BFLIB API: Beta release

Robert P. Dougherty

OptiNav, Inc.

March 23, 2018

Contents

Basic concepts

BFLIB is a library of C functions for processing acoustic camera microphone data. There is assumed to be an array of N microphones for receiving airborne sound. The acoustic camera is assumed to include a data acquisition system that simultaneously measures the microphone pressures at “sampleRate” samples per second. A value of blockSize is selected to be power of 2, usually between 128 and 4096. At an average rate of sampleRate/blocksize, the user of BFLIB obtains a block of pressure data from the acoustic camera and passes it to BFLIB by calling the function “processData.” An evolving array Cross Spectral Matrix (CSM) is maintained within BFLIB and updated by exponential averaging when each block of array is supplied. A ring buffer of CSMs is stored to make it convenient to analyze data from the recent past. Alternatively, CSMs can be computed externally and supplied to BFLIB. Also, BFLIB can compute a synthetic CSM for use in modelling the performance of an array design. The function syntheticContributionToCSM supports this operation.

The user defines a grid of M 3D beamforming source points and communicates the grid to BFLIB. The grid points are usually arranged in a 2D pattern that matches the field of view of the optical camera, with a lower density of points. For example, if the optical image has 640 x 480 pixels, it may be appropriate for the acoustic grid to be have 64 x 48 points, so M = 3072. The usual way to arrange the grid points is to space them on a 2D plane that is perpendicular to the central line of sight of the optical camera and set far enough from the camera so that the acoustic sources are approximately in the plane, but different arrangements may sometimes be more appropriate.

BFLIB has several Beamformer objects that can process different frequency bands in parallel. The function “getMap” with one its arguments set to iBF causes the Beamformer numbered iBF to run and return the map of M sound pressure levels (in dB re 20 µPa) corresponding to the beamforming grid points for a specified frequency band and a specified CSM in the ring buffer. In real time acoustic camera operation, getMap is typically called about 6-10 times per second for each frequency band being monitored. In the example code provided, two threads call getMap as fast as the computer can support. The speed depends on the power of computer, especially the number of parallel threads it can support, the number of grid point, and other factors.

BFLIB also provides focused time series and several types of 1D frequency spectra.

The calling routine is responsible for making use of the map values and other acoustic results from BFLIB. In acoustic camera processing, one common function is to convert the acoustic image from 32 bit float to 24 bit RGB using a color Look Up Table, obtain an optical frame from the video camera, merge the optical and acoustic images, and display the result. BFLIB does not perform these functions, but they are simple with a graphics library such as OpenCV. The example program illustrates a way to display greyscale images with openCV.

The functions use C-style arrays which must be declared with the required storage by the calling routing. An access violation will probably occur if the arrays are not large enough.

This is a beta release of BFLIB. Some of the functions, in particular the focus functions, have not been tested yet.

Functions

List of functions

The BFLIB functions are

```
extern DWORD buildRectGrid(DWORD mX, DWORD mY, float xFOVdeg, float yFOVdeg, float z,
       float* xGrid)
extern DWORD BFLIB_API exit();
extern DWORD BFLIB_API getMap(DWORD iBF, bool optiNavBF, DWORD contrastChoice,
        DWORD blockNum, DWORD nThreads, float* map, float* diagonal);
extern DWORD BFLIB_API getSpectra(DWORD blockNum, float* spectra);
extern DWORD BFLIB_API getNFreq();
extern DWORD BFLIB_API initialize(float sampleRate, DWORD blockSize, DWORD nCSMbuffer,
         DWORD N, DWORD maxM,  bool compensate, float maxFreq, DWORD nBF);
extern DWORD BFLIB_API processData(float alpha, float* dataBlock, float* outCos,
       float* outSin, float* focusedSignal, float* focusedSpectrum);
extern DWORD BFLIB_API setArray(float* xMic);
extern DWORD BFLIB_API setFocusPoint(DWORD jFocus);
extern DWORD BFLIB_API setFocusWeights(float* focusWeights);
extern DWORD BFLIB_API setFrequency(DWORD iBF, float freq, DWORD bandwidthChoice);
extern DWORD BFLIB_API setGrid(DWORD M, float* xGrid, float tempC);
extern DWORD BFLIB_API syntheticContributionToCSM(float* spectrum, float* xSource,
        float tempC);
extern DWORD BFLIB_API zeroCSM();
```

