Data Analysis Working Group

Task 2. Bad phase flag problem.

Author: Pasha Ponomarenko

Date: 20 August 2013

1.1. Description:

The initial estimate of the error is made in **omega_guess.c** based on at least three sets of consecutive good lags and then it is used in **fit_acf.c** to calculate the spread in the slope estimates through fitting OMEGA_LOC ± OMEGA_ERR_LOC. If the difference between the respective minimum and maximum slope estimates is larger than the fitting error, then the output error value is equivalent to the OMEGA_ERR_LOC.

1.2. Implications

The problem here is that if there are less than three pairs of consecutive good lags, this parameter is assigned a very large number (9999), which then used to calculate the output velocity errors, even is the fit itself was good. In this case the information about fitting errors is lost. I don't think that the mere fact that there are less than tree pairs of the consecutive good lags means that the fitted data are useless.

1.3. Proposed action

I propose to use the fitting errors all the time, and if there is not enough pairs of consecutive lags than just flag that fact by using negative error values. In order to do that, **fit_acf.c** has to be modified at line 377 from

```
377      ptr->v_err = fabs(omega_high - omega_low);
```

to

```
377      ptr->v_err = - ptr->v_err;
```

In this case we still preserve the information on the fitting errors but also use the minus sign to notify the users that there was no error calculated from the pairs of consecutive lags.

Attention: in the github FITACF2.5 depository this is line 406!!!

Ultimately, we can just abandon the second (purely empirical) way of estimating velocity errors using OMEGA_LOC ± OMEGA_ERR_LOC which would simplify the code and speed up its execution (a single run of DO_PHASE_FIT.C instead of running it three times)

