Press

Data Fitting and Uncertainty

A practical Introduction to Weighted Least Squares and beyond

2nd Edition

2015

by

Tilo Strutz

Appendix S - The Source Code

Appendix S

The Source Code

S.1 Licence and Disclaimer

The source code was written, assembled and tested carefully by the author. Nevertheless the code might contain minor errors. It is not guaranteed that this software will work under arbitrary conditions and especially using odd data as input. The author and the publisher take no responsibility and offer no warranties on this software. In case that you recognise bugs or unexpected behaviour of the fitting procedure using this software, please report to the author directly or via the publisher.

1. The usage of the source code for academic use is free.
2. The redistribution of source code either in original form, modified form, binary form, or part of other software is not allowed without explicit permission given by the author.
3. All advertising materials or publications mentioning features or use of this software must cite the source properly as
   "Strutz, T.: Data fitting and Uncertainty. Springer Fachmedien, 2015".
4. If commercial usage is planned, please contact the author to get information about details of a corresponding commercial licence.

S.2 Main functions

```
/*****************************************************************
 *
 * File....:  fitting.c
 * Function:  data fitting with least squares
 * Author..:  Tilo Strutz
 * Date....:  28.09.2009
 *
 * changes:
 * 20.08.2012 implementation of RANSAC, M-score
 * 29.04.2013 bugfix MAX_CONDITIONS vs M_MAX
 * 28.01.2014 new option cw
 * 09.12.2014 output of weighting and outlier-detection mode
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#ifndef WIN32
  #include <unistd.h>
#endif
#include "get_option.h"
#include "errmsg.h"
#include "matrix_utils.h"
#include "functions.h"
#include "functions_NIST.h"
#include "macros.h"
#include "prototypes.h"
#include "defines.h"

/* #define OUTPUT_DEVIATES */

/* model functions */
#define CONSTANT        0   /* constant value */
#define LINEAR          1   /* f(x|a) = a1 + SUM_j a_j * x_(j-1)  */
#define LINEAR_2        3   /* f(x|a) =      SUM_j a_j * x_(j-1)  */
#define COSINE_LIN      2   /* cosine linear  */
#define COSINE          5   /* cosine nonlinear */
#define EXPONENTIAL     6   /* exponential  */
#define LOGARITHM       7   /*    */
#define GAUSSIAN_2      8   /* superposition of two Gaussians */
#define EXPONENTIAL2    9   /* exponential 2  */
#define EXPONENTIAL2_LIN  10    /* exponential 2, linearised */
#define GEN_LAPLACE     11  /* generalised Laplacian distribution */
#define COSINE_TREND    12  /* cosine with linear trend */
#define GAUSSIAN_1      13  /* Gaussian */
#define POLYNOM_2NDORD  16  /* 2nd order polynomial */
#define POLYNOM_3RDORD  17  /* 3rd order polynomial */
#define POLYNOMIAL      18  /* multi-order polynomial */
#define POLYNOMIAL_REG  19  /* regularised */
#define QUAD_SURFACE    20  /* parabolic 2D surface */
#define COORD_TRANSF    21  /* rotation + translation */
#define TRIGONOMETRIC1  22  /* trigonometric polynom, 1st order */
#define TRIGONOMETRIC2  23  /* trigonometric polynom, 2st order */
#define CIRCLE          24  /* circle */
#define CIRCLE_LIN      25  /* circle, linearised */
#define CIRCLE_TLS      26  /* circle, total least squares */
#define NN_3x3x1        30  /* NeuralNet 3x3x1 */
#define NN_3x2x1        31  /* NeuralNet 3x2x1 */
#define NN_1x2x1        32  /* NeuralNet 1x2x1 */
#define NN_2x2x1        33  /* NeuralNet 2x2x1 */
#define NN_1x3x1        34  /* NeuralNet 1x3x1 */
#define NIST_THURBER    40  /* NIST data set */
#define NIST_MGH09      41  /* NIST data set */
#define NIST_RAT42      42  /* NIST data set */
#define NIST_RAT43      43  /* NIST data set */
#define NIST_ECKERLE4   44  /* NIST data set */
#define NIST_MGH10      45  /* NIST data set */
#define NIST_BENNETT5   46  /* NIST data set */
#define NIST_BOXBOD     47  /* NIST data set  */

/* defined in singvaldec.c  */
double euclid_dist( double a, double b);

/*---------------------------------------------------------------
 *  main()
 *--------------------------------------------------------------*/
int
main( int argc, char *argv[])
{
  char *rtn = "main";
  char *field;  /* used for reading text files  */
  char *inname = NULL;  /* filename of input data */
  char *outname = NULL; /* filename of results    */
  /* string with list of columns containing conditions */
  char *column_cond_str=NULL;
  /* used for reading text files  */
  char line[MAXLINELENGTH+1], *ptr;
  /* pointer to model function */
  double (*funct) (int, double*, double*) = NULL;
  /* pointer to its derivative */
  double (*funct_deriv) (double(*)(int,double*,double*),
          int,int,int,double*,double*) = NULL;
  double (*funct_deriv2) (double(*)(int,double*,double*),
          int,int,int,int,double*,double*) = NULL;
  /* pointer to initialisation function */
  int (*init) (int, double*,double*,double*,
              unsigned char*,FILE*) = NULL;
  int err = 0, i, j, o;
  int cnt;    /* counter for observations */
  int column_cond[MAX_CONDITIONS], col, ch;
  int column_weights = 0; /* column containing the weights */
  int column_obs = 0; /* column containing the observations */
  int cond_dim = 1;   /* dimensionality of conditions */
  int obs_dim = 1;    /* dimensionality of observations */
  int type = 0;       /* type of model function */
  int N;              /* number of observations */
  int M;              /* number of model parameters */
  int M_flag=0;       /* flag for model LINEAR, POLYNOMIAL_REG  */
  int numerical_flag = 0; /* force numerical derivation */
  int scale_flag = 0;   /* enable scaling of conditions */
  int forget_flag = 0;  /* enables reset of weights after
                         * outlier removal  */
  int num_outlier = 0;  /* number of outliers */
  int out_mode = 0;   /*
                       * 0 .. no outlier removal
                       * 1 .. enable z-score + Chauvenet's
                       * 2 .. enable ClubOD
                       * 3 .. enable M-score + Chauvenet's
                       * 4 .. enable RANSAC
                       */
  int weight_mode = 0;/*
                       * 0 .. use equal weights (no weighting)
                       * 1 .. enable deviates based weighting
                       * 2 .. weighting by binning
                       */
  int obs_per_bin = 50; /* observations per bin for
                         * weight_mode = 2
                         */
  int algo_mode = 1;  /*
                       * 0 .. use simple matrix inversion, M <= 5
                       * 1 .. singular value decomposition
                       * 2 .. LU decomposition
                       */
  int iter;        /* counter for weight iterations */
  int iter_wmax = 120;  /* number of iterations for weights
                           estimation */
  int iter_stop;  /* stop iteration when convergence is
                     reached */
  int iter_final; /* last iteration after convergence is
                     reached */
  int num_outliers;     /* number of detected outliers */
  int out_detect_flag;  /* indicates, whether outlier detection
                         * has been performed
                         */
  int argc_orig;
  double **jacob = NULL;  /* Jacobian matrix J    */
  double **covar = NULL;  /* covariance matrix C  */
  double *weights = NULL; /* vector for weights   */
  double *weights_old = NULL; /* vector for weights */
  double *obs = NULL;   /* observations */
  double *datac = NULL; /* calculated values based on parameters
                         */
  double *cond = NULL;  /* conditions X */
  double a[M_MAX];      /* parameter of model function    */
  unsigned char a_flag[M_MAX];    /* corresponding flags  */
  double *deviate = NULL;       /* = [obs - f(x|a) ]    */
  double *deviates_abs = NULL;  /* = abs[obs - f(x|a) ] */
  double *deltasq = NULL;       /* = [obs - f(x|a) ] ^2 */
  double chisq, sum, variance, energy, mean, mean_weights;
  double chisq_target;
  double gfit;        /* goodness of fit  */
  double uncertainty; /* sigma_y  */
  double scale_fac = 1.;  /* factor for scaling of conditions */
  LS_FLAG lsq_flag, *ls_flag;
  FILE *in = NULL;
  FILE *out = stdout;

  errno = 0; /* reset global error number, is checked in ls.c */
  /* reset all flags, assume that initial parameters must
   * be determined by the programm itself
   */
  for ( j = 0; j < M_MAX; j++)
  {
    a_flag[j] = 0;
  }

  /* set pointer to object of flags and initialise flags */
  ls_flag = &lsq_flag;
  ls_flag->linear = 1;
  ls_flag->svd = 1;   /* use SVD for linear systema as default  */
  ls_flag->LM = 1;    /* use Levenberg-Marquardt as default */
  ls_flag->chisq_target = 0;
  ls_flag->trueH = 0;
  ITERAT_MAX = 2000;  /* declared in ls.c */
```

```
190:
191: #ifdef TESTT
192:   {
193:     double **a, **b;
194:     double  det;
195:     a = matrix( 5, 5);  /*  matrix */
196:     b = matrix( 5, 5);  /*  matrix */
197:
198: a[0][0] = 2.; a[0][1] = 2.; a[0][2] = 3.; a[0][3] = 4.; a[0][4] = 5.;
199: a[1][0] = 2.; a[1][1] = 3.; a[1][2] = 5.; a[1][3] = 5.; a[1][4] = 5.;
200: a[2][0] = 1.; a[2][1] = 4.; a[2][2] = 4.; a[2][3] = 4.; a[2][4] = 2.;
201: a[3][0] = 1.; a[3][1] = 2.; a[3][2] = 1.; a[3][3] = 5.; a[3][4] = 3.;
202: a[4][0] = 3.; a[4][1] = 3.; a[4][2] = 3.; a[4][3] = 2.; a[4][4] = 6.;
203:
204:     det = inverse_5x5( a, b);
205:     fprintf( stderr, "\n det = %f\n", det);
206:     for ( i = 0; i < 5; i++)
207:     {
208:       for ( j = 0; j < 5; j++)
209:       {
210:         fprintf( stderr, " %6.2f", b[i][j]);
211:       }
212:       fprintf( stderr, "\n ");
213:     }
214:
215:     free_matrix( &a);
216:     free_matrix( &b);
217:     exit( 1);
218:   }
219: #endif
220:   argc_orig = argc; /* remember the number of arguments */
221:   /* check command-line parameters */
222:   while (( optstr =
223:     ( char*)get_option( argc, (const char **)argv)) != NULL)
224:   {
225:     switch (optstr[1])
226:     {
227:       case 'a':
228:         switch (optstr[2])
229:         {
230:           case '\0':
231:             algo_mode = atoi( OptArg);
232:             /* 0 .. use simple matrix inversion, M <= 3
233:              * 1 .. singular value decomposition
234:              * 2 .. LU decomposition
235:              */
236:             break;
237:           default:
238:             /* take number after 'a' as number of parameter
239:              * limited to 1..9
240:              * since OPTIONSTRING (usage.c) does not contain 'a0'
241:              * the domain of definition of j is limited to 1...9
242:              */
243:             j = atoi( &(optstr[2]));
244:             a[j-1] = atof( OptArg);
245:             a_flag[j-1] = 1;
246:         }
247:         break;
248:       case 'b':
249:         obs_per_bin = atoi( OptArg); /* observations per bin */
250:         break;
251:       case 'c':
252:         switch (optstr[2])
253:         {
254:           case 'c': /* string of comma-separated column numbers */
255:             column_cond_str = OptArg;
256:             break;
257:           case 'o': /* column number of observations */
258:             column_obs = atoi( OptArg);
259:             break;
260:           case 'w': /* column number of weights */
261:             column_weights = atoi( OptArg);
262:             break;
263:           default: /* option '-c' */
264:             scale_flag = 1;
265:             break;
266:         }
267:         break;
268:       case 'm': /* model function */
269:         type = atoi( OptArg);
270:         break;
271:       case 'n': /* force usage of numerical derivation  */
272:         numerical_flag = 1;
273:         break;
274:       case 'H': /* use true Hessian matrix  */
275:         ls_flag->trueH = 1;
276:         break;
277:       case 'I': /* maximum number of iterations */
278:         ITERAT_MAX = atoi( OptArg);
279:         break;
280:       case 'M': /* number of parameters (type == LINEAR)  */
281:         M = atoi( OptArg);
282:         M_flag = 1;
283:         break;
284:       case 'L': /* use plain Gauss-Newton */
285:         ls_flag->LM = 0;
286:         break;
287:       case 'w':
288:         weight_mode = atoi( OptArg);
289:         /* 0 ... equal weights; 1 ... deviates based;
290:          * 2 ... Bin-wise
291:          */
292:         break;
293:       case 'i':
294:         inname = OptArg;
295:         break;
296:       case 'o':
297:         outname = OptArg;
298:         break;
299:       case 't':
300:         chisq_target = atof( OptArg);
301:         ls_flag->chisq_target = 1;
302:         break;
303:       case 'x':
304:         out_mode = atoi( OptArg);
305:         /* 0 ... no removal;
306:          * 1 ... z-score + Chauvenet's;
307:          * 2 ... CluBOD;
308:          * 3 ... M-score + Chauvenet's;
309:          * 4 ... RANSAC;
310:          */
311:         break;
312:       case 's':
313:         ls_flag->svd = 0;
314:         /* disable special SVD function for solving linear model
315:          */
316:         break;
317:       case 'f':
318:         forget_flag = 1;
319:         /* forget weights after outlier removal
320:          */
321:         break;
322:       case '?':
323:       default:
324:         usage( argv[0]); /* provides  help */
325:         err = 11;
326:         goto endfunc;
327:     }
328:   }
329:
330:   /* check, whether all mandatory options were given */
331:   err = check_opt( argv[0]);
332:   if (err)
333:   {
334:     fprintf( stderr, "\n command:");
335:     for ( i = 0; i < argc_orig; i++)
336:     {
337:       fprintf( stderr, " %s", argv[i]);
338:     }
339:     fprintf( stderr, "\n");
340:     goto endfunc;
341:   }
342:
343:   /*
344:    * evaluation of programm options
345:    */
346:   if (outname == NULL)
347:   {
348:     fprintf( stderr, "\n Name of output file missing!");
349:     usage( argv[0]);
350:     goto endfunc;
351:   }
352:   if (inname == NULL)
353:   {
354:     fprintf( stderr, "\n Name of input file missing!");
355:     usage( argv[0]);
356:     goto endfunc;
357:   }
358:   if ((type == LINEAR || type == POLYNOMIAL ||
359:        type == POLYNOMIAL_REG) && M_flag == 0)
360:   {
361:     fprintf( stderr, "\n You have chosen mode '-m %d', ", type);
362:     fprintf( stderr,
363:              "but forgotten to set the function order '-M'");
364:     usage( argv[0]);
365:     goto endfunc;
366:   }
367:
368:   /* initialise default columns of conditions  */
369:   for ( i = 0; i < MAX_CONDITIONS; i++)
370:   {
371:     /* conditions in increasing order */
372:     column_cond[i] = i+1;
373:   }
374:
375:   /* if comma-separated list of columns is given */
376:   if (column_cond_str != NULL)
377:   {
378:     /* convert column string into numbers */
379:     i = col = 0;
380:     do
381:     {
```

```
382:       /* loop until all columns are read or
383:        * maximal number of columns is reached
384:        */
385:       ptr = &(column_cond_str[i]);
386:       sscanf( ptr, "%d", &(column_cond[col]));
387:       do
388:       { /* go to next number */
389:         i++;
390:         ch = column_cond_str[i];
391:       } while( ch != '\0' && ch != ',');
392:       i++;
393:       col++;
394:     } while ( ch != '\0' && col < MAX_CONDITIONS);
395:     for ( i = col; i < MAX_CONDITIONS; i++)
396:     {
397:         column_cond[i] = column_cond[i-1]+1;
398:     }
399:   }
400:
401:   /*
402:    * open the input file
403:    * determine the number of data sets
404:    */
405:   in = fopen( inname, "rt");
406:   if (in == NULL)
407:   {
408:     err = errmsg( ERR_OPEN_READ, rtn, inname, 0);
409:     goto endfunc;
410:   }
411:   /* open out file */
412:   out = fopen( outname, "wt");
413:   if (out == NULL)
414:   {
415:     err = errmsg( ERR_OPEN_WRITE, rtn, outname, 0);
416:     goto endfunc;
417:   }
418:
419:   fprintf( out, "# %s ==============================", rtn);
420:   fprintf( out, "\n# use data file: %s", inname);
421:
422:   /* determine number of observations by counting of valid lines */
423:   N = 0;
424:   while (( ptr = fgets( line, MAXLINELENGTH, in)) != NULL)
425:   {
426:     /* skip comment lines (starting with '#') and empty ones */
427:     if ( is_data_line( line, MAXLINELENGTH) )
428:     {
429:       N++;
430:       if (strlen( line) == MAXLINELENGTH-1)
431:       {
432:         fprintf( stderr,
433:             "\n lines of input file are too long (>%d)",
434:             MAXLINELENGTH);
435:         fprintf( stderr, ", increase MAXLINELENGTH");
436:       }
437:     }
438:   }
439:   fclose( in);
440:
441:   fprintf( stderr, "\n datafile contains %d data points\n", N);
442:
443:   /*
444:    * set number of parameters and redirect pointer to functions
445:    */
446:   switch (type)
447:   {
448:     case CONSTANT:
449:       /* y = a1 */
450:       fprintf( out, "\n# constant function");
451:       printf( "\n constant function");
452:       funct_deriv = fconstant_deriv;
453:       M = 1;
454:       break;
455:
456:     case LINEAR:
457:       /* f(x|a) = a1 + Sum_j(a_j*x_j)  */
458:       fprintf( out, "\n# linear in x, order %d", M-1);
459:       printf( "\n linear in x, order %d", M-1);
460:       funct_deriv = flin_deriv;
461:       /* M is set via program parameter */
462:       cond_dim = M - 1; /* first parameter a1 is just an offset
463:                          * w/o corresponding condition
464:                          */
465:       break;
466:     case LINEAR_2:
467:       /* f(x|a) = Sum_j(a_j*x_j)  */
468:       fprintf( out, "\n# linear in x, order %d, w/o a_1", M);
469:       printf( "\n linear in x, order %d, w/o a_1", M);
470:       funct_deriv = flin2_deriv;
471:       /* M is set via program parameter */
472:       cond_dim = M; /* first parameter a1 is not used  */
473:       break;
474:     case COSINE_LIN:
475:       /*
476:        * f(x|b) = b1 + b2 * cos( x - b3)
477:        * f(x|a) = a1 + a2 * cos( x) + a3 * sin( x)
478:        * a2 = b2 * cos(b3), a3 = b2 * sin(b3)
479:        * a1 = b1
480:        */
481:       fprintf( out, "\n# cosine (linear)");
482:       printf( "\n cosine (linear)");
483:       funct_deriv = fcosine_deriv;
484:       M = 3;
485:       break;
486:     case COSINE:
487:       /*
488:        * f(x|a) = a1 + a2 * cos( x - a3)
489:        */
490:       fprintf( out, "\n# cosine nonlinear");
491:       printf( "\n cosine nonlinear");
492:       if (numerical_flag)
493:         funct_deriv = f_deriv;/* use numerical differentiation */
494:       else
495:         funct_deriv = fcosine_nonlin_deriv;
496:       funct = fcosine_nonlin;
497:       init = init_cosine_nonlin;
498:       M = 3;
499:       ls_flag->linear = 0;  /* nonlinear */
500:       break;
501:     case COSINE_TREND:
502:       /*
503:        * f(x|a) = a1 + a2 * x + a3 * cos( x - a4)
504:        */
505:       fprintf( out, "\n# cosine with trend");
506:       printf( "\n cosine with trend");
507:       if (numerical_flag)
508:         funct_deriv = f_deriv;/* use numerical differentiation */
509:       else
510:         funct_deriv = fcosine_trend_deriv;
511:       funct = fcosine_trend;
512:       init = init_cosine_trend;
513:       M = 4;
514:       ls_flag->linear = 0;  /* nonlinear */
515:       break;
516:     case LOGARITHM:
517:       /* f(x|a) = a1 + a2 * exp( a3 * x) */
518:       fprintf( out, "\n# log( a1 * x)");
519:       printf( "\n log( a1 * x)");
520:       if (numerical_flag)
521:         funct_deriv = f_deriv;/* use numerical differentiation */
522:       else
523:         funct_deriv = flogarithmic_deriv; /*  */
524:       funct_deriv2 = flogarithmic_deriv2; /*  */
525:       funct = flogarithmic;
526:       init = init_logarithmic;
527:       M = 1;
528:       ls_flag->linear = 0;  /* nonlinear */
529:       break;
530:     case EXPONENTIAL:
531:       /* f(x|a) = a1 + a2 * exp( a3 * x) */
532:       fprintf( out, "\n# exponential");
533:       printf( "\n exponential");
534:       funct_deriv = f_deriv;  /* use numerical differentiation */
535:       funct = fexponential;
536:       init = init_exponential;
537:       M = 3;
538:       ls_flag->linear = 0;  /* nonlinear */
539:       break;
540:     case GEN_LAPLACE:
541:       /* f(x|a) = a1 * exp( -|x|^a2 * a3) */
542:       fprintf( out, "\n# gen. Laplacian");
543:       printf( "\n gen. Laplacian");
544:       if (numerical_flag)
545:         funct_deriv = f_deriv;/* use numerical differentiation */
546:       else
547:         funct_deriv = fgen_laplace_deriv;
548:       funct_deriv = f_deriv;
549:       funct_deriv2 = NULL;
550:       funct = fgen_laplace;
551:       init = init_gen_laplace;
552:       M = 3;
553:       ls_flag->linear = 0;  /* nonlinear */
554:       break;
555:     case GAUSSIAN_1:
556:       /*
557:        * f(x|a) = a1 * exp( a2 * (x-a3)^2) +
558:        */
559:       fprintf( out, "\n# single Gaussian");
560:       printf( "\n single Gaussian");
561:       if (numerical_flag)
562:         funct_deriv = f_deriv;/* use numerical differentiation */
563:       else
564:         funct_deriv = fgauss_deriv;
565:       funct_deriv2 = fgauss_deriv2;
566:       funct = fgauss1;
567:       init = init_gauss;
568:       M = 3;
569:       ls_flag->linear = 0;  /* nonlinear */
570:       break;
571:     case GAUSSIAN_2:
572:       /*
573:        * f(x|a) = a1 * exp( a2 * (x-a3)^2) +
```

```
574:        *          a4 * exp( a5 * (x-a6)^2)
575:        */
576:       fprintf( out, "\n# two Gaussians");
577:       printf( "\n two Gaussians");
578:       funct_deriv = f_deriv;  /* use numerical differentiation */
579:       funct = fgauss2;
580:       init = init_gauss2;
581:       M = 6;
582:       ls_flag->linear = 0;  /* nonlinear */
583:       break;
584:     case EXPONENTIAL2:
585:       /* f(x|a) = a2 * exp( a3 * x) */
586:       fprintf( out, "\n# exponential 2");
587:       printf( "\n exponential 2");
588:       if (numerical_flag)
589:         funct_deriv = f_deriv;/* use numerical differentiation */
590:       else
591:         funct_deriv = fexpon2_deriv;
592:       funct = fexpon2;
593:       init = init_expon2;
594:       M = 2;
595:       ls_flag->linear = 0;  /* nonlinear */
596:       break;
597:     case EXPONENTIAL2_LIN:
598:       /* ln(f(x|a)) = ln(a2) + a3 * x */
599:       fprintf( out, "\n# exponential 2, linearised");
600:       printf( "\n exponential 2, linearised");
601:       if (numerical_flag)
602:         funct_deriv = f_deriv;/* use numerical differentiation */
603:       else
604:         funct_deriv = flin_deriv;
605:       M = 2;
606:       break;
607:     case POLYNOM_2NDORD:
608:       /*
609:        * f(x|a) = a1 + a2 * x + a3 * x^2
610:        */
611:       fprintf( out, "\n# polynomial of 2nd order");
612:       printf( "\n polynomial of 2nd order");
613:       funct_deriv = fpolynom2_deriv;
614:       M = 3;
615:       break;
616:     case POLYNOM_3RDORD:
617:       /*
618:        * f(x|a) = a1 + a2 * x + a3 * x^2 + a4 * x^3
619:        */
620:       fprintf( out, "\n# polynomial of 3rd order");
621:       printf( "\n polynomial of 3rd order");
622:       funct_deriv = fpolynom3_deriv;
623:       M = 4;
624:       break;
625:     case POLYNOMIAL:
626:       /*
627:        * f(x|a) = a1 + a2 * x + a3 * x^2 + ...
