

VSIPL++ Demonstration: Minimum Variance Beamformer

Randall Judd

Dennis Cottel

Introduction	3
<i>Disclaimer</i>	3
Make file	4
Main	5
Parameter class	13
<i>Parameter file</i>	13
<i>Parameter interface file (param_mvdr.h)</i>	15
<i>Parameter class implementation (param_mvdr.cpp)</i>	17
Data input class	24
<i>Data Input interface file (data_input.h)</i>	24
<i>Data input class implementation (data_input.cpp)</i>	25
Array class	29
<i>Array Sensor File</i>	30
<i>Array class interface file (array.h)</i>	30
<i>Array class implementation (array.cpp)</i>	31
Beam-Steering coefficient class	35
<i>Beam-Steering coefficient interface (beam_steer_coef.h)</i>	35
<i>Beam steering coefficient class implementation (beam_steer_coef.cpp)</i>	36
Phase array table class	39
<i>Phase array table interface (phat.h)</i>	39
<i>Phase array table implementation (phat.cpp)</i>	40
Cube class	43
<i>Cube interface and implementation (cube.h)</i>	43
Vector/Matrix print utility	46
<i>Vector/Matrix print implementation (VUPP_print.h)</i>	46

Introduction

This document presents source code for a minimum variance beamformer algorithm. This is a generic algorithm available in standard beamforming texts (for instance Johnson & Dudgeon *Array signal processing: Concepts and Techniques* ISBN 0-13-048513-6).

The purpose of this code is to demonstrate the use of the VSIP++ signal processing library. The specification for this library is available on the internet at hpec-si.org.

Disclaimer

For this example code neither the United States Government, the United States Navy, nor any of their employees, makes any warranty, express or implied, including the warranties of merchantability and fitness for a particular purpose, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights.

Make file

This is an example makefile which the authors used to create executable code.

(Makefile)

```
## Minimum Variance Beamformer Example

## $Revision: 1.2 $
## $Date: 2005/05/05 17:41:05 $
## $Author: judd $

## replace RDIR with correct directory where VSIPL is for your machine
RDIR=/Users/judd/local

LDIR=-L$(RDIR)/lib
IDIR=-I$(RDIR)/include
## LIBS (names) will be machine dependent. also FFTW may not be used.
LIBS=-lvsip_fftw -lvsipp -lfftw -lm

CPPFILES = BeamformEx.cpp param_mvdr.cpp Data_input.cpp array.cpp \
           beam_steer_coef.cpp phat.cpp
INTERFACE = array.h Data_input.h param_mvdr.h beam_steer_coef.h \
           cube.h phat.h

BeamformEx: $(CPPFILES) $(INTERFACE)
            g++ -O2 -o BeamformEx $(CPPFILES) $(LDIR) $(IDIR) $(LIBS)

clean:
    rm -f BeamformEx *.o
```

Main

This is the top level algorithm containing the *main()* portion of the code.

(BeamformerEx.cpp)

```
/* Minimum Variance Beamformer Example */
/* main */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:05 $
    $Author: judd $
*/

#include<cstdlib>
#include<ctime>
#include<string>
#include<iostream>
#include<fstream>
#include<vsip/initfin.hpp>
#include<vsip/support.hpp>
#include<vsip/vector.hpp>
#include<vsip/matrix.hpp>
#include<vsip/solvers.hpp>
#include<vsip/signal.hpp>
#include<vsip/math.hpp>
#include"param_mvdr.h"
#include"data_input.h"
#include"array.h"
#include"beam_steer_coef.h"
#include"VUPP_print.h"

int
main( int argc, char* argv[])
{
    vsip::vsipl myvsip;
    std::string myfile(argv[1]);
    Param_mvdr param(myfile);
    Data_input tsdatain(param);
```

```

Array myarray(param.getName_f_array_xypos(),
              param.get_n_sen(),
              param.get_hyd_array(),
              param.get_n_hyd());
Beam_steer_coef steering_vectors(param, myarray);

int n = param.get_fft_length(); // length of FFT
int nh = param.get_n_hyd(); // number of hydrophones used
int nang = param.get_n_beams(); // number of beams formed
int ibin1 = param.get_bin_low(); // beginning frequency bin
int ibin2 = param.get_bin_high(); // end frequency bin
vsip::scalar_f fs = param.get_samp_rate(); // sample rate of input data
/*
 * overlap of input time series data
 * if nover were zero then no input data would be reused in consecutive FFT's
 * here nover is calculated using array dimensions using the formula below.
 * I don't know the origin of this formula, however its result is
 * an integer equal to the number of samples that would be collected
 * during the time an acoustic wave transits the longest portion of
 * the array (+ 1 sample since the int causes a floor operation)
 */
int nover =(int)(1 + fs * myarray.max() / param.get_sv());

std::ofstream
    bm_ts_output((param.getName_f_output_bts()).c_str(),std::ios::binary);

int mysize = n - 2 * nover;
vsip::scalar_f *bm_ts_buff = new vsip::scalar_f[nang * (mysize)];
vsip::scalar_f *bm_ts_ptr = bm_ts_buff;

std::ofstream btr_output((param.getName_f_output_btr()).c_str(),std::ios::binary);
vsip::scalar_f *btr_buff = new vsip::scalar_f[nang];

/* navg (call navg) is number of input data sets to use when averaging power for
the BTR */
int navg = param.get_n_cov_avg();
vsip::scalar_f navg_scale = 1.0/(vsip::scalar_f)(navg);

```

```

/*
 * nshift is the distance between individual data sets where averaging is started
 * if nshift equals navg then there is no overlap if nshift is greater
 * than navg then some data is not processed. This case is not handled here.
 * If nshift equals zero then the same data set would be processed
 * repeatedly. This case is not handled.
 * So nshift == 1 implies maximum data processing
 * and nshift == navg implies minimum data shift. 1 <= nshift <= navg
 */

int nshift = param.get_n_fft_shift();

// the weight Matrix is where the steering vectors reside
vsip::Matrix<vsip::cscalar_f> w(nang,nh);           // weight matrix
vsip::Matrix<vsip::cscalar_f> w_trans(nh,nang);    // weight matrix (transpose)

// S holds periodogram data
vsip::Matrix<vsip::cscalar_f> S(nh * navg, n/2 + 1);

// S_bmf holds complex beamformed data
vsip::Matrix<vsip::cscalar_f> S_bmf(nang * navg, n/2 + 1);

vsip::Matrix<vsip::scalar_f> TS_bm_out(nang, n); // Real Time series out
vsip::Matrix<vsip::scalar_f> my_magsq(nang * navg, n/2 + 1);

/* btr_scale corrects the BTR power */
vsip::scalar_f btr_scale = 1.0 / ((vsip::scalar_f)(nh * nh * (nshift) * \
                                   (ibin2 - ibin1)) * fs / (vsip::scalar_f)n);

/* bm_ts_scale corrects the beam time series power */
vsip::scalar_f bm_ts_scale = 1.0 / (vsip::scalar_f)(nh * n);

/* FFT for real input data to complex input data for each phone */
vsip::Fftm< vsip::scalar_f,
            vsip::cscalar_f,
            vsip::row,vsip::fft_fwd,
            vsip::by_reference,
            0, vsip::alg_space>