Appendix 1

```
/* omega_guess.c
   =============
   Author: R.J.Barnes & K.Baker
*/

/*
 Copyright 2004 The Johns Hopkins University/Applied Physics Laboratory.
 All rights reserved.

 This material may be used, modified, or reproduced by or for the U.S.
 Government pursuant to the license rights granted under the clauses at
DFARS
 252.227-7013/7014.

 For any other permissions, please contact the Space Department
 Program Office at JHU/APL.

 This Distribution and Disclaimer Statement must be included in all
copies of
 "Radar Software Toolkit - SuperDARN Toolkit" (hereinafter "the
Program").

 The Program was developed at The Johns Hopkins University/Applied
Physics
 Laboratory (JHU/APL) which is the author thereof under the "work made
for
 hire" provisions of the copyright law.

 JHU/APL assumes no obligation to provide support of any kind with regard
to
 the Program.  This includes no obligation to provide assistance in using
the
 Program or to provide updated versions of the Program.

 THE PROGRAM AND ITS DOCUMENTATION ARE PROVIDED AS IS AND WITHOUT ANY
EXPRESS
 OR IMPLIED WARRANTIES WHATSOEVER.  ALL WARRANTIES INCLUDING, BUT NOT
LIMITED
 TO, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE ARE
 HEREBY DISCLAIMED.  YOU ASSUME THE ENTIRE RISK AND LIABILITY OF USING
THE
 PROGRAM TO INCLUDE USE IN COMPLIANCE WITH ANY THIRD PARTY RIGHTS.  YOU
ARE
 ADVISED TO TEST THE PROGRAM THOROUGHLY BEFORE RELYING ON IT.  IN NO
EVENT
 SHALL JHU/APL BE LIABLE FOR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT
 LIMITATION, ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR
 CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OR INABILITY TO USE THE
 PROGRAM."
```

```
 43
 44 */
 45
 46 /*
 47  $Log: omega_guess.c,v $
 48  Revision 1.7  2004/05/04 22:54:21  barnes
 49  Changed math header name.
 50
 51  Revision 1.6  2004/04/26 22:14:07  barnes
 52  Audit - Enforced warning "all".
 53
 54  Revision 1.5  2004/01/08 22:20:39  barnes
 55  Kile Baker's Modifications for improved velocity error and ground
scatter
 56  determination.
 57
 58  Revision 1.4  2003/09/13 22:39:29  barnes
 59  Modifications to use the new data structures.
 60
 61  Revision 1.3  2001/06/27 20:48:31  barnes
 62  Added license tag
 63
 64  Revision 1.2  2001/01/29 18:11:53  barnes
 65  Added Author Name
 66
 67  Revision 1.1  1998/06/05 19:56:46  barnes
 68  Initial revision
 69
 70  */
 71
 72
 73 /* Revision 1.4 corrected the method of weighting the
 74    estimate for omega to be consistent with the way
 75    it is done in fit_acf and do_phase_fit.
 76
 77    The error on any given point is estimated to be
 78    delta_phase[i] = <delta_phase>*<P>/P[i]
 79 */
 80
 81 #include <math.h>
 82 #include "rmath.h"
 83
 84 double omega_guess(struct complex *acf,double *tau,
 85                    int *badlag,double *phi_res,
 86                    double *omega_err,int mpinc,int mplgs) {
 87
 88   int i,j,nave=0;
 89   double delta_tau, delta_phi, omega=0.0,
 90          omega2=0.0, average=0.0, sigma, tau_lim=1.0;
 91   double sum_W=0.0, W;
 92   register double temp;
 93   double two_sigma;
 94
 95   two_sigma = sigma = 2*PI;
 96   *omega_err = 9999.;
 97
 98   while (tau_lim < 3 && nave < 3) {
```

```
 99      for (j=1; j<=tau_lim; ++j)
100        for (i=0; i< mplgs - j; ++i) {
101          if (badlag[i+j] || badlag[i]) continue;
102          delta_tau = tau[i+j] - tau[i];
103          if (delta_tau != tau_lim) continue;
104          delta_phi = phi_res[i+j] - phi_res[i];
105          W = (cabs(acf[i]) + cabs(acf[i+j]))/2.0;
106          W = W*W;
107
108          if (delta_phi > PI) delta_phi = delta_phi - 2*PI;
109          if (delta_phi < -PI) delta_phi = delta_phi + 2*PI;
110
111          if ((average != 0.0) && (fabs(delta_phi - average) > two_sigma))
112            continue;
113          temp = delta_phi/tau_lim;
114          omega = omega + temp*W;
115          omega2 = omega2 + W*(temp*temp);
116          sum_W = sum_W + W;
117          nave++;
118        }
119
120      if (nave >= 3 && (sigma == 2*PI)) {
121        average = omega/sum_W;
122        sigma = ((omega2/sum_W) - average*average)/(nave-1);
123        sigma = (sigma > 0.0) ? sqrt(sigma) : 0.0;
124        two_sigma = 2.0*sigma;
125        omega = 0.0;
126        omega2 = 0.0;
127        sum_W = 0;
128        nave = 0;
129        tau_lim = 1;
130      } else if (nave >=3) {
131        omega = omega/sum_W;
132        omega = omega/(mpinc*1.0e-6);
133        sigma = ((omega2/sum_W) - average*average)/(nave-1);
134        sigma = (sigma > 0.0) ? sqrt(sigma) : 0.0;
135        *omega_err = sigma/(mpinc*1.0e-6);
136        return omega;
137      }
138      else ++tau_lim;
139    }
140    return 0.0;
141  }
```