628:        */
629:       fprintf( out, "\n# polynomial of %dth order", M-1);
630:       printf( "\n polynomial of %dth order", M-1);
631:       funct_deriv = fpolynomial_deriv;
632:       /* M is set via program parameter */
633:       break;
634:     case POLYNOMIAL_REG:
635:       /*
636:        * f(x|a) = a1 + a2 * x + a3 * x^2 + ...
637:        */
638:       if (M == 2)
639:       {
640:         fprintf( out, "\n# polynomial of 1st order");
641:         printf( "\n polynomial of 1st order");
642:       }
643:       else if (M==3)
644:       {
645:         fprintf( out, "\n# polynomial of 2nd order");
646:         printf( "\n polynomial of 2nd order");
647:       }
648:       else if (M==3)
649:       {
650:         fprintf( out, "\n# polynomial of 3rd order");
651:         printf( "\n polynomial of 3rd order");
652:       }
653:       else
654:       {
655:         fprintf( out, "\n# polynomial of %dth order", M-1);
656:         printf( "\n polynomial of %dth order", M-1);
657:       }
658:       fprintf( out, ", regularised (nonlinear)");
659:       printf( ", regularised (nonlinear)");
660:
661:       if (numerical_flag)
662:         funct_deriv = f_deriv;/* use numerical differentiation */
663:       else
664:         funct_deriv = fpolynomial_deriv;
665:       funct = fpolynomial;
666:       init = init_polynomial;
667:       /* M is set via program parameter */
668:       ls_flag->linear = 0;  /* nonlinear */
669:       break;
670:     case QUAD_SURFACE:
671:       /*
672:        * f(x|a) = a1 + a2*x1 + a3*x1^2 + a4*x2 +a5*x2^2
673:        */
674:       fprintf( out, "\n# quadratic surface");
675:       printf( "\n quadratic surface");
676:       funct_deriv = fquadsurface_deriv;
677:       cond_dim = 2;
678:       M = 5;
679:       break;
680:     case COORD_TRANSF:
681:       /*
682:        * f1(x|a) = a1 + cos(a3) * x1 - sin(a3) * x2
683:        * f2(x|a) = a2 + sin(a3) * x1 + cos(a3) * x2
684:        */
685:       fprintf( out, "\n# rotation");
686:       printf( "\n rotation");
687:       funct_deriv = f_deriv;  /* use numerical differentiation */
688:       funct_deriv = frotation_deriv;
689:       funct = frotation;
690:       init = init_rotation;
691:       M = 3;
692:       cond_dim = 2;
693:       obs_dim = 2;
694:       ls_flag->linear = 0;  /* nonlinear */
695:       break;
696:     case TRIGONOMETRIC1:
697:       /*
698:        * f(x|a) = a1 + a2*cos(a3*x-a4)
699:        */
700:       fprintf( out, "\n# trigonometric 1st order");
701:       printf( "\n trigonometric 1st order");
702:       funct_deriv = f_deriv;  /* use numerical differentiation */
703:       funct = ftrigonometric1;
704:       init = init_trigonometric1;
705:       M = 4;
706:       cond_dim = 1;
707:       obs_dim = 1;
708:       ls_flag->linear = 0;  /* nonlinear */
709:       break;
710:     case TRIGONOMETRIC2:
711:       /*
712:        * f(x|a) = a1 + a2*cos(a3*x-a4) + a5*cos(2*a3*x-a6)
713:        */
714:       fprintf( out, "\n# trigonometric 2nd order");
715:       printf( "\n trigonometric 2nd order");
716:       funct_deriv = f_deriv;  /* use numerical differentiation */
717:       funct = ftrigonometric2;
718:       init = init_trigonometric2;
719:       M = 6;
720:       cond_dim = 1;
721:       obs_dim = 1;
722:       ls_flag->linear = 0;  /* nonlinear */
723:       break;
724:     case CIRCLE:
725:       /*
726:        * f(x|a) = 0 = (x1 - a1)^2 + (x2 - a2)^2 - a3*a3
727:        */
728:       fprintf( out, "\n# circle");
729:       printf( "\n circle");
730:       if (numerical_flag)
731:         funct_deriv = f_deriv;/* use numerical differentiation */
732:       else
733:         funct_deriv = fcircle_deriv;
734:       funct = fcircle;
735:       init = init_circle;
736:       M = 3;
737:       cond_dim = 2;
738:       obs_dim = 1;
739:       ls_flag->linear = 0;  /* nonlinear */
740:       break;
741:     case CIRCLE_TLS:
742:       /*
743:        * f(x|a) = 0 = (sqrt[(x1 - a1)^2 + (x2 - a2)^2] - a3)^2
744:        */
745:       fprintf( out, "\n# circle, TLS");
746:       printf( "\n circle, TLS");
747:       if (numerical_flag)
748:         funct_deriv = f_deriv;/* use numerical differentiation */
749:       else
750:         funct_deriv = fcircleTLS_deriv;
751:       funct = fcircleTLS;
752:       init = init_circle;
753:       M = 3;
754:       cond_dim = 2;
755:       obs_dim = 1;
756:       ls_flag->linear = 0;  /* nonlinear */
757:       break;
758:     case CIRCLE_LIN:
759:       /*
760:        * f(x|a) = 0 = (x1 - a1)^2 + (x2 - a2)^2 - a3*a3
761:        * f(x|b) = x1^2 + x2^2 = b1*x1 + b2*x2 - b3
762:        * b1 = 2*a1, b2 = 2*a2, b3 = a1^2 + a2^2 - a3^2
763:        */
764:       fprintf( out, "\n# circle, linearised");
765:       printf( "\n circle, linearised");
```

```
766:       funct_deriv = fcirclelin_deriv;
767:       init = init_circlelin;
768:       M = 3;
769:       cond_dim = 2;
770:       obs_dim = 1;
771:       ls_flag->linear = 1;  /* linear */
772:       break;
773:     case NN_3x3x1:
774:       /*
775:        * f(x|a) = neural network 3x3x1
776:        */
777:       fprintf( out, "\n# NN 3x3x1");
778:       printf( "\n NN 3x3x1");
779:       funct_deriv = f_deriv;  /* use numerical differentiation */
780:       funct = fNN_3_3;
781:       init = init_NN3x3x1;
782:       M = 16;
783:       cond_dim = 3;
784:       obs_dim = 1;
785:       ls_flag->linear = 0;  /* nonlinear */
786:       break;
787:     case NN_3x2x1:
788:       /*
789:        * f(x|a) = neural network 3x2x1
790:        */
791:       fprintf( out, "\n# NN 3x2x1");
792:       printf( "\n NN 3x2x1");
793:       funct_deriv = f_deriv;  /* use numerical differentiation */
794:       funct = fNN_3_2;
795:       init = init_NN;
796:       M = 11;
797:       cond_dim = 3;
798:       obs_dim = 1;
799:       ls_flag->linear = 0;  /* nonlinear */
800:       break;
801:     case NN_1x2x1:
802:       /*
803:        * f(x|a) = neural network 1x2x1
804:        */
805:       fprintf( out, "\n# NN 1x2x1");
806:       printf( "\n NN 1x2x1");
807:       funct_deriv = f_deriv;  /* use numerical differentiation */
808:       funct = fNN_1_2;
809:       init = init_NN;
810:       M = 7;
811:       cond_dim = 1;
812:       obs_dim = 1;
813:       ls_flag->linear = 0;  /* nonlinear */
814:       break;
815:     case NN_2x2x1:
816:       /*
817:        * f(x|a) = neural network 2x2x1
818:        */
819:       fprintf( out, "\n# NN 2x2x1");
820:       printf( "\n NN 2x2x1");
821:       funct_deriv = f_deriv;  /* use numerical differentiation */
822:       funct = fNN_2_2;
823:       init = init_NN;
824:       M = 9;
825:       cond_dim = 2;
826:       obs_dim = 1;
827:       ls_flag->linear = 0;  /* nonlinear */
828:       break;
829:     case NN_1x3x1:
830:       /*
831:        * f(x|a) = neural network 1x2x1
832:        */
833:       fprintf( out, "\n# NN 1x3x1");
834:       printf( "\n NN 1x3x1");
835:       funct_deriv = f_deriv;  /* use numerical differentiation */
836:       funct = fNN_1_3;
837:       init = init_NN1x3x1;
838:       M = 10;
839:       cond_dim = 1;
840:       obs_dim = 1;
841:       ls_flag->linear = 0;  /* nonlinear */
842:       break;
843:     case NIST_THURBER:
844:       /*
845:        * f(x|a) =(a1 + a2*x + a3*x**2 + a4*x**3) /
846:        *          (1 + a5*x + a6*x**2 + a7*x**3)
847:        */
848:       fprintf( out, "\n# NIST_THURBER");
849:       printf( "\n NIST_THURBER");
850:       if (numerical_flag)
851:         funct_deriv = f_deriv;/* use numerical differentiation */
852:       else
853:         funct_deriv = fNIST_thurber_deriv;
854:       funct = fNIST_thurber;
855:       init = init_NIST_thurber;
856:       M = 7;
857:       cond_dim = 1;
858:       obs_dim = 1;
859:       ls_flag->linear = 0;  /* nonlinear */
860:       break;
861:     case NIST_MGH09:
862:       /*
863:        * f(x|a) =a1 * (x**2 + a2*x) / (x*x + a3*x + a4)
864:        */
865:       fprintf( out, "\n# NIST_MGH09");
866:       printf( "\n NIST_MGH09");
867:       if (numerical_flag)
868:         funct_deriv = f_deriv;/* use numerical differentiation */
869:       else
870:         funct_deriv = fNIST_MGH09_deriv;
871:       funct = fNIST_MGH09;
872:       init = init_NIST_MGH09;
873:       M = 4;
874:       cond_dim = 1;
875:       obs_dim = 1;
876:       ls_flag->linear = 0;  /* nonlinear */
877:       break;
878:     case NIST_RAT42:
879:       /*
880:        * f(x|a) = a1 / (1 + exp(a2 - a3*x))
881:        */
882:       fprintf( out, "\n# NIST_Rat42");
883:       printf( "\n NIST_Rat42");
884:       if (numerical_flag)
885:         funct_deriv = f_deriv;/* use numerical differentiation */
886:       else
887:         funct_deriv = fNIST_Rat42_deriv;
888:       funct = fNIST_Rat42;
889:       init = init_NIST_Rat42;
890:       M = 3;
891:       cond_dim = 1;
892:       obs_dim = 1;
893:       ls_flag->linear = 0;  /* nonlinear */
894:       break;
895:     case NIST_RAT43:
896:       /*
897:        * f(x|a) = a1 / [1 + exp(a2 - a3*x)]^(1/a4)
898:        */
899:       fprintf( out, "\n# NIST_Rat43");
900:       printf( "\n NIST_Rat43");
901:       if (numerical_flag)
902:         funct_deriv = f_deriv;/* use numerical differentiation */
903:       else
904:         funct_deriv = fNIST_Rat43_deriv;
905:       funct = fNIST_Rat43;
906:       init = init_NIST_Rat43;
907:       M = 4;
908:       cond_dim = 1;
909:       obs_dim = 1;
910:       ls_flag->linear = 0;  /* nonlinear */
911:       break;
912:     case NIST_ECKERLE4:
913:       /*
914:        * f(x|a) = a1 / a2 * exp(-0.5*((x -a3)/ a2)^2)
915:        */
916:       fprintf( out, "\n# NIST_ECKERLE4");
917:       printf( "\n NIST_ECKERLE4");
918:       if (numerical_flag)
919:         funct_deriv = f_deriv;/* use numerical differentiation */
920:       else
921:         funct_deriv = fNIST_Eckerle4_deriv;
922:       funct = fNIST_Eckerle4;
923:       init = init_NIST_Eckerle4;
924:       M = 3;
925:       cond_dim = 1;
926:       obs_dim = 1;
927:       ls_flag->linear = 0;  /* nonlinear */
928:       break;
929:     case NIST_MGH10:
930:       /*
931:        * f(x|a) = a1 * exp( a2 / (x+a3))
932:        */
933:       fprintf( out, "\n# NIST_MGH10");
934:       printf( "\n NIST_MGH10");
935:       if (numerical_flag)
936:         funct_deriv = f_deriv;/* use numerical differentiation */
937:       else
938:         funct_deriv = fNIST_MGH10_deriv;
939:       funct_deriv2 = fNIST_MGH10_deriv2;
940:       funct = fNIST_MGH10;
941:       init = init_NIST_MGH10;
942:       M = 3;
943:       cond_dim = 1;
944:       obs_dim = 1;
945:       ls_flag->linear = 0;  /* nonlinear */
946:       break;
947:     case NIST_BENNETT5:
948:       /*
949:        * f(x|a) = a1 * (x+a2)^(-1/a3)
950:        */
951:       fprintf( out, "\n# NIST_BENNETT5");
952:       printf( "\n NIST_BENNETT5");
953:       if (numerical_flag)
954:         funct_deriv = f_deriv;/* use numerical differentiation */
955:       else
956:         funct_deriv = fNIST_Bennett5_deriv; /* */
957:       funct = fNIST_Bennett5;
```

```
      init = init_NIST_Bennett5;
      M = 3;
      cond_dim = 1;
      obs_dim = 1;
      ls_flag->linear = 0;  /* nonlinear */
      break;
    case NIST_BOXBOD:
      /* f(x|a) = a1 *(1 - exp( -a2 * x) */
      fprintf( out, "\n# NIST_BOXBOD");
      printf( "\n NIST_BOXBOD");
      if (numerical_flag)
        funct_deriv = f_deriv;/* use numerical differentiation */
      else
        funct_deriv = fNIST_BoxBOD_deriv;
      funct = fNIST_BoxBOD;
      init = init_NIST_BoxBOD;
      M = 2;
      ls_flag->linear = 0;  /* nonlinear */
      break;
    default:
      err = errmsg( ERR_NOT_DEFINED, rtn, "-m ", type);
      usage( argv[0]);
      goto endfunc;
  }

  if (ls_flag->linear && numerical_flag)
  {
    fprintf( stderr, "\n Numerical derivation is not implemented");
    fprintf( stderr, "   for linear model functions! -------\n");
    numerical_flag = 0;
  }

  /* if column for observation is not given explicitely by a
   * command-line parameter, then assume the column following
   * the conditions
   */
  if (column_obs == 0) column_obs = cond_dim + 1;
  printf( "\n");
  fflush( stdout);

  if (M > 5 && algo_mode == 0)
  {
     fprintf( stderr, "\n too much  parameters (%d) ", M);
     fprintf( stderr, "for standard matrix inversion");
     usage( argv[0]);
     err = 42;
     goto endfunc;
   }

   if (M > MAX_CONDITIONS)
   {
     fprintf( stderr, "\n too much  parameters (%d) ", M);
     fprintf( stderr, "maximum is %d", MAX_CONDITIONS);
     err = 43;
     goto endfunc;
   }

   if (N < M)
   {
     fprintf( stderr, "\nToo less observations (%d) compared ", N);
     fprintf( stderr, "to number of model parameters (%d)\n", M);
     err = 44;
     goto endfunc;
   }
   fprintf( out, "\n# Number of observations: %d", N);
   fprintf( out, "\n# Number of parameters  : %d", M);

   if (ls_flag->linear && ls_flag->svd && algo_mode != 1)
   {
     fprintf( stderr, "\n# option '-a %d' is ignored, ", algo_mode);
     fprintf( stderr, "since special SVD approach is used!");
     algo_mode = 1;
   }
   if (ls_flag->trueH && funct_deriv2 == NULL)
   {
     fprintf( stderr,
       "\n### function for 2nd derivativ was not initialised!");
     err = 45;
     goto endfunc;
   }

   /*
    * allocate memory
    */
   jacob = matrix( N * obs_dim, M);  /* Jacobian */
   covar = matrix( M, M);  /* covariance matrix */

   obs = vector( N * obs_dim); /* observations */
   datac = vector( N * obs_dim); /* calculated data using
                                    f(x|a) */
   cond = vector( N * cond_dim); /* conditions x */
   weights = vector( N * obs_dim); /* weights */
   weights_old = vector( N * obs_dim); /* weights one step back
                                          */
   deviate = vector( N * obs_dim); /* remaining differences */
   /* remaining absolute differences */
   deviates_abs = vector( N * obs_dim);
   deltasq = vector( N * obs_dim); /* remaining squared
                                      differences */

   /* open input file again */
   in = fopen( inname, "rt");
   if (in == NULL)
   {
     err = errmsg( ERR_OPEN_READ, rtn, inname, 0);
     perror( "\nReason");
     goto endfunc;
   }

   fprintf( out, "\n# condition columns: ");
   for (j = 0; j < cond_dim; j++)
   {
     fprintf( out, "%d ", column_cond[j]);
   }

   fprintf( out, "\n# observations column: ");
   for (o = 0; o < obs_dim; o++)
   {
     fprintf( out, "%d ", column_obs + o);
   }

   if (column_weights)
     fprintf( out, "\n# weights column: %d", column_weights);

   fprintf( out, "\n#");
   if (!ls_flag->linear || (ls_flag->linear && !ls_flag->svd) )
   {
     fprintf( out, "\n# algorithm for inversion: ");
     if (algo_mode == 0)
       fprintf( out, "Cofactor");
     else if (algo_mode == 1)
       fprintf( out, "SVD");
