Constrained Least Squares Fit of a Filter Bank to an Arbitrary Magnitude Frequency Response

Jon Dattorro

0. Introduction

There are numerous applications of filter banks in audio signal processing. A common example is the use of a Graphic Equalizer to tailor home or automobile frequency response for the reproduction of music. The Graphic Equalizer is a filter bank comprising a multiplicity of distinct low-order filters (bank-filters), arranged in parallel, each predominating in only a fixed portion of the audio spectrum. The Graphic Equalizer is one instance where the composite filter-bank transfer function is used as compensation to approximate some desired frequency response.

A natural question arises pertaining to the optimum setting of each bank-filter. The answer encompasses known results in the branch of mathematics called Linear Algebra. Two innovations are required, however: 1**)** a method of synthesizing the desired phase response when given only the magnitude of the desired frequency response, 2**)** a method of bounding the solution while still yielding the optimum bank-filter setting.

1. Problem Statement

We are given a fixed bank of filters, together spanning some large bandwidth in the audio spectrum. The characteristics of each individual filter from the bank (filter selectivity, center frequency, magnitude/phase response, etc.) are predetermined and beyond our control at this point. The only control available to us is the level of the output of each individual filter from the bank. These levels are not without bounds and we will map the full scale output of any individual bank-filter to a level of 1, while the inverted full scale output of any individual bank-filter will be mapped to a level of -1. (Level inversion provides us with the capability of subtracting an individual bank-filter output from the composite transfer function.)

We are also given only the magnitude of a desired frequency response that we interpret as the frequency response magnitude of some desired high-order transfer function. This desired response is, in general, beyond the capability of any one bank-filter to match in any sense. Further, the desired magnitude response will not necessarily have any characteristics in common with any one of the bank-filters.

The problem we are presented with is how to adjust each individual bank-filter level so that the composite magnitude response of the bank matches, as closely as possible in some sense, the desired magnitude response.

2. Classical Solution

The problem, as stated above, can be formulated using Linear Algebra. Linear Algebra comprises all of matrix theory. The classical problem of Linear Algebra is the solution of simultaneous equations, typically formulated as:

$$\mathbf{A}\,\mathbf{x} = \mathbf{b} \qquad (0.1)$$

where $\mathbf{A}$ is a matrix, $\mathbf{x}$ and $\mathbf{b}$ are vectors. The vector $\mathbf{x}$ is the unknown here, while $\mathbf{A}$ and $\mathbf{b}$ are given. When there are as many equations as unknowns and all the equations are independent, the matrix $\mathbf{A}$ is square and the solution is simply:

$$\mathbf{x} = \mathbf{A}^{-1}\,\mathbf{b} \qquad (0.2)$$

Quite often, as in our case, there are more equations than unknowns and the simple solution in (0.2) can no longer be applied because $\mathbf{A}$, not being square, is no longer invertible. Further, there can be no exact solution to (0.1) if $\mathbf{b}$ is not in the range spanned by the columns of $\mathbf{A}$. When $\mathbf{A}$ has more rows than columns, this case is called "over-determined" and solved using the method of *least squares*:

$$\mathbf{x} = (\mathbf{A}^{H}\,\mathbf{A})^{-1}\,\mathbf{A}^{H}\,\mathbf{b} \qquad (0.3)$$

Superscript H denotes the conjugate transpose of the matrix $\mathbf{A}$. That solution is well known, [Strang] and works when $(\mathbf{A}^{H}\,\mathbf{A})$ is invertible (when the columns of $\mathbf{A}$ are independent) whether or not $\mathbf{b}$ is in the range of $\mathbf{A}$.

2.1 Relation to the Problem at Hand

We will solve our problem in the frequency domain. The bank-filter levels that are under our control act as scalars in both the time and frequency domains. In the time domain each level multiplies a real audio signal emanating from its respective bank-filter output. But in the frequency domain, we use each level to scale the *complex*-valued response of each bank-filter; *i.e.*, each bank-filter has both a magnitude and a phase response. It is the magnitude that is scaled by the corresponding level.

We now relate our problem to least squares of linear algebra:

We first load each individual column of the **A** matrix with the full scale frequency response (sampled in the frequency domain) of one filter from the bank; one filter per column, each column entry being a complex sample of the filter response at a particular frequency. In other words, each column of the **A** matrix holds the sampled frequency response for a different bank-filter. Hence, there are as many columns as there are bank-filters p, while there are as many rows of the **A** matrix as there are frequencies of evaluation N; **A** is of dimension [N x p].

The vector **b** holds the desired magnitude response, which was given, sampled at the same frequencies corresponding to the rows of **A**. The number of elements in **b** is therefore the same as the number of frequencies of evaluation; **b** is [N x 1].

The vector **x** will hold the levels of the bank-filters that that we seek. **x** has as many elements as there are bank-filters; **x** is [p x 1]. When the number of frequencies of evaluation N exceeds the number of elements in **x**, we must use the Least Squares solution (0.3). We will assume N > p throughout.

2.2 Deficiencies of Classical Solution:
We are ready to solve (0.3), and we can do so using only the information we have been given.

1**)** We note that we have been given only the magnitude of the desired response, and we wonder if a more complete specification, including the desired phase response, would result in a better approximation, **x**. The answer is in the affirmative as that becomes our first deficiency. In fact, if we unwittingly use only the desired magnitude response, we are implicitly demanding that the phase of the filter bank become zero over the entire frequency range! That unrealistic assumption pulls the solution away from the optimal solution by a wide margin simply because it is impossible to produce a zero-phase transfer function using the filters which physically comprise the bank.

2**)** The second deficiency that we discover is sometimes the solution **x** demands that the absolute level of some of the individual bank-filters exceed 1. That means the solution demands more output magnitude from a bank-filter than it is physically capable of.

This latter deficiency is independent of the first, so we have two separate problems. We point out that any fitting process that is not given desired phase information, and whose solution can be formulated as a system of linear equations, will be required to solve the former deficiency unless a sub-optimal solution is satisfactory. Also, any fitting process whose solution is bounded by physical constraints, and whose solution can be formulated as a system of linear equations, cannot ignore the latter deficiency.