```

```

        rcffftop((vsip::Domain<2>(nh, n)),static_cast<vsip::scalar_f>(1.0));

/* FFT for complex beam data to real beam data for each beam */
vsip::Fftm< vsip::cscalar_f,
            vsip::scalar_f,
            vsip::row, vsip::fft_inv,
            vsip::by_reference,
            0,vsip::alg_space>
    crffftop((vsip::Domain<2>(nang, n)),static_cast<vsip::scalar_f>( bm_ts_scale));

int i_fft = 0;

/* Initialize input data object for "nover" and get first set of data */
tsdatain.get_xdata(nover); // data contained in tsdatain object
if(tsdatain.fail()){
    printf("input failure for initital data input\n");
    exit(-1);
}

/* calculate periodogram for first data set. Store in (top nh rows of) S */
rcffftop(
    tsdatain.x_data_mat(), // returns matrix of input data
    S(vsip::Domain<2>(vsip::Domain<1>(0,1,nh),vsip::Domain<1>(0,1,n/2+1)))
);

i_fft++; // This guy just counts the number of rcffftop calls

int start = 1; // we now have one set of data so the loop starts at one to navg
data sets.
int bts_start = 0; // need to write all the beam time series out and not skip the
first one.

/* below is done until data dies or nfft's are done */
while(i_fft < param.get_n_fft()){ // Start main processing loop

    /* Transform from time domain to Frequency Domain */
    for(int i = start; i<navg; i++){ // start depends upon nshift
        tsdatain.get_xdata(); // get some data
    }
}

```

```

if(tsdain.fail()){
    std::cout << "data input failure\n" << std::endl;
    std::cout << "number of FFT's : " << i_fft << std::endl;
    exit(-1);
}
/* FFT Data */
/* calculate periodogram for data set i. Store in S */
rcffftop(
    tsdain.x_data_mat(),
    S(vsip::Domain<2>(vsip::Domain<1>(i * nh ,1,nh),\
        vsip::Domain<1>(0,1,n/2+1)))
    );

    i_fft++; // This guy just counts the number of rcffftop calls
} // completed input of data and conversion to periodogram. nave data sets
ready

/* Do beamforming */
for(int f_bin = ibin1; f_bin <= ibin2; f_bin++){ // for each frequency bin

    vsip::Matrix<vsip::cscalar_f>::col_type s_col(S.col(f_bin));
    vsip::Matrix<vsip::cscalar_f>::col_type s_bmf_col(S_bmf.col(f_bin));

    /* beamform each beam */

    // Create a cholesky object
    vsip::chold<vsip::cscalar_f, vsip::by_reference>
        chold_object(vsip::chold<vsip::cscalar_f, vsip::by_reference>::lower,nh);

    /* calculate the covariance martrix */
    // A matrix to hold the covariance estimate
    vsip::Matrix<vsip::cscalar_f> cov(nh,nh);
    cov = vsip::cscalar_f(0.0,0.0); // Initialize to zero;

    // average together navg covariance estimates for frequency f_bin
    for(int set=0; set < navg; set++){

        vsip::Matrix<vsip::cscalar_f>::col_type::subview_type \

```

```

        sv(s_col(vsip::Domain<1>(set * nh, 1 , nh)));

    COV +=
        vsip::outer<vsip::cscalar_f>(vsip::cscalar_f(navg_scale,0.0),sv,sv);
}
// add a fudge factor for matrix stability
cov.diag(0) += sumval(cov.diag(0)).real()/nh * param.get_diagl();
// completed covariance estimate

chold_object.decompose(cov); // Decomposition of covariance

w = (steering_vectors.get_xy(f_bin-ibin1));
for(int row=0; row <nang; row++) w_trans.col(row) = w.row(row);
chold_object.solve(w_trans,w_trans); // Solve for steering vectors

/* normalize steering vectors */
for(int row=0; row<nang; row++)
    w_trans.col(row) /= cvjdot(w.row(row), w_trans.col(row)).real();

/* transpose */
for(int row=0; row <nang; row++)
    w.row(row) = conj(w_trans.col(row));

for(int j=bts_start; j<navg; j++){ // Beamform
    vsip::Matrix<vsip::cscalar_f>::col_type::subview_type \
        sv(s_col(vsip::Domain<1>(j * nh, 1 , nh)));
    vsip::Matrix<vsip::cscalar_f>::col_type::subview_type \
        sv_bmf(s_bmf_col(vsip::Domain<1>(j * nang, 1 , nang)));
    //(Matrix of steering vectors ) * (column at f_bin)
    sv_bmf = conj(prod(w,sv)); // mvprod
}
}

// calculate BTR
my_magsq = magsq(S_bmf);
for(int iang=0; iang < nang; iang++){
    vsip::scalar_f val = 0.0;
    for(int k=navg - nshift; k<navg; k++){

```

```

        val += sumval(my_magsq.row(iang + k * nang));
    }
    btr_buff[iang] = val * btr_scale;
}
btr_output.write((char*)btr_buff, nang * sizeof(vsip::scalar_f));
btr_output.flush();

// calculate beam time series
// only need the new ones
for(int bmset = bts_start; bmset < navg; bmset++){

    crffftop(S_bmf(vsip::Domain<2>(vsip::Domain<1>(bmset * nang, 1, nang),
        (vsip::Domain<1>(0, 1, n/2+1))))), TS_bm_out);

    //write TS_bm_out
    bm_ts_ptr = bm_ts_buff;
    for(int i=nover; i < n - nover; i++){
        for(int iang = 0; iang < nang; iang++){
            *bm_ts_ptr = TS_bm_out.get(iang, i);
            bm_ts_ptr++;
        }
    }
    bm_ts_output.write((char*) bm_ts_buff, \
        nang * mysize * sizeof(vsip::scalar_f));
    bm_ts_output.flush();
}

/* Reorg the data and set up for next output set */
int moveto = 0;
if(nshift > navg){ // move to new starting place
    for(int shift = 0; shift < nshift - navg; shift++){
        tsdatain.get_xdata();
        if(tsdatain.fail()){
            std::cout << "data input failure\n" << std::endl;
            std::cout << "number of FFT's : " << i_fft << std::endl;
            exit(0);
        }
    }
}

```

```

} else { // move freq domain data to be reused up in S and S_bmf
    for(int movefrom=nshift; movefrom < navg; movefrom++){

        S(vsip::Domain<2>(vsip::Domain<1>(moveto * nh,1,nh),\
            vsip::Domain<1>(0,1,n/2+1))) = \
            S(vsip::Domain<2>(vsip::Domain<1>(movefrom * nh,1,nh),\
                vsip::Domain<1>(0,1,n/2+1)));

        S_bmf(vsip::Domain<2>(vsip::Domain<1>(moveto * nang,1,nang),\
            vsip::Domain<1>(0,1,n/2+1))) =
            S_bmf(vsip::Domain<2>(vsip::Domain<1>(movefrom * nang,1,nang),\
                vsip::Domain<1>(0,1,n/2+1)));

        moveto++;
    }
}

start      = moveto; // next set of data loads here
bts_start  = moveto; // next set of data loads here
} // End main loop

bm_ts_output.close();
btr_output.close();
delete bm_ts_buff;
delete btr_buff;
}

```

Parameter class

The parameter class is used to input data from a parameter file. The parameter object contains the data from the parameter file and has functionality to allow the program to access the parameters. The interface to the parameter class is fairly simple and should be easy to discern with a review of the interface (header) file below.

