYSIS ******************* */
			/* do transform */
			smbFft(gFFTworksp, fftFrameSize, -1);

			/* this is the analysis step */
			for (k = 0; k <= fftFrameSize2; k++) {

				/* de-interlace FFT buffer */
				real = gFFTworksp[2*k];
				imag = gFFTworksp[2*k+1];
```

```
            /* compute magnitude and phase */
            magn = 2.*sqrt(real*real + imag*imag);
            phase = atan2(imag,real);

            /* compute phase difference */
            tmp = phase - gLastPhase[k];
            gLastPhase[k] = phase;

            /* subtract expected phase difference */
            tmp -= (double)k*expct;

            /* map delta phase into +/- Pi interval */
            qpd = tmp/M_PI;
            if (qpd >= 0) qpd += qpd&1;
            else qpd -= qpd&1;
            tmp -= M_PI*(double)qpd;

            /* get deviation from bin frequency from the +/- Pi interval */
            tmp = osamp*tmp/(2.*M_PI);

            /* compute the k-th partials' true frequency */
            tmp = (double)k*freqPerBin + tmp*freqPerBin;

            /* store magnitude and true frequency in analysis arrays */
            gAnaMagn[k] = magn;
            gAnaFreq[k] = tmp;

    }

    /* ***************** PROCESSING ******************* */
    /* this does the actual pitch shifting */
    memset(gSynMagn, 0, fftFrameSize*sizeof(float));
    memset(gSynFreq, 0, fftFrameSize*sizeof(float));
    for (k = 0; k <= fftFrameSize2; k++) {
            index = k*pitchShift;
            if (index <= fftFrameSize2) {
                    gSynMagn[index] += gAnaMagn[k];
                    gSynFreq[index] = gAnaFreq[k] * pitchShift;
            }
    }

    /* ***************** SYNTHESIS ******************* */
    /* this is the synthesis step */
    for (k = 0; k <= fftFrameSize2; k++) {

            /* get magnitude and true frequency from synthesis arrays */
            magn = gSynMagn[k];
            tmp = gSynFreq[k];

            /* subtract bin mid frequency */
            tmp -= (double)k*freqPerBin;

            /* get bin deviation from freq deviation */
            tmp /= freqPerBin;

            /* take osamp into account */
            tmp = 2.*M_PI*tmp/osamp;

            /* add the overlap phase advance back in */
            tmp += (double)k*expct;

            /* accumulate delta phase to get bin phase */
            gSumPhase[k] += tmp;
            phase = gSumPhase[k];
```

```
				/* get real and imag part and re-interleave */
				gFFTworksp[2*k] = magn*cos(phase);
				gFFTworksp[2*k+1] = magn*sin(phase);
			}

			/* zero negative frequencies */
			for (k = fftFrameSize+2; k < 2*fftFrameSize; k++) gFFTworksp[k] = 0.;

			/* do inverse transform */
			smbFft(gFFTworksp, fftFrameSize, 1);

			/* do windowing and add to output accumulator */
			for(k=0; k < fftFrameSize; k++) {
				window = -.5*cos(2.*M_PI*(double)k/(double)fftFrameSize)+.5;
				gOutputAccum[k] += 2.*window*gFFTworksp[2*k]/(fftFrameSize2*osamp);
			}
			for (k = 0; k < stepSize; k++) gOutFIFO[k] = gOutputAccum[k];

			/* shift accumulator */
			memmove(gOutputAccum, gOutputAccum+stepSize, fftFrameSize*sizeof(float));

			/* move input FIFO */
			for (k = 0; k < inFifoLatency; k++) gInFIFO[k] = gInFIFO[k+stepSize];
		}
	}
}

// -----------------------------------------------------------------------------------------------------------------
---------------


void smbFft(float *fftBuffer, long fftFrameSize, long sign)
/*
	FFT routine, (C)1996 S.M.Bernsee. Sign = -1 is FFT, 1 is iFFT (inverse)
	Fills fftBuffer[0...2*fftFrameSize-1] with the Fourier transform of the
	time domain data in fftBuffer[0...2*fftFrameSize-1]. The FFT array takes
	and returns the cosine and sine parts in an interleaved manner, ie.
	fftBuffer[0] = cosPart[0], fftBuffer[1] = sinPart[0], asf. fftFrameSize
	must be a power of 2. It expects a complex input signal (see footnote 2),
	ie. when working with 'common' audio signals our input signal has to be
	passed as {in[0],0.,in[1],0.,in[2],0.,...} asf. In that case, the transform
	of the frequencies of interest is in fftBuffer[0...fftFrameSize].
*/
{
	float wr, wi, arg, *p1, *p2, temp;
	float tr, ti, ur, ui, *p1r, *p1i, *p2r, *p2i;
	long i, bitm, j, le, le2, k;

	for (i = 2; i < 2*fftFrameSize-2; i += 2) {
		for (bitm = 2, j = 0; bitm < 2*fftFrameSize; bitm <<= 1) {
			if (i & bitm) j++;
			j <<= 1;
		}
		if (i < j) {
			p1 = fftBuffer+i; p2 = fftBuffer+j;
			temp = *p1; *(p1++) = *p2;
			*(p2++) = temp; temp = *p1;
			*p1 = *p2; *p2 = temp;
		}
	}
	for (k = 0, le = 2; k < (long)(log(fftFrameSize)/log(2.)+.5); k++) {
		le <<= 1;
		le2 = le>>1;
		ur = 1.0;
```

```
                ui = 0.0;
                arg = M_PI / (le2>>1);
                wr = cos(arg);
                wi = sign*sin(arg);
                for (j = 0; j < le2; j += 2) {
                        p1r = fftBuffer+j; p1i = p1r+1;
                        p2r = p1r+le2; p2i = p2r+1;
                        for (i = j; i < 2*fftFrameSize; i += le) {
                                tr = *p2r * ur - *p2i * ui;
                                ti = *p2r * ui + *p2i * ur;
                                *p2r = *p1r - tr; *p2i = *p1i - ti;
                                *p1r += tr; *p1i += ti;
                                p1r += le; p1i += le;
                                p2r += le; p2i += le;
                        }
                        tr = ur*wr - ui*wi;
                        ui = ur*wi + ui*wr;
                        ur = tr;
                }
        }
}


// -------------------------------------------------------------------------------------------------------------

/*

    12/12/02, smb

    PLEASE NOTE:

    There have been some reports on domain errors when the atan2() function was used
    as in the above code. Usually, a domain error should not interrupt the program flow
    (maybe except in Debug mode) but rather be handled "silently" and a global variable
    should be set according to this error. However, on some occasions people ran into
    this kind of scenario, so a replacement atan2() function is provided here.

    If you are experiencing domain errors and your program stops, simply replace all
    instances of atan2() with calls to the smbAtan2() function below.

*/


double smbAtan2(double x, double y)
{
  double signx;
  if (x > 0.) signx = 1.;
  else signx = -1.;

  if (x == 0.) return 0.;
  if (y == 0.) return signx * M_PI / 2.;

  return atan2(x, y);
}


// -------------------------------------------------------------------------------------------------------------
// -------------------------------------------------------------------------------------------------------------
// -------------------------------------------------------------------------------------------------------------
```