Appendix 2

```
/* fit_acf.c
   =========
   Author: R.J.Barnes & K.Baker & P.Ponomarenko
*/

/*
 Copyright 2004 The Johns Hopkins University/Applied Physics Laboratory.
 All rights reserved.

 This material may be used, modified, or reproduced by or for the U.S.
 Government pursuant to the license rights granted under the clauses at
DFARS
 252.227-7013/7014.

 For any other permissions, please contact the Space Department
 Program Office at JHU/APL.

 This Distribution and Disclaimer Statement must be included in all
copies of
 "Radar Software Toolkit - SuperDARN Toolkit" (hereinafter "the
Program").

 The Program was developed at The Johns Hopkins University/Applied
Physics
 Laboratory (JHU/APL) which is the author thereof under the "work made
for
 hire" provisions of the copyright law.

 JHU/APL assumes no obligation to provide support of any kind with regard
to
 the Program.  This includes no obligation to provide assistance in using
the
 Program or to provide updated versions of the Program.

 THE PROGRAM AND ITS DOCUMENTATION ARE PROVIDED AS IS AND WITHOUT ANY
EXPRESS
 OR IMPLIED WARRANTIES WHATSOEVER.  ALL WARRANTIES INCLUDING, BUT NOT
LIMITED
 TO, PERFORMANCE, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE ARE
 HEREBY DISCLAIMED.  YOU ASSUME THE ENTIRE RISK AND LIABILITY OF USING
THE
 PROGRAM TO INCLUDE USE IN COMPLIANCE WITH ANY THIRD PARTY RIGHTS.  YOU
ARE
 ADVISED TO TEST THE PROGRAM THOROUGHLY BEFORE RELYING ON IT.  IN NO
EVENT
 SHALL JHU/APL BE LIABLE FOR ANY DAMAGES WHATSOEVER, INCLUDING, WITHOUT
 LIMITATION, ANY LOST PROFITS, LOST SAVINGS OR OTHER INCIDENTAL OR
 CONSEQUENTIAL DAMAGES, ARISING OUT OF THE USE OR INABILITY TO USE THE
 PROGRAM."
```

```
 43
 44 */
 45
 46 /*
 47  $Log: fit_acf.c,v $
 48  Revision 1.11  2007/02/02 21:38:42  code
 49   Modifications made to leave fluctuation level(s) intact and to subtract
lag 0 n
 50  oise from R(0).
 51
 52
 53  Revision 1.10  2005/06/30 23:28:27  barnes
 54  Corrected problems discovered in the IDL version of the code.
 55
 56  Revision 1.9  2005/06/30 16:50:46  barnes
 57  Minor bug fixes.
 58
 59  Revision 1.8  2004/05/04 22:54:21  barnes
 60  Changed math header name.
 61
 62  Revision 1.7  2004/04/26 22:14:07  barnes
 63  Audit - Enforced warning "all".
 64
 65  Revision 1.6  2004/01/08 22:20:39  barnes
 66  Kile Baker's Modifications for improved velocity error and ground
scatter
 67  determination.
 68
 69  Revision 1.5  2003/09/13 22:39:29  barnes
 70  Modifications to use the new data structures.
 71
 72  Revision 1.4  2002/03/05 16:59:40  barnes
 73  Fixed bug that meant nonsense was returned if a power fit failed.
 74
 75  Revision 1.3  2001/06/27 20:48:31  barnes
 76  Added license tag
 77
 78  Revision 1.2  2001/01/29 18:11:53  barnes
 79  Added Author Name
 80
 81  Revision 1.1  1998/06/05 19:56:46  barnes
 82  Initial revision
 83
 84  */
 85
 86 #include <math.h>
 87 #include <stdio.h>
 88
 89 #include "rmath.h"
 90 #include "limit.h"
 91 #include "badsmp.h"
 92 #include "fitblk.h"
 93
 94
 95 #include "acf_preproc.h"
 96 #include "calc_phi_res.h"
 97 #include "omega_guess.h"
```

```c
#include "badlags.h"
#include "more_badlags.h"
#include "do_phase_fit.h"


#define determ(aa,bb,cc,dd) (aa*dd - bb*cc)

int fit_acf (struct complex *acf,int range,
             int *badlag,struct FitACFBadSample *badsmp,int lag_lim,
                    struct FitPrm *prm,
                double noise_lev_in,char xflag,double xomega,
                struct FitRange *ptr) {

  double sum_np,sum_w,sum_wk,sum_wk2,sum_wk2_arr[LAG_SIZE],sum_wk4;
  double sum_p,sum_pk,sum_pk2,sum_phi,sum_kphi;
  double t0,t2,t4,phi_res[LAG_SIZE];
  int j;
  long k;
  int npp;
  double tau[LAG_SIZE], tau2[LAG_SIZE];
  double phi_k[LAG_SIZE];
  double w[LAG_SIZE], pwr[LAG_SIZE];
  double wt[LAG_SIZE], wt2[LAG_SIZE], wp[LAG_SIZE];
  double e2;

  double c_log,c_log_err,d;
  double omega_loc, omega_err_loc, phi_loc, noise_lev;
  double omega_base, omega_high, omega_low;
  double phase_sdev;
  double temp;
  double phi_err, omega_err;
  double wbar;

/*  The following variables have been added for version 2.0 of cfitacf */

  double /*P0, */ P0n;   /* this is the power level where the acf levels
off */
  int last_good;
  int bad_pwr[LAG_SIZE];

  int status;

/* the following array has been added to support preprocessing of the
   acf */

  int acf_stat=ACF_UNMODIFIED;

/* ----------------End of declarations ---------------------------------
*/

  if (cabs(acf[0]) < noise_lev_in) {