Parameter file

This section could be used as a parameter file. Below is a Sample of what a parameter file looks like. Note that only the line following a row of dashes contains parameter input. All other text in a parameter file is comment. The actual data input into the parameter object is color coded red to make it clear. For this example we assume a generic linear array of 10 equally spaced (1 meter apart) sensors along the x axis from 0 to 9 meters sampled at 1 kHz .

```
Name of input time series file.  
The file must contain float (real*4) binary data;  
No headers, buffer counters, time stamps, etc. are allowed.  
Note that input data directly follows a line of dashes.
```

```
-----  
/Users/judd/input_data_files/time_series_data
```

```
Name of output Bearing Time Record (BTR) file  
This is a float (real*4) binary file  
The file will have number of beam columns  
and number of rows will correspond to time.  
The data is column major.  
This file will have "nan" columns, see line (6) below.  
The number of rows will depend on the amount of data  
processed, but will be written out when the program ends.)
```

```
-----  
/Users/judd/output_data_files/btr_data
```

```
Name of output beam time series data  
This is a float (real*4) binary file  
The number of beams is along the column  
The time is along the rows  
thi file is column major
```

```
-----  
/Users/judd/output_data_files/beam_data
```

```
Name of input file for array sensor position data  
This is an ascii file x, y, z sensor positions on each line; one line per sensor
```

```
-----  
/Users/judd/input_data_files/array_sensor_positions
```

Physical Parameters

sv = Sound speed (Units consistent with sensor positions).
samp_rate = Sample rate (Hz).
n_sen = No. of sensors in full array.
sv samp_rate n_sen

1500.0 1000 10

Processing Parameters

low_freq = Lowest frequency to process (Hz).
high_freq = Highest frequency to process (Hz).
fft_length = FFT length (Must be power of 2).
n_cov_avg = No. of FFTs to average per covariance matrix (0 => 1).
n_fft_shift = No. of FFTs to shift between start of successive
data sets (0 => 1).
n_beams = No. of beamforming bearings (cosine spaced).
(Beam #1 is North, and Beam #(n_beams) is South.)
low_freq high_freq fft_length n_cov_avg n_fft_shift n_beams

10.0 400.0 1024 4 2 10

Other Parameters

diagl = Diagonal loading factor for adaptive beamformer (The
quantity: diagl*[average phone power] is added to the
diagonal of the covariance matrix before inversion).
n_ptskp = No. of data points to skip before start of processing
n_fft = No. of FFTs to compute (0 => 1000000000).
diagl n_ptskp n_fft

0.1 400 0

Array data. These are the numbers of the hydrophones to process

arrayld_hyd = A list of the hydrophones in array. (Note that the
first phone is number 1, not 0.)

If there was a bad sensor (or some channels had data not
associated with a sensor) then we can account for that here by
leaving out the number corresponding to that bad data.

1 2 3 4 5 6 7 8 9 10

Parameter interface file (param_mvdr.h)

```
/* Minimum Variance Beamformer Example */
/* Param_mvdr API */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:06 $
    $Author: judd $
*/
*/
/*
 * Param_mvdr.h
 *
 * Created by Randall Judd on Fri Mar 26 2004.
 * Copyright (c) 2004 __SSC-SD__. All rights reserved.
 *
 */
#ifndef PARAM_MVDR
#define PARAM_MVDR

#include<iostream>
#include<fstream>
#include<string>

#define DELIMITER "-----"

class Param_mvdr
{
public:
    Param_mvdr(std::string); // Constructor, get file name from arg list
    ~Param_mvdr();

    std::string getName_f_input_data() const;
    std::string getName_f_output_btr() const;
    std::string getName_f_output_bts() const;
    std::string getName_f_array_xypos() const;

    int get_n_sen() const;
    int get_n_ptskp() const;
    int get_fft_length() const;
    int get_n_fft() const;
    float get_samp_rate() const;
    float get_low_freq() const;
};
```

```

float get_high_freq() const;

int get_n_cov_avg() const;
int get_n_fft_shift() const;
int get_n_beams() const;
float get_diagl() const;
float get_sv() const;
int get_phone(int) const;
int* get_hyd_array() const;

int get_n_hyd() const; // Number of Hydrophones

float get_delf() const;
int get_bin_low() const;
int get_bin_high() const;
int get_n_freq() const;
void print();

private:
std::string pfile;
// FILE NAMES IN FIRST FIVE LINES
std::string f_input_data; // name of input file containing sensor time series
std::string f_output_btr; // name of output file for BTR data
std::string f_output_bts; // name of output file for beam time series data
std::string f_array_xypos; // name of input file with hydrophone positions
int n_sen; // number of sensors in full array
int n_ptskip; // data points to skip before start of processing
int fft_length; // FFT length (must be power of 2);
int n_fft; // Number of FFTs to compute (0 => a lot )
float samp_rate; // sample rate
float low_freq; // lowest frequency to process
float high_freq; // highest frequency to process
int n_cov_avg; // number of covariance matrices to average
int n_fft_shift; // numberof FFTs to shift between sart of successive data
sets
int n_beams; // number of beams
float diagl;
float sv; // sound velocity

int *hyd_array;
int n_hyd; // inferred from input; NUMBER OF SENSORS USED

// calculated parameters;

```

```

    float delf;
    int bin_low;
    int bin_high;
    int n_freq;

};

#endif

```

Parameter class implementation (param_mvdr.cpp)

```

/* Minimum Variance Beamformer Example */
/* Param_mvdr Class implementation */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:06 $
    $Author: judd $
*/
/*
* param_mvdr.cpp
*
* Created by Randall Judd
* Copyright (c) 2004 SSC-SD. All rights reserved.
*
*/

#include<iostream>
#include<iomanip>
#include "param_mvdr.h"

Param_mvdr::Param_mvdr(std::string param_file){
    std::string aline;
    std::string delim(DELIMITER);

    pfile = param_file;
    std::ifstream param(param_file.c_str());