3. Innovations:

3.1 Phase Least-Squares (PhiLS)

The synthetic desired phase response that we beset upon ourselves to make is found empirically to be unique; *i.e.*, there is one synthetic phase response which when chosen results in a closer fit $\mathbf{x}$ than could ever be attained having chosen any other. One possible choice that comes to mind is the minimum-phase transfer function corresponding to the desired magnitude response. It is easily found [Opp] and definitely unique. It is not the one we want, unfortunately, because the bank-filters themselves are not necessarily minimum phase.

We derive the optimum synthetic desired phase response by iteration; that is, we define the initial condition:

$$\mathbf{b}^{(0)} = \mathbf{b} \qquad (0.4)$$

This says we start off by placing the given (zero-phase) desired magnitude response $\mathbf{b}$ into $\mathbf{b}^{(0)}$. We then solve for $\mathbf{x}^{(1)}$, using (0.3);

$$\mathbf{x}^{(\mathbf{i})} = (\mathbf{A}^{H}\,\mathbf{A})^{-1}\,\mathbf{A}^{H}\,\mathbf{b}^{(\mathbf{i}-1)} \qquad ; \text{ i = 1 -> infinity .} \qquad (0.5)$$

$\mathbf{x}^{(\mathbf{i})}$ denotes that solution found at our $\mathbf{i}^{th}$ iteration of the classical Least Squares solution. (Note the the classical Least Squares solution (0.3) requires no iteration; we are imposing the iterations.) Once we have $\mathbf{x}^{(1)}$, we then calculate $\mathbf{b}^{(1)}$ using (0.1). From that calculation we retain only the phase response information; we replace the magnitude response in $\mathbf{b}^{(1)}$ with the original desired magnitude response $\mathbf{b}$ thereby yielding our desired $\mathbf{b}^{(1)}$. We then use $\mathbf{b}^{(1)}$ to calculate $\mathbf{x}^{(2)}$ as in (0.5), and so on... The process repeats until $\mathbf{b}^{(\mathbf{i})}$ converges, which we do not prove here. When convergence is attained, the optimal synthetic desired phase response will have been found, but, more importantly, the optimal corresponding $\mathbf{x}$ vector that holds all the bank-filter levels will also have been found.

3.2 Level-Boundary Projection

The obvious remedy for bounding the solution **x** is to clip the requested filter levels in **x** to +/-1. That appealing solution is far from optimal.

The solution we will use has its roots in a branch of mathematics called Computational Geometry. [Geo] This mathematical technique requires us to think of the columns of the [N x p] matrix **A** as vectors in hyperspace (a vector space of more than 3 dimensions). Assuming the columns are independent, the dimension of the subspace spanned by the columns is equal to the number of columns p. (N > p) The non-orthogonal vectors that we assume span that entire subspace are simply the columns of **A**.

For the sake of example, let us assume that p = 3; *i.e.*, our matrix **A** has only three columns. Then the vectors that are the columns of **A** span a 3-dimensional subspace, but the vectors are pointing in capricious directions. Further, let us suppose that each of these vectors is, in general, of different but finite length. The entire range of levels that could possibly be found in **x** then determines precisely which points **A x** in 3-space can be reached by a linear combination of the **A**-column vectors.

If we constrain the allowable range of levels **x** to reside within particular bounds (say, -1 -> +1), then the reachable set of points in 3-space becomes a distinct three-dimensional object! All the surfaces of this object lay in a finite region of 3-space because the bounds on **x** prevents any of the **A**-column vectors from reaching out to infinity.

Our solution then is simply this: When some requested filter level in a Least Squares solution **x** exceeds the prescribed range, we project the point **A x** onto the closest point lying upon the 3-dimensional object described above. The idea of projection is not foreign to the theory of Least Squares. There, one is accustomed to projecting points onto planes or higher-dimensional subspaces stretching out to infinity. The innovation we have developed is how to project a point onto an object contained by that space. (See C program in Appendix.) When the dimension of the space exceeds 3, the same arguments apply but are more difficult to visualize.

If the technique of PhiLS (described above) is being used, the projection would be performed after PhiLS is completed. PhiLS is not required, however; the projection technique described will work for any solution **x** found by any means to the problem formulated as **A x** = **b**.

In the past, Least Squares problems having constraints such as this were solved using a technique called Linear Programming. That technique is akin to an exhaustive search, so it is expensive computationally. The method of

constraining the solution that we have proposed above requires no iteration, hence no search is required.

References

[Geo] Toussaint, G.T., "Scanning the Issue, Computational Geometry", Proceedings of the IEEE , Sept.'92, vol.80, no.9, pg.1347

[Opp] Oppenheim, Alan V., and Schafer, Ronald W., *Discrete Time Signal Processing,* 2nd ed., Prentice Hall, Englewood Cliffs, NJ, 1989

[Strang] Strang, Gilbert, *Linear Algebra and Its Applications*, 2nd ed., Academic Press, New York City, 1980