     else if (algo_mode == 2)
       fprintf( out, "LU decomposition");
   }

   /* put statement about weighting and outlier detection scheme */
   fprintf( out, "\n# mode of weighting:");
   switch (weight_mode)
   {
     case 0:   if (column_weights)
               {
                 fprintf( out, "  weights have been provided");
               }
               else
               {
                 fprintf( out, "  no weighting used");
               }
               break;
     case 1:   fprintf( out,
                 "  estimate weights based on deviates"); break;
     case 2:   fprintf( out,
                 "  estimate weights based on binning"); break;
     default: break;
   }
   if (forget_flag)
       fprintf( out, "\n# reset weights after outlier removal");
   fprintf( out, "\n# mode of outlier detection:");
   switch (out_mode)
   {
     case 0: fprintf( out, "  no outlier detection"); break;
     case 1: fprintf( out,
               "  based on z-score and Chauvenet's criterion");
             break;
     case 2: fprintf( out, "  based on ClubOD"); break;
     case 3: fprintf( out,
               "  based on M-score and Chauvenet's criterion");
             break;
     case 4: fprintf( out, "  based on RANSAC"); break;
     default: break;
   }

   if (ls_flag->linear)
   {
     fprintf( out, "\n# fitting a linear system");
     if (ls_flag->svd)
       fprintf( out, "\n# use special SVD based algorithm");
   }
   else
   {
     fprintf( out, "\n# fitting a nonlinear system");
     if (ls_flag->LM)
       fprintf( out, "\n# use Levenberg-Marquardt method");
     else
       fprintf( out, "\n# use Gauss-Newton method");
     if (ls_flag->chisq_target)
       fprintf( out, "\n# chisq must be lower than %f",
         chisq_target);
   }
   /* write LS flags in output */

   /* read the conditions and observations */
```

```
1150:   for (i = 0; i < N; i++)
1151:   {
1152:     /* jump over comments and empty lines */
1153:     do
1154:     {
1155:       ptr = fgets( line, MAXLINELENGTH, in);
1156:     } while ( !is_data_line( line, MAXLINELENGTH) );
1157:
1158:     /* loop over all conditions */
1159:     for (j = 0; j < cond_dim; j++)
1160:     {
1161:       /* if 0 was given, then assume that conditions are just
1162:        * serial numbers 1,2,3,...
1163:        */
1164:       if (column_cond[j] == 0)
1165:       {
1166:         if (cond_dim > 1)
1167:         {
1168:           fprintf( stderr,
1169:             "\n There is more than one condition (%d)!", cond_dim);
1170:           fprintf( stderr,
1171:             "\n Columns of conditions must be given via '-cc'!\n");
1172:           goto endfunc;
1173:         }
1174:         cond[cond_dim * i + j] = i+1;
1175:       }
1176:       else
1177:       {
1178:         /* get string starting from desired column */
1179:         field = get_nth_field( line, column_cond[j]);
1180:         if (field != NULL)
1181:         {
1182:           /* multidimensional conditions are stored one after each
1183:              other */
1184:           sscanf( field, "%lf", &( cond[cond_dim * i + j]));
1185:         }
1186:         else
1187:         {
1188:           fprintf( stderr, "\n\n === %d th column does not exist",
1189:             column_cond[j]);
1190:           err = 13;
1191:           goto endfunc;
1192:         }
1193:       }
1194:     }
1195:     /* loop over all observations */
1196:     for (o = 0; o < obs_dim; o++)
1197:     {
1198:       /* get string starting from desired column */
1199:       field = get_nth_field( line, column_obs + o);
1200:       if (field != NULL)
1201:       {
1202:         sscanf( field, "%lf", &( obs[obs_dim * i + o]));
1203:       }
1204:       else
1205:       {
1206:         fprintf( stderr, "\n\n === %d th column does not exist",
1207:           column_obs);
1208:         err = 13;
1209:         goto endfunc;
1210:       }
1211:     }
1212:
1213:     /* get string starting from desired column */
1214:     if (column_weights > 0)
1215:     {
1216:       field = get_nth_field( line, column_weights);
1217:       if (field != NULL)
1218:       {
1219:         sscanf( field, "%lf", &( weights[i]));
1220:       }
1221:       else
1222:       {
1223:         fprintf( stderr,
1224:           "\n\n === %d th column does not exist in line %d",
1225:           column_obs, i);
1226:         err = 13;
1227:         goto endfunc;
1228:       }
1229:     }
1230:   }
1231:   fclose( in);
1232:
1233:   /*---------------------------------------------
1234:    * scaling of conditions, if enabled
1235:    */
1236:   if (scale_flag)
1237:   {
1238:     /* get maximum absolute value */
1239:     double max_cond, abs_cond;
1240:     switch (type)
1241:     {
1242:       case LINEAR:
1243:         max_cond = fabs( cond[0]);
1244:         for (i = 1; i < N*cond_dim; i++)
1245:         {
1246:           abs_cond = fabs( cond[i]);
1247:           if ( max_cond < abs_cond) max_cond = abs_cond;
1248:         }
1249:         scale_fac = 1./max_cond;
1250:
1251:         /* do scaling */
1252:         for (i = 0; i < N*cond_dim; i++)
1253:         {
1254:           cond[i] *= scale_fac;
1255:         }
1256:         fprintf( out, "\n# scaling activated");
1257:         fprintf( out, ", scale_fac = %f", scale_fac);
1258:         /* For cond_dim > 1 it would be even better to scale each
1259:          * condition separately. This would require cond_dim
1260:          * different scaling factors
1261:          */
1262:         break;
1263:       case POLYNOMIAL:
1264:       /* scaling has no positive effect for these nonlinear
1265:        * model functions */
1266:       case NIST_RAT42:
1267:       case NIST_ECKERLE4:
1268:       case NIST_MGH10:
1269:         max_cond = fabs( cond[0]);
1270:         for (i = 1; i < N; i++)
1271:         {
1272:           abs_cond = fabs( cond[i]);
1273:           if ( max_cond < abs_cond) max_cond = abs_cond;
1274:         }
1275:         scale_fac = 1./max_cond;
1276:
1277:         /* do scaling */
1278:         for (i = 0; i < N; i++)
1279:         {
1280:           cond[i] *= scale_fac;
1281:         }
1282:         fprintf( out, "\n# scaling activated");
1283:         fprintf( out, ", scale_fac = %f", scale_fac);
1284:         break;
1285:       default:
1286:         fprintf( out, "\n# scaling not supported for '-m %d'", type);
1287:         fprintf( stderr,
1288:           "\n#### scaling not supported for '-m %d'###\n", type);
1289:     }
1290:   }
1291:
1292:   /* check input data */
1293:   if (type == COSINE_LIN || type == COSINE)
1294:   {
1295:     /* conditions in degree ? */
1296:     double max_cond, abs_cond;
1297:     max_cond = fabs( cond[0]);
1298:     for (i = 1; i < N; i++)
1299:     {
1300:       abs_cond = fabs( cond[i]);
1301:       if ( max_cond < abs_cond) max_cond = abs_cond;
1302:     }
1303:     if ( max_cond < 2*M_PI)
1304:     {
1305:       fprintf( stderr, "\n====================================");
1306:       fprintf( stderr, "\n== range of degrees is very small!! ==");
1307:       fprintf( stderr, "\n== Mismatch with radians??          ==");
1308:       fprintf( stderr, "\n== Please check.\n                 ==");
1309:       fprintf( stderr, "\n====================================");
1310:     }
1311:   }
1312:
1313:   /* prepare input data */
1314:   if (type == CIRCLE_LIN)
1315:   {
1316:     for ( i = 0; i < N; i++)
1317:     {
1318:       obs[i] = cond[2*i] * cond[2*i] + cond[2*i+1] * cond[2*i+1];
1319:     }
1320:   }
1321:
1322:   /* initialize weights */
1323:   if (column_weights == 0) /* no weights given  */
1324:   {
1325:     for (i = 0; i < N * obs_dim; i++)
1326:     {
1327:       /* sum of all weights must be equal to N minus number of
1328:        * outliers
1329:        */
1330:       weights[i] = 1.0;
1331:       weights_old[i] = 1.0;
1332:     }
1333:     mean_weights = 1;
1334:   }
1335:   else
1336:   {
1337:     mean_weights = 0;
1338:     for (i = 0; i < N; i++)
1339:     {
1340:       mean_weights += weights[i];
1341:     }
```

```
1342:   }
1343:
1344:   /*
1345:    * pre-processing if necessary
1346:    */
1347:
1348:   /* linearisation */
1349:   if (type == EXPONENTIAL2_LIN)
1350:   {
1351:     /* ln(f(x|a)) = ln(a1) + (a2 * x) */
1352:     /* estimates for parameters */
1353:     for (i = 0; i < N; i++)
1354:     {
1355:       if (obs[i] <= 0.0)
1356:       {
1357:         /* now we have a problem; this observation is invalid */
1358:         weights[i] = 0;
1359:         obs[i] = -9999.;
1360:       }
1361:       else
1362:         obs[i] = log( obs[i]);
1363:     }
1364:   }
1365:
1366:   /*
1367:    * set initial parameters for nonlinear functions
1368:    * parameter values given on command line wont be changed
1369:    */
1370:   if (!ls_flag->linear)
1371:   {
1372:     init( N, obs, cond, a, a_flag, out);
1373:
1374:     fprintf( out, "\n# initial Parameters\n# ");
1375:     /* write initial parameters to output */
1376:     for (i = 0; i < M; i++)
1377:     {
1378:       fprintf( out, "a%d=%.9f, ", i+1, a[i]);
1379:     }
1380:     for (i = M; i < M_MAX; i++)
1381:     {
1382:       /* zero out unnecessary parameters
1383:        * required for POLYNOMIAL_REG
1384:        */
1385:       a[i] = 0.;
1386:     }
1387:     /* If scaling is enabled, all initial parameters, which are set
1388:      * independently on the condition values, must be corrected
1389:      */
1390:     if (scale_flag)
1391:     {
1392:       switch ( type)
1393:       {
1394:       /* scaling has no positive effect for these nonlinear
1395:        * model functions */
1396:         case NIST_RAT42:
1397:           a[2] /= scale_fac;
1398:           break;
1399:         case NIST_ECKERLE4:
1400:           a[0] *= scale_fac;
1401:           a[1] *= scale_fac;
1402:           a[2] *= scale_fac;
1403:           break;
1404:         case NIST_MGH10:
1405:           a[1] *= scale_fac;
1406:           a[2] *= scale_fac;
1407:           break;
1408:       }
1409:       fprintf( out, "\n# scaled\n# ");
1410:       /* write initial parameters to output */
1411:       for (i = 0; i < M; i++)
1412:       {
1413:         fprintf( out, "a%d=%.9f, ", i+1, a[i]);
1414:       }
1415:     }
1416:   }
1417:
1418:   /*
1419:    * prepare Jacobian matrix containing
1420:    * first derivatives of target function
1421:    */
1422:   for (i = 0; i < N * obs_dim; i++)
1423:   {
1424:     for (j = 0; j < M; j++)
1425:     {
1426:       jacob[i][j] = funct_deriv( funct, i, j, M, cond, a);
1427:     }
1428:   }
1429:
1430:   /* debugging output, because of interleaved observations */
1431:   if (type == COORD_TRANSF)
1432:   {
1433:     fprintf( out, "\n#\n#== Obs ===== Jacobian ==========");
1434:     for (i = 0; i < MIN(10,N); i++)
1435:     {
1436:       fprintf( out, "\n# %8.1f", obs[i]);
1437:       for (j = 0; j < M; j++)
1438:       {
1439:         fprintf( out, " %8.1f", jacob[i][j]);
1440:       }
1441:     }
1442:   }
1443:
1444:   /*
1445:    *  estimation of weights if required
1446:    */
1447:   iter_stop = 0;
1448:   if (column_weights > 0) /* weights given  */
1449:   {
1450:     if (weight_mode > 0)
1451:       fprintf( out,
1452:       "\n! weights were read from file. set weight_mode = 0 !\n");
1453:     weight_mode = 0; /* overwrite weights mode  */
1454:   }
1455:
1456:   if (weight_mode == 0)
1457:   {
1458:     iter_stop = 1; /* only one run */
1459:     iter_final = 1; /* only one run for ls and outlier removal */
1460:   }
1461:   else if (weight_mode == 2)
1462:   {
1463:     /* weights can be estimated beforehand via binning */
1464:     est_weights2( N * obs_dim, cond, obs, weights,
1465:                   obs_per_bin, out);
1466:     iter_stop = 1; /* only one run */
1467:     iter_final = 1; /* only one run of least squares */
1468:   }
1469:   else
1470:       iter_final = 0;
1471:
1472:   /* outlier detection has not be performed yet */
1473:   out_detect_flag = 0;
1474:
1475:
1476:   /* ------------------------------------------------- */
1477:   /* loop for weights estimation */
1478:   for (iter = 0;
1479:        (iter <= iter_wmax /* cont. as long max. number of
1480:                              iterations has not reached     */
1481:              && !iter_stop)  /* the stop flag is not set   */
1482:              || (iter_final);/* or it is the last round    */
1483:            iter++)
1484:   {
1485:     if (iter_final && iter_stop)
1486:       iter_final = 0; /* if final round reached, then reset flag */
1487:
1488:     /* feedback on console */
1489:     printf( "\r iterations: %3d", iter);
1490:
1491:     /* estimate weights in all but the last iteration */
1492:     fprintf( out, "\n#\n#==================================");
1493:     if (!iter_stop)
1494:       fprintf( out, "\n# %s: weights iteration #: %d", rtn, iter);
1495:     else
1496:     {
1497:       fprintf( out,
1498:       "\n# %s: approximation with final weights #: %d", rtn, iter);
1499:     }
1500:
1501:     /* do the least squares approximation   */
1502:     err =
1503:       ls( funct, funct_deriv, funct_deriv2, init, N * obs_dim, M,
1504:           obs, cond, jacob, weights, a, a_flag, algo_mode, ls_flag,
1505:           chisq_target, covar, out);
1506:
1507:     /* compute weighted and squared differences, chi-squared */
1508:     chisq = energy = mean = 0.0;
1509:     cnt = 0;
1510:     if (ls_flag->linear)
1511:     {
1512:       /* separate for linear and nonlinear, because funct() is
1513:          not defined for linear models */
1514:       for (i = 0; i < N * obs_dim; i++)
1515:       {
1516:         /* get calculated data points dependent on current
1517:            parameters */
1518:         datac[i] = 0.0;
1519:         for (j = 0; j < M; j++)
1520:         {
1521:           datac[i] += a[j] * jacob[i][j];
1522:         }
1523:         deviate[i] = obs[i] - datac[i];
1524:         deviates_abs[i] = fabs( deviate[i]);
1525:         /* weighted and squared differences */
1526:         deltasq[i] = deviate[i] * deviate[i];
1527:         if (weights[i] > 0.)
1528:         {
1529:           /* exclude outliers, i.e. weigths == 0 */
1530:           chisq += weights[i] * deltasq[i];
1531:           energy += deltasq[i];
1532:           mean += deviates_abs[i];
1533:           cnt++;
```

```
1534:         }
1535:       }
1536:     }
1537:     else /* if not linear */
1538:     {
1539:       for (i = 0; i < N * obs_dim; i++)
1540:       {
1541:         /* get calculated data points dependent on current
1542:            parameters */
1543:         datac[i] = funct( i, cond, a);
1544:
1545:         deviate[i] = obs[i] - datac[i];
1546:         deviates_abs[i] = fabs( deviate[i]);
1547:         /* weighted and squared differences */
1548:         deltasq[i] = deviate[i] * deviate[i];
1549:         if (weights[i] > 0.)