All of the functions are blocking. With the exceptions of processData and getNfreq, all of functions return a DWORD 0 if the operation was successful and an error code if not. The value returned by processData, blockNum, is the position in the CSM ring buffer that has just been updated. Functions that use the CSMs, getMap, and getSpectra, take an argument blockNum to select a particular CSM. To use the latest CSM, they should specify blockNum as the latest output from processData. To go back in time to an earlier block, they can use a different value of blockNum = (unwrapped block index) % nCSMbuffer, where the length of the stored history in blocks, nCSMbuffer, is specified in "initialize."

Detailed operation

Several of the functions depend on the number of narrowband frequency bins nFreq = (int)(2*maxFreq* blockSize /sampleRate) or blockSize, whichever is smaller. Here maxFreq is a maximum frequency for processed spectra. The number of microphones, N is also important. The parameters N, blockSize, sampleRate and maxFreq are all communicated to BFLIB by the function initialize.

buildRectGrid

```
extern DWORD buildRectGrid(DWORD mX, DWORD mY, float xFOVdeg, float yFOVdeg, float z,
float* xGrid)
```

Create a rectangular beamforming grid, xGrid, for input to setGrid. See setGrid for the description of the format. The inputs are mX and mY, the number of grid points in the horizontal and vertical directions, xFOVdeg and yFOVdeg, the extent of the field of view in the horizontal and vertical directions, z, the z coordinate of all of the grid points, in meters. The layout assumes that the microphones are in the x-y plane at z = 0 (see setArray). The output array is assumed to be declared with 3*M = 3*mX*mY float values. BFLIB will crash if xGrid does not have this much space. Typical inputs are (64, 48, 65.24f, 51.28f, 2.0f, xGrid). This gives a moderate grid with M = 3072 points.

Exit

```
extern DWORD BFLIB_API exit();
```

Call this when you are finished with a BFLIB session to clean up the memory. On the development machine using Visual Studio, this causes an error as Visual Studio seems to try to delete arrays that have already been deleted. This is believed to be a bug in Visual Studio.

getMap

```
extern DWORD BFLIB_API getMap(DWORD iBF, bool optiNavBF, DWORD contrastChoice, DWORD
DWORD blockNum, DWORD nThreads, DWORD M, float* map, float* diagonal);
```

The function processes one of the CSMs in the ring buffer to produce an acoustic map.

The input parameters, such as `optiNavBF`, can be changed in between calls.

iBF

iBF selects the Beamfomer to use. It should be in the range of 0 ≤ iBf < nBF . See nBF in initialize. The frequency and bandwidth for the this beamformer must have been set by a previous call to setFreqeuncy.

optinavBF

optiNavBF causes getMap to use the OptiNav beamforming method if true. This method is related to both Adaptive Beamforming and Functional Beamforming and is the main algorithm of BeamformX. If optiNavBF is false, then getMap uses a form of conventional Frequency Domain Beamforming.

contrastChoice

contrastChoice controls a tradeoff between resolution and dynamic range for the OptiNav beamforming method. The choices are NORMAL, EXTRA, LOTS, and LOW. NORMAL is best in most cases. LOTS puts more emphasis on resolution and EXTRA gives the most resolution. LOW should be selected when very weak sources need to be seen simultaneously with strong sources or the when the sources are poorly represented as point sources, such when the sound is propagating through a complicated structure on the way to the array.

blockNum

blockNum is the identification number of the CSM in the ring buffer to use. To use the newest data, it should be the latest return from processData.

nThreads

nThreads is the number of parallel threads that this instance of getMap will use to speed up some of the processing steps. Ideally, nThreads*nBF (see initialize) should be less than or equal to the number of threads the computer can run minus 2-4. (The additional threads support processData in BFLIB and presumably optical camera, acoustic array, and image processing and display functions by the calling routine. An error is returned if nThreads exceeds 6.