Appendix - C program
This code is extraordinarily well tested. Not all subroutines are presented.

```
//PhiLS and saturation projection -Jon Dattorro
#include <windows.h>
#include <essdll.h>
#include <stdio.h>
#include <math.h>

#include "ess_win.h"
#include "ess.h"
#include "curves.h"
#include "dataset.h"
#include "cnf.h"
#include "wnds.h"
#include "mathfit.h"
#include "mathintr.h"
#include "fatttbl.h"
#include "peaks.h"
#include "rescurs.h"
#include "convert.h"
#include "regs.h"
#include "messages.h"
#include "mathgain.h"
#include "vents.h"
#include "gaintbl.h"
#include "calctbl.h"
#include "mathmatx.h"
#include "mathchip.h"
#include "mathtarg.h"

#ifdef DBOUT
#include "debugout.h"
#endif

#define fmax(a,b) (((a)>(b))?(a):(b))
#define fmin(a,b) (((a)<(b))?(a):(b))
#define sqr(x)                                              (x)*(x)

#define WARP_FACTOR 1.0

extern  HANDLE  hMemCalcTable;

extern DATA_SET_TYPE DataSet;  /* Working Data Set */
extern WINDOW_TABLE  Wnds;
extern double H_real;
```

```
extern double H_imag;
extern double VentCutOff[NUM_VENTSIZES];
extern BOOL bUNIPOLAR;
extern BOOL bRosco[2];

// switch to let us check validity of saturation projection
extern BOOL bJONSWITCH;

/* Answer to BIG question:   pointer[NUM_FILTERS] are filter gains.*/
/* This program currently provides positive and negative */
/* gains in the pointer[] array.  */
/* The spline_curve() subroutine returns Desired Magnitudes*/
/* in the range of 0. to 1.  Discontinuities make the curve fit problem more  */
/* difficult.  Regions of zero magnitude are also intensive to fit.*/
/*     At the very least, spline_curve() must be piecewise linear (as */
/* opposed to piecewise constant), and the number of dB/octave transitions    */
/* must be watched.  Otherwise, this program will begin to clip pointer[]    */
/* values as they attempt to exceed |1| to meet the curve demands. */


int fit(LOCALHANDLE hMFitData,
    double *fitval,
    WORD Side,
    double Fmin,
    double Fmax,
    BYTE FATTIndex)
{

CALC_TABLE FAR  *CalcTable;

BOOL ret_val;
register int i, j, k;
int numits, criterion_met;
int total_load;
int dimension, basis[NUM_FILTERS], eliminated_vector, howmany_negs, howmany_sats;
double min_angle, basis_angle;
double tempest;
double imag, real;
double FAR *pointer;
double FAR *pointer1;
double FAR *pointer2;
double pia2;
double error, tempd, current_err, old_err, tempp, tempe;
double projection[2*NUM_FREQS], projection_trans[2*NUM_FREQS];
double quantize(), perturbation(), saturate(), reduction(), signum();
double pointer_copy[NUM_FILTERS], pointer_trans[NUM_FILTERS];
```

```
struct mag_phase {
       double magtu[NUM_FREQS];
       double phase[NUM_FREQS];
                  } b;

FitData  FAR *lpFitData;
GLOBALHANDLE hFitData;
MFitData *pMFitData;
double max_magtu, fatt_diff;


#if defined(DBOUT)
DebugOut("\r\n\r\n\r\n\r\nSize needed %u   ",sizeof(FitData));
DebugOut("Size available  %lu\r\n",GlobalCompact((DWORD)sizeof(FitData)));
#endif


  hFitData = GlobalAlloc(GMEM_MOVEABLE, (DWORD)sizeof(FitData));
  lpFitData = (FitData FAR *) GlobalLock(hFitData);

  pMFitData = (MFitData *) LocalLock(hMFitData);

/****************************** init ***********************************/
  ret_val = FIT_OK;
  pia2 = 8.*atan(1.);

  for(i=0; i<NUM_FILTERS; i++)
  {
    lpFitData->K[i] = 1.;
    lpFitData->K_buf[i] = 0.;
  }
  pointer1 = lpFitData->K_buf;
  pointer2 = lpFitData->K;
  pointer = lpFitData->K;

  for(i=0; i<NUM_FREQS; i++)
  {  /* Get magnitude curve, form Least Squares inverse weighting. */
    /* magnitude limiting is performed in mathtarg.c */
    b.magtu[i] = pMFitData->Mags[i];

/* WEIGHT[] actually holds desired weights squared.        */
    if(pMFitData->Freqs[i] < Fmin) lpFitData->WEIGHT[i] = 0.;
    else if(pMFitData->Freqs[i] > Fmax) lpFitData->WEIGHT[i] = 0.;
//    else if (1./b.magtu[i] < 1.0) lpFitData->WEIGHT[i] = 1.;
    else lpFitData->WEIGHT[i] = 1./b.magtu[i];
```

```
/* Inverse IS equal weighting across frequency. */

    lpFitData->WEIGHT[i] *= pMFitData->Weighted[i];
    lpFitData->WEIGHT[i] = pow(lpFitData->WEIGHT[i], WARP_FACTOR);

#if defined(DBOUT)
DebugOut("freq %g   ",pMFitData->Freqs[i]);
DebugOut("mag %g  ",b.magtu[i]);
DebugOut("Weighted %g  ",pMFitData->Weighted[i]);
DebugOut("WEIGHT %g\r\n",lpFitData->WEIGHT[i]);
#endif
  }
/****************************** END init **********************************/



  if(ESCPressed())
  {
    ret_val = CANCELED;
    goto LBL_END;
  }

/*****************************************************************************/
/***************** BEGIN SIMULTANEOUS EQUATIONS SOLUTION *******************/
/***** The weighting matrix here is pre-squared as H in Strang  ************/
/**** TEMP = (Hc^T X WEIGHT X Hc)^(-1) X Hc_T X WEIGHT *******(pg.93)****/
/************** first calculate the Hc(z) matrix. (pg.92) *********/
/*** applied weighting function; see Strang book pg.148 ***********/

  /* Get the H_real and H_imag data out of CalcTable */
  CalcTable = (CALC_TABLE FAR *)LockResource(hMemCalcTable);

  for(i=0; i<NUM_FREQS; i++)
    for(j=0; j<NUM_FILTERS; j++)
    {
//      Hiz(pMFitData->Freqs[i], j, pia2);
      lpFitData->Hc[i][j] = (*CalcTable)[j][i].H_real;