    for (j=0; j<prm->mplgs; j++) badlag[j]=3;
    ptr->p_l = 0.0;
    ptr->p_s = 0.0;
    ptr->p_l_err = 0.0;
    ptr->p_s_err = 0.0;
    ptr->v = 0.0;
```

```
153     ptr->v_err = 0.0;
154     ptr->w_l = 0.0;
155     ptr->w_l_err = 0.0;
156     ptr->w_s = 0.0;
157     ptr->w_s_err = 0.0;
158     ptr->phi0 = 0.0;
159     ptr->phi0_err = 0.0;
160     ptr->sdev_l = 0.0;
161     ptr->sdev_s = 0.0;
162     return 2;
163   }
164
165   /* initialize the table of abs(acf[k]) and log(abs(acf[k])) */
166
167   FitACFCkRng(range, badlag, badsmp, prm);
168
169   /* Save the original ACF in a new variable so we can try some
170      preprocessing on it.
171   */
172
173 /*  for (k=0; k < prm->mplgs; k++) {
174     orig_acf[k].x = acf[k].x;
175     orig_acf[k].y = acf[k].y;
176   }
177 */
178
179   /* This next statement provides a hook for a routine to pre-process
180     the ACF and return a modified ACF that will actually be fitted.
181
182     The function acf_preproc should return a 0 if no change was made
183     and a non-zero value otherwise - the actual non-zero value may be
184     used to provide information about the change that was made.
Specifically
185     the following values should be defined:
186        ACF_UNMODIFIED        the ACF was unchanged
187        ACF_GROUND_SCAT    the preprocessing indicates the ACF is ground
188                                                scatter.
189        ACF_ION_SCAT          the preprocessing indicates the ACF is
Ionospheric
190        ACF_MIXED_SCAT        the preprocessing indicates a mixture of
191                                                ionospheric and ground
scatter.
192
193     The original acf will be in the array "orig_acf" and the modified
194     acf will be in "acf".
195
196     To write an acf_preprocessor you must use the calling convention
below.
197     The revised acf will be returned in the array acf.  You must also
provide
198     the value of the noise_level (noise_lev_in), the range number, the
badlag
199     table, and the number of lags
200   */
201
202   noise_lev = noise_lev_in;
203 /*
```

```
204   if (noise_lev != 0.0) acf_stat = acf_preproc (acf, orig_acf,
&noise_lev,
205                     range, badlag, prm->mplgs);
206 */
207   for (k=0; k<prm->mplgs; k++) {
208     tau[k] = prm->lag[1][k] - prm->lag[0][k];
209     tau2[k] = tau[k] * tau[k];
210     w[k] = cabs(acf[k]); /* w[k] = cabs(acf[k])- noise_lev; */
211      if (w[k] <= noise_lev) w[k] = 0.1; /* if (w[k] <= 0.0) w[k] = 0.1;
*/
212   }
213
214   /* we now have to compute the amount of power to subtract off the
215    power level at each range.  The amouont to be subtracted is
P(0)/sqrt(nave)
216    which is the approximate expectation value of the power level after
the
217    ACF has decorrelated.
218
219    [ To derive this, determine the expectation value of
220    P**2 = R(tau)*conj(R(tau))]
221   */
222
223   P0n = w[0]/sqrt((double) prm->nave);
224
225
226   if ((w[0] - P0n) < noise_lev) return 2;
227     /* give up if left over pwr is too low */
228
229
230   /*  identify any additional bad lags */
231
232   sum_np = more_badlags(w, badlag, noise_lev, prm->mplgs,prm->nave);
233
234   ptr->nump = (char) sum_np;
235
236   /*  We must have at least lag_lim good lags */
237
238   if (sum_np < lag_lim) return 4;
239
240   if (noise_lev <= 0.0) noise_lev = 0.1;
241     /* this is required to make logs ok */
242   w[0]=w[0]-prm->noise; /* This is to remove background delta-correlated
noise from lag 0 power (version 2.0)*/
243
244   /* OK, now we have determined the good lags for the phase fit.
245      Now subtract of P0n from the power profile */
246
247   /* calculate the power values for each lag.  'w' is the linear power.
248      wt is the power times the lag.  wt2 is power times lag**2.
249      pwr is the log of the power. wp is the linear power times the log of
250      the power.  The items that involve the linear power are all parts of
251      the least squares fits with the weighting done by the linear power.
*/
252
253   for (k=0; k<prm->mplgs; k++) {
254
```

```
255     if (w[k] <= P0n) w[k] = 0.1; /* if (w[k] <= 0.0) w[k] = 0.1; */
256     wt[k] = w[k]*w[k]*tau[k];
257     wt2[k] = wt[k]*tau[k];
258     pwr[k] = log(w[k]);
259     wp[k] = w[k]*w[k]*pwr[k];
260   }
261   /* we now have to check to see how many additional bad lags have been
262      introduced by subtracting off P0n. */
263
264   for (k=0, npp=0; k < prm->mplgs; k++) {