M

M is the number of points in the acoustic image (see setGrid). BFLIB knows the value of M by this point, and an error code is returned if the input M does not match the expected value.

map

map is the output array for the acoustic image results. The values are in dB re 20 µPa. map must be declared be large enough to store the M values.

diagonal

diagonal is an output array of N float values that receives the autospectral values of the N microphones, processes with the same CSM data and analysis band as used for map. Ideally, if only a single acoustic source is present, the values of diagonal should equal the peak value in map. In most cases, the diagonal values will not all be the same, due to differences between the microphones, and the median of the diagonal values should be less than the peak value in map. BFLIB will crash if diagonal is not NULL and is not declared large enough to hold N float values.

See getSpectra for discussion of different types of spectral products.

getSpectra

```
extern DWORD BFLIB_API getSpectra(DWORD blockNum, float* spectra);
```

spectra

Returns an array of nFreq*N narrowband spectral levels (in dB re 20 µPa) from the CSM blockNum. The parameter blockNum has the same meaning as in getMap. The SPL for microphone i and frequency index iF is returned in spectra[i + N*iF], i = 0,…,(N-1); iF = 0,…,(nFreq-1).

The output of getSpectra differs from the diagonal result of getMap in that the array returned for getMap is only for one frequency band and this band may be narrowband like the output of getSpectra or 1/12 or 1/3 octave band, according to the argument bandwidthChoice of getMap. The output focusedSpectrm from processData is a narrowband spectrum in dB, but it has blockSize values, going up the the Nyquist frequency, which may be greater than maxFreq, and this spectrum relates to the focus point. Other outputs of processData, outCos and outSin, also depend on frequency extending up to the Nyquist frequency, but these are the real and imaginary parts of complex Fourier coefficients.

initialize

```
extern DWORD BFLIB_API initialize(float sampleRate, DWORD blockSize, DWORD extending ,
DWORD N, DWORD maxM, bool compensate, float maxFreq, DWORD nBF);
```

sampleRate and blockSize

This is the first function to call. The sample rate should be set according to the acoustic camera, in the range 22,000 ≤ sampleRate ≤ 192,000. The blockSize should be a power of 2, specifically 128, 256, 512, 1024, 2048 or 4096, also chosen according to the capability of the acoustic camera. The bandwidth of the narrowband results will be sampleRate/blockSize.

The number of narrowband output spectra becomes nFreq = (int)(2*maxFreq* blockSize /sampleRate). If this value exceeds blockSize, then nFreq is replaced by blockSize and the maximum frequency is sampleRate/2.   See getSpectra.

nCSMbuffer

nCSMbuffer specifies the number of CSMs that BFLIB stores in a ring buffer. See the discussion of blockNum in getMap. The smallest value of nCSMbuffer is 1. The upper limit is controlled by computer memory, but an error is returned if nCSMbuffer exceeds 1024. Each CSM requires nFreq*N*N*16 bytes, where nFreq = (int)(2*maxFreq* blockSize /sampleRate), or blockSize, whichever is smaller.

N

N is the number of microphones in the array. 16 ≤ N ≤ 168.

maxM

maxM is the maximum number of grid points for subsequent calls to setGrid. For a real-time, 2D grid, it should probably not be more than about 307200. A value closer to 3072 is more normal. It might be a lot larger for 3D gird, but this will make getMap slow.

compensate

compensate is a bool that determines whether the input pressure values are divided by to 2 to compensate for the pressure doubling by a hard acoustic camera face.

maxFreq

maxFreq is used to limit the number of frequencies used by getSpectra and syntheticContributionToCSM, to nFreq = (int)(2*maxFreq* blockSize /sampleRate) or blockSize, whichever is smaller. maxFreq must be in the range 1/ sampleRate ≤ maxFreq ≤ 20000.

nBF

nBF is the number of Beamformer objects that BFIB will create and store internally. This is largest number of frequency bands that can be processed simultaneously by different instances of getMap. The minimum number is 1. The maximum value might consider the number of threads that the computer can support. It is an error to enter nBF > 24. See also nThreads in getMap.

processData

```
extern DWORD BFLIB_API processData(float alpha, float* dataBlock, float* outCos, float*
outSin, float* focusedSignal, float* focusedSpectrum);
```

This function is used by the calling routine to communicate array microphone pressure data to BFLIB. BFLIB uses this information to update the stored CSM data. The value returned by the function, totalBlocks, is discussed below.