1550:         {
1551:           /* exclude outliers in final iteration*/
1552:           chisq += weights[i] * deltasq[i];
1553:           energy += deltasq[i];
1554:           mean += deviates_abs[i];
1555:           cnt++;
1556:         }
1557:       }
1558:     }
1559:     num_outliers = N * obs_dim - cnt;
1560:
1561:     /* estimate of k,  w_i= k/sigma^2_i */
1562:     gfit = chisq / (double)( cnt - M);  /* goodness of fit  */
1563:     mean = mean / (double)( cnt);
1564:     variance = energy / (double)cnt - mean * mean;
1565:
1566:     fprintf( out, "\n#\n# %s\n# Parameters:  ", rtn);
1567:     for (i = 0; i < M; i++)
1568:     {
1569:       fprintf( out, "a%d=%.6f, ", i+1, a[i]);
1570:     }
1571:
1572:     fprintf( out, "\n#\n# | chisq: %f", chisq);
1573:     fprintf( out, "\n# | mean of |deviates|: %f", mean);
1574:     fprintf( out, "\n# | variance of |deviates|: %f", variance);
1575:     fprintf( out, "\n# | goodnes of fit: %f", gfit);
1576:     fprintf( out, "\n# | number of outliers: %d", num_outliers);
1577:
1578:
1579:     /* estimate weights, but not in last iteration */
1580:     if (!iter_stop && weight_mode)
1581:     {
1582:       fprintf( out, "\n#\n# enter weight estimation");
1583:       /* estimation of weights based on absolute deviates */
1584:       if (weight_mode == 1)
1585:       {
1586:         est_weights1( N * obs_dim, deviates_abs, weights, out);
1587:       }
1588:       /* no iterative weighting in weight_mode == 2 */
1589:
1590:       fprintf( out, "\n#\n# %s\n#  i       observ         ", rtn);
1591:       fprintf( out, "calc      deviates    weights");
1592:
1593:       /* get mean of weights and output current values */
1594:       mean_weights = 0;
1595:       cnt = 0;
1596:       for (i = 0; i < N * obs_dim; i++)
1597:       {
1598:         mean_weights += weights[i]; /* compute sum of all weights */
1599:         if (i < MAX_LINES) /* limits the number of output lines */
1600:         {
1601:           fprintf( out, "\n# %2d  %12.5f %12.5f %12.5f %14f", i,
1602:             obs[i], datac[i], deviates_abs[i], weights[i]);
1603:         }
1604:         if (weights[i] > 0.0)
1605:         {
1606:           cnt++;  /* count used observations  */
1607:         }
1608:       }
1609:       /* mean of all weights > 0.; it is used later on */
1610:       mean_weights /= (double)cnt;
1611:
1612:       /* compare new and old weights */
1613:       sum = 0;
1614:       for (i = 0; i < N * obs_dim; i++)
1615:       {
1616:         /* watch changes of weights */
1617:         sum += fabs( weights[i] - weights_old[i]);
1618:         weights_old[i] = weights[i];  /* remember for next
1619:                                         iteration */
1620:       }
1621:       sum /= (double)N *obs_dim;  /* mean difference in
1622:                                      weights */
1623:       /* criterion of convergence */
1624:       if (sum < 0.0001)
1625:       {
1626:         if (!iter_stop)
1627:         {
1628:           iter_final = 1; /* go to last iteration */
1629:         }
1630:         iter_stop = 1; /* last iteration has been performed */
1631:
1632:         fprintf( out, "\n#\n# convergence of weights");
1633:       } /* if (sum < 0.0001)*/
1634:
1635:       if (iter == iter_wmax && !iter_stop)
1636:       {
1637:         fprintf( out,
1638:               "\n#\n# maximum number of iterations reached");
1639:         fprintf( out, "\n# no convergence of weights");
1640:         iter_final = 1; /* go to last iteration */
1641:       } /* if (iter == iter_wmax && !iter_stop)*/
1642:     }  /* if (!iter_stop && weight_mode) */
1643:
1644: #ifdef OUTPUT_DEVIATES
1645:     /* write deviates in separate file */
1646:     if (iter_stop == 1)
1647:     {
1648:           char dev_name[500];
1649:           int i, len;
1650:           FILE *out_dev;
1651:
1652:           len = strlen(outname);
1653:           /* copy filename w/o extension */
1654:           for (i = 0; i<len; i++)
1655:           {
1656:             if ( outname[i] == '.') break;
1657:             dev_name[i] = outname[i];
1658:           }
1659:           dev_name[i] = '_';
1660:           dev_name[i+1] = 'd';
1661:           dev_name[i+2] = 'e';
1662:           dev_name[i+3] = 'v';
1663:           dev_name[i+4] = '.';
1664:           dev_name[i+5] = 'x';
1665:           dev_name[i+6] = 'y';
1666:           dev_name[i+7] = '\0';
1667:           /* open out file */
1668:           out_dev = fopen( dev_name, "wt");
1669:           if (out_dev == NULL)
1670:           {
1671:             err = errmsg( ERR_OPEN_WRITE, rtn, dev_name, 0);
1672:             goto endfunc;
1673:           }
1674:           fprintf( out_dev, "#deviates for file ");
1675:           fprintf( out_dev, "%s ", outname);
1676:           fprintf( out_dev, "\n# before outlier detection");
1677:           fprintf( out_dev, "\n# i          deviate");
1678:           if (type == COORD_TRANSF)
1679:           {
1680:             fprintf( out_dev, "X     deviateY");
1681:             for ( i = 0; i < N*obs_dim; i+=2)
1682:             {
1683:               fprintf( out_dev, "\n%4d \t %.4e\t %.4e", i,
1684:                     deviate[i], deviate[i+1]);
1685:             }
1686:           }
1687:           else
1688:           {
1689:             for ( i = 0; i < N*obs_dim; i++)
1690:             {
1691:               fprintf( out_dev, "\n%4d \t %.4e", i, deviate[i]);
1692:             }
1693:           }
1694:           fclose(out_dev);
1695:     } /* end of deviates output */
1696: #endif
1697:
1698:     /* enter outlier detection only, if it was not done before */
1699:     if (iter_stop && out_mode && (!out_detect_flag))
1700:     {
1701:         num_outlier = 0;
1702:         out_detect_flag = 1;
1703:         /* do the outlier detection */
1704:         if (out_mode == 1)
1705:         {
1706:           /* do the outlier detection via z-score */
1707:           /* we can exploit the value of gfit,
1708:            * because sigma^2 = gfit / mean(weigts)
1709:            */
1710:           num_outlier = outlier_detection1( N * obs_dim,
1711:                 sqrt( gfit/mean_weights), deviates_abs,
1712:                 weights, 0.15, out);
1713:         }
1714:         else if (out_mode == 2)
1715:         {
1716:           /* do the cluster-based outlier detection */
1717:           num_outlier = outlier_detection2( N * obs_dim,
1718:                               deviates_abs, weights, out);
1719:         }
1720:         else if (out_mode == 3)
1721:         {
1722:           /* do the MAD outlier detection */
1723:           num_outlier = outlier_detection3( N * obs_dim,
1724:                 deviates_abs, weights, 0.15, out);
1725:         }
```

```
1726:         else if (out_mode == 4)
1727:         {
1728:           /* do the RANSAC outlier detection based on
1729:            * least-squares approximation  on subsets
1730:            */
1731:           if( obs_dim==1  || 1)
1732:           {
1733:             num_outlier =
1734:             ransac( funct, funct_deriv, funct_deriv2, init, N, M,
1735:                 obs, cond, jacob, weights, a, a_flag, algo_mode,
1736:                 ls_flag, chisq_target, covar, out, deviates_abs,
1737:                 cond_dim, obs_dim);
1738:             /* do a last round of ls in order to get a correct
1739:              * covariance matrix
1740:              */
1741:             iter_final = 1;
1742:           }
1743:           else
1744:           {
1745:             fprintf( out,
1746:               "# RANSAC is not implemented for multi-variate data\n");
1747:           }
1748:         }
1749:         /* if outliers have been found, do an additional final
1750:          * least square approx.
1751:          */
1752:         if (num_outlier > 0)
1753:         {
1754:           iter_final = 1;
1755:           if (forget_flag == 1)
1756:           {
1757:             fprintf( out, "\n#forget weights");
1758:             /* set  all weights to 1. */
1759:             for (i = 0; i < N * obs_dim; i++)
1760:             {
1761:                 if (weights[i] > 0.0)
1762:                 weights[i] = (float)1.0;
1763:             }
1764:             mean_weights = 1.0;
1765:           }
1766:         }
1767:     } /* if out_mode */
1768:   }    /* for iter */
1769:   /* ------------------------------------------------------ */
1770:
1771:   /*
1772:    * evaluation of results
1773:    */
1774:
1775:   /*
1776:    * since the weights are already normalised to their
1777:    * mean value (w = k/sigma^2), gfit is also equal to
1778:    * the variance of observations
1779:    */
1780:
1781:   /* uncertainty in observations */
1782:   uncertainty = sqrt( gfit/ mean_weights);
1783:
1784:   fprintf( out, "\n#\n# evaluation of results");
1785:   fprintf( out, "\n#  number of outliers: %d", num_outliers);
1786:   fprintf( out, "\n#  parameters:");
1787:   for (j = 0; j < M; j++)
1788:   {
1789:     fprintf( out, " a%d=%f", j+1, a[j]);
1790:   }
1791:   fprintf( out, "\n#  chi square.................: %.12G", chisq);
1792:   fprintf( out, "\n#  goodness of fit............: %.12G", gfit);
1793:   fprintf( out, "\n#  uncertainty in observations: %.12G",
1794:     uncertainty);
1795:   fprintf( out, "\n#\n#  (co)variance of parameters:");
1796:   {
1797:     int flag = 0;
1798:     for (i = 0; i < M; i++)
1799:     {
1800:       fprintf( out, "\n#");
1801:       for (j = 0; j < M; j++)
1802:       {
1803:         fprintf( out, " %15.9G", covar[i][j]);
1804:         if ( (i == j) && (covar[i][j] < 0.) ) flag = 1;
1805:       }
1806:     }
1807:     if (flag)
1808:     {
1809:       printf( "\n#  negative parameter variance");
1810:       printf( "\n#  probably ill-conditioned problem");
1811:       if (!ls_flag->svd)
1812:       {
1813:         printf( "\n#  solution is probably wrong");
1814:         printf( "\n#  disable option '-s'");
1815:         fprintf( out, "\n####  solution is probably wrong       #");
1816:         fprintf( out, "\n####  disable option '-s'");
1817:       }
1818:       fprintf( out, "\n####  negative parameter variance      #");
1819:       fprintf( out, "\n####  probably ill-conditioned problem #");
1820:     }
1821:   }
1822:   fprintf( out, "\n\n#  (co)variance of parameters multiplied");
1823:   fprintf( out, " with goodness of fit");
1824:   for (i = 0; i < M; i++)
1825:   {
1826:     fprintf( out, "\n#");
1827:     for (j = 0; j < M; j++)
1828:     {
1829:       covar[i][j] *= gfit;
1830:       fprintf( out, " %12.6G", covar[i][j]);
1831:     }
1832:   }
1833:   fprintf( out, "\n\n#  resulting uncertainty of parameters \n#");
1834:   for (j = 0; j < M; j++)
1835:   {
1836:     if (covar[j][j] >= 0)
1837:       fprintf( out, " %15.9G", sqrt( covar[j][j] ));
1838:     else
1839:       fprintf( out, "     ??      ");
1840:   }
1841:
1842:
1843:   /*-------------------------------------------------------
1844:    * post-processing
1845:    */
1846:
1847:   /* correction of parameters, before determination of relative
1848:    * uncertainty, because of phase shift
1849:    */
1850:   if (type == COSINE) /* nonlinear cosine model */
1851:   {
1852:     if (a[1] < 0 ) /* avoid negative amplitude/radius */
1853:     {
1854:         a[1] = -a[1];
1855:         a[2] = a[2] - 180; /* phase shift of 180 degrees */
1856:     }
1857:     fprintf( out, "\n#\n# corrected Parameters\n# ");
1858:     for (j = 0; j < M; j++)
1859:     {
1860:       fprintf( out, "a%d=%16.12G, ", j + 1, a[j]);
1861:     }
1862:     fprintf( out, "\n#");
1863:   }
1864:   else if (type == TRIGONOMETRIC2)
1865:   {
1866:     if (a[3] >  2*M_PI) a[3] -= 2*M_PI;
1867:     else if  (a[3] < 0) a[3] += 2*M_PI;
1868:     if (a[5] >  2*M_PI) a[6] -= 2*M_PI;
1869:     else if  (a[5] < 0) a[6] += 2*M_PI;
1870:     fprintf( out, "\n#\n# corrected Parameters\n# ");
1871:     for (j = 0; j < M; j++)
1872:     {
1873:       fprintf( out, "a%d=%16.12G, ", j + 1, a[j]);
1874:     }
1875:     fprintf( out, "\n#");
1876:   }
1877:
1878:   /* check the uncertainty in parameters */
1879:   {
1880:     int flag = 0;
1881:     double val;
1882:     fprintf( out, "\n#");
1883:
1884:     for (j = 0; j < M; j++)
1885:     {
1886:       if (covar[j][j] >= 0)
1887:       {
1888:         val = sqrt( covar[j][j]) * 100. / fabs(a[j]);
1889:         fprintf( out, "%15.5G%%", val);
1890:         if (val > 10) flag++;
1891:       }
1892:       else
1893:       {
1894:         fprintf( out, "    ??      %%");
1895:       }
1896:     }
1897:
1898:     if (flag == 1)
1899:     {
1900:       fprintf( out,
1901:         "\n# One parameter has relative high uncertainty !");
1902:       fprintf( stdout,
1903:         "\n# One parameter has relative high uncertainty !");
1904:       fprintf( stdout, "\n# Please inspect file %s !", outname);
1905:     }
1906:     else if (flag > 1)
1907:     {
1908:       fprintf( out,
1909:         "\n# %d parameters have relative high uncertainty !",
1910:         flag);
1911:       fprintf( stdout,
1912:         "\n# %d parameters have relative high uncertainty !",
1913:         flag);
1914:       fprintf( stdout, "\n# Please inspect file %s !", outname);
1915:     }
1916:   }
1917:
```

```
1918:   if (num_outliers)
1919:   {
1920:     fprintf( stdout, "\n# detection of %d outliers!", num_outliers);
1921:   }
1922:
1923:   /* unscale the parameters */
1924:   if (scale_flag)
1925:   {
1926:     /* correction of parameters */
1927:     fprintf( out, "\n# undo the scaling");
1928:     switch (type)
1929:     {
1930:       case LINEAR:
1931:         for ( j = 1; j < M; j++)
1932:         {
1933:           a[j] *= scale_fac;
1934:         }
1935:         break;
1936:       case POLYNOMIAL:
1937:         for ( j = 1; j < M; j++)
1938:         {
1939:           a[j] *= pow( scale_fac, (double)j);
1940:         }
1941:         break;
1942:       case NIST_RAT42:
1943:         a[2] *= scale_fac;
1944:         break;
1945:       case NIST_ECKERLE4:
1946:         a[0] /= scale_fac;
1947:         a[1] /= scale_fac;
1948:         a[2] /= scale_fac;
1949:         break;
1950:       case NIST_MGH10:
1951:         a[1] /= scale_fac;
1952:         a[2] /= scale_fac;
1953:         break;
1954:       default:
1955:         fprintf( out, "\n# scaling not supported for '-m %d'", type);
1956:         fprintf( stderr,
1957:           "\n#### scaling not supported for '-m %d'###\n", type);
1958:     }
1959:     /* unscale conditions for output  */
1960:     for (i = 0; i < N*cond_dim; i++)
1961:     {
1962:       cond[i] /= scale_fac;
1963:     }
1964:   }
1965:
1966:   fprintf( out, "\n#\n# Final_Parameters ");
1967:   for (i = 0; i < M; i++)
1968:   {
1969:     fprintf( out, "a%d= %.12G ", i+1, a[i]);
1970:   }
1971:   if (type == POLYNOMIAL)
1972:   {
1973:     fprintf( out, "\n# f%d(x) = %.14G", M, a[0]);
1974:     if (M > 1)
1975:     {
1976:       fprintf( out, " + %.14G*x", a[1]);
1977:     }
1978:     for ( j = 2; j < M; j++)
1979:     {
1980:         fprintf( out, " + %.14G * x**%d",  a[j], j);
1981:     }
1982:   }
1983:
1984:   /* map parameters to original model function */
1985:   if (type == COSINE_LIN)
1986:   {
1987:     double r0, phi0;
1988:
1989:     /* f(x|b) = b1 + r0 * cos( x - phi0)
1990:      * f(x|a) = a1 + a2 * cos( x) + a3 * sin( x)
1991:      */
1992:     r0 = sqrt( a[1] * a[1] + a[2] * a[2]);  /* radius a2 */
1993:     phi0 = 0;
1994:     if (r0 > 0.)
1995:     {
1996:       double pc1, pc2, ps1, ps2;
1997:       double d11, d12, d21, d22;
1998:
1999:       /* solve ambiguity of angles */
2000:       pc1 = acos( a[1] / r0); /* 1st solution */
2001:       pc2 = -pc1; /* 2nd solution */
2002:       ps1 = asin( a[2] / r0); /* 3rd solution */
2003:       if (ps1 < 0)  /* 4th solution */
2004:         ps2 = -M_PI - ps1;
2005:       else
2006:         ps2 = M_PI - ps1;
2007:
2008:       d11 = fabs( pc1 - ps1); /* two of four must be equal */
2009:       d12 = fabs( pc1 - ps2); /* take differences */
2010:       d21 = fabs( pc2 - ps1);
2011:       d22 = fabs( pc2 - ps2);
2012:       /* look for smallest difference */
2013:       if (d11 < d12 && d11 < d21 && d11 < d22)
2014:       {
2015:         phi0 = 0.5 * (pc1 + ps1);
2016:       }
2017:       if (d12 < d11 && d12 < d21 && d12 < d22)
2018:       {
2019:         phi0 = 0.5 * (pc1 + ps2);
2020:       }
2021:       if (d21 < d11 && d21 < d12 && d21 < d22)
2022:       {
2023:         phi0 = 0.5 * (pc2 + ps1);
2024:       }
2025:       if (d22 < d11 && d22 < d12 && d22 < d21)
2026:       {
2027:         phi0 = 0.5 * (pc2 + ps2);
2028:       }
2029:       a[1] = r0;
2030:       a[2] = phi0 * 180 / M_PI;
2031:     }
2032:     fprintf( out, "\n#\n# corrected Parameters ");
2033:     fprintf( out, "according to f(x)=b1+b2*cos(x-b3)\n# ");
2034:     for (j = 0; j < M; j++)
2035:     {
2036:       fprintf( out, "b%d=%.9f, ", j + 1, a[j]);
2037:     }
2038:     fprintf( out, "\n#");
2039:   }
2040:   else if (type == EXPONENTIAL2_LIN)
2041:   {
2042:     /* convert observations back */
2043:     for (i = 0; i < N; i++)
2044:     {
2045:       obs[i] = exp( obs[i]);
2046:     }
2047:     a[0] = exp( a[0]);
2048:
2049:     fprintf( out, "\n#\n# corrected Parameters\n# ");
2050:     for (j = 0; j < M; j++)
2051:     {
2052:       fprintf( out, "a%d=%.9f, ", j + 1, a[j]);
2053:     }
2054:     fprintf( out, "\n#\n# uncertainties of corrected Parameters ");
2055:     fprintf( out, "are not available yet\n# ");
2056:
2057:     for (i = 0; i < N; i++)
2058:     {
2059:       /* get calculated data points dependent on corrected
2060:          parameters */
2061:       datac[i] = fexpon2( i, cond, a);
2062:     }
2063:   }
2064:   else if (type == COORD_TRANSF)
2065:   {
2066:     /* print angle in degrees */
2067:
2068:     fprintf( out, "  (%.4f degrees)", a[2] * 180. / M_PI);
2069:   }
2070:   else if (type == CIRCLE || type == CIRCLE_TLS)
2071:   {
2072:     /* evaluate result in terms of mean squared distance
2073:      * of points to circle
2074:      */
2075:     double eval = 0, d, delta;
2076:
2077:     cnt = 0;
2078:     for ( i = 0; i < N; i++)
2079:     {
2080:       if (weights[i] > 0.)