265     if (w[k] < noise_lev+P0n && !badlag[k]) bad_pwr[k] = 1; /* if (w[k] <
noise_lev && !badlag[k]) bad_pwr[k] = 1; */
266     else bad_pwr[k] = 0;
267     if (! (badlag[k] || bad_pwr[k])) ++npp;
268   }
269
270  /* initialize the sums */
271
272   sum_np = 1;
273   sum_w = w[0]*w[0];
274   sum_wk = 0;
275   sum_wk2 = 0;
276   sum_wk2_arr[0] = 0;
277   sum_wk4 = 0;
278   sum_p = w[0]*w[0]*pwr[0];
279   sum_pk = 0;
280   sum_pk2 = 0;
281   phi_loc = atan2(acf[0].y, acf[0].x);
282   sum_kphi = 0;
283   t0 =  prm->mpinc * 1.0e-6;
284   t2 = t0 * t0;
285   t4 = t2 * t2;
286
287   /* calculate all the residual phases */
288   /* if calc_phi_res returns a bad status abort the fit */
289
290   if (calc_phi_res(acf, badlag, phi_res, prm->mplgs) != 0){
291     return 2;
292   }
293
294   if (!xflag) {
295     if (acf_stat == ACF_GROUND_SCAT) omega_loc = 0.0;
296     else omega_loc = omega_guess(acf, tau, badlag, phi_res,
297                                  &omega_err_loc,prm->mpinc,prm->mplgs);
298     phi_k[0] = 0;
299     sum_phi = 0;
300   } else {
301     phi_k[0] = phi_loc;
302     sum_phi = phi_loc * w[0]*w[0];
303     omega_loc = xomega;
304   }
305
306
307   /*  The preliminaries are now over.
308   Now start the fitting process */
309
310   /* first, calculate the sums needed for the phase fit */
```

```
311
312   for (k=1; k<prm->mplgs; k++) {
313     if (badlag[k]) {
314       sum_wk2_arr[k] = sum_wk2_arr[k-1];
315       continue;
316     }
317     sum_w = sum_w + w[k]*w[k];
318     sum_np = sum_np + 1;
319     sum_wk = sum_wk + w[k]*w[k]*tau[k];
320     sum_wk2 = sum_wk2 + wt2[k];
321     sum_wk2_arr[k] = sum_wk2;
322   }
323
324   /* Now do the phase fit using the best initial guess for omega */
325
326
327   status = do_phase_fit (omega_loc, xflag, prm->mplgs, acf, tau,
328                     w, sum_wk2_arr, phi_res, badlag, t0,
329                     sum_w, sum_wk, sum_wk2,
330                     &omega_base, &phi_loc, &phase_sdev,
331                     &phi_err, &omega_err);
332
333   ptr->phi0 = phi_loc;
334   ptr->v = omega_base;
335   ptr->sdev_phi = phase_sdev;
336   ptr->phi0_err = phi_err;
337   ptr->v_err = omega_err;
338
339   /* check the status of the phase fit to see if it was actually OK.
340      if not, set error bars to HUGE_VAL */
341
342   if (status != 0) {
343     ptr->sdev_phi = HUGE_VAL;
344     ptr->v_err = HUGE_VAL;
345     if (xflag) ptr->phi0_err = HUGE_VAL;
346   }
347
348   /* OK, we now have our baseline value for omega.  Now re-do the
349      phase fit, but using omega_loc + omega__err_loc. */
350
351
352   if (!xflag && (status == 0)) {
353   status = do_phase_fit (omega_loc + omega_err_loc,
354                          xflag, prm->mplgs, acf, tau,
355                     w, sum_wk2_arr, phi_res, badlag, t0,
356                     sum_w, sum_wk, sum_wk2,
357                     &omega_high, &phi_loc, &phase_sdev,
358                     &phi_err, &omega_err);
359
360   status = do_phase_fit (omega_loc - omega_err_loc,
361                          xflag, prm->mplgs, acf, tau,
362                     w, sum_wk2_arr, phi_res, badlag, t0,
363                     sum_w, sum_wk, sum_wk2,
364                     &omega_low, &phi_loc, &phase_sdev,
365                     &phi_err, &omega_err);
366
367   /* if the difference between the high and low values of omega
```

```
368       is greater than the error estimate of the original fit,
369       we will use the original fit as our best guess for the
370       velocity, but we'll set the error to be the difference between
371       the high and low values.  Actually, at this point we should have
372       non-symmetric error bar, but the file format has no provision
373       for that. */
374
375     if (fabs(omega_high - omega_low) >= 2*ptr->v_err) {
376       ptr->v = omega_base;
377       ptr->v_err = fabs(omega_high - omega_low);
378       }
379     }
380
381
382     /* POWER FITS:  We now turn to the power fits.  The sums have to be
383     partially redone, since we have subtracted P0n. */
384
385     /* We are now faced with the question of what to do if we don't have
enough
386     lags left to do a fit.  we can't abaondon the data because the phase
fit is
387     actually ok.  we have to have at least 3 points to do the fit and
estimate
388     an error on the fit.
389
390     If we don't have at least 3 good points, then simply set the lamda and
391     sigma powers both to the power_lag0 level.  If there are only 2 good