```

```

if (!param.is_open()){
    std::cout << "Error (" << __FILE__
        << "): can't open parameter file \"" << param_file.c_str() <<
        "\" -- aborting." << std::endl;
    exit(-1);
}

```

```

int mycount = 0;
while (!param.eof()){
    getline(param,aline);
    if (aline.find(delim) != std::string::npos){
        switch(mycount){
            case 0: // name of file with input time series in it
                param >> f_input_data;
                mycount++; break;
            case 1: // name of output BTR file
                param >> f_output_btr;
                mycount++; break;
            case 2: // name of output beam time series file
                param >> f_output_bts;
                mycount++; break;
            case 3: // name of sensor position file
                param >> f_array_xypos;
                mycount++; break;
            case 4:
                param >> sv;
                param >> samp_rate;
                param >> n_sen;

                mycount++; break;
            case 5:
                param >> low_freq;
                param >> high_freq;
                param >> fft_length;
                param >> n_cov_avg;
                param >> n_fft_shift;
                param >> n_beams;
                mycount++; break;

```

```

    case 6:
        param >> diag1;
        param >> n_ptskp;
        param >> n_fft;
        if (n_fft <= 0 ) n_fft = 1000000000;
        mycount++; break;
    case 7:
        hyd_array = new int[n_sen];
        n_hyd = 0;
        while(param.peek() != '\n'){
            param >> hyd_array[n_hyd];
            if((hyd_array[n_hyd] <= 0) || (hyd_array[n_hyd] > n_sen)) break;
            n_hyd ++;
            if(n_hyd >= n_sen) break;
        }
        mycount++; break;
    case 8:
        std::cerr << "to many input delimiters" << std::endl;
        break;
} // end switch
} // end if statement;
if (param.fail() && !param.eof()) {
    std::cout << " mycount" << mycount;
    std::cout << " parameter stream failure " << std::endl;
    break;
}
} // end input data (while statement done )

```

```

// calculated parameters
delf = samp_rate/fft_length;
bin_low = (int)(0.5 + low_freq/delf);
if (bin_low < 1) bin_low = 1;
bin_high = (int)(0.5 + high_freq/delf);
if (bin_high >= (fft_length/2)) bin_high = fft_length/2 - 1;
n_freq = bin_high + 1 - bin_low;
}

```

```

Param_mvdr::~Param_mvdr(){

```

```

    delete [] hyd_array;
}

std::string Param_mvdr::getName_f_input_data() const { return f_input_data;}
std::string Param_mvdr::getName_f_output_btr() const { return f_output_btr;}
std::string Param_mvdr::getName_f_output_bts() const { return f_output_bts;}
std::string Param_mvdr::getName_f_array_xypos() const { return f_array_xypos;}

float Param_mvdr::get_sv() const { return sv; }
float Param_mvdr::get_samp_rate() const { return samp_rate;}
int Param_mvdr::get_n_sen() const { return n_sen;}

float Param_mvdr::get_low_freq() const { return low_freq;}
float Param_mvdr::get_high_freq() const { return high_freq;}
int Param_mvdr::get_fft_length() const { return fft_length;}
int Param_mvdr::get_n_cov_avg() const { return n_cov_avg; }
int Param_mvdr::get_n_fft_shift() const { return n_fft_shift;}
int Param_mvdr::get_n_beams() const { return n_beams; }

float Param_mvdr::get_diagl() const { return diagl; }
int Param_mvdr::get_n_ptskp() const { return n_ptskp;}
int Param_mvdr::get_n_fft() const { return n_fft;}

int Param_mvdr::get_phone(int i) const {return hyd_array[i]; }
int *Param_mvdr::get_hyd_array() const {return hyd_array; }
int Param_mvdr::get_n_hyd() const {return n_hyd; }
float Param_mvdr::get_delf() const { return delf;}
int Param_mvdr::get_bin_low() const { return bin_low;}
int Param_mvdr::get_bin_high() const { return bin_high;}
int Param_mvdr::get_n_freq() const { return n_freq;}

void Param_mvdr::print(){
    std::string delim(DELIMITER);
    std::cout << std::setiosflags(std::ios::fixed | std::ios::showpoint);
    std::cout << std::setprecision(1);
    std::cout << "Input parameters:" << std::endl;
    std::cout << " This parameter file: " << pfile << std::endl;
    std::cout << std::endl;
}

```

```

std::cout << " Input time series file: " << std::endl;
std::cout << delim << std::endl; // case 0
std::cout << f_input_data << std::endl;

std::cout << " Output BTR file: " << std::endl;
std::cout << delim << std::endl; // case 1
std::cout << f_output_btr << std::endl;

std::cout << " Output Beamformed time series file: " << std::endl;
std::cout << delim << std::endl; // case 2
std::cout << f_output_bts << std::endl;

std::cout << " Sensor position file: " << std::endl;
std::cout << delim << std::endl; //case 3
std::cout << f_array_xypos << std::endl;

std::cout << "\n Sound speed: " \
    << sv << std::endl;
std::cout << " Sample rate: " \
    << samp_rate << std::endl;
std::cout << " Number of sensors: " \
    << n_sen << std::endl;
std::cout << delim << std::endl; //case 4
std::cout << sv << " " << samp_rate << " " << n_sen << " " << std::endl;

std::cout << "\n Requested lowest frequency to process: " \
    << low_freq << std::endl;
std::cout << " Requested highest frequency to process: " \
    << high_freq << std::endl;
std::cout << " FFT length: " \
    << fft_length << std::endl;
std::cout << " Number of FFTs to average per covariance matrix: " \
    << n_cov_avg << std::endl;
std::cout << " Number of FFTs to shift between data sets: " \
    << n_fft_shift << std::endl;
std::cout << " Number of beamforming angles: " \
    << n_beams << std::endl;
std::cout << delim << std::endl; //case 5

```

```

std::cout << low_freq << " ";
std::cout << high_freq << " ";
std::cout << fft_length << " ";
std::cout << n_cov_avg << " ";
std::cout << n_fft_shift << " ";
std::cout << n_beams << std::endl;

std::cout << "\n Diagonal loading factor:           " \
    << diagl << std::endl;
std::cout << " Number of data points to skip before processing:   " \
    << n_ptskp << std::endl;
std::cout << " Number of FFTs to be done:                           " \
    << n_fft << std::endl;
std::cout << delim << std::endl; //case 6
std::cout << diagl << " ";
std::cout << n_ptskp << " ";
std::cout << ((n_fft > 1000) ? 0 : n_fft) << std::endl;

std::cout << "\nArray sensors included in processing:" << std::endl;
std::cout << delim << std::endl; //case 7
for (int i=0; i < n_hyd; i++){
//     if (i % 20 == 0)
//         std::cout << std::endl << " "; // new output line every so often
    std::cout << hyd_array[i] << " ";
}
std::cout << std::endl;

std::cout << "\nValues calculated from input parameters:" << std::endl;
std::cout << " Number of sensors to input: " << n_hyd << std::endl;
std::cout << " Low frequency bin number (bin_low):           " \
    << bin_low << std::endl;
std::cout << " High frequency bin number (bin_high):           " \
    << bin_high << std::endl;
std::cout << " delf (fs/n):                                     " << delf <<
std::endl;
std::cout << " Actual lowest frequency to process:           " \
    << delf * bin_low << std::endl;
std::cout << " Actual highest frequency to process:           " \