      lpFitData->Hc[i+NUM_FREQS][j] = (*CalcTable)[j][i].H_imag;
      lpFitData->W[i][j] = (*CalcTable)[j][i].H_real *
                              lpFitData->WEIGHT[i];   /* have W = WEIGHT^T X Hc */
      lpFitData->W[i+NUM_FREQS][j] = (*CalcTable)[j][i].H_imag *
                                       lpFitData->WEIGHT[i];
    }

  UnlockResource(hMemCalcTable);
```

```
/********************** Multiply W^T X Hc ***********************************/
/********* dimensions of PROD are [NUM_FILTERS X NUM_FILTERS] ***************/
  for(i=0; i<NUM_FILTERS; i++)
    for(j=0; j<NUM_FILTERS; j++)
    {
      lpFitData->PROD[i+1][j+1] = 0.0;
      for(k=0; k<(2*NUM_FREQS); k++)
     lpFitData->PROD[i+1][j+1] += lpFitData->W[k][i]*lpFitData->Hc[k][j];
    /* PROD[1..n][1..n] */
    }
/************************ Find the inverse of the matrix PROD ***************/
  ludcmp(lpFitData->PROD,
         NUM_FILTERS,
     lpFitData->indx, Side);    /* PROD is destroyed */
  for(j=1; j<=NUM_FILTERS; j++)
  {
    for(i=1; i<=NUM_FILTERS; i++) lpFitData->col[i] = 0.0;
    lpFitData->col[j] = 1.0;
    lubksb(lpFitData->PROD, NUM_FILTERS, lpFitData->indx, lpFitData->col);
    for(i=1; i<=NUM_FILTERS; i++) lpFitData->INV[i][j] = lpFitData->col[i];
    /* INV[1..n][1..n] */
  }
/******************** Multiply INV by Hc^T **********************************/
/********* dimensions of TEMP are [NUM_FILTERS X 2*NUM_FREQS] ***************/
  for(i=0; i<NUM_FILTERS; i++)
    for(j=0; j<(2*NUM_FREQS); j++)
    {
      lpFitData->TEMP[i][j] = 0.0;
      for(k=0; k<NUM_FILTERS; k++)
    lpFitData->TEMP[i][j] += lpFitData->INV[i+1][k+1]*lpFitData->Hc[j][k];
      lpFitData->TEMP[i][j] *= lpFitData->WEIGHT[j%NUM_FREQS];  /* apply weighting
                                                                   function */
    }
/************************ END NON-ITERATIVE PART OF SIMUL EQ ****************/
/****************************************************************************/

/****************************************************************************/
/********************** BEGIN ITERATIVE estimate of phase *******************/
  for(numits=0; numits<NUM_IT_MAX; numits++)
  {    /* begin big iteration loop */
    for(i=0; i<NUM_FREQS; i++)
    {
      imag = real = 0.;
      for(j=0; j<NUM_FILTERS; j++)
      { // =A x_w
    real += pointer[j]*lpFitData->Hc[i][j];
```

```
    imag += pointer[j]*lpFitData->Hc[i+NUM_FREQS][j];
      }
      b.phase[i] = atan2(imag, real);
    }
    pointer  = pointer1;
    pointer1 = pointer2;   /* exchange pointers */
    pointer2 = pointer;

/***** Now have list of frequencies, desired magnitudes and estimated phases. */
/********************** K = TEMP X b ***********************************/
    if(numits < EMPIRICAL)  /* vascillation bug fix. */
       total_load = calculate_load(pointer, 0, pointer[0]);

    for(i=0; i<NUM_FILTERS; i++)
    {
      pointer[i] = 0.0;
      for(k=0; k<NUM_FREQS; k++) {
    tempest = Compensate(total_load, k, Side);
    pointer[i] += (b.magtu[k]*(lpFitData->TEMP[i][k]*cos(b.phase[k])
       + lpFitData->TEMP[i][k+NUM_FREQS]*sin(b.phase[k])))
                                                                / tempest;

                                                            }
    }
/************** determine if more iterations are needed ***********************/
    criterion_met = TRUE;
    for(i=0; i<NUM_FILTERS; i++)
      if(pointer[i] != 0.0)
    if(fabs((pointer1[i] - pointer2[i])/pointer[i]) > CRITERION)
    {
      criterion_met = FALSE;
      break;
    }
    if(criterion_met) break;

    if(ESCPressed())
    {
      ret_val = CANCELED;
      goto LBL_END;
    }
  }    /* end iteration loop */
/********************** END ITERATIVE estimate of phase *******************/
/*****************************************************************************/
```

```
/*****************************************************************************/
/************** BEGIN BIPOLAR COEFFICIENT SATURATION PROJECTION ***************/
/* ARE ANY COEFFICIENTS EXCESSIVE */
    for(i=0,k=0; i<NUM_FILTERS; i++)
      if(fabs(pointer[i]) > 1.0) {
           k++;
           break;
                                }
/* if no excessive coefficients, this entire procedure unnecessary. */
/****************** pg.149 ***************************************/
    if(k && !bUNIPOLAR)
    {
      for(i=0; i<NUM_FILTERS; i++)

        {
         if(pointer[i] == 0.0) pointer[i] = 1e-17;
         pointer_copy[i] = pointer[i];
         }
#if defined(DBOUT)
 DebugOut("pre-saturation-map bipolar coefs\r\n");
 for(i=0; i<NUM_FILTERS; i++)
 {
   DebugOut("K[%d]", i);
   DebugOut(" = %lf\r\n",pointer[i]);
 }
#endif
/***************************** BEGIN SATURATION MAP ITERATION ******************/
      while(TRUE)
      {
/***************** find translation point  q  in A-space ********************/

        for(i=0; i<NUM_FILTERS; i++)
 //assumption: active basis vectors are nonZero
              pointer_trans[i] = reduction(pointer[i]);
     for(i=0; i<(2*NUM_FREQS); i++)
     {
       projection[i] = 0.0;
       for(j=0; j<NUM_FILTERS; j++)
               projection[i] += lpFitData->Hc[i][j]*pointer_trans[j];  // q = A x_w