points
392     then calculate the value of sigma and lamda, but set the error estimate
393     to HUGE_VAL.
394
395     If we end up with only lag-0 being good, then flag the width estimate
396     by setting it to a large negative value.
397
398  */
399
400     if (npp < 3) {
401       c_log = pwr[0];
402
403       /* if c_log < 0 it means that after subtracting the noise and P0n,
404        the result is less than 1.0.  This must really be pretty meaningless
405        It shouldn't even be possible since we have supposedly already
checked
406        this at the beginning. */
407
408       if (c_log < 0 ) return 2;
409
410       ptr->p_l = c_log;
411       ptr->p_s = c_log;
412
413       /* find the last good lag */
414       last_good = -1;
415       for (k= 0; k < prm->mplgs; k++) if (!badlag[k]) last_good = k;
416
417       /* if there are no good lags, or only lag-0 is good, set the width
418          to a high negative value, by setting the last_good lag to 1
419          */
```

```
420
421     if (last_good <=0 ) {
422       ptr->w_l = -9999.0;
423       ptr->w_s = -9999.0;
424       ptr->p_l_err = HUGE_VAL;
425       ptr->p_s_err = HUGE_VAL;
426       ptr->w_l_err = HUGE_VAL;
427       ptr->w_s_err = HUGE_VAL;
428       ptr->sdev_l = HUGE_VAL;
429       ptr->sdev_s = HUGE_VAL;
430     } else {
431       /* now calculate the width as the lag-0 power divided by the
432        time to the last good lag. */
433
434       ptr->w_l = c_log/(tau[last_good]*t0);
435       ptr->w_s = c_log/(tau2[last_good]*t2);
436
437       /* set the errors to the maximum value */
438
439       ptr->p_l_err = HUGE_VAL;
440       ptr->p_s_err = HUGE_VAL;
441       ptr->w_l_err = HUGE_VAL;
442       ptr->w_s_err = HUGE_VAL;
443       ptr->sdev_l = HUGE_VAL;
444       ptr->sdev_s = HUGE_VAL;
445     }
446   } else {
447     /*  Calculate the sums that were not used in the phase fit */
448     for (k=1; k < prm->mplgs; k++) {
449       if (badlag[k] || bad_pwr[k]) continue;
450       sum_p = sum_p + wp[k];
451       sum_pk = sum_pk + pwr[k]*wt[k];
452       sum_pk2 = sum_pk2 + pwr[k]*wt2[k];
453       sum_wk4 = sum_wk4 + wt2[k]*tau2[k];
454     }
455
456     /* Now adjust the sums that were used in the phase fit, but that
457        have changed because of additional bad lags */
458
459     for (k=1; k< prm->mplgs; k++) {
460       if (bad_pwr[k]) {
461         sum_w = sum_w - w[k]*w[k];
462         sum_np = sum_np - 1;
463         sum_wk = sum_wk - w[k]*w[k]*tau[k];
464         sum_wk2 = sum_wk2 - wt2[k];
465       }
466     }
467
468 /*  start with the lamda fit */
469
470     d = determ(sum_w,-t0*sum_wk,t0*sum_wk,-t2*sum_wk2);
471     c_log = determ(sum_p,-t0*sum_wk,t0*sum_pk,-t2*sum_wk2)/d;
472
473     ptr->p_l = c_log;
474
475     ptr->w_l = determ(sum_w,sum_p,t0*sum_wk,t0*sum_pk)/d;
476
```

```
    if (sum_np > 3) {
      e2 = 0.;
      wbar = 0.;
      npp = 0;
      for (k=0; k<prm->mplgs; k++)
        if ((badlag[k] == 0) && (bad_pwr[k] == 0)) {
              temp = pwr[k] - (c_log - tau[k]*t0* (ptr->w_l));
              e2 = e2 + w[k]*w[k]*(temp*temp);
              wbar = wbar + w[k];
              npp++;
        }
      wbar = wbar/npp;
      ptr->sdev_l = sqrt(e2/sum_w/(npp - 2));

      if ((sum_w*sum_wk2 - sum_wk*sum_wk) <=0) {
        ptr->p_l_err = HUGE_VAL;
        ptr->w_l_err = HUGE_VAL;
        ptr->sdev_l = HUGE_VAL;
      } else {
        c_log_err = ptr->sdev_l * wbar *
            sqrt(sum_wk2/(sum_w*sum_wk2 - sum_wk*sum_wk));
        ptr->p_l_err = c_log_err;

        ptr->w_l_err = ptr->sdev_l * wbar *
             sqrt(sum_w/(t2*(sum_w*sum_wk2 -
                         sum_wk*sum_wk)));
      }
    } else {
      ptr->p_l_err = HUGE_VAL;
      ptr->w_l_err = HUGE_VAL;
      ptr->sdev_l = HUGE_VAL;
    }


/* ----------------now do the sigma fit ----------------------- */

    d = determ(sum_w,-t2*sum_wk2,t2*sum_wk2,-t4*sum_wk4);
    c_log = determ(sum_p,-t2*sum_wk2,t2*sum_pk2,-t4*sum_wk4)/d;

    ptr->p_s = c_log;

    ptr->w_s = determ(sum_w,sum_p,t2*sum_wk2,t2*sum_pk2)/d;

    if (sum_np > 3) {
      e2 = 0.;
      wbar = 0.;
      npp = 0;
      for (k=0; k<prm->mplgs; k++)
        if ((badlag[k] == 0) && (bad_pwr[k] == 0)) {
              temp = pwr[k] - (c_log - tau2[k]*t2* (ptr->w_s));
              e2 = e2 + w[k]*w[k]*(temp*temp);
              wbar = wbar + w[k];
              npp++;
        }
      wbar = wbar/npp;
      ptr->sdev_s = sqrt(e2/sum_w/(npp - 2));