```

```
    << delf * bin_high << std::endl;
std::cout << " Actual number of frequency bins to process: " \
    << n_freq << std::endl;
}
```

Data input class

A parameter object is input to create a data input object. On successful creation the data input object will contain an input stream from the data file.

```
Data_input input_data(param_object);
```

To bring data into the data input object use `get_xdata`

```
input_data.get_xdata(overlap); // set overlap parameter, get data
```

Note that the `overlap` is sticky and need not be set each call. Generally the call above will be used first and then the data object is initialized and the overlap will not need to be changed.

```
input_data.get_xdata(); // get the next set of data
```

The input data object keeps track of where it is in the data file. There is no random access to the data file available for this function. All input parameters except for the overlap parameter are set on object creation.

To get a matrix of the current input data suitable for use in the program use `x_data_mat()`. This will return a row major matrix with time samples along the row and sensors down the column. This includes all the data needed for the selected number of covariance averages.

```
input_data.x_data_mat(); // Matrix of input data
```

Note that the input data object will de-mean the data so that the matrix returned will have zero mean for each row. A side affect is that the data in the date matrix probably won't exactly match the data in the input file since the mean has been removed.

Data Input interface file (data_input.h)

```
/* Minimum Variance Beamformer Example */
/* Data Input API */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:06 $
    $Author: judd $
*/
/*
 * Data_input
 */
#ifndef DATA_INPUT
#define DATA_INPUT

#include"param_mvdr.h"
```

```

#include<iostream>
#include<fstream>
#include<string>
#include<vsip/math.hpp>

class Data_input
{
public:
    Data_input(const Param_mvdr&);
    ~Data_input();
    float **get_xdata(int);
    float **get_xdata();
    bool fail();
    vsip::Matrix<vsip::scalar_f> const & x_data_mat() const;
private:
    std::ifstream *input;
    float **xdata;
    int length;
    int N;           // length of entire data buffer row_length * col_length
    int nsen;       // row_length row major, length of row
    int n;          // col_length data length, row_length stride
    int nstart;     // start of next input segment (row number);
    int noverlap;   // start of overlap section (row) to be copied to front;
    int Nnew;       // number of new data points to read in;
    int Noverlap;
    int nover;
    int nh;
    int *ihyd;
    bool status;
    vsip::Matrix<vsip::scalar_f> x_data_mat_;
};

#endif

```

Data input class implementation (data_input.cpp)

```

/* Minimum Variance Beamformer Example */
/* Data_input Class implementation */

```

```

/*
$Revision: 1.2 $
$Date: 2005/05/05 17:41:06 $
$Author: judd $
*/
/*
 * Data_input.cpp
 */
#include"data_input.h"

#define MYFALSE 0

Data_input::Data_input(const Param_mvdr &param)
:x_data_mat_(param.get_n_hyd(),param.get_fft_length())
{
    input = new std::ifstream(param.getName_f_input_data().c_str(),std::ios::binary);
    nsen = param.get_n_sen();
    n =param.get_fft_length();
    N = n * nsen;
    nstart = 0;    // location of row to start new input
    noverlap = n - nstart;    // starts out pointing one past end of array
    Nnew = (n - nstart) * nsen;    // number of new data points to input
    Noverlap = N-Nnew;    // number of overlap values
    float *dataptr = new float[N];
    xdata = new float*[n];
    for(int i=0; i<n; i++) xdata[i]=dataptr + i * nsen;
    nh = param.get_n_hyd();
    ihyd = new int[nh];
    for(int i=0; i<nh; i++) ihyd[i] = param.get_hyd_array()[i];
    status = MYFALSE;
}

Data_input::~Data_input(){
    delete input;
    delete xdata[0];
    delete xdata;
    delete ihyd;
}

```

```

}

float **Data_input::get_xdata(int nover){
    // copy old data to start of array
    float *ptr_front = xdata[0];
    float *ptr_end   = xdata[noverlap];
    for(int i=0; i< N-Nnew; i++){ // if noverlap = col_length then N-Nnew will be 0
        *ptr_front = *ptr_end;
        ptr_front++; ptr_end++;
    }
    input->read((char *) xdata[nstart], Nnew * sizeof(float));
    if( input->eof() || input->fail()) status = !MYFALSE;
    // calculate new Nnew, nstart
    nstart = 2 * nover;
    Nnew = (n - nstart) * nsen;
    Noverlap = N-Nnew;
    noverlap = n - nstart;
    if(!status){
        vsip::index_type i=0,j=0;
        for(i=0; i<this->x_data_mat_.size(0); i++){
            vsip::index_type ind = ihyd[i] - 1;
            for(j=0; j<this->x_data_mat_.size(1); j++){
                this->x_data_mat_.put(i,j,xdata[j][ind]); // note: corner turn here
            }
            //de-mean
            this->x_data_mat_.row(i) -= vsip::meanval(this->x_data_mat_.row(i));
        }
    }
    return xdata;
}
}

```

```

float **Data_input::get_xdata(){
    float *ptr_front = xdata[0];
    float *ptr_end = xdata[noverlap];
    for(int i=0; i<Noverlap; i++){
        *ptr_front = *ptr_end;
        ptr_front++; ptr_end++;
    }
}

```

```

input->read((char *) xdata[nstart], Nnew * sizeof(float));
if( input->eof() || input->fail()) status = !MYFALSE;
if(!status){ // copy data do matrix and demean rowise
    vsip::index_type i=0,j=0;
    for(i=0; i<this->x_data_mat_.size(0); i++){
        vsip::index_type ind = ihyd[i] - 1;
        for(j=0; j<this->x_data_mat_.size(1); j++){
            this->x_data_mat_.put(i,j,xdata[j][ind]); // note: corner turn here
        }
        //de-mean
        this->x_data_mat_.row(i) -= vsip::meanval(this->x_data_mat_.row(i));
    }
}
return xdata;
}

bool Data_input::fail() { return status;}

vsip::Matrix<vsip::scalar_f> const&
Data_input::x_data_mat() const
{
    return this->x_data_mat_;
}

```

Array class

This class is used to create an object to hold array sensor position information. On instantiation an array file name is passed to the class. The file has x, y, z position information for each sensor, one sensor on each line. The file is opened and the data is input into the create routine. A spatial transformation is done on the array so that the first and last sensors lie equidistant from the origin along the y axis so that the first sensor is on the south end of the translated array. The longest distance (distance between the two sensors that lie farthest apart) through the array is also calculated.