dataBlock

The input array dataBlock should contain blockSize*N 32-bit float microphone pressure values in Pa. (blocksize and N are specified by “initialize”.) The pressure from microphone i and sample number it into the block is expected to be in dataBlock[i + N*it], i = 0,…,N-1; it = 0,…,blockSize-1. The array dataBlock is not changed by processData.

outCos and outSin

On each call to processData, the dataBlock is appended to the data from the previous call to implement a block overlap operation, and a Hanning window is applied to the time resulting time series for each of the N channels. An FFT is computed and the blockSize*N real and imaginary Fourier coefficients are returned in the arrays outCos and outSin. In each array, the Fourier cosine or sine coefficient is returned in index [i + N*iF], i = 0,…,N-1; iF = 0,…,blockSize-1. These FFT coefficients are intermediate results in the beamforming analysis, but they are returned so that the calling routine can, for example, monitor for the sudden appearance of a tone. The float[blockSize*N] arrays outCos and outSin are required even if the output values are not needed.

focusedSignal and focusedSpectrum

Time domain signals of length blockSize are synthesized by Delay And Sum (DAS) beamforming for a specific focusing point and the time series and frequency spectrum from this operation are returned in focusedSignal and focusedSpectrum, respectively. The focusing point is set by setFocusPoint and the microphone weights applied in the sum are specified by setFocusWeights. The number of values returned in focusedSignal is focusedSpectrum is blockLength, each. An access violation is likely if either of these is not NULL and does not have enough space to accept the results. The values in focusedSignal are in Pa, scaled the center of the microphone array, representing a single time series at the rate sampleRate. The results in focusedSpectrum are narrowband SPL values in in dB re 20 µPa, again scaled for the center of the microphone array. The temperature assumed for this operation is the most recent value of tempC supplied to setGrid, or 22° C if setGid has not been called.

focusedSignal and focusedSpectrum have not yet been tested in this version of BFLIB.

alpha

To implement an exponential filter for the CSMs with a decay time of tau seconds, the appropriate value of alpha to specify is exp(-1/ (tau * sampleRate / blockSize)). This controls how quickly the successive CSMs in the ring buffer change in response to changing acoustic inputs. A typical value of tau is 0.3 s. On the next call to processData after calling zeroCSM, the value of alpha is overridden by 0. This initializes the exponential filter at a full snapshot instead of 0.

Returned value

The return value of processData is the index in the CSM ring buffer, blockNum, where the CSM for this block has been stored. This value can be input to getMap or getSpecrtra to select this particular CSM.

Note concerning timing

The intention is that processData should run in less time than the duration of a block of data, blockSize/sampleRate so that processData can keep with the acoustic camera. Because of the unpredictable timing of a PC, it likely that there will be some exceptions. Sometimes processData will not complete before the next block is available from the array. The calling program should be written so as to discard one or more blocks of data in that case, rather than trying to get processData to catch up. The example code arrayDataThread below illustrates the idea. processData will not run much faster than real time.

The most time consuming job for processData is filling the elements of the narrowband CSM. In order to save time, processData only updates CSMs for frequency bands that have been assigned by calls to setFrequency. Higher maxFreq, larger blockSize, larger nBF, and larger bandwidhChoice all increase the number of narrowband frequencies in use and increase the time required by processData. The number of microphones, N, is also has a major impact on the speed of processData. Depending on the speed of the computer it may be necessary to constrain some of the parameters above to keep processData from falling behind.

setArray

```
extern DWORD BFLIB_API setArray(float* xMic);
```

xMic

The input array xMic is assumed to contain 3*N float values. The microphone array coordinates are assumed to be xi = xMic[i + 3*N], yi = xMic[i+1 + 3*N], zi = xMic[i+2 + 3*N], i = 0,..,N-1 in meters. If xMic does not contain 12*N bytes, the BFLIB will crash.

BFLIB is intended to support most known acoustic camera designs, but not line arrays. Supported array designs are 2D planar arrays, including “arm” arrays, and 3D spheres. setArray analyzes xMic to classify the array shape and verify that the RMS distance of the microphones from the center of mass is between 0.05 m and 1 m for planar arrays and 0.1 m and 0.32 m for spherical arrays. The location and orientation of the array are not important, but the grid points, xGrid, to be subsequently specified by setGrid, must be in the same coordinate system as xMic so that the required distances between the grid points and the microphones can be computed. If the array design does not meet the requirements, then setArray returns with error code 2.