2081:       {
2082:         d = euclid_dist( (cond[2*i]-a[0]), (cond[2*i+1]-a[1]));
2083:         /* orthogonal distance to curve of circle */
2084:         delta = d - a[2];
2085:         eval += delta * delta; /* sum up squared distances */
2086:         cnt++;
2087:       }
2088:     }
2089:     fprintf( out, "\n# mean squared distance to circle:  %.6f\n#",
2090:           eval/cnt);
2091:   }
2092:   else if (type == CIRCLE_LIN)
2093:   {
2094:     double a1, a2, a3;
2095:     double eval = 0, d, delta;
2096:     /* convert parameters back  */
2097:     a1 = 0.5 * a[0];
2098:     a2 = 0.5 * a[1];
2099:     a3 = sqrt( a1*a1 + a2*a2 - a[2]);
2100:
2101:     fprintf( out, "\n#\n# corrected Parameters\n# ");
2102:     fprintf( out, "a1=%.9f, ", a1);
2103:     fprintf( out, "a2=%.9f, ", a2);
2104:     fprintf( out, "a3=%.9f", a3);
2105:     fprintf( out, "\n#");
2106:
2107:     /* evaluate result in terms of mean squared distance
2108:      * of points to circle
2109:      */
```

```
2110:     cnt = 0;
2111:     for ( i = 0; i < N; i++)
2112:     {
2113:       if (weights[i] > 0.)
2114:       {
2115:         d = euclid_dist( (cond[2*i]-a1), (cond[2*i+1]-a2));
2116:         /* orthogonal distance to curve of circle */
2117:         delta = d - a3;
2118:         eval += delta * delta; /* sum up squared distances */
2119:         cnt++;
2120:       }
2121:     }
2122:     fprintf( out, "\n# mean squared distance to circle:  %.6f\n#",
2123:           eval/cnt);
2124:   }
2125:
2126:   /* ---------------------------------------- */
2127:   fprintf( out, "\n#  ");
2128:   for (j = 0; j < cond_dim; j++)
2129:   {
2130:     fprintf( out, "cond%d        ", j + 1);
2131:   }
2132:   fprintf( out, "observed     fitted   weight uncertainty ");
2133:   fprintf( out, "glob.uncertainty difference");
2134:
2135:   /*
2136:    * estimated weight should be w = 1/sigma^2
2137:    * thus estimated uncertainty is sigma = 1/ sqrt(weight)
2138:    * make nice output
2139:    */
2140:   if (type == COORD_TRANSF)
2141:   {
2142:     fprintf( out, "  observed2     fitted2   weight2 ");
2143:     fprintf( out, "uncertainty2 glob.uncertainty2 difference2");
2144:   }
2145:
2146:   /*
2147:    * output of final results
2148:    */
2149:   for (i = 0; i < (int)N * obs_dim; i += obs_dim)
2150:   {
2151:     double uncert;
2152:
2153:     fprintf( out, "\n");
2154:     for (j = 0; j < cond_dim; j++)
2155:     {
2156:       fprintf( out, "%12.6f ",
2157:         cond[cond_dim * (i / obs_dim) + j]);
2158:     }
2159:     /* put both fitting results for fixed conditions in same row
2160:      */
2161:     for (o = 0; o < (int)obs_dim; o++)
2162:     {
2163:       int k;
2164:       k = i + o;
2165:       if (weights[k] > 0)
2166:         uncert = 1. / sqrt( weights[k]);
2167:       else
2168:         uncert = 9999.;
2169:       fprintf( out, "%12.6f ", obs[k]);
2170:       fprintf( out, "%12.6f %11.6f %12.6f  %.6f",
2171:         datac[k], weights[k], uncert, uncertainty);
2172:       fprintf( out, "%+12.6f ", (obs[k] - datac[k]));
2173:     }
2174:     /* make a empty line to enforce plotting a mesh */
2175:     if (type == LINEAR && M == 3 &&
2176:       ( cond[cond_dim * i] != cond[cond_dim * (i + 1)]))
2177:     {
2178:       /* plane approximation */
2179:       fprintf( out, "\n");
2180:     }
2181:     if (type == QUAD_SURFACE  &&
2182:       ( cond[cond_dim * i+1] != cond[cond_dim * (i + 1)+1]))
2183:     {
2184:       /* surface approximation */
2185:       fprintf( out, "\n");
2186:     }
2187:   }
2188:   fprintf( out, "\n");
2189:
2190: endfunc:
2191:   printf( "\n");
2192:   fflush( stdout);
2193:
2194:   free_vector( &weights);
2195:   free_vector( &weights_old);
2196:   free_vector( &obs);
2197:   free_vector( &datac);
2198:   free_vector( &cond);
2199:   free_vector( &deviate);
2200:   free_vector( &deviates_abs);
2201:   free_vector( &deltasq);
2202:   free_matrix( &jacob);
2203:   free_matrix( &covar);
2204:
2205:   if (argc > 2)
2206:   {
2207:     if (out != NULL)
2208:       fclose( out);
2209:   }
2210:   if (in != NULL)
2211:     fclose( in);
2212:
2213:   if (err)
2214:   {
2215:     fprintf( stderr, "\n failed.\n");
2216:     return err;
2217:   }
2218:   else
2219:   {
2220:     fprintf( stderr, "\n ready.\n");
2221:     return 0;
2222:   }
2223: }
2224:
2225: /*-------------------------------------------------------------
2226:  *  is_data_line()
2227:  *-------------------------------------------------------------*/
2228: int
2229: is_data_line( char *line, int N)
2230: {
2231:   int i;
2232:
2233:   /* scan leading white spaces  */
2234:   for (i = 0; i < N; i++)
2235:   {
2236:       if (line[i] != ' '      /* space  */
2237:           && line[i] != '\t'  /* tab    */
2238:           )
2239:           break;
2240:   }
2241:   /* check next character */
2242:   if (line[i] == '#')   /* comment */
2243:   {
2244:     return 0;
2245:   }
2246:   if (line[i] == '\n')  /* newline */
2247:   {
2248:     return 0;
2249:   }
2250:   if (line[i] == '\r')  /* carrige return (Windows)  */
2251:   {
2252:     return 0;
2253:   }
2254:
2255:   return 1; /* is data line */
2256: }
2257:
2258: /*-------------------------------------------------------------
2259:  *  get_nth_field()
2260:  * scans a string up to the desired column (field)
2261:  *-------------------------------------------------------------*/
2262: char *
2263: get_nth_field( char *line, int n)
2264: {
2265:   char *ptr = NULL;
2266:   int ch, i = 0, loop_flag, cnt, field_flag;
2267:
2268:   if (line == NULL)
2269:     return ptr;
2270:
2271:   loop_flag = 1;
2272:   field_flag = 0;
2273:   cnt = 0;
2274:   do
2275:   {
2276:     ch = line[i];
2277:     if (ch == '\0')
2278:     {
2279:       loop_flag = 0;
2280:       break;
2281:     }
2282:     if (!field_flag)
2283:     {
2284:       if (ch != ' ' && ch != '\t' && ch != '\r' && ch != '\n')
2285:       {
2286:         field_flag = 1;
2287:         cnt++;
2288:         if (cnt == n)
2289:         {
2290:           /* desired field is found */
2291:           ptr = &( line[i]);
2292:           loop_flag = 0;
2293:         }
2294:       }
2295:     }
2296:     else
2297:     {
2298:       /* search next white space */
2299:       if (ch == ' ' || ch == '\t' || ch == '\r' || ch == '\n')
2300:       {
2301:         field_flag = 0;
```

```
2302:       }
2303:     }
2304:     i++;
2305:   } while (loop_flag);
2306:   return ptr;
2307: }
```

```
 0: /*****************************************************************
 1:  *
 2:  * File........:  prototypes.h
 3:  * Function....:  proto typing for different functions
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  27.01.2010, 30.3.2011
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *****************************************************************/
14: #ifndef PROTO_H
15: #define PROTO_H
16:
17: typedef struct {
18:   int linear;     /* linear model */
19:   int svd;        /* special computation for linear models */
20:   int LM;         /* 0 .. Gaus-Newton, 1 .. Levenberg-Marquardt */
21:   int chisq_target; /* indicates that 'chisq_target' was set  */
22:   int trueH;        /* use true Hessian matrix  */
23: } LS_FLAG;
24:
25: /* least squares routine */
26: int
27: ls( double (*funct) (int,double*,double*),
28:     double (*funct_deriv) (double(*)(int,double*,double*),
29:           int,int,int,double*,double*),
30:     double (*funct_deriv2) (double(*)(int,double*,double*),
31:     int,int,int,int,double*,double*),
32:     int (*init)(int, double*,double*,double*,
33:             unsigned char*,FILE*),
34:     int N, int M, double *obs, double *cond, double **jacob,
35:     double *weights, double *a, unsigned char* a_flag,
36:     int algo_mode, LS_FLAG *ls_flag,
37:     double chisq_target, double **covar, FILE *out);
38:
39: int svd_inversion( int N, double **normal, double **normal_i,
40:                   FILE *out);
41: int IsFiniteNumber(double x);
42: int
43: solve_lin( int N, int M, double *obs, double *weights,
44:               double **jacob, double **covar, double *a,
45:               FILE *out);
46:
47: /* parsing of command-line parameters */
48: char* get_nth_field( char *line, int n);
49: int is_data_line( char *line, int N);
50:
51: /* matrix inversion */
52: int singvaldec( double **a, int N, int M, double w[], double **v);
53: void  backsub_LU( double **lu, int N, int *indx, double back[]);
54: int decomp_LU( double **normal, int M, int *indx, int *s);
55: void heap_sort_d_(unsigned long N, double ra[], int idx[]);
56: void heap_sort_d(unsigned long N, double ra[]);
57:
58: /* estimation of weights */
59: void est_weights1( int N, double *deltasq,
60:           double *weights, FILE *out);
61: void est_weights2( int N, double *cond, double *obs,
62:               double *weights, int obs_per_bin, FILE *out);
63: int outlier_detection1( int N, double sigma_y, double *deltasq,
64:           double *weights, double nu, FILE *out);
65: int outlier_detection2( int N, double *deltasq,
66:           double *weights, FILE *out);
67: int outlier_detection3( int N, double *deltasq,
68:           double *weights, double nu, FILE *out);
69:
70: int
71: ransac( double (*funct) (int,double*,double*),
72:    double (*funct_deriv) (double(*)(int,double*,double*),
73:    int,int,int,double*,double*),
74:    double (*funct_deriv2) (double(*)(int,double*,double*),
75:    int,int,int,int,double*,double*),
76:    int (*init)(int, double*,double*,double*,
77:    unsigned char*,FILE*),
78:    int N, int M, double *obs, double *cond, double **jacob,
79:    double *weights, double *a, unsigned char* a_flag,
80:    int algo_mode, LS_FLAG *ls_flag,
81:    double chisq_target, double **covar, FILE *out,
82:    double *deviates_abs,
83:    int cond_dim,
84:    int obs_dim);
85:
86:    void
87: ls_straightline(
88:   int N, /* number of entries */
89:   double cond[], /* vector of conditions */
90:   double obs[],   /* vector of observations */
91:   double a[]      /* container for parameters to be estimated */
92:   );
93:
94: #endif
```

```
/*****************************************************************
 *
 * File........:  ls.c
 * Function....:  least squares with alternative matrix inversion
 * Author......:  Tilo Strutz
 * last changes:  05.02.2008, 28.09.2009, 25.01.2010
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
*****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include <float.h>
#include "errmsg.h"
#include "matrix_utils.h"
#include "defines.h"
#include "macros.h"
#include "prototypes.h"
#include "functions.h"
#ifndef WIN32
#include <sys/time.h>
#else
#include <time.h>
#define random rand
#endif

long ITERAT_MAX;
double NU_FAC;


/*---------------------------------------------------------------
* ls()
*
*--------------------------------------------------------------*/
int
ls( double (*funct) (int,double*,double*),
    double (*funct_deriv) (double(*)(int,double*,double*),
    int,int,int,double*,double*),
    double (*funct_deriv2) (double(*)(int,double*,double*),
    int,int,int,int,double*,double*),
    int (*init)(int, double*,double*,double*,
    unsigned char*,FILE*),
    int N, int M, double *obs, double *cond, double **jacob,
    double *weights, double *a, unsigned char* a_flag,
    int algo_mode, LS_FLAG *ls_flag,
    double chisq_target, double **covar, FILE *out)
{
  char *rtn = "ls";
  int err = 0, i, j, k, n;
  int Nfree;
  int stop_flag;  /* supports convergence criterion   */
  int iter_cnt;   /* counter for nonlinear iterations */
  int iter_max=ITERAT_MAX;  /* maximum number of iterations */
  double **cofac = NULL;  /* cofactor matrix for matrix
                             inversion */
  double **normal = NULL;     /* N = J^(T) * W * J  */
  double **normal_i = NULL;   /* inverse of N       */
  double **tmpmat = NULL;     /* temporary matrix   */
  double *tmpvec = NULL;      /* J^(T) * W * r      */
  double *datac = NULL; /* calculated values based on parameters
                           */
  double *da = NULL;    /* parameter update deltasq a */
  double *deltasq = NULL; /* = w * [obs - f(x|a) ] ^2 */
  double chisq, det, tmp, residual, deriv_2nd, variance;
  double min_chisq=0, *min_a=NULL; /* remember best result */
  double mu_fac = 0;    /* factor for Levenberg-Marquardt */
  double max_diag;  /* maximum value of Njj */
  double diag[M_MAX]; /* vector of diagonal of Normal matrix  */

  fprintf( out,
    "\n#  -- %s - start  ------------------------------", rtn);

  /*
  * allocate memory
  */

  /* normal matrix N = J^(T) * W * J, its inverse */
  normal = matrix( M, M);
  normal_i = matrix( M, M);

  cofac  = matrix( M, M); /* cofactor matrix */
  tmpvec = vector( M);    /* container for J^(T) * W * G  */
  datac  = vector( N);    /* calculated data using f(x|a) */
  deltasq = vector( N);   /* remaining differences  */
  da = vector( M);        /* model parameter update */
  min_a = vector( M);     /* remember best parameter set */

  iter_cnt = 1;
  if (!ls_flag->linear)
  {
    /* initialise minimum (best) values */
    for (j = 0; j < M; j++)
    {
      min_a[j] = a[j];
    }
    min_chisq = -1.;
  }
  else
  {
    /* nothing to iterate for linear models */
  }

  if (ls_flag->svd && ls_flag->linear)
  {
    /* This function exploits the SVD for solving linear problems.