     }

/************** find minimum angle associated with excess-coef basis vectors ***/
     min_angle = 4.0*atan(1.0) + 1.0;
     howmany_sats = 0;
     for(i=0; i<NUM_FILTERS; i++)
```

```
      if(fabs(pointer[i]) > 1.0)
      {
         howmany_sats++;
         /********** calculate angle of p with current basis vector ****/
         basis_angle = 0.0;
         tempd = 0.0;
         tempe = 0.0;
         for(j=0; j<(2*NUM_FREQS); j++)
         {
         // WEIGHT[] = Strang's H = Strang W*W
         tempd += sqr(lpFitData->Hc[j][i])*lpFitData->WEIGHT[j%NUM_FREQS];
         tempe += sqr(projection[j])*lpFitData->WEIGHT[j%NUM_FREQS];
         basis_angle += lpFitData->Hc[j][i]*
                          projection[j]*
                          lpFitData->WEIGHT[j%NUM_FREQS];
         }
         basis_angle = acos((signum(pointer[i])*basis_angle)/sqrt(tempd*tempe));
         if(basis_angle < min_angle)
         {
                                                          min_angle = basis_angle;
                                                          eliminated_vector = i;
         }
      }
    if(!howmany_sats) break; /* out of while(1) */

#if defined(DBOUT)
DebugOut("min angle = %lf\r\n", min_angle);
DebugOut("reduced vector = %d\r\n", eliminated_vector);
#endif
/*********** reduce basis_m coef magnitude ******************************/
         pointer_copy[eliminated_vector] = pointer[eliminated_vector];
    pointer[eliminated_vector] = reduction(pointer[eliminated_vector]);

/**************** find reduced-coef projection point  p'  in A-space
********************/
    for(i=0; i<(2*NUM_FREQS); i++)
    {
      projection[i] = 0.0;
      for(j=0; j<NUM_FILTERS; j++)
              projection[i] += lpFitData->Hc[i][j]*pointer[j];  // p' = A x_w
    }

/*********** reduce dimension of space ***********************************/
    pointer[eliminated_vector] = 0.0;
```

```
/************ determine dimension of reduced space ********************/
     dimension = 0;
     for(i=0; i<NUM_FILTERS; i++)
       if(pointer[i] != 0.0)
         basis[dimension++] = i;   /* list of active basis vectors */

/****************** form reduced B matrix from A matrix *********************/
     for(i=0; i<NUM_FREQS; i++)
       for(j=0; j<dimension; j++)
       {
        lpFitData->HcPjx[i][j] = lpFitData->Hc[i][basis[j]];
        lpFitData->HcPjx[i+NUM_FREQS][j] = lpFitData->Hc[i+NUM_FREQS][basis[j]];
        lpFitData->W[i][j] = lpFitData->Hc[i][basis[j]]*lpFitData->WEIGHT[i];
 lpFitData->W[i+NUM_FREQS][j] = lpFitData->Hc[i+NUM_FREQS][basis[j]]*lpFitData->WEIGHT[i];
       }

     for(i=0; i<dimension; i++)
       for(j=0; j<dimension; j++)
       {
         lpFitData->PROD[i+1][j+1] = 0.0;
         for(k=0; k<(2*NUM_FREQS); k++)
    /* PROD[1..n][1..n] */
            lpFitData->PROD[i+1][j+1] += lpFitData->W[k][i]*lpFitData->HcPjx[k][j];

       }

     ludcmp(lpFitData->PROD, dimension, lpFitData->indx,Side);
     for(j=1; j<=dimension; j++)
     {
     for(i=1; i<=dimension; i++) lpFitData->col[i] = 0.0;
     lpFitData->col[j] = 1.0;
                                                              /* INV[1..n][1..n] */
     lubksb(lpFitData->PROD, dimension, lpFitData->indx, lpFitData->col);
     for(i=1; i<=dimension; i++) lpFitData->INV[i][j] = lpFitData->col[i];
     }

     for(i=0; i<dimension; i++)
       for(j=0; j<(2*NUM_FREQS); j++)
       {
         lpFitData->TEMP[i][j] = 0.0;
         for(k=0; k<dimension; k++)
           lpFitData->TEMP[i][j] += lpFitData->INV[i+1][k+1]*lpFitData->HcPjx[j][k];
         lpFitData->TEMP[i][j] *= lpFitData->WEIGHT[j%NUM_FREQS];
       }
```

```
/***************** calculate reduced space coefficients *****************/
     for(i=0; i<dimension; i++)
     {
       pointer[basis[i]] = 0.0;
       for(k=0; k<(2*NUM_FREQS); k++)
         pointer[basis[i]] += lpFitData->TEMP[i][k]*projection[k];
     }
      if(ESCPressed())
        {
        ret_val = CANCELED;
        goto LBL_END;
        }

      }  /** end while(TRUE) **/
/****************************** END SATURATION MAP ITERATION ******************/
    for(i=0; i<NUM_FILTERS; i++)
     if(pointer[i] == 0.0)
         pointer[i] = signum(pointer_copy[i]);
    }   /* end if-excessive coefficients */
/************** END BIPOLAR COEFFICIENT SATURATION PROJECTION ***************/
/****************************************************************************/

/****************************************************************************/
/************** BEGIN 1st QUADRANT SATURATION PROJECTION *********************/
/* ARE ANY COEFFICIENTS NEGATIVE or POSITIVE EXCESSIVE */
    for(i=0,k=0; i<NUM_FILTERS; i++)
      if((pointer[i] < 0.0) || (pointer[i] > 1.0))
        {
         k++;
         break;
        }
/* if no negative or excessive coefs, this entire procedure unnecessary. */
/****************** pg.147 ***************************************/
    if(bUNIPOLAR && k)
    {
      for(i=0; i<NUM_FILTERS; i++)

        {
         if(pointer[i] == 0.0) pointer[i] = 1e-17;
         pointer_copy[i] = pointer[i];
         }
#if defined(DBOUT)
 DebugOut("bipolar coefficients\r\n");
 for(i=0; i<NUM_FILTERS; i++)