Be sure to call setGrid after calling setArray, otherwise setArray will have no effect.

setFocusPoint

```
extern DWORD BFLIB_API setFocusPoint(DWORD jFocus);
```

The argument should satisfy 0 ≤ jFocus < M, where M is the number of grid points set by setGrid. This selects that grid point for focusing the array data by DAS to produce the focusedSignal, and focusedSpectrum outputs of processData. This beamforming is different from the frequency domain beamforming of getMap that is the main product of BFLIB.

setFocusWeights

```
extern DWORD BFLIB_API setFocusWeights(float* focusWeights);
```

The argument is an input array of N float values that are used a weights in the DAS beamforming by processData for its focusedSignal, and focusedSpectrum outputs. By default, the focusWeight values are all 1.f.

setFrequency

```
extern DWORD BFLIB_API setFrequency(DWORD iBF, float freq, DWORD bandwidthChoice);
```

```
Sets the center frequency and the bandwidth for a beamformer. For real time processing,
it is usually appropriate to call zeroCSM after setFrequency.
```

setFrequency can be called at any time after initialize, but it has some cost. To monitor several frequencies bands, it is better to create several beamfomers. After setFrequeny, it is usually appropriate to call zeroCSM. To perform a frequency sweep, the calling rountine would need to build the CSM several times by multiple calls to processData, using buffered time series data. BFLIB does not buffer time series data.

iBF

iBF selects the Beamfomer to use. It should be in the range of 0 ≤ iBf ≤ (nBF – 1). See nBF in initialize.

freq

freq is the center frequency of the desired analysis band in Hz. An error is returned if freq is not in the range (sampleRate/(2*blockSize) ≤ freq ≤ maxFreq. (See initialize.)

bandwidthChoice

bandwidthChoice selects the analysis bandwidth. The value of bandwidthChoice can be NARROWBAND, TWELTH_OCTAVE, or THRID_OCTAVE. The constants are defined in the header file.

setGrid

```
extern DWORD BFLIB_API setGrid(DWORD M, float* xGrid, float tempC);
```

This function sets M and the beamforming grid of M 3D points according to xj = xMic[j + 3*M], yj = xMic[j+1 + 3*M], zj = xMic[j+3 + 3*M], j = 0,..,M-1 in meters. If xGrid does not contain at least 12*M bytes, then BFLIB will crash. As discussed previously, the grid points may be chosen to cover the field of view of the video camera, if present, but usually at lower resolution.

setArray must be called before setGrid.

syntheticContributionToCSM

```
extern DWORD BFLIB_API syntheticContributionToCSM(float* spectrum, float* xSource, float
tempC);
```

This function creates a simple CSM for a point sources and adds it to the current CSM. It does not change the pointer to the CSM ring buffer. The spectrum of the source, (in dB re 20 µPa) is supplied in the array "spectrum" with length nFreq. The normalization is such that this spectrum would be seen from this source at the center of the array. The location of the source is xs = xSource[0], ys = xSource[1], zs = xSource[2]. The temperature in Celsius is specified by tempC. Note that it is possible for this temperature to differ from the temperature used for beamforming, which is specified in setGrid.

It is possible to call syntheticContributionToCSM more than once to model complicated distributions of incoherent sources.

zeroCSM

```
extern DWORD BFLIB_API zeroCSM();
```

Sets the current CSM and the spectrum to 0. Used as the reset to start creating a new CSM with one or more calls to `syntheticContributionToCSM`. Also appropriate when changing the frequency `for getMap. After calling zeroCSM, the value of alpha applied for the next call to processData is 0, regardless of the input value of alpha, so the exponential filter is initialized by the next snapshot (block) instead of 0.`