    * Inverting of the normal matrix is not required, which
    * sometimes runs into problems when inverting the normal matrix
    * of difficult model functions as, for example, polynomials of
    * high order. It returns the estimated parameters in a and the
    * covariance matrix covar
    */
    err = solve_lin( N, M, obs, weights, jacob, covar, a, out);
    if (err)
    {
      fprintf( stderr,
        "\n\n### Unable to solve this linear system!");
      fprintf( stderr, "\n Abort!\n");
      goto endfunc;
    }
  }
  else
  {
    mu_fac = 0.0001; /* initial factor for Levenberg-Marquard */

    /* iteration if nonlinear */
    do
    {
      /* feedback on console */
      printf( "\r\t\t %4d", iter_cnt);


      /*
       * calculate normal matrix N
       * N = J^(T) * W * J
       */
       if (errno)
       {
         perror( "\n### ");
         fprintf( stderr, "     errno = %d", errno);
         fprintf( out, "\n Error in computation (%d), ", errno);
         fprintf( out, "see standard output (console)\n");
         err = errno;
         errno = 0;
         goto endfunc;
       }
      max_diag = 0;
      for (j = 0; j < M; j++)
      {
        for (i = 0; i < M; i++)
        {
          normal[j][i] = 0.;
          for (n = 0; n < N; n++)
          {
            normal[j][i] += jacob[n][j] * jacob[n][i] * weights[n];
          }
          /* overflow test  */
          if (!IsFiniteNumber( normal[j][j]))
          {
            err = errmsg( ERR_IS_INFINITE, rtn, "normal", 0);
            goto endfunc;
          }
        }
        /* only for nonlinear models of importance:
         * get maximum value on main diagonal
         */
        diag[j] = normal[j][j];
        if (max_diag < normal[j][j])
        {
          max_diag = normal[j][j];
        }
        if (errno)
        {
          perror( "\n### ");
          fprintf( stderr, "     errno = %d", errno);
          fprintf( out, "\n Error in computation (%d), ", errno);
          fprintf( out, "see standard output (console)\n");
          err = errno;
          errno = 0;
          goto endfunc;
        }
      }

      /* K = J^(T) * W * r */
      if (ls_flag->linear)
      {
        /* tmpvec = J^(T) * W * y */
```

```
192:         for (j = 0; j < M; j++)
193:         {
194:           tmpvec[j] = 0.;
195:           for (n = 0; n < N; n++)
196:           {
197:             tmpvec[j] += jacob[n][j] * obs[n] * weights[n];
198:           }
199:         }
200:       }
201:       else  /* nonlinear */
202:       {
203:         if (errno)
204:         {
205:           perror( "\n### ");
206:           fprintf( stderr, "     errno = %d", errno);
207:           fprintf( out, "\n Error in computation (%d), ", errno);
208:           fprintf( out, "see standard output (console)\n");
209:           err = errno;
210:           errno = 0;
211:           goto endfunc;
212:         }
213:         /* tmpvec = J^(T) * W * r */
214:         /* r contains residuals */
215:         for (j = 0; j < M; j++)
216:         {
217:           tmpvec[j] = 0.;
218:           for (i = 0; i < N; i++)
219:           {
220:             residual = obs[i] - funct( i, cond, a);
221:             tmpvec[j] += jacob[i][j] * residual * weights[i];
222:           }
223:         }
224:         if (errno)
225:         {
226:           perror( "\n### ");
227:           fprintf( stderr, "     errno = %d", errno);
228:           fprintf( out, "\n Error in computation (%d), ", errno);
229:           fprintf( out, "see standard output (console)\n");
230:           err = errno;
231:           errno = 0;
232:           goto endfunc;
233:         }
234:
235:         /* add Levenberg-Marquardt term on main diagonal */
236:         if (ls_flag->LM)
237:         {
238:           for (j = 0; j < M; j++)
239:           {
240:             normal[j][j] += mu_fac * max_diag;
241:           }
242:         }
243:         /* add Q term ==> true Hessian matrix */
244:         if (ls_flag->trueH)
245:         {
246:           for (i = 0; i < N; i++)
247:           {
248:             residual = obs[i] - funct( i, cond, a);
249:             for (j = 0; j < M; j++)
250:             {
251:               for (k = 0; k < M; k++)
252:               {
253:                 deriv_2nd = funct_deriv2( funct, j, k, M, i,cond,a);
254:                 normal[j][k] += deriv_2nd * residual;
255:               }
256:             }
257:           }
258:         }
259:       }
260:
261:       /*
262:        * inversion of normal matrix
263:        * (cofactor method, LU decomposition, or SVD)
264:        *
265:        */
266:       if (algo_mode == 0)
267:       {
268:         switch (M)
269:         {
270:         case 1:
271:           /* y = a1 */
272:           /* df/da1 = 1 */
273:           det = normal[0][0]; /* determinant */
274:           if (det != 0)
275:           {
276:             cofac[0][0] = 1.;
277:           }
278:           else
279:           {
280:             err = errmsg( ERR_IS_ZERO, rtn, "determinant", 0);
281:             goto endfunc;
282:           }
283:           break;
284:
285:         case 2:
286:           det = determinant_2x2( normal); /* determinant */
287:           coFactor_2x2( normal, cofac);   /* coFactor matrix */
288:           break;
289:
290:         case 3:
291:           det = determinant_3x3( normal); /* determinant */
292:           coFactor_3x3( normal, cofac); /* coFactor matrix */
293:           break;
294:         case 4: /* need 4 Parameters */
295:           det = inverse_4x4( normal, cofac);
296:           break;
297:         case 5: /* need 5 Parameters */
298:           det = inverse_5x5( normal, cofac);
299:           break;
300:
301:         default:
302:           fprintf( stderr, "\n too much  parameters (%d) ", M);
303:           fprintf( stderr, "for standard matrix inversion");
304:           err = errmsg( ERR_CALL, rtn,
305:             "check command-line parameters ", 0);
306:           goto endfunc;
307:         } /* switch */
308:         if (fabs( det) > 1.0e-20)
309:         {
310:           for (i = 0; i < M; i++)
311:           {
312:             for (j = 0; j < M; j++)
313:             {
314:               normal_i[i][j] = cofac[i][j] / det;
315:             }
316:           }
317:         }
318:         else
319:         {
320:           err = errmsg( ERR_IS_ZERO, rtn, "determinant", 0);
321:           fprintf( stderr,
322:               "\n Please, consider to use option '-a 1' ! ");
323:           fprintf( out,
324:             "\n#\n### determinant is zero ! ");
325:           fprintf( out,
326:             "\n### Please, consider to use option '-a 1' !\n#");
327:           goto endfunc;
328:         }
329:       }
330:       else if (algo_mode == 1)  /* SVD */
331:       {
332:         err = svd_inversion( M, normal, normal_i, out);
333:         if (err)
334:         {
335:           /* take best parameter */
336:           for (j = 0; j < M; j++)
337:           {
338:             a[j] = min_a[j];
339:           }
340:           break;
341:         }
342:       }  /* algo_mode SVD */
343:       else if (algo_mode == 2)  /* LU decomposition */
344:       {
345:         int *indx = NULL, s;
346:         double *column = NULL;
347:
348:         indx = ivector( M);
349:         column = vector( M);
350:
351:         /* decompose the matrix */
352:         err = decomp_LU( normal, M, indx, &s);
353:         if (err)
354:         {
355:           if (err == 6)
356:             fprintf( out, ERR_IS_ZERO_MSG,
357:             "decomp_LU", "'max_element'");
358:           free_ivector( &indx);
359:           free_vector( &column);
360:           goto endfunc;
361:         }
362:         /* find inverse by back-substitution of columns */
363:         for (j = 0; j < M; j++)
364:         {
365:           for (i = 0; i < M; i++)
366:             column[i] = 0.0;
367:           column[j] = 1.0;
368:
369:           backsub_LU( normal, M, indx, column);
370:
371:           for (i = 0; i < M; i++)
372:             normal_i[i][j] = column[i];
373:         }
374:         free_vector( &column);
375:         free_ivector( &indx);
376:       }  /* LU decomp */
377:
378:
379:       /* final matrix multiplication to get parameter updates */
380:       for (j = 0; j < M; j++)
381:       {
382:         da[j] = 0.;
383:         for (i = 0; i < M; i++)
```

```
384:         {
385:           da[j] += normal_i[j][i] * tmpvec[i];
386:         }
387:       }
388:         if (errno)
389:         {
390:           perror( "\n### ");
391:           fprintf( stderr, "     errno = %d", errno);
392:           fprintf( out, "\n Error in computation (%d), ", errno);
393:           fprintf( out, "see standard output (console)\n");
394:           err = errno;
395:           errno = 0;
396:           goto endfunc;
397:         }
398:       if (ls_flag->linear)
399:       {
400:         /* linear: parameter = update */
401:         for (j = 0; j < M; j++)
402:         {
403:           a[j] = da[j];
404:         }
405:       }
406:       else  /* nonlinear */
407:       {
408:         stop_flag = 0;
409:
410:         if (iter_cnt < 3000)
411:         {
412:           fprintf( out, "\n#\n#  Iteration of least squares: %d",
413:                    iter_cnt);
414:           fprintf( out, "\n#  Updates:  ");
415:         }
416:         for (j = 0; j < M; j++)
417:         {
418:           if (iter_cnt < 3000)
419:             fprintf( out, "da%d=%.14G, ", j+1, da[j]);
420:           /* if changes are negigible */
421:           if ( fabs(da[j]) < TOL) /* case a[j] == 0 */
422:             stop_flag++;
423:         }
424:
425:         /* adjust parameters */
426:         for (j = 0; j < M; j++)
427:         {
428:           /* adjust */
429:           a[j] += da[j];
430:         }
431:
432:         /* update Jacobian matrix for nonlinear models */
433:         for (i = 0; i < N; i++)
434:         {
435:           for (j = 0; j < M; j++)
436:           {
437:             jacob[i][j] = funct_deriv( funct, i, j, M, cond, a);
438:             if (!IsFiniteNumber( jacob[i][j]))
439:             {
440:               fprintf( stderr,
441:                 "\n#\n####  Divergence of approximation ");
442:               fprintf( out,
443:                 "\n#\n#  Divergence of approximation ");
444:               if (errno)
445:               {
446:                 perror( "\n### ");
447:                 fprintf( stderr, "     errno = %d", errno);
448:                 fprintf( out, "\terrno = %d", errno);
449:                 fprintf( out, "\tsee standard output (console)\n");
450:                 err = errno;
451:                     errno = 0;
452:                 goto endfunc;
453:               }
454:               err = 59;
455:               goto endfunc;
456:             }
457:           }
458:         }
459:       }
460:
461:       /* compute weighted and squared differences chi-squared */
462:       chisq = 0.0;
463:       Nfree = -M; /* reduce by number of parameters */
464:       if (ls_flag->linear)
465:       {
466:         for (i = 0; i < N; i++)
467:         {
468:           /* get calculated data points dependent on current
469:           parameters */
470:           datac[i] = 0.0;
471:           for (j = 0; j < M; j++)
472:           {
473:             datac[i] += a[j] * jacob[i][j];
474:           }
475:           tmp = fabs( obs[i] - datac[i]);
476:           /* weighted and squared differences */
477:           deltasq[i] = tmp * tmp;
478:           chisq += weights[i] * deltasq[i];
479:           if (weights[i] > 0.)
480:             Nfree++;
481:         }
482:       }
483:       else  /* nonlinear */
484:       {
485:         for (i = 0; i < N; i++)
486:         {
487:           /* get calculated data points dependent on current
488:           parameters */
489:           datac[i] = funct( i, cond, a);
490:
491:           tmp = fabs( obs[i] - datac[i]);
492:           /* weighted and squared differences */
493:           deltasq[i] = tmp * tmp;
494:           chisq += weights[i] * deltasq[i];
495:           if (weights[i] > 0.)
496:             Nfree++;
497:         }
498:         if (min_chisq < 0) /* initial value */
499:         {
500:           min_chisq = chisq;
501:         }
502:         else if (min_chisq > chisq)
503:         {
504:           /* new result is closer to minimum */
505:           min_chisq = chisq;
506:           /* copy parameters */
507:           for (j = 0; j < M; j++)
508:           {
509:             min_a[j] = a[j];
510:           }
511:           if (mu_fac > 2. * DBL_EPSILON)
512:             mu_fac *= 0.5; /* decrease Lev-Mar parameter */
513:           /* improvement */
514:         }
515:         else
516:         {
517:           if (ls_flag->LM)
518:           {
519:             if (iter_cnt < 3000)
520:               fprintf( out, "\n#\n#  rejection, chisq=%.14G",
521:                        chisq);
522:             if (mu_fac < 1.e+8)
523:             {
524:               /* increase Lev-Mar parameter */
525:               /* should not be a multiple of decreasing factor */
526:               /* Thurber.dat was not satisfied with 3.  */
527:               mu_fac *= 9.;
528:             }
529:             /* restore old parameters */
530:             for (j = 0; j < M; j++)
531:             {
532:               a[j] = min_a[j];
533:             }
534:           }
535:           else
536:           {
537:             if (iter_cnt < 3000)
538:               fprintf( out, "\n#\n#  uphill, chisq=%.14G", chisq);
539:           }
540:         }
541:       } /* if nonliner */
542:
543:       variance = min_chisq / (double)Nfree;
544:
545:       if (!ls_flag->linear)
546:       {
547:         if (iter_cnt < 3000)
548:         {
549:           fprintf( out, "\n#\n#  Parameters:  ");
550:           for (i = 0; i < M; i++)
551:           {
552:             fprintf( out, "a%d=%.12G, ", i+1, a[i]);
553:           }
554:
555:           fprintf( out, "\n#\n#  chisq: %.14G", min_chisq);
556:           if (ls_flag->LM) /* Lev-Marquardt method */
557:           {
558:             fprintf( out, "\t mu_fac: %e", mu_fac);
559:           }
560:         }
561:         printf( "  chisq = %.15G       ", min_chisq);
562:         /* set break condition */
563:         if (min_chisq == 0.0) stop_flag = M;
564:         fflush(stdout);
565:         if (errno)
566:         {
567:           perror( "\n### ");
568:           fprintf( stderr, "     errno = %d", errno);
569:           fprintf( out, "\n Error in computation (%d), ", errno);
570:           fprintf( out, "see standard output (console)\n");
571:           err = errno;
572:           errno = 0;
573:           goto endfunc;
574:         }
575:
```

```
576:       /* test of convergence */
577:       if (stop_flag == M) /* all adjustments are negligible */
578:       {
579:         fprintf( out, "\n#\n#  convergence after %d iterations",
580:           iter_cnt);
581:         if (ls_flag->chisq_target)
582:         {
583:           fprintf( out, "\n#  check target (%f) ", chisq_target);
584:           /* check, whether maximum target error is reached */
585:           if (chisq <= chisq_target)
586:           {
587:             /* yes, we can stop the optimisation */
588:             break;
589:           }
590:           /* avoid randomisation just before max. iteration*/
591:           else if (iter_cnt < iter_max-1)
592:           {
593:             printf( "local minimum\n");
594:             /* no, lets re-initialise the parameters  */
595:             fprintf( out, "\n#  re-initialise parameters ");
596:             /* set flags and randomise parameters */
597:             for (j = 0; j < M; j++)
598:             {
599:               float z;
600:               /* -1...+1 */
601:               z = (2.F * (float)random()/ (float)RAND_MAX - 1.F);
602:               /* a_flag[j] = 1; */
603:               a[j] += a[j] * z /2.; /* max. change 50% */
604:             }
605:             /* err = init( N, obs, cond, a, a_flag, out); */
606:
607:             /* reset minimum (best) values */
608:             for (j = 0; j < M; j++)
609:             {
610:               min_a[j] = a[j];
611:             }
612:             min_chisq = 99999999.;
613:             fprintf( out, "\n#  Parameters:  ");
614:             for (i = 0; i < M; i++)
615:             {
616:               fprintf( out, "a%d=%.8f, ", i+1, a[i]);
617:             }
618:             /* update Jacobian matrix for nonlinear models */
619:             for (i = 0; i < N; i++)
620:             {
621:               for (j = 0; j < M; j++)
622:               {
623:                 jacob[i][j] = funct_deriv( funct, i, j, M,cond,a);
624:               }
625:             }
626:             min_chisq = -1;
627:           } /* else if (iter_cnt < iter_max-1)  */
628:         } /* if (ls_flag->chisq_target) */
629:         else
630:           break;
631:       } /* if (stop_flag == M)  */
632:     } /* if (!ls_flag->linear)  */
633:
634:     iter_cnt++;
635:     if (!ls_flag->linear && iter_cnt >= iter_max)
636:     {
637:       /* take best parameter */
638:       for (j = 0; j < M; j++)
639:       {
640:         a[j] = min_a[j];
641:       }
642:       fprintf( stderr,
643:         "\n\n     no convergence after %d iterations",
644:         iter_cnt);
645:       fprintf( stderr, "\n     chisq x 1000 = %f",
646:         chisq * 1000);
647:       fprintf( out,
648:           "\n#\n#     no convergence after %d iterations",
649:           iter_cnt);
650:       fprintf( out, "\n#     chisq x 1000 = %f",
651:         chisq * 1000);
652:
653:       /* ### needs more elaboration ! #####*/
654:       if (ls_flag->chisq_target)
655:       {
656:         fprintf( out, "\n#  check target (%f) ", chisq_target);
657:         /* check, whether maximum target error is reached */
658:         if (chisq <= chisq_target)
659:         {
660:           /* yes, we can stop the optimisation */
661:           break;
662:         }
663:         /* no, lets re-initialise the parameters  */
664:         fprintf( out, "\n#  re-initialise parameters ");
665:         /* modifies only values NOT given on command line!  */
666:         init( N, obs, cond, a, a_flag, out);
667:         /* update Jacobian matrix for nonlinear models */
668:         for (i = 0; i < N; i++)
669:         {
670:           for (j = 0; j < M; j++)
671:           {
672:             jacob[i][j] = funct_deriv( funct, i, j, M, cond, a);
673:           }
674:         }
675:         iter_cnt = 1;
676:         min_chisq = -1;
677:         mu_fac = 0.0001;
678:       }
679:       else
680:         break;
681:     }
682:   }
683:   while (!ls_flag->linear);
684:   /* iterate if nonlinear */
685: }
686:
687: /*
688:  * determination of covariance matrix
689:  * if not used solve_lin()
690:  */
691: if (!ls_flag->svd || !ls_flag->linear)
692: {
693:   /* compute normal w/o mu !!!! */
694:   for (j = 0; j < M; j++)
695:   {
696:     for (i = 0; i < M; i++)
697:     {
698:       normal[j][i] = 0.;
699:       for (n = 0; n < N; n++)
700:       {
701:         normal[j][i] += jacob[n][j] * jacob[n][i] * weights[n];
702:       }
703:     }
704:   }
705:
706:   /* inversion
707:    * for ill-conditioned problems (e.g. polynomials of high
708:    * order) the inversion of the normal matrix might fail
709:    *
710:    */
711:   err = svd_inversion( M, normal, covar, out);
712:   if (err)
713:   {
714:     /* take best parameter */
715:     for (j = 0; j < M; j++)
716:     {
717:       a[j] = min_a[j];
718:     }
719:   }
720: }
721: endfunc:
722:   fprintf( out,
723:     "\n#  -- %s - end  ------------------------------", rtn);
724:
725:   free_vector( &min_a);
726:   free_vector( &da);
727:   free_vector( &tmpvec);
728:   free_vector( &datac);
729:   free_vector( &deltasq);
730:   free_matrix( &normal);
731:   free_matrix( &normal_i);
732:   free_matrix( &cofac);
733:   free_matrix( &tmpmat);
734:
735:   return err;
736: }
```

```
 0: /*************************************************************
 1:  *
 2:  * File........:  ls_straightline.c
 3:  * Function....:  least-squares solution for straight lines
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  20.10.2007
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *************************************************************/
14: #include <stdio.h>
15: #include <stdlib.h>
16: #include <math.h>
17:
18: /*-------------------------------------------------------------
19:  *  ls_straightline()
20:  *-------------------------------------------------------------*/
21: void
22: ls_straightline(
23:   int N,          /* number of entries */
24:   double cond[],  /* vector of conditions */
25:   double obs[],   /* vector of observations */
26:   double a[]      /* container for parameters to be estimated */