```

```
534       if ((sum_w*sum_wk4 - sum_wk2*sum_wk2) <= 0.0 ) {
535         ptr->p_s_err = HUGE_VAL;
536         ptr->w_s_err = HUGE_VAL;
537         ptr->sdev_s = HUGE_VAL;
538       } else {
539         c_log_err = ptr->sdev_s * wbar *
540           sqrt(sum_wk4/(sum_w*sum_wk4 - sum_wk2*sum_wk2));
541         ptr->p_s_err = c_log_err;
542         ptr->w_s_err = ptr->sdev_s * wbar *
543            sqrt(sum_w/(t4*(sum_w*sum_wk4 - sum_wk2*sum_wk2)));
544
545
546       }
547     } else {
548       ptr->p_s_err = HUGE_VAL;
549       ptr->w_s_err = HUGE_VAL;
550       ptr->sdev_s = HUGE_VAL;
551     }
552
553     /* finally check for ground scatter fit */
554
555     /*  first, see if an ACF preprocessor has already identified the
556       scatter as being ground scatter.  */
557
558     if (acf_stat == ACF_GROUND_SCAT) ptr->gsct = 1;
559     else {
560       ptr->gsct = 0;
561     }
562   }
563
564   /* all done - return code = 1 */
565   if (npp < 1) return 4;
566   else return 1;
567 }
```