Error codes

1	Calling a function other than initialize first
2	Unsupported array design specified in setArray
3	sampleRate out of range in initialize
4	blockSize is not a power of 2 or not in the allowed range in initialize
5	nCSMbuffer out of range in initialize
6	Number of microphones, N, out of range in initialize
7	Number of Beamformers, nBF, requested in initialize out of range
8	Calling processData with dataBlock, cosSpect, or sinSpect == NULL
9	iBF not in the range 0 ≤ iBF < nFB in getMap or setFrequency
10	Calling syntheticContributionToCSM or zeroCSM before initialize
11	contrastChoice out of range in getMap
12	maxFreq out of range in initialize
13	Number of threads requested in getMap exceeds limit
14	Frequency out of range `sampleRate/(2*blockSize)` ≤ freq ≤ maxFreq in setFrequency
15	bandwidthChoice out of range in getMap
16	M in setGrid not in the range 1 ≤ M ≤ maxM
17	alpha in processData does not satisfy 0 ≤ alpha ≤ 1
18	getMap called for a beamformer that is unexpectedly not set up
19	jFocus does not satisfy 0 ≤ jFocus < M in setFocusPoint
20	M in setGrid is less than 1 or larger than the value maxM provided in initialize
21	Internal inconsistency in grid found in getMap
22	setGrid called before successful call to setArray

Use of Eigen

BFLIB internally makes use of the C++ template linear library Eigen, which is linear algebra package. Eigen is licensed under MPL2. Eigen source code is not distribute with BFLIB. BFLIB was compiled with the EIGEN_MPL2_ONLY preprocessor defined. MPL2 requires that the source code for Eigen be made available to users of BFLIB. This source code is linked from the Eigen web site: http://eigen.tuxfamily.org.

Variants of BFLIB

There are currently three variants of BFLIB, numbered 1-3. The variant number is in the second place after the decimal in the version number, e.g. a future version 20.43 of BFLIB would be variant 3.

The variant number limits the allowed array shape. setArray analyzes the input array design and determines whether it meets the requirements. Linear arrays are never supported. All three variants support planar arrays. In this case, an ellipse is fit to the coordinates, and the eccentricity of the ellipse is computed. The eccentricity must not be greater than 0.9. Variant 1also supports open spherical arrays.

The limits of the array diameter and the inputs to "initialize" are given in Table 1.

Table 1

Variant	Array shapes	Max diam.	Max freq.	Max #of mics.	Max sampleRate	Max blockSize
1	planar spherical	2.5 m 0.7 m	20 kHz	168	96 kS/s	4096
2	planar	1 m	20 kHz	64	96 kS/s	4096
3	planar	0.4 m	10 kHz	32	48 kS/s	2048

Example calling program

The following example illustrates the use of BFLIB to monitor two frequency bands. The frequencies are initially 3 and 8 kHz, but it would be feasible to make the respective frequencies global variables, external to beamformingThread, so they could be changed during run.

The library cv refers to openCV, which may be convenient for displaying image results. It is not part of or required for BFLIB.

Includes

… Edit for your environment…

```
#include "BfLib.h"
#include //… other includes, such as acoustic camera library
#include <opencv2/highgui/highgui.hpp> //Use openCV to display the image maps
#include <opencv2/imgproc/imgproc.hpp>
#include <iostream>
#include <thread>
```

```
#include <vector>
//The following 2 lines needed for ignore in main()
#include <limits>
#undef max
using namespace std;
```

Global variables to control BFLIB

```
DWORD nCSMbuffer = 6;
float maxFreq = 12000.f;
bool compensate = true;
DWORD blockSize = 1024;
DWORD nBF = 2;
DWORD N;
float* xMic;
float sampleRate;
float tauDecaySec = 0.3f;
DWORD nThreads = 2;
DWORD contrastChoice = 0;
DWORD bandwidthChoice = 2;
float alpha = 0;
DWORD mX = 64;
DWORD mY = 48;
DWORD M = mX*mY;
DWORD maxM = M;
float xFOVdeg = 65.24f;
float yFOVdeg = 51.28f;
float z = 1.f;
float* xGrid = new float[3*mX*mY];
bool arrayKeepGoing = true;
bool keepBeamforming = true;
DWORD blockNum = 0;
```