 {
   DebugOut("K[%d]", i);
   DebugOut(" = %lf\r\n",pointer[i]);
```

```
 }
#endif
/***************************** BEGIN UNIPOLAR CASE ITERATION *****************/
      numits = 0;
      while(bUNIPOLAR)
      {
/***************** find projection point  p  in A-space ********************/
     for(i=0; i<(2*NUM_FREQS); i++)
     {
       projection[i] = 0.0;
       for(j=0; j<NUM_FILTERS; j++)
        projection[i] += lpFitData->Hc[i][j]*pointer[j];  // =A x_w
     }
/***************** find translation point  q  in A-space ********************/

        for(i=0; i<NUM_FILTERS; i++)
 //assumption: active basis vectors are nonZero
              pointer_trans[i] = reduction(pointer[i]);
     for(i=0; i<(2*NUM_FREQS); i++)
     {
       projection_trans[i] = 0.0;
       for(j=0; j<NUM_FILTERS; j++)
               projection_trans[i] += lpFitData->Hc[i][j]*pointer_trans[j];  // q = A x_w
     }

/************** find minimum angle associated with basis vectors ***/
     min_angle = 4.0*atan(1.0) + 1.0;
     howmany_negs = 0;
     for(i=0; i<NUM_FILTERS; i++)
       if((pointer[i] < 0.0) || (pointer[i] > 1.0))
       {
          howmany_negs++;
          /********** calculate angle of p with current basis vector ****/
          basis_angle = 0.0;
          tempd = 0.0;
          tempe = 0.0;
          for(j=0; j<(2*NUM_FREQS); j++)
          {
              if(pointer[i] < 0.0)
                  {
          // WEIGHT[] = Strang's H = Strang W*W
          tempd += sqr(lpFitData->Hc[j][i])*lpFitData->WEIGHT[j%NUM_FREQS];
          tempe += sqr(projection[j])*lpFitData->WEIGHT[j%NUM_FREQS];
          basis_angle += lpFitData->Hc[j][i]*
                           projection[j]*
                           lpFitData->WEIGHT[j%NUM_FREQS];
```

```
                }
            else
                {
         tempd += sqr(lpFitData->Hc[j][i])*lpFitData->WEIGHT[j%NUM_FREQS];
         tempe += sqr(projection_trans[j])*lpFitData->WEIGHT[j%NUM_FREQS];
         basis_angle += lpFitData->Hc[j][i]*
                         projection_trans[j]*
                         lpFitData->WEIGHT[j%NUM_FREQS];
                }
        }
                             /* signum() not required */
            basis_angle = acos(basis_angle/sqrt(tempd*tempe));
            if(pointer[i] < 0.0)
           basis_angle = 4.0*atan(1.0) - basis_angle;
        if(basis_angle < min_angle)
        {
          min_angle = basis_angle;
          eliminated_vector = i;
        }
      }
    if(!howmany_negs) break; /* out of while(bUNIPOLAR) */

#if defined(DBOUT)
DebugOut("min angle = %lf\r\n", min_angle);
DebugOut("elim vector = %d\r\n", eliminated_vector);
#endif
/*********** reduce basis_m coef magnitude ******************************/
         pointer_copy[eliminated_vector] = pointer[eliminated_vector];

/**************** find reduced-coef projection point  p'  in A-space
*********************/
         if(pointer_copy[eliminated_vector] > 1.0)
            {
        pointer[eliminated_vector] = reduction(pointer[eliminated_vector]);
        for(i=0; i<(2*NUM_FREQS); i++)
           {
          projection[i] = 0.0;
          for(j=0; j<NUM_FILTERS; j++)
                  projection[i] += lpFitData->Hc[i][j]*pointer[j];  // p' = A x_w
           }
             }

/*********** reduce dimension of space ***********************************/
     pointer[eliminated_vector] = 0.0;

/************ determine dimension of reduced space ********************/
```

```
     dimension = 0;
     for(i=0; i<NUM_FILTERS; i++)
       if(pointer[i] != 0.0)
         basis[dimension++] = i;   /* list of active basis vectors */

/****************** form reduced B matrix from A matrix *********************/
     for(i=0; i<NUM_FREQS; i++)
       for(j=0; j<dimension; j++)
       {
        lpFitData->HcPjx[i][j] = lpFitData->Hc[i][basis[j]];
        lpFitData->HcPjx[i+NUM_FREQS][j] = lpFitData->Hc[i+NUM_FREQS][basis[j]];
        lpFitData->W[i][j] = lpFitData->Hc[i][basis[j]]*lpFitData->WEIGHT[i];
 lpFitData->W[i+NUM_FREQS][j] = lpFitData->Hc[i+NUM_FREQS][basis[j]]*lpFitData->WEIGHT[i];
       }

     for(i=0; i<dimension; i++)
       for(j=0; j<dimension; j++)
       {
         lpFitData->PROD[i+1][j+1] = 0.0;
         for(k=0; k<(2*NUM_FREQS); k++)
    /* PROD[1..n][1..n] */
           lpFitData->PROD[i+1][j+1] += lpFitData->W[k][i]*lpFitData->HcPjx[k][j];

       }

     ludcmp(lpFitData->PROD, dimension, lpFitData->indx,Side);
     for(j=1; j<=dimension; j++)
     {
     for(i=1; i<=dimension; i++) lpFitData->col[i] = 0.0;
     lpFitData->col[j] = 1.0;
                                                          /* INV[1..n][1..n] */
     lubksb(lpFitData->PROD, dimension, lpFitData->indx, lpFitData->col);
     for(i=1; i<=dimension; i++) lpFitData->INV[i][j] = lpFitData->col[i];
     }

     for(i=0; i<dimension; i++)
       for(j=0; j<(2*NUM_FREQS); j++)
       {
         lpFitData->TEMP[i][j] = 0.0;
         for(k=0; k<dimension; k++)
           lpFitData->TEMP[i][j] += lpFitData->INV[i+1][k+1]*lpFitData->HcPjx[j][k];
         lpFitData->TEMP[i][j] *= lpFitData->WEIGHT[j%NUM_FREQS];
       }


/***************** calculate reduced space coefficients ******************/
```