In the prototype below the file name is a string indicating the array sensor file. The number of sensors is the number of sensors in the file. The array of valid sensors is a standard "C" integer array containing the numbers of valid sensors that will be processed. For instance sensor 5 would correspond to line 5 of the array sensor file. The number of valid sensors is the length of this array. For most simple cases the number of sensors in the array sensor file and the sensor numbers listed in the array of valid sensors will be the same and will just be [1 2 3 ... nsen].

The array sensor file must have a sensor position line for every row of the time series data input file. The array of valid sensors controls which rows are actually read in. The array held in the array object will correspond to the valid sensors.

Note that all inputs to the array class are available in the parameter object.

```
Array myarray(  
    file_name,  
    number_of_sensors,  
    array_of_valid_sensors,  
    number_of_valid_sensors);  
myarray.x() // return a vector of x array sensor locations  
myarray.y() // return a vector of y array sensor locations  
myarray.z() // return a vector of z array sensor locations  
myarray.max() // return size of array
```

Array Sensor File

The array sensor file is ASCII and holds sensor position data for the array. For the example array described above (in the parameter file) the sensor position file would look like

```
0.0 0.0 0.0
1.0 0.0 0.0
2.0 0.0 0.0
3.0 0.0 0.0
4.0 0.0 0.0
5.0 0.0 0.0
6.0 0.0 0.0
7.0 0.0 0.0
8.0 0.0 0.0
9.0 0.0 0.0
```

Array class interface file (array.h)

```
/* Minimum Variance Beamformer Example */
/* Array API */

/*
   $Revision: 1.2 $
   $Date: 2005/05/05 17:41:05 $
   $Author: judd $
*/

#ifndef ARRAY_HEADER_GUARD
#define ARRAY_HEADER_GUARD

#include <string>
#include <vsip/vector.hpp>

class Array
{
public:

    Array(
        std::string array_file,
        vsip::length_type nsen,
        const int *ihyd,
        int nh);

    vsip::Vector<vsip::scalar_f> const& x () const;
```

```
vsip::Vector<vsip::scalar_f> const& y () const;
vsip::Vector<vsip::scalar_f> const& z () const;
vsip::scalar_f max() const;
void print(const int*, int);
```

private:

```
vsip::Vector<vsip::scalar_f> x_;
vsip::Vector<vsip::scalar_f> y_;
vsip::Vector<vsip::scalar_f> z_;
vsip::scalar_f max_;
```

```
};
```

```
#endif
```

Array class implementation (array.cpp)

```
/* Minimum Variance Beamformer Example */
/* Array Class implementation          */
/*
/*
/*
  $Revision: 1.2 $
  $Date: 2005/05/05 17:41:05 $
  $Author: judd $
*/

/* Note that as part of the array object instantiation the
 * Array class translates and rotates the input array so
 * that the first and last phones are on the y axis,
 * phone 1 at the south end, with equal negative and
 * positive y values for the first and last phone. */

#include <iostream>
#include <fstream>
#include <iomanip>
#include <string>
```

```

#include <vsip/math.hpp>

#include "array.h"

Array::Array(
    std::string array_file,
    vsip::length_type nsen,
    const int *ihyd,
    int nh)
: x_(nsen), y_(nsen), z_(nsen)
{
    // get array data from file
    std::ifstream myarray(array_file.c_str());
    vsip::scalar_f xt,yt,zt;
    for (unsigned int i = 0; i < nsen; i++)
    {
        myarray >> xt >> yt >> zt;
        this->x_.put(i,xt);
        this->y_.put(i,yt);
        this->z_.put(i,zt);
    }

    // calculate max dimension for overlap calculation
    max_ = 0;
    for (unsigned int i = 0; i < nsen-1; i++)
    {
        vsip::scalar_f xt_i= this->x_.get(i);
        vsip::scalar_f yt_i= this->y_.get(i);
        vsip::scalar_f zt_i= this->z_.get(i);
        for (unsigned int j = i + 1; j < nsen; j++)
        {
            xt = xt_i - this->x_.get(j);
            yt = yt_i - this->y_.get(j);
            zt = zt_i - this->z_.get(j);
            vsip::scalar_f atest = xt * xt + yt * yt; // + zt * zt;
            if(atest > max_) max_ = atest;
        }
    }
}

```

```

max_ = (vsip::scalar_f) sqrt(max_);

// calculate array transformation
vsip::index_type i1 = (vsip::index_type) (ihyd[0] - 1);
vsip::index_type i2 = (vsip::index_type) (ihyd[nh-1] - 1);

xt = this->x_.get(i1);
yt = this->y_.get(i1);
this->x_ = - xt + this->x_;
this->y_ = - yt + this->y_;

xt = this->x_.get(i2); yt = this->y_.get(i2);
vsip::scalar_f fac = (vsip::scalar_f)sqrt(xt * xt + yt * yt);
xt /= fac; yt /= fac;
vsip::Vector<vsip::scalar_f> temp(nsen);
temp = xt * this->x_ + yt * this->y_;
this->y_ = xt * this->y_;
this->x_ = -yt * this->x_;
this->x_ = -(this->y_ + this->x_);
this->y_ = temp;

fac = 0.5 * (this->y_.get(i1) + this->y_.get(i2));
this->y_ = -fac + this->y_;
}

vsip::Vector<vsip::scalar_f> const&
Array::x () const
{
    return this->x_;
}

vsip::Vector<vsip::scalar_f> const&
Array::y () const
{
    return this->y_;
}

vsip::Vector<vsip::scalar_f> const&

```

```

Array::z () const
{
    return this->z_;
}

vsip::scalar_f
Array::max() const
{
    return max_;
}

void
Array::print(const int *ihyd, int nh)
{
    for (unsigned int i = 0; i < nh; i++)
    {
        vsip::index_type ind = ihyd[i] - 1;
        vsip::scalar_f x = this->x_.get((vsip::index_type) ind);
        vsip::scalar_f y = this->y_.get((vsip::index_type) ind);
        std::cout << std::setiosflags(std::ios::fixed | std::ios::showpoint);
        std::cout << std::setw(4) << i+1;
        std::cout << std::setw(4) << ind + 1;
        std::cout << std::setprecision(2) << std::setw(8) << x;
        std::cout << std::setprecision(2) << std::setw(8) << y;
        std::cout << std::endl;
    }
}

// end Array class

```

Beam-Steering coefficient class

The beam steering coefficient object is a Cube object (see section below). The Beam_steer_coef class instantiates a cube having matrices of size (number of sensors) by (number of beams); one matrix for each frequency to be processed. Each matrix is filled with the beam steering coefficients for a frequency of interest.