27:   )
28: {
```

```
29:   int i;
30:   double Sxx, Sx, Sy, Sxy, tmp;
31:
32:     Sx = Sxx = Sy = Sxy = 0.;
33:     for (i = 0; i < N; i++)
34:     {
35:       Sx += cond[i];
36:       Sy += obs[i];
37:       Sxx += cond[i] * cond[i];
38:       Sxy += cond[i] * obs[i];
39:     }
40:
41:
42:     tmp = N * Sxx - Sx * Sx;
43:     a[0] = (Sxx * Sy - Sx * Sxy) / tmp;
44:     a[1] = (N * Sxy - Sx * Sy) / tmp;
45: }
```

```
 0: /*****************************************************************
 1:  *
 2:  * File.........:  solve_lin.c
 3:  * Function....:  solving linear least squares via SVD
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  25.01.2010
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *****************************************************************/
14: #include <stdio.h>
15: #include <stdlib.h>
16: #include <string.h>
17: #include <math.h>
18: #include "errmsg.h"
19: #include "matrix_utils.h"
20: #include "macros.h"
21: #include "prototypes.h"
22:
23: //#define QDEBUG
24:
25: /*---------------------------------------------------------------
26:  *  solve_lin()
27:  *
28:  *---------------------------------------------------------------*/
29: int
30: solve_lin( int N, int M, double *obs, double *weights,
31:               double **jacob, double **covar, double *a,
32:               FILE *out)
33: {
34:   char *rtn = "solve_lin";
35:   int i, j, n, err = 0;
36:   double thresh, smax, sqrtwe;
37:   double **tmpmat = NULL; /* temporary matrix */
38:   double **tmpmat2 = NULL;  /* temporary matrix */
39:   double *s = NULL; /* singular values */
40:   double **V = NULL;  /* V matrix */
41:   double **WJ = NULL; /* W*J matrix */
42:
43:     V = matrix( M, M);  /* V matrix for SVD */
44:     s = vector( M); /* singular values for SVD */
45:     WJ = matrix( N, M); /* temporary matrix */
46:     tmpmat = matrix( M, M); /* temporary matrix */
47:     tmpmat2 = matrix( M, N);  /* temporary matrix */
48:
49:     /* WJ = sqrt(W)*J  */
50:     for (i = 0; i < N; i++)
51:     {
52:       sqrtwe = sqrt( weights[i]);
53:       for (j = 0; j < M; j++)
54:       {
55:         WJ[i][j] = sqrtwe * jacob[i][j];
56:       }
57:     }
58:
59: #ifdef QDEBUG
60:     fprintf( out, "\n#\n#== Weight * Jacobian ================");
61:     for (i = 0; i < N; i++)
62:     {
63:       fprintf( out, "\n# ");
64:       for (j = 0; j < M; j++)
65:       {
66:         fprintf( out, " %8.5f", WJ[i][j]);
67:       }
68:     }
69: #endif
70:
71:     /* do the SVD */
72:     err = singvaldec( WJ, N, M, s, V);
73:     if (err)
74:     {
75:         free_matrix( &V);
76:         free_vector( &s);
77:         free_matrix( &WJ);
78:         free_matrix( &tmpmat);
79:         free_matrix( &tmpmat2);
80:         goto endfunc;
81:     }
82:
83: #ifdef QDEBUG
84:     fprintf( out, "\n#\n#== U ================");
85:     for (i = 0; i < N; i++)
86:     {
87:       fprintf( out, "\n# ");
88:       for (j = 0; j < M; j++)
89:       {
90:         fprintf( out, " %8.5f", WJ[i][j]);
91:       }
92:     }
93:     fprintf( out, "\n#\n#== V ================");
94:     for (i = 0; i < M; i++)
95:     {
96:       fprintf( out, "\n# ");
97:       for (j = 0; j < M; j++)
98:       {
99:         fprintf( out, " %8.5f", V[i][j]);
100:      }
101:    }
102: #endif
103:
104:    /* check the singular values  */
105:    smax = 0.0;
106:    for (j = 0; j < M; j++)
107:    {
108:      if (s[j] > smax)  smax = s[j];
109:    }
110:    if (smax < TOL_S)
111:    {
112:      fprintf( stderr,
113:        "\n###\n###     singular matrix, smax = %f",smax);
114:      fprintf( out,
115:        "\n###\n###     singular matrix, smax = %f",smax);
116:
117:      err = 1;
118:      goto endfunc;
119:    }
120:    else if (smax > 1.e+31)
121:    {
122:      fprintf( stderr,
123:        "\n###\n###     degraded matrix, smax = huge");
124:      fprintf( out,
125:        "\n###\n###     degraded matrix, smax = huge");
126:      err = 1;
127:      goto endfunc;
128:    }
129:
130:    thresh = MIN( TOL_S * smax, TOL_S);
131:
132:    fprintf( out, "\n#\n#  singular values (thresh = %.14G)\n#  ",
133:        thresh);
134:    for (j = 0; j < M; j++)
135:    {
136:        fprintf( out, "s%d=%.14G, ", j+1, s[j]);
137:    }
138:
139:    /* invert singular values */
140:    for (j = 0; j < M; j++)
141:    {
142:      /* <= in case of smax =0 */
143:      if (s[j] <= thresh)
144:          s[j] = 0.0;
145:      else
146:          s[j] = 1. / s[j];
147:    }
148:
149:    /* V * [diag(1/s[j])] */
150:    for (i = 0; i < M; i++)
151:    {
152:      for (j = 0; j < M; j++)
153:      {
154:        tmpmat[i][j] = V[i][j] * s[j];
155:      }
156:    }
157:
158: #ifdef QDEBUG
159:    fprintf( out, "\n#\n#== V * inv(S) ================");
160:    for (i = 0; i < M; i++)
161:    {
162:      fprintf( out, "\n# ");
163:      for (j = 0; j < M; j++)
164:      {
165:        fprintf( out, " %8.5f", tmpmat[i][j]);
166:      }
167:    }
168: #endif
169:
170:    /* multiplication of tmpmat with transposed of U  */
171:    /* result is: inv(W*J) = (V*inv(S)) * U'  */
172:    for (i = 0; i < M; i++)
```

```
    {
      for (j = 0; j < N; j++)
      {
        tmpmat2[i][j] = 0.;
        for (n = 0; n < M; n++)
        {
          tmpmat2[i][j] += tmpmat[i][n] * WJ[j][n];
        }
      }
    }

#ifdef QDEBUG
    fprintf( out, "\n#\n#== V * inv(S) * U' ================");
    for (i = 0; i < M; i++)
    {
      fprintf( out, "\n# ");
      for (j = 0; j < N; j++)
      {
        fprintf( out, " %8.5f", tmpmat2[i][j]);
      }
    }
#endif

    /* compute the parameter vector a = inv(W*J)*W*y */
    for (j = 0; j < M; j++)
    {
      a[j] = 0.0;
      for (i = 0; i < N; i++)
      {
        a[j] += tmpmat2[j][i] *  sqrt(weights[i]) * obs[i];
      }
    }

    /* compute covariance matrix V*S^(-2) */
    for (i = 0; i < M; i++)
    {
      for (j = 0; j < M; j++)
      {
        tmpmat[i][j] = V[i][j] * s[j] * s[j];
      }
    }
    /* compute covariance matrix tmpmat * V' */
    multmatsqT( M, covar, tmpmat, V);

endfunc:
  free_vector( &s);
  free_matrix( &V);
  free_matrix( &tmpmat);
  free_matrix( &tmpmat2);
  free_matrix( &WJ);

  return err;
}
```

```
/*****************************************************************
 *
 * File........:  est_weights.c
 * Function....:  estimation of weights
 * Author......:  Tilo Strutz
 * last changes:  03.07.2009
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
 *****************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "errmsg.h"
#include "matrix_utils.h"
#include "macros.h"
#include "prototypes.h"
#include "defines.h"

/*---------------------------------------------------------------
 *  est_weights1()
 *
 *  weights estimation based on deviates, 50% equal weights
 *  no dependence on standard uncertainty of observations
 *
 *---------------------------------------------------------------*/
void
est_weights1( int N, double *deviates, /* absolute deviates ! */
              double *weights, FILE *out)
{
  char *rtn="est_weights1";
  int i;
  int *idx_dev=NULL;
  double *dev_sort = NULL;
  double lambda_L, lambda_L2, lambda_L2_inv;
  double max_deviate, bound;
  const double kappa_L = 0.05;

  fprintf( out, "\n# -- %s - start ----------------------", rtn);
  /*
   * memory allocation
   */
  idx_dev = ivector( N);
  dev_sort = vector( N);

  /* ascending sorting of deviates and indices */
  memcpy( dev_sort, deviates, sizeof(double) * N);
  for (i = 0; i < N; i++)
  {
    if (weights[i] == 0.)
    {
      /* weights can already be set to zero by purpose
       * in the linearisation process
       * corresponding deviates must be ignored somehow
       */
      dev_sort[i] = 0.;
    }
  }
  /* sorting of absolute deviates and of an index array */
  heap_sort_d_( N, dev_sort, idx_dev);

  /* get max. of all absolute deviates */
  max_deviate = dev_sort[N-1];

  fprintf( out, "\n#  Number of observations: %d", N);
  fprintf( out, "\n#  max_deviate: %f", max_deviate);

  /*
   * check values
   */
  if (max_deviate == 0.0)
  {
    fprintf( out, "\n#\n#  all deviates are equal to zero!");
    fprintf( out, "\n#  nothing to weight, perfect fit!");
  }
  else
  {
    /*
     * initialisation of threshold
     */
    /* keep half of the values with equal weights */
    lambda_L = dev_sort[N/2];
    bound = max_deviate * kappa_L;
    /* ensure minimum value for this threshold */
    if (lambda_L < bound)
    {
      /* value for worst case, stabilising LS */
      lambda_L = bound;
    }
    lambda_L2 = lambda_L * lambda_L;
    fprintf( out, "\n#  lambda_L = %f", lambda_L);

    /*
     * do the weighting
     */
    /* inspect unsorted data */
    lambda_L2_inv = 1. / lambda_L2;
    for ( i = 0; i < N; i++)
    {
      if (weights[i] > 0.)
      {
        /* weights can already be set to zero by purpose
         * in the linearisation process
         */
        if (deviates[i] < lambda_L)
        {
          /* fixed weight */
          weights[i] = lambda_L2_inv;
        }
        else
        {
          /* adapted weight */
          weights[i] = 1. / (deviates[i] * deviates[i]);
        }
      }
    }

  } /*  if max_deviate > 0 */

#ifdef NORM_W
    /* may not used when evaluating the goodness-of-fit */
    {
      int cnt;
      double sum;
      /* Normalisation of weights */
      sum = 0.;
      cnt = 0;
      for (i = 0; i < N; i++)
      {
        if (weights[i])
        {
          sum += weights[i];
```

```
137:           cnt++;
138:         }
139:       }
140:       for (i = 0; i < N; i++)
141:       {
142:         weights[i] = weights[i] * cnt / sum;
143:       }
144:     }
145: #endif
146:
147:     /* output information for debugging */
148:     fprintf( out, "\n#\n#  dev_sort[i]          weights[ idx[i] ]");
149:     for (i = 0; i < N && i < MAX_LINES_W; i++)
150:     {
151:       fprintf( out, "\n#%4d  %14.9e %16.8f", i,
152:           dev_sort[i], weights[ idx_dev[i] ]);
153:     }
154:
155:     free_vector( &dev_sort);
156:     free_ivector( &idx_dev);
157:     fprintf( out, "\n# -- %s - end  ------------------------",rtn);
158: }
159:
160: /*-------------------------------------------------------------
161:  *  est_weights2()
162:  *
163:  *  weights estimation based on binning
164:  *
165:  *-------------------------------------------------------------*/
166: void
167: est_weights2( int N, double *cond, double *obs,
168:               double *weights, int obs_per_bin, FILE *out)
169: {
170:   char *rtn="est_weights2";
171:   int i, n, m0, m, b, cnt;
172:   int num_bins;
173:   int *idx=NULL;
174:   double w, var, diff, a[2], last_cond;
175:   double *cond_sort=NULL;
176:   double *bin_cond = NULL;
177:   double *bin_obs = NULL;
178:
179:     fprintf( out,
180:       "\n#  -- %s - start ------------------------------", rtn);
181:
182:     /* vector of conditions in a single bin */
183:     bin_cond = vector( obs_per_bin);
184:     /* vector of observations in a single bin */
185:     bin_obs = vector( obs_per_bin);
186:
187:     /* determine the number of bins */
188:     num_bins = N / obs_per_bin;
189:     /* one bin for the rest */
190:     if (N % obs_per_bin) num_bins++;
191:
192:     cond_sort = vector(N); /* array for sorted conditions */
193:     idx = ivector( N);     /* vector for sorted indices   */
194:
195:     /* copy all conditions */
196:     memcpy( cond_sort, cond, sizeof(double)*N);
197:
198:     /* ascending sorting of conditions and indices */
199:     heap_sort_d_( N, cond_sort, idx);
200:
201:     /* initialise last condition value; just for output */
202:     last_cond = cond_sort[0];
203:
204:     fprintf( out, "\n#  Number of bins: %d", num_bins);
205:     fprintf( out, "\n#  bin number_of_observations   a1        ");
206:     fprintf( out, "a2             last_cond   variance");
207:
208:     n = 0; /* n... already used observations */
209:
210:     /* for all bins */
211:     for ( b = 0; b < num_bins; b++)
212:     {
213:       if ( N-n < obs_per_bin/ 2)
214:       {
215:         /* applies for the last 2 bins:
216:          * if too less remaining observation,
217:          * then let bins overlap
218:          */
219:         m0 = n - obs_per_bin/2;
220:         obs_per_bin = N-m0;
221:       }
222:       else m0 = n;
223:
224:       /*
225:        * piece-wise linear approximation per bin
226:        */
227:
228:       /* copy bin related values */
229:       cnt = 0;
230:       for ( i = 0, m = m0; i < obs_per_bin && m < N; i++, m++)
231:       {
232:         bin_cond[i] = cond_sort[m];
233:         /* bin_cond[i] = cond[ idx[m] ];  same */
234:         bin_obs[i] = obs[ idx[m] ];
235:         cnt++;
236:       }
237:       /* determine parameters for piece-wise linear fit */
238:       ls_straightline( cnt, bin_cond, bin_obs, a);
239:
240:       /* determine uncertainty */
241:       m = m0;
242:       var = 0.;
243:       for ( i = 0; i < obs_per_bin && m < N; i++, m++)
244:       {
245:         diff = obs[ idx[m] ] - (a[0] + a[1] * bin_cond[i]);
246:         var += diff * diff;
247:       }
248:       var = var / (double)cnt;
249:       if (var > 0.)
250:         w = 1. / var;
251:       else w = 0.;
252:       fprintf( out,
253:           "\n#  %4d %10d    %14.4e %14.4e %14.4e %12.3e",
254:           b, i, a[0], a[1], bin_cond[i-1], var);
255:       fprintf( stdout,
256:         "\n  g%d(x) = (x > %.4f && x < %.4f) ? %.4f +x*  %.4f : 0",
257:         b, last_cond, bin_cond[i-1],  a[0], a[1]);
258:
259:       last_cond = bin_cond[i-1];
260:
261:       /* klone determined bin weight for all corresponding
262:        * observations
263:        */
264:       for ( i = 0, m = m0; i < obs_per_bin && m < N; i++, m++)
265:       {
266:           weights[ idx[m] ] = w;
267:       }
268:       n = m; /* update number of already used observations */
269:     }
270:
271:     free_vector( &bin_cond);
272:     free_vector( &bin_obs);
273:     free_vector( &cond_sort);
274:     free_ivector( &idx);
275:     fprintf( out,
276:       "\n#  -- %s - end  ------------------------------", rtn);
277: }
```

```
  0: /*************************************************************
  1:  *
  2:  * File....:  outlier_detection.c
  3:  * Function:  outlier detection
  4:  * Author..:  Tilo Strutz
  5:  * Date....:  03.07.2009, 3
  6:  *
  7:  * changes:
  8:  * 10.03.2011, 3.4.2011
  9:  * 20.08.2012 implementation of RANSAC, M-score
 10:  *
 11:  * LICENCE DETAILS: see software manual
 12:  *  free academic use
 13:  *  cite source as
 14:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 15:  *  2nd edition 2015"
 16:  *
 17:  *************************************************************/
 18: #include <stdio.h>
 19: #include <stdlib.h>
 20: #include <string.h>
 21: #include <assert.h>
 22: #include <math.h>
 23: #include <time.h>
 24: #include "errmsg.h"
 25: #include "matrix_utils.h"
 26: #include "macros.h"
 27: #include "prototypes.h"
 28: #include "erf.h"
 29: #include "defines.h"
 30: /* disables output from outlier detection */
 31: /* #define COMPTEST */
 32:
 33: /*-------------------------------------------------------------
 34:  *  outlier_detection1()
 35:  *
 36:  *  outlier_detection based on standard deviation of obs.
 37:  *  plus re-weighting
 38:  *-------------------------------------------------------------*/
 39: int
 40: outlier_detection1( int N,/* number of observations */
 41:         double sigma_y,   /* standard uncertainty */
 42:         double *deviates, /* absolute deviates    */
 43:         double *weights,  /* weights of observations  */
 44:         double nu,        /* Chauvenet's parameter    */
 45:         FILE *out)
 46: {
 47:   char *rtn = "outlier_detection1";
 48:   int i, err;
```

```
  int count_outlier; /* counts the outliers */
  double erfval;
  double lambda_0;
  double kappa_0;

#ifndef COMPTEST
  fprintf( out,
    "\n#\n# -- %s - start ------------------------------", rtn);
#endif

  if (nu > 0. && nu < 1.0)
  {
    /* set threshold according to Chauvenet's criterion,
     * if nu is inside correct range
     */
    err = erfinv( 1 - nu / N, &erfval);
    if (err) return 0;
    kappa_0 = sqrt(2.) * erfval;
  }
  else
  {
    kappa_0 = 4;  /* use fixed conservative threshold */
  }

  /* set breakdown point (threshold)  */
  lambda_0 = sigma_y * kappa_0;

#ifndef COMPTEST
  fprintf( out, "\n#  sigma_y = %f, kappa_0 = %f, lambda_0 = %f",
           sigma_y, kappa_0, lambda_0);
#endif

  /* compare all deviates with threshold  */
  count_outlier = 0;
  for (i = 0; i < N; i++)
  {
    if (deviates[i] >= lambda_0)  /* take as outlier */
    {
      weights[i] = 0.0;
      count_outlier++;
    }
  }

#ifndef COMPTEST
  fprintf( out,
    "\n# -- %s - end  ------------------------------", rtn);
#endif

  return count_outlier;
}

/*-------------------------------------------------------------------
 *  outlier_detection2()
 *
 *  cluster-based outlier detection (ClubOD)
 *
 *-------------------------------------------------------------------*/
int
outlier_detection2( int N,/* number of observations */
        double *deviates, /* absolute deviates    */
        double *weights,  /* weights of observations  */
        FILE *out)
{
  char *rtn="outlier_detection2";
  int i, j;
  int start_i;  /* range for outlier search */
  int stop; /* flag */
  int cntZ; /* counter for distances being equal to zero  */
  int count_outlier; /* counts the outliers */
  int *idx_dev=NULL;
  double relation, relation_max;
  double dist_loc, dist_ave;
  double *dist_glob = NULL;
  double *dist_all = NULL, dist_all_max;
  double *dist_sort = NULL;
  double *dev_sort = NULL;
  double lambda_0;
  double dist_thresh;
  double kappa1; /* kappa to be used  */

  /* number of observations for determination of kappa_1  */
  const int NN[19]=
    {  8,  11,  16,    23,  32,    45,   64,   91, 128,  181, 256,
      362, 512, 724, 1024, 1448, 2048, 2896, 4096
    };
  /* kappa_1 values according to NN[] */
  const double ka1[19]=
  { 7.3, 7.7, 10.1, 11.8, 14.1, 16.7, 20.3, 25.2, 31.5, 39.6,  51.3,
    66.6, 86.4, 112, 150, 198.0, 261.0, 351.0,
  };
  const double kappa2 = 2.0;
  /* proportion of observations that should belong to the
   * bulk of good observations
   */
  const double kappa4 = 0.6;


#ifndef COMPTEST
    fprintf( out, "\n# -- %s - start ----------------------", rtn);
#endif

    /*
     * memory allocation
     */
    idx_dev = ivector( N);  /* index array  */
    dev_sort = vector( N);  /* sorted absolute deviations */
    dist_all = vector( N);  /* distances between deviations */
    dist_sort = vector( N); /* sorted distances */
    dist_glob = vector( N); /* global distances */

    /*
     * determination of kappa_1 in dependent on number of
     * observations (threshold for suspicious distances)
     */
    if (N > NN[0])
    {
      if (N >= NN[18])
          kappa1 = ka1[18];
      else
      {
        for (i = 1; i < 19; i++)
        {
          if ( N <= NN[i])
          {
            double frac;

            frac = (double)(NN[i] - N) / (double)(NN[i] - NN[i-1]);
            /* linear interpolation */
            kappa1 = ka1[i] * (1-frac) + frac * ka1[i-1];
            break;
          }
        }
      }
    }
    else kappa1 = ka1[0];

    /* ascending sorting of deviates and indices */
    memcpy( dev_sort, deviates, sizeof(double) * N);
    heap_sort_d_( N, dev_sort, idx_dev);

    /*
     * determine all distances
     */
    for (i = N-1; i > 0; i--)
    {
      dist_all[i] = dev_sort[i] - dev_sort[i-1];
    }

    /* dummy */
    dist_all[0] = 0.;

    /* ascending sorting of distances in order to get
     * the median distance
     */
    memcpy( dist_sort, dist_all, sizeof(double) * N);
    heap_sort_d( N, dist_sort);

#ifndef COMPTEST
    fprintf( out, "\n#  Number of observations: %d", N);
    fprintf( out, "\n#  kappa1: %f", kappa1);
    fprintf( out, "\n#  kappa2: %f", kappa2);
#endif

    /* keep major part as good observations */
    start_i = (int) ceil( kappa4 * N);

    /* determination of dist_glob,
     * separate value for each deviate
     */
    {
      double arg, wj, sum_wj;
      /* for all observations, which have to be tested  */
      for ( i = start_i; i < N; i++)
      {
        /* compute weighted average as d_glob,
         * 2* sigma = N, variance = N*N/4
         */
        arg = -0.5 * 4. / (N * N);
        sum_wj = 0.;
        dist_glob[i] = 0;

        /* take all distances into account, exclude [0] */
        for (j = 1; j < i; j++)
        {
          wj =  exp( j*j * arg);
          /* dist_all[] sorted according increasing deviates
           * largest fac2 for distance of highes deviate
           */
          dist_glob[i] += dist_all[i-j] * wj;
          sum_wj += wj;
        }
        if (sum_wj > 0.)