Thread for acquiring data and processing into CSMs

```
void arrayDataThread() {
       DWORD blockSizeN = blockSize*N;
       float* outCos = new float[blockSizeN];
       float* outSin = new float[blockSizeN];
       float* focusedSignal = NULL;
       float* focusedSpectrum = NULL;
       DWORD blockTimeMicros = (1000000 * blockSize) / sampleRate;
       chrono::microseconds timespan(blockTimeMicros/16);

       …Start the acoustic camera here…

       while (arrayKeepGoing) {
              DWORD blocksInCamera = …Get number of blocks in the camera buffer
              if (blocksInCamera > 0) {
                     for (int i = 0; i < blocksInCamera; i++) {
                            … Read a block from the camera into dataBlock
                     }
                     //Note only processing the newest data block.  Others discarded.

                     blockNum = BfLib::processData(alpha, dataBlock, outCos, outSin,
                                   focusedSignal, focusedSpectrum);
              } else {
                     this_thread::sleep_for(timespan);
              }
```

```
        }

        …Stop and shut down the acoustic camera here…

        delete[] dataBlock;
        delete[] outCos;
        delete[] outSin;
}
```

Utility functions for graphics

```
float normalize(float dr, DWORD M,  float* map) {
        float max = map[0];
        int iMax = 0;
        for (int i = 1; i < M; i++) {
                if (map[i] > max)max = map[i];
                iMax = i;
        }
        float min = max - dr;
        for (int i = 0; i < M; i++) {
                if (map[i] < min) {
                        map[i] = 0.f;
                }else{
                        map[i] = (map[i] - min) / dr;
                }
        }
        return max;
}
cv::Mat fillCV(int mX, int mY, float* map) {
        cv::Mat bfFrame;
        bfFrame.create(mY, mX, CV_32F);
        float* framePtr = (float*)bfFrame.data;
        int stepY = bfFrame.step[0] / sizeof(float);
        int stepX = bfFrame.step[1] / sizeof(float);
        int ind = 0;
        for (int j = 0; j < mY; j++) {
                for (int i = 0; i < mX; i++) {
                        framePtr[i*stepX + j*stepY] = map[ind++];
                }
        }
        return bfFrame;
}
```

Thread for beamforming

```
void beamformingThread(string title, DWORD iBF, bool optiNavBF, DWORD nThreads, DWORD
         contrastChoice, DWORD bandwidthChoice, float freq) {
        float* map = new float[M];
        float* diagonal = new float[N];
        BfLib::setFrequency(iBF, freq, bandwidthChoice);
        BfLib::zeroCSM();
        while (keepBeamforming) {
                BfLib::getMap(iBF, optiNavBF, contrastChoice, blockNum, nThreads, map,
                        diagonal);
                normalize(10, M, map);
                normalize(10, M, map);
                cv::Mat mat32 = fillCV(mX, mY, map);
                cv::imshow(title, mat32);
                cv::waitKey(30);
        }
        delete[] map;
        delete[] diagonal;
}
```

Main program

```
int main() {
        N = …
        xMic = …
        alpha = exp(-1.f / (tauDecaySec * sampleRate / blockSize));

        …Set up the acoustic camera here…

        BfLib::initialize(sampleRate, blockSize, nCSMbuffer,  N, maxM,
                             compensate,  maxFreq, nBF);
        arrayKeepGoing = true;
        thread aT(arrayDataThread);
        BfLib::setArray(xMic);
        BfLib::buildRectGrid(mX, mY, xFOVdeg, yFOVdeg, z, xGrid);
        BfLib::setGrid( M, xGrid, 25.f);
        keepBeamforming = true;
        thread bT0(beamformingThread, "3 kHz", 0, true, nThreads, contrastChoice,
                bandwidthChoice, 3000.f);
        thread bT1(beamformingThread, "8 kHz", 1, true, nThreads, contrastChoice,
               bandwidthChoice, 8000.f);
        cout << "Press ENTER to quit...";
        //Wait for the user to press ENTER
        cin.ignore(std::numeric_limits <std::streamsize>::max(), '\n');
        arrayKeepGoing = false;
        keepBeamforming = false;
        aT.join();
        bT0.join();
        bT1.join();
        cout << "Done ENTER to exit...";
        //Wait for the user to press ENTER
        cin.ignore(std::numeric_limits <std::streamsize>::max(), '\n');
        delete[] xMic;
        BfLib::exit();
        return 0;
}
```