Usage:

```
Beam_steer_coef steering_vectors(parameter_object, array_object);
/* Note here "freq" is an index into a column for the frequency */
steering_vectors.get_xy(freq); // return matrix for frequency
```

Beam-Steering coefficient interface (beam_steer_coef.h)

```
/* Minimum Variance Beamformer Example */
/* Beam_steer_coef API */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:05 $
    $Author: judd $
*/
/*
 * Beam_steer_coef.h
 *
 * Created by Dennis Cottel on Mon Apr 19 2004.
 * Copyright (c) 2004 SSC-SD. All rights reserved.
 *
 */

#ifndef BEAM_STEER_H
#define BEAM_STEER_H

#include "param_mvdr.h"
#include "array.h"
#include "cube.h"
#include "phat.h"

class Beam_steer_coef : public Cube<vsip::cscalar_f>
{
```

```
public:
```

```
    Beam_steer_coef(Param_mvdr &parameters, const Array &myarray);  
    ~Beam_steer_coef();  
    void print_phat();
```

```
private:
```

```
    Phat* myphat;
```

```
};
```

```
#endif
```

Beam steering coefficient class implementation (beam_steer_coef.cpp)

```
/* Minimum Variance Beamformer Example */  
/* Beam_steer_coef Class implementation */  
  
/*  
    $Revision: 1.2 $  
    $Date: 2005/05/05 17:41:05 $  
    $Author: judd $  
*/  
/*  
 * beam_steer_coef.cpp  
 *  
 * Created by Dennis Cottel on Mon Apr 19 2004.  
 * Copyright (c) 2004 SSC-SD. All rights reserved.  
 *  
 * Build a data cube to hold the complex float replica data. The  
 * three dimensions are:  
 *   number of hydrophones (nh) -- unit stride (adjacent elements)  
 *   number of angles (nang) -- stride = nh * element_size  
 *   number of frequencies (nfreq) -- stride = nh * nang * element_size  
 */  
  
#include<vsip/selgen.hpp>  
#include "beam_steer_coef.h"  
//#include "phat.h"  
//#include "cube.h"  
  
#ifndef M_PI  
#define M_PI (3.14159265358979323846)
```

```

#endif

/*-----*/
/* Beam_steer_coef constructor */
/*-----*/

Beam_steer_coef::Beam_steer_coef(Param_mvdr &parameters, const Array &myarray):
    Cube<vsip::cscalar_f>(parameters.get_n_freq(),
                        parameters.get_n_beams(),
                        parameters.get_n_hyd())
{
    // Variables derived from the input parameter object

    const float delf      = parameters.get_delf ();
    const float sv        = parameters.get_sv ();
    const int   nh        = parameters.get_n_hyd ();
    const int   nang      = parameters.get_n_beams ();
    const int   *ihyd     = parameters.get_hyd_array ();
    const int   ibin1     = parameters.get_bin_low ();
    const int   nfreq     = parameters.get_n_freq ();

    // Phase array tables

    myphat = new Phat(parameters.get_n_beams());
    vsip::Vector<vsip::scalar_f> k( vsip::ramp ((vsip::scalar_f) (delf * ibin1),
                                             (vsip::scalar_f) delf, nfreq));
    k *= (2.0 * M_PI/sv);
    /* Main frequency loop */

    for (int ifreq = 0; ifreq < nfreq; ifreq++) {
        // Get access to the (nh x nang) matrix for this frequency.

        vsip::Matrix<vsip::cscalar_f> replica = this->get_xy(ifreq);
        const float waveno = k.get(ifreq); // 2.0 * M_PI * freq / sv
        /* Simply doing the dot product of the unit vector pointing to the
           planewave source with the vector pointing to the hydrophone, and
           then multiplying this by square root of -1 times the wave number
           gives the argument for the exponential of the planewave replica. */
    }
}

```

```

/* Loop over bearings */
for (int j = 0; j < nang; j++) {
    /* Calculate replicas for the array */
    float s = myphat->s().get(j);
    float c = myphat->c().get(j);
    for (int i = 0; i < nh; i++) {
        const int nphone = ihyd[i] - 1;
        const float fac = s * myarray.x().get(nphone) + c *
myarray.y().get(nphone);
        const float arg = waveno * fac;

        replica.put(j /*angle*/, i /*hydrophone*/,
vsip::cscalar_f(cos(arg),sin(arg)));
    }
} // end frequency loop

return;

} // Beam_steer_coef()

/*-----*/
/* Beam_steer_coef destructor */
/*-----*/

Beam_steer_coef::~Beam_steer_coef() {
    delete myphat;
} // ~Beam_steer_coef()

/*-----*/
/* print the phase array table */
/*-----*/
void
Beam_steer_coef::print_phat() {
    // print phase array table print for debugging and reassurance
    myphat->print();
} // print_phat()
// end beam_steer_coef.cpp

```

Phase array table class

The phase array creates an object which contains two vectors, one each for sine and cosine values for the beamforming phase delays.

Usage:

```
Phat myphat(number_of_beams);
myphat.c(); // return the cosine vector
myphat.s(); // return the sine vector
```

Phase array table interface (phat.h)

```
/* Minimum Variance Beamformer Example */
/* Phat API */

/*
$Revision: 1.2 $
$date: 2005/05/05 17:41:06 $
$Author: judd $
*/

/* Set up cosine and sine tables for the beamformer
bearings. Beams are equally spaced in cosine space.
Beam #1 is North, and beam #(1 + nang / 2) is South
if there are an even number of beams or else there
is no beam that points directly South. */

#ifndef PHAT_H
#define PHAT_H

#include <vsip/vector.hpp>

class Phat
{
public:

    Phat (vsip::length_type nang);
    vsip::Vector<vsip::scalar_f> const& c () const;
    vsip::Vector<vsip::scalar_f> const& s () const;
```

```

void print();

private:

    vsip::Vector<vsip::scalar_f> cphat_;
    vsip::Vector<vsip::scalar_f> sphat_;

};

#endif

```

Phase array table implementation (phat.cpp)

```

/* Minimum Variance Beamformer Example */
/* Phat Class implementation          */
/* Phase Array Table Calculation     */

/*
   $Revision: 1.2 $
   $Date: 2005/05/05 17:41:06 $
   $Author: judd $
*/

/* Set up cosine and sine tables for the beamformer
   bearings. Beams are equally spaced in cosine space.
   Beam #1 is North, and beam #(1 + nang / 2) is South
   if there are an even number of beams or else there
   is no beam that points directly South. */

#include <iostream>
#include <iomanip>
#include <vsip/math.hpp>

#include "phat.h"

//-----
// Phat constructor.
//-----