```
     for(i=0; i<dimension; i++)
     {
       pointer[basis[i]] = 0.0;
       for(k=0; k<(2*NUM_FREQS); k++)
         pointer[basis[i]] += lpFitData->TEMP[i][k]*projection[k];
     }
      if(ESCPressed())
        {
        ret_val = CANCELED;
        goto LBL_END;
        }

#if defined(DBOUT)
DebugOut("UNIPOLAR coefficients. %d\r\n",numits++);
for(i=0; i<NUM_FILTERS; i++)
{
  DebugOut("K[%d]",i);
  DebugOut(" = %lf\r\n",pointer[i]);
}
#endif
      }  /** end while(bUNIPOLAR) **/
/****************************** END UNIPOLAR CASE ITERATION *****************/
    for(i=0; i<NUM_FILTERS; i++)
     if(pointer[i] == 0.0)

           {
             if(pointer_copy[i] <= 0.0);
             else if(pointer_copy[i] > 1.0) pointer[i] = 1.0;
            }
    }                                                        /* end if bUNIPOLAR */
/************** END 1st QUADRANT SATURATION PROJECTION *******************************/
/*************************************************************************/

#if defined(DBOUT)
DebugOut("\r\nNumber of PhiLS iterations = %d\r\n", numits+1);
error = 0.0;
total_load = calculate_load(pointer, 0, pointer[0]);
for(i=0; i<NUM_FREQS; i++)
  {
    imag = real = 0.;
    for(j=0; j<NUM_FILTERS; j++)
    {
      real += pointer[j]*lpFitData->Hc[i][j];
      imag += pointer[j]*lpFitData->Hc[i+NUM_FREQS][j];
    }
    tempest = Compensate(total_load, i, Side);
    tempe = b.magtu[i] - tempest*sqrt(real*real+imag*imag);
```

```
    error += fabs(tempe)*lpFitData->WEIGHT[i];  //OK
  }
DebugOut("Post Clipping coefficients.\r\n");
for(i=0; i<NUM_FILTERS; i++)
{
  DebugOut("K[%d]",i);
  DebugOut(" = %lf\r\n",pointer[i]);
}
if (bUNIPOLAR)  DebugOut("UNIPOLAR + ");
else           DebugOut("BIPOLAR + ");
DebugOut("PhiLS + clip error = %.6lf\r\n", error);
#endif

/****************************** END PHASE LEAST SQUARES *********************/
/****************************************************************************/
/************************** FATT ITERATION ******************************/
/* if were not already on the largest FATT */
  if (FATTIndex < MAX_FATT_INDEX)
  {
  /* find max magnitude */
    max_magtu = 0.0;
    for (i=0; i<NUM_FILTERS; i++)
    {
      max_magtu = fmax(fabs(pointer[i]),max_magtu);
    }

  /* find the difference between the current FATT and the next */
    fatt_diff = GetFattGain(FATTIndex + 1,bRosco[Side]) -
                              GetFattGain(FATTIndex,bRosco[Side]);

#if defined(DBOUT)
DebugOut("max_magtu in dB %g  ",20. * log10(max_magtu));
DebugOut("fatt_diff %g\r\n",fatt_diff);
#endif

//  /* if there is room for the next FATT get out and redo */
//    if ((fabs(fatt_diff) * FATT_THRESHHOLD) < fabs(20. * log10(max_magtu)))
//    {
//      ret_val = INCREMENT_FATT;
//      goto LBL_END;
//    }
  }
/********************** END FATT ITERATION ******************************/


/****************************************************************************/
```

```
/************************ BEGIN PERTURBATION ANALYSIS ************************/
/************** quantize coefs and find total load ***************************/
  for(i=0; i<NUM_FILTERS; i++)
    pointer[i] = quantize(pointer[i]);
  total_load = calculate_load(pointer, 0, pointer[0]);

#if defined(DBOUT)
DebugOut("total loading at outset is %d\r\n", total_load);
#endif

/***************** calculate error due to quantization and loading **********/
  error = 0.0;
  for(i=0; i<NUM_FREQS; i++)
  {
    imag = real = 0.;
    for(j=0; j<NUM_FILTERS; j++)
    {
      real += pointer[j]*lpFitData->Hc[i][j];
      imag += pointer[j]*lpFitData->Hc[i+NUM_FREQS][j];
    }
    tempest = Compensate(total_load, i,Side);
    tempe = b.magtu[i] - tempest*sqrt(real*real+imag*imag);
    error += fabs(tempe)*lpFitData->WEIGHT[i];
  }
  current_err = error;
  old_err = 0.;

#if defined(DBOUT)
for(i=0; i<NUM_FILTERS; i++)
{
  DebugOut("K[%d]",i);
  DebugOut(" = %lf\r\n",pointer[i]);
}
DebugOut("Quantization error = %.6lf\r\n", error);
#endif

  numits = 0;
  while(fabs(current_err - old_err) > CRITERION)
  {
    if(ESCPressed())
    {
      ret_val = CANCELED;
      goto LBL_END;
    }

    old_err = current_err;
```

```
#if defined(DBOUT)
DebugOut("%d\r\n",numits);
#endif

//    for(k=NUM_FILTERS-1; k>=0; k--) //top down
    for(k=0;k<NUM_FILTERS;k++) //bottom up
    {
      error = 0.0;
        /* UPWARD */
      for(i=0; i<NUM_FREQS; i++)
      {
    imag = real = 0.;
    for(j=0; j<NUM_FILTERS; j++)
    {
      tempd = pointer[j];
      if(j == k)
      {
        tempd += perturbation(tempd, UP);
        tempd = saturate(tempd);
        /*total_load = calculate_load(pointer, k, tempd);*/
        tempp = tempd;
      }
      real += tempd*lpFitData->Hc[i][j];
      imag += tempd*lpFitData->Hc[i+NUM_FREQS][j];
    }
    tempest = Compensate(total_load, i,Side);
    tempe = b.magtu[i] - tempest*sqrt(real*real+imag*imag);
    error += fabs(tempe)*lpFitData->WEIGHT[i];
      }
      if(error < current_err)
      {
#if defined(DBOUT)
DebugOut("%dth upward perturbation error = ", k);
DebugOut("%.6lf\r\n",error);
#endif

    current_err = error;
    pointer[k] = tempp;
      }
      else
      {
    error = 0.0;
          /* DOWNWARD */