```

```

Phat::Phat (vsip::length_type nang) : cphat_ (nang), sphat_ (nang)
{
  if (nang % 2){ // nang must be odd
    const int ceil_nang_over2 = nang/2 + 1;
    const vsip::scalar_f delta = \
      (2.0F + 2.0F/(vsip::scalar_f)nang) / ((vsip::scalar_f)ceil_nang_over2);
    this->cphat_.put(0,-1.0); this->sphat_.put(0,0.0);
    for (unsigned int j = 1; j < (nang + 1) / 2; j++) {
      const vsip::scalar_f a = -(1.0F - delta * (vsip::scalar_f) j);
      const vsip::scalar_f b = sqrt((1.0F+a) * fabs(1.0 - a));
      this->cphat_.put(j,a);
      this->sphat_.put(j,-b);
      this->cphat_.put(nang-j,a);
      this->sphat_.put(nang-j,b);
    }
  } else { // nang must be even
    const vsip::scalar_f delta = 4.0F / (vsip::scalar_f) (nang);
    this->cphat_.put(0,-1.0); this->sphat_.put(0,0.0);
    this->cphat_.put(nang/2,1.0); this->sphat_.put(nang/2,0.0);
    for (unsigned int j = 1; j < nang / 2; j++) {
      vsip::scalar_f a = -(1.0F - delta * (vsip::scalar_f) j);
      vsip::scalar_f b = sqrt((1.0F+a) * fabs(1.0 - a));
      this->cphat_.put(j,a);
      this->sphat_.put(j,-b);
      this->cphat_.put(nang-j,a);
      this->sphat_.put(nang-j,b);
    }
  }
}

//-----
// Return the cosine array.
//-----
vsip::Vector<vsip::scalar_f> const&
Phat::c() const
{
  return this->cphat_;
}

```

```

//-----
// Return the sine array.
//-----
vsip::Vector<vsip::scalar_f> const&
Phat::s() const
{
    return this->sphat_;
}

//-----
// Print the array contents.
//-----
void
Phat::print()
{
    #ifndef M_PI
    #define M_PI (3.14159265358979323846)
    #endif
    for (unsigned int i = 0; i < this->cphat_.size(); i++) {
        vsip::scalar_f c = this->cphat_.get((vsip::index_type) i);
        vsip::scalar_f s = this->sphat_.get((vsip::index_type) i);
        vsip::scalar_f a = 180.0 * atan2(-s, -c) / M_PI;
        std::cout << std::setiosflags(std::ios::fixed | std::ios::showpoint);
        std::cout << "cphat[" << std::setw(2) << i << "] = ";
        std::cout << std::setprecision(5) << std::setw(8) << c;
        std::cout << " sphat[" << std::setw(2) << i << "] = ";
        std::cout << std::setprecision(5) << std::setw(8) << s;
        std::cout << " angle = " << std::setprecision(3);
        std::cout << std::setw(8) << a << std::endl;
    }
}
// end phat.cpp

```

Cube class

The Cube class was done to allow an indexed set of (equal size) matrices to be defined. To instantiate a cube object which contains L matrices of size M rows by N columns use

```
Cube<vsip::scalar_f> mycube(L,M,N);
```

To retrieve a pointer to the “ i ’th” matrix use.

```
mycube.get_xy(i);
```

Cube interface and implementation (cube.h)

```
/* Minimum Variance Beamformer Example */
/* Cube API and implementation      */

/*
    $Revision: 1.2 $
    $Date: 2005/05/05 17:41:06 $
    $Author: judd $
*/
/*
 * Cube.h
 *
 * This is a simple way to get some of the functionality of Tensors
 * which are not available in the initial VSIPL++ reference library.
 *
 * This class should eventually be removed and replaced with Tensors.
 *
 * Created by Dennis Cottle on Mon Apr 19 2004.
 * Copyright (c) 2004 SSC-SD. All rights reserved.
 */

#ifndef CUBE_HEADER_GUARD
#define CUBE_HEADER_GUARD

#include "vsip/vector.hpp"
#include "vsip/matrix.hpp"

template <class T>
class Cube
```

```

{
public:
    //-----
    // Cube constructor.
    //-----

Cube<T> (vsip::length_type nmatrices, vsip::length_type y, vsip::length_type x)
{
    // Allocate an array to hold pointers to nmatrices VSIPL++ Matrix objects

    number_of_matrices = nmatrices;

    the_cube = new Matrixp_t[number_of_matrices];

    for (unsigned int ii = 0; ii < number_of_matrices; ii++)
    {
        the_cube[ii] = new vsip::Matrix<T> (y, x);
    }
}

//-----
// Cube destructor.
//-----

~Cube<T> ()
{
    for (unsigned int ii = 0; ii < number_of_matrices; ii++)
    {
        delete the_cube[ii];
    }
    delete [] the_cube;
}

//-----
// Return the x-y matrix at the given matrix index.
//
// NOTE: The resulting Matrix view will still refer to the
// original memory locations.

```

```

//-----

vsip::Matrix<T> get_xy (const vsip::index_type which_matrix) const
{
    return *(the_cube[which_matrix]);
}

//-----
// Return a row_type subview of the row vector at the given
// (row,matrix) index.
//
// NOTE: If assigning the result directly to a variable,
// make sure the variable is a subview and not a Vector
// view. Otherwise a copy will occur and the Vector does
// not reference the original memory locations.
//-----

typename vsip::Matrix<T>::row_type
    get_x (const vsip::index_type which_row,
          const vsip::index_type which_matrix) const
{
    return the_cube[which_matrix]->row(which_row);
}

private:

    typedef vsip::Matrix<T> *Matrixp_t;

    Matrixp_t *the_cube;

    vsip::length_type number_of_matrices;

}; // end class Cube

#endif

```

Vector/Matrix print utility

This is a utility that allows one to print vectors or matrices to standard output. It is not used in the program but is included (in *main*) for debugging and data output if needed.

Vector/Matrix print implementation (VUPP_print.h)

```
/*
   $Revision: 1.2 $
   $Date: 2005/05/05 17:41:05 $
   $Author: judd $
*/

/* This utility routine will allow VSIPL vectors *
 * and matrices to be easily printed using      *
 * the operator<<.                               */

/* Jeffrey Oldham provided the basic recipe for this code */

#ifndef VUPP_PRINT_H
#define VUPP_PRINT_H

#include<iostream>
#include<vsip/vector.hpp>
#include<vsip/matrix.hpp>

/* Print a Vector to an ostream. */
template <typename T,
          typename Block>
inline std::ostream&
operator<< (std::ostream& out,
           const vsip::Vector<T, Block>& v) VSIP_NOTHROW
{
    out << "[";
    for (vsip::index_type i = 0; i < v.size(); ++i)
        out << v.get(i) << std::endl;
    return out << "]" << std::endl;
}
```

```

template <typename T, typename Block>
inline std::ostream&
operator<< (std::ostream& out, const vsip::Matrix<T, Block>& v) VSIP_NOTHROW
{
    vsip::index_type row, col;
    out << "[";
    for(row=0; row<v.size(0); row++){
        for(col=0; col<v.size(1); col++)
            out << v.get(row,col) << " ";
        out << std::endl;
    }
    return out << "]" << std::endl;
}

#endif

```