    for(i=0; i<NUM_FREQS; i++)
    {
```

```
      imag = real = 0.;
      for(j=0; j<NUM_FILTERS; j++)
      {
        tempd = pointer[j];
        if(j == k)
        {
          tempd -= perturbation(tempd, DOWN);
          tempd = saturate(tempd);
          /*total_load = calculate_load(pointer, k, tempd);*/
          tempp = tempd;
        }
        real += tempd*lpFitData->Hc[i][j];
        imag += tempd*lpFitData->Hc[i+NUM_FREQS][j];
      }
      tempest = Compensate(total_load, i,Side);
      tempe = b.magtu[i] - tempest*sqrt(real*real+imag*imag);
      error += fabs(tempe)*lpFitData->WEIGHT[i];
    }
    if(error < current_err)
    {
#if defined(DBOUT)
DebugOut("%dth downward perturbation error = ", k);
DebugOut("%.6lf\r\n",error);
#endif
      current_err = error;
      pointer[k] = tempp;
    }
      }
       /* end down else */
    }         /* end once around k coefs */
    if(++numits >= NUM_IT_MAX) break;  /* timeout */
  }  /* end while */
         /******* end diminishing returns *******/
#if defined(DBOUT)
DebugOut("\r\nNumber of PERTURBATION iterations = %d\r\n", numits);
#endif

/************************* FATT ITERATION ********************************/
/* if were not already on the largest FATT */
if (!bJONSWITCH)
  if (FATTIndex < MAX_FATT_INDEX)
  {
  /* find max magnitude */
    max_magtu = 0.0;
    for (i=0; i<NUM_FILTERS; i++)
    {
```

```
      max_magtu = fmax(fabs(pointer[i]),max_magtu);
    }

  /* find the difference between the current FATT and the next */
    fatt_diff = GetFattGain(FATTIndex + 1,bRosco[Side]) -
GetFattGain(FATTIndex,bRosco[Side]);

#if defined(DBOUT)
DebugOut("max_magtu in dB %g  ",20. * log10(max_magtu));
DebugOut("fatt_diff %g\r\n",fatt_diff);
#endif

  /* if there is room for the next FATT get out and redo */
    if ((fabs(fatt_diff) * FATT_THRESHHOLD) < fabs(20. * log10(max_magtu)))
    {
      ret_val = INCREMENT_FATT;
      goto LBL_END;
    }
  }
/********************** END FATT ITERATION *******************************/

for(i=0; i<NUM_FILTERS; i++)
  {
    fitval[i] = pointer[i];

#if defined(DBOUT)
DebugOut("K[%d] =",i);
DebugOut("%lf\r\n", pointer[i]);
#endif
  }

#if defined(DBOUT)
error = 0.0;
total_load = calculate_load(pointer, 0, pointer[0]);
DebugOut("final loading is %d\r\n", total_load);

for(i=0; i<NUM_FREQS; i++)
{
  imag = real = 0.;
  for(j=0; j<NUM_FILTERS; j++)
  {
    real += pointer[j]*lpFitData->Hc[i][j];
    imag += pointer[j]*lpFitData->Hc[i+NUM_FREQS][j];
  }
  tempest = Compensate(total_load, i,Side);
  tempe = b.magtu[i] - tempest*sqrt(real*real+imag*imag);
  error += fabs(tempe)*lpFitData->WEIGHT[i];
```

```
  DebugOut("freq %lf  ",pMFitData->Freqs[i]);
  DebugOut("tempest %lf  ",tempest);
  DebugOut("ETarg %lf  ",20.*log10(b.magtu[i]/tempest) + 70.);
  DebugOut("EFit %lf\r\n",10.*log10(real*real+imag*imag) + 70.);
}
DebugOut("after-perturbation error = %.6lf\r\n", error);
#endif
/******************************** END PERTURBATION ANALYSIS ******************/
/*****************************************************************************/

LBL_END:
  LocalUnlock(hMFitData);

  GlobalUnlock(hFitData);
  GlobalFree(hFitData);

/* ANSWER is fitval[NUM_FILTERS] */
  return(ret_val);
}


/******************************** SUBROUTINES ********************************/
/*****************************************************************************/
/*double quantize(double argum)
{
  short quan;
  int load;
  double sign, gain;

  gain = 1./DECIMOUS;
  sign = -1.;
  load = 0;
  quan = Magnitude2FilterSetting(argum);

  if(quan & SIGNUM)
  {
    sign = 1.0;
    quan ^= SIGNUM;
  }
  if(quan & G5)
  {
    gain = 1.0;
    load = 1;
    quan ^= G5;
  }
```

```
  return((sign*gain*quan)/BINARIOUS);
}
*/
double quantize(double argum)
{
  short quan;
  double gain, signum();

  gain = 1./DECIMOUS;
  quan = Magnitude2FilterSetting(argum);

  if(abs(quan) & G5)
  {
    gain = 1.0;
    quan = (short)signum((double)quan) * (abs(quan) ^ G5);
  }

  return((gain*quan)/BINARIOUS);
}


/* assumption is that quantity is previously quantized */
double perturbation(double quantity, int direction)
{
  if( !((!(direction == UP) && (quantity >= 0.0))
    || ((direction == UP) && !(quantity >= 0.0))) )
  {
    if(fabs(quantity) < (BINARIOUS - 0.5)/(DECIMOUS*BINARIOUS))
      return(1./(DECIMOUS*BINARIOUS));
    if(fabs(quantity) > (BINARIOUS + 0.5)/(DECIMOUS*BINARIOUS))
      return(1./BINARIOUS);
    return(FIRST_OVERLAP/(double)BINARIOUS - 1./DECIMOUS);
  }
  else
  {
    if(fabs(quantity) < (BINARIOUS + 0.5)/(DECIMOUS*BINARIOUS))
      return(1./(DECIMOUS*BINARIOUS));
    if(fabs(quantity) < (FIRST_OVERLAP + 0.5)/BINARIOUS)
      return(FIRST_OVERLAP/(double)BINARIOUS - 1./DECIMOUS);
    return(1./BINARIOUS);
  }
}


double saturate(double ragu)
```

```
{
  if(ragu > 1.0) return(1.0);
  if(bUNIPOLAR && (ragu < 0.0)) return(0.0);
  if(ragu < -1.0) return(-1.0);
  return(ragu);
}


double reduction(double redu)
/* used in saturation mapping */
{
  if(redu == 0.0) return(redu);
  if(redu > 0.0) return(redu - 1.0);
  return(redu + 1.0);
}

double signum(double argu)
{
  if(argu == 0.0) return(0.0);
  if(argu > 0.0) return(1.0);
  return(-1.0);
}
```