          dist_glob[i] /= sum_wj; /* normalise by sum */
```

```
241:         else
242:           dist_glob[i] = 0.0;
243:       }
244:     }
245:
246:     if (dist_glob[start_i] == 0.0) /* check for smallest deviate */
247:     {
248: #ifndef COMPTEST
249:       fprintf( out, "\n#\n#  all distances are equal to zero!");
250:       fprintf( out, "\n#  nothing to weight, perfect fit!");
251: #endif
252:     }
253:     else
254:     {
255:       /* initialisation of thresholds */
256:       /* reset breakdown point */
257:       lambda_0 = 0;
258:
259:       /*
260:        * search for suspicious distance
261:        */
262:
263: #ifndef COMPTEST
264:       fprintf( out,
265:         "\n#\n#   n     deviate         d[n]          d_glob");
266:       fprintf( out, "           d[n]/d_glob    d_loc   d[n]/d_loc ");
267:       fprintf( out,
268:         "\n# ----------------------------------------------");
269:       fprintf( out, "------------------------------");
270: #endif
271:
272:       stop = 0;
273:       relation_max = 0.;
274:       dist_all_max = 0;
275:
276:       /* check only upper portion of sorted deviates  */
277:       for (i = start_i; i < N && !stop; i++)
278:       {
279:         dist_thresh = dist_glob[i] * kappa1;
280:
281: #ifndef COMPTEST
282:         fprintf( out, "\n# %3d\t %.3e\t %.3e", i, dev_sort[i],
283:                                                 dist_all[i]);
284:         fprintf( out, "\t %8.2e\t %8.2f",
285:           dist_glob[i], dist_all[i] / dist_glob[i]);
286: #endif
287:
288:         /* if global condition is fulfilled, then check also local
289:          * one
290:          */
291:         if (dist_all[i] >= dist_thresh)
292:         {
293:           double arg, wj, sum_wj;
294:
295:           /* determination of local distance value,
296:            * 12* sigma = N, variance = N*N/144
297:            */
298:           arg = -0.5 * 144. / (N * N);
299:           cntZ = 0;
300:           sum_wj = 0;
301:           dist_ave = 0;
302:           for (j = 1; j < i; j++)
303:           {
304:             wj =  exp( j*j * arg);
305:             /* largest fac2 for closest neighbour */
306:             dist_ave += dist_all[i-j] * wj;
307:             sum_wj += wj;
308:             /* count distances equal to zero  */
309:             if (dist_all[i-j] == 0.0) cntZ++;
310:           }
311:           if (sum_wj)
312:             dist_loc = dist_ave / sum_wj;
313:           else
314:             dist_loc = 0.0;
315:
316:           /* exception handling, if more than 50% of distances
317:            * are equal to zero
318:            */
319:           if (cntZ <= i/2)
320:           {
321:             /* normal calculation */
322:             if (dist_loc > 0.0)
323:               relation = dist_all[i] / dist_loc;
324:             else
325:               relation = 0.0;
326:           }
327:           else /* exception handling  */
328:           {
329:             /* current distance is > zero, border case */
330:             if (dist_all[i] > 0)
331:             {
332:               relation = kappa2;
333:             }
334:             else
335:             {
336:               /* not considered further */
337:               relation = 0;
338:             }
339:           }
340:
341: #ifndef COMPTEST
342:           fprintf( out, "\t %.3e", dist_loc);
343:           fprintf( out, "\t %f", relation);
344: #endif
345:
346:           /* check second condition */
347:           if (relation >= kappa2)
348:           {
349:             /* outliers found */
350:
351: #ifndef COMPTEST
352:             fprintf( out, " *");
353: #endif
354:             /* look for the highest relation (change in distances)*/
355:             if (relation > relation_max)
356:             {
357: #ifndef COMPTEST
358:               fprintf( out, " ##");
359: #endif
360:               /* deviate of current position is taken as
361:                * breakdown point
362:                */
363:               lambda_0 = dev_sort[i];
364:               /* store current releation as maximal one */
365:               relation_max = relation;
366:               /* store current distance as maximum distance */
367:               dist_all_max = dist_all[i];
368:             }
369:             else if (relation == relation_max)
370:             {
371:               /* can happen, for instance, if relation was several
372:                * times equal to kappa2
373:                */
374:               /* if equal, then take teh one corresponding to the
375:                * larger absolute deviate
376:                */
377:               if (dist_all[i] >= dist_all_max)
378:               {
379:                 /* take the one with higher distance */
380: #ifndef COMPTEST
381:                 fprintf( out, " ##");
382: #endif
383:                 lambda_0 = dev_sort[i];
384:                 relation_max = relation;
385:                 dist_all_max = dist_all[i];
386:               }
387:             }
388:             /*stop = 1; */
389:           }
390:         } /* if (dist_sort[i] > dist_thresh) */
391:       } /* for i  */
392:
393:       /* check whether there was at least one relation
394:        * higher than kappa2
395:        */
396: #ifndef COMPTEST
397:       if (relation_max > 0.0)
398:       {
399:         fprintf( out, "\n#\n#  outliers detected !");
400:         fprintf( out, "\t  lambda_0 = %f", lambda_0);
401:       }
402: #endif
403:
404:       /* inspect unsorted data, set weights of outliers to zero*/
405:       count_outlier = 0;
406:       if (lambda_0 > 0)
407:       {
408:         for ( i = 0; i < N; i++)
409:         {
410:           if (deviates[i] >= lambda_0) /* outliers */
411:           {
412:               weights[i] = 0.;
413:               count_outlier++;
414:           }
415:         }
416:       }
417:     } /*  if (enough observations) */
418:
419:     /* output information for debugging */
420: #ifndef COMPTEST
421:     fprintf( out, "\n#\n#   n  deviate      ");
422:     fprintf( out, "           d[n]         weights[ idx[i] ]");
423:     for (i = 0; i < N && i < MAX_LINES_W; i++)
424:     {
425:       fprintf( out, "\n#%4d  %14.9e %14.9e %16.8f", i,
426:           dev_sort[i], dist_all[i],
427:           weights[ idx_dev[i] ]);
428:     }
429: #endif
430:
431:     free_vector( &dist_glob);
432:     free_vector( &dev_sort);
```

```
433:     free_vector( &dist_sort);
434:     free_vector( &dist_all);
435:     free_ivector( &idx_dev);
436:
437: #ifndef COMPTEST
438:     fprintf( out, "\n# -- %s - end  ----------------------",rtn);
439: #endif
440:
441:     return count_outlier;
442: }
443:
444:
445: /*-------------------------------------------------------------
446:  *  outlier_detection3()
447:  *
448:  *  outlier_detection based on Median Absolute Deviation (MAD)
449:  *  plus re-weighting
450:  *-------------------------------------------------------------*/
451: int
452: outlier_detection3( int N,/* number of observations */
453:         double *deviates, /* absolute deviates    */
454:         double *weights,  /* weights of observations  */
455:         double nu,        /* Chauvenet's parameter    */
456:         FILE *out)
457: {
458:   char *rtn = "outlier_detection3";
459:   int i, err;
460:   int count_outlier; /* counts the outliers */
461:   double erfval;
462:   double lambda_0;
463:   double kappa_0;
464:   double *dev_sort = NULL, median_dev;
465:
466: #ifndef COMPTEST
467:   fprintf( out,
468:          "\n# -- %s - start ------------------------------", rtn);
469: #endif
470:
471:   if (nu > 0. && nu < 1.0)
472:   {
473:     /* set threshold according to Chauvenet's criterion,
474:      * if nu is inside correct range
475:      */
476:     err = erfinv( 1 - nu / N, &erfval);
477:     if (err) return 0;
478:     kappa_0 = sqrt(2.) * erfval;
479:   }
480:   else
481:   {
482:     kappa_0 = 4;  /* use fixed conservative threshold */
483:   }
484:
485:   /*
486:    * determine median of absolute deviates
487:    */
488:   dev_sort = vector( N);  /* sorted absolute deviations */
489:   /* ascending sorting of deviates and indices */
490:   memcpy( dev_sort, deviates, sizeof(double) * N);
491:   heap_sort_d( N, dev_sort);
492:   median_dev = dev_sort[N/2];
493:
494:   /* set breakdown point (threshold)  */
495:   /* z-score: lambda_0 = sigma_y * kappa_0; */
496:   lambda_0 = 1.4826 * median_dev * kappa_0;
497:
498: #ifndef COMPTEST
499:   fprintf( out,
500:   "\n#  1.4826 * median_dev = %f, kappa_0 = %f, lambda_0 = %f",
501:           1.4826 * median_dev, kappa_0, lambda_0);
502: #endif
503:
504:   /* compare all deviates with threshold  */
505:   count_outlier = 0;
506:   for (i = 0; i < N; i++)
507:   {
508:     if (deviates[i] >= lambda_0)  /* take as outlier */
509:     {
510:       weights[i] = 0.0;
511:       count_outlier++;
512:     }
513:   }
514:
515:   free_vector( &dev_sort);
516:
517: #ifndef COMPTEST
518:   fprintf( out,
519:     "\n# -- %s - end  ------------------------------", rtn);
520: #endif
521:
522:   return count_outlier;
523: }
524:
525: /*-------------------------------------------------------------
526:  *  ransac()
527:  *
528:  *  outlier_detection based on RANSAC/MSAC method
529:  * This is actually not a method for outlier detection, but
530:  * a method selecting the most suitable modell-parameter vector
531:  * out of many hypothesis of parameters.
532:  *
533:  *-------------------------------------------------------------*/
534: int
535: ransac( double (*funct) (int,double*,double*),
536:    double (*funct_deriv) (double(*)(int,double*,double*),
537:    int,int,int,double*,double*),
538:    double (*funct_deriv2) (double(*)(int,double*,double*),
539:    int,int,int,int,double*,double*),
540:    int (*init)(int, double*,double*,double*,
541:    unsigned char*,FILE*),
542:    int N, int M, double *obs, double *cond, double **jacob,
543:    double *weights, double *a, unsigned char* a_flag,
544:    int algo_mode, LS_FLAG *ls_flag,
545:    double chisq_target, double **covar, FILE *out,
546:    double *deviates_abs,
547:    int cond_dim,
548:    int obs_dim)
549: {
550:   char *rtn = "ransac outlier detection";
551:   int err = 0;
552:   int iter, num_iterat;
553:   int i, j, im, rn, rm, idx;
554:   int Ns;
555:   int num_outlier, cnt_inliers, cnt_inliers_best;
556:   double rin = 0.5;   /* percentage of inliers  */
557:   double s_a[M_MAX];      /* parameter of model function    */
558:   double s_a_best[M_MAX]; /* best parameter of model function */
559:   double costs, min_costs, threshold;
560:   double **s_jacob=NULL;/* array for subset of Jacobian matrix */
561:   double *s_obs = NULL; /* array for subset of observations */
562:   double *s_cond = NULL; /* array for subset of conditions */
563:   double *s_weights = NULL; /* array for subset of weights */
564:   /* array for sorted original deviates */
565:   double *dev_sort = NULL;
566:   /* array for deviates based on subset approximation */
567:   double *deviates = NULL;
568:   /* array of deviates based on best subset */
569:   double *deviates_best = NULL;
570:   /* array for indices of selected data points */
571:   unsigned int *s_idx = NULL;
572:   /* array for indices of selected data points of best subset */
573:   unsigned int *s_idx_best = NULL;
574:
575: #ifndef COMPTEST
576:   fprintf( out,
577:          "\n# -- %s - start ------------------------------", rtn);
578: #endif
579:
580:   /* set size of subset */
581:   Ns = MIN( 2*M, N/3);
582:   if (Ns < M) Ns = M;   /* at least equal to number of parameters */
583:
584:   if (ls_flag->linear)
585:   {
586:     Ns = M; /* original setup of RANSAC:
587:             * take minimum number of data required to fit the model
588:             */
589:   }
590:   else
591:   {
592:     Ns = M+1; /* one more than necessary in case of nonlinear models
593:                * leads to somewhat higher numerical stability
594:                */
595:   }
596:
597:
598:   /* 0.999 = probability that at least one subset is outlier-free */
599:   num_iterat = (int)(log(1-0.999) / log(1-pow(rin, Ns)));
600:   fprintf( out, "\n# required trials: %d", num_iterat);
601:
602:   deviates = vector( N * obs_dim);
603:   deviates_best = vector( N * obs_dim);
604:   s_obs = vector( Ns * obs_dim);
605:   s_cond = vector( Ns * cond_dim);
606:   s_weights = vector( Ns * obs_dim);
607:   s_idx = uivector( Ns);
608:   s_idx_best = uivector( Ns);
609:   s_jacob = matrix( Ns * obs_dim, M); /* Jacobian */
610:
611:   /*
612:    * determine threshold based on median of absolute deviates
613:    * based on current approximation
614:    * threshold is a very critical parameter
615:    *  if it is too low, then to many points are declared as outliers
616:    *  if it is too high, then we might miss some outliers
617:    */
618:   dev_sort = vector( N * obs_dim);  /* sorted absolute deviations */
619:   /* ascending sorting of deviates and indices */
620:   memcpy( dev_sort, deviates_abs, sizeof(double) * N * obs_dim);
621:   heap_sort_d( N * obs_dim, dev_sort);
622:   threshold = 3. * dev_sort[N * obs_dim/2];
623:   fprintf( out, "\n#    threshold: %f", threshold);
624:
```

```
625:   /* zero out unnecessary parameters;
626:    * required for POLYNOMIAL_REG
627:    */
628:   for (i = M; i < M_MAX; i++)
629:   {
630:     s_a[i] = 0.;
631:   }
632:
633:   /*
634:    * create hypothesis based on randomly chosen subsets of
635:    * the entire set of data points
636:    */
637:   srand(time(NULL));   /* Seed the random number generator. */
638:   idx = 0;
639:   min_costs = N*9999.; /* arbitrary large number */
640:   cnt_inliers_best = 0;
641:   for ( iter = 0; iter < num_iterat; iter++)
642:   {
643:     /* genetrate sub-set */
644:     /* The Knuth algorithm.
645:      * This is a very simple algorithm with a complexity of O(N)
646:      * The algorithm works as follows:
647:      * iterate through all numbers from 1 to N and
648:      * select the current number with probability rm / rn,
649:      * where rm is how many numbers we still need to find,
650:      * and rn is how many numbers we still need to iterate through.
651:      */
652:     im = 0;
653:     /* create array of indices (LUT)  */
654:     fprintf( out, "\n# subset: ");
655:     for ( i = 0; i < N && im < Ns; i++)
656:     {
657:       rn = N - i;
658:       rm = Ns - im;
659:       if (rand() % rn < rm)
660:       {
661:         /* Take it */
662:         s_idx[im++] = i;
663:         fprintf( out, "%d ", i);
664:       }
665:     }
666:     assert( im == Ns);
667:
668:     /* copy selected data */
669:     for ( i = 0; i < Ns; i++)
670:     {
671:       for ( j = 0; j < obs_dim; j++)
672:       {
673:         /* observations */
674:         s_obs[i * obs_dim+j] = obs[s_idx[i] * obs_dim+j];
675:         /* weights  */
676:         s_weights[i * obs_dim+j] = weights[s_idx[i] * obs_dim+j];
677:       }
678:       for ( j = 0; j < cond_dim; j++)
679:       {
680:         /* conditions */
681:         s_cond[i * cond_dim+j] = cond[s_idx[i]*cond_dim + j];
682:       }
683:     }
684:
685:     if (!ls_flag->linear) /* nonlinear model is used */
686:     {
687:       if (cnt_inliers_best < 0.8 * N)
688:       {
689:         /* too less good data points; < 80% */
690:         for ( i = 0; i < M; i++)
691:         {
692:           s_a[i] = a[i]; /* reset to originally estimated values  */
693:         }
694:       }
695:       else /* use parameter set of based model */
696:       {
697:         for ( i = 0; i < M; i++)
698:         {
699:           s_a[i] = s_a_best[i];
700:         }
701:       }
702:
703:       fprintf( out, "\n# initial Parameters\n# ");
704:       /* write initial parameters to output */
705:       for (i = 0; i < M; i++)
706:       {
707:         fprintf( out, "a%d=%.9f, ", i+1, s_a[i]);
708:       }
709:       /*
710:        * prepare Jacobian matrix containing
711:        * first derivatives of target function
712:        * based on newly estimated parameters
713:        */
714:       for (i = 0; i < Ns*obs_dim; i++)
715:       {
716:         for (j = 0; j < M; j++)
717:         {
718:           s_jacob[i][j] = funct_deriv( funct, i, j, M, s_cond, s_a);
719:         }
720:       }
721:     }
722:     else /* if (!ls_flag->linear)*/
723:     {
724:       /* copy selected data from Jacobian matrix */
725:       for ( i = 0; i < Ns; i++)
726:       {
727:         for ( j = 0; j < M; j++)  /* elements of Jacobian matrix  */
728:         {
729:           s_jacob[i][j] = jacob[s_idx[i]][j];
730:         }
731:       }
732:     }
733:
734:     /* do the least-squares approximation based on subset
735:      * do not overwrite original Parameters in a
736:      */
737:     err =
738:       ls( funct, funct_deriv, funct_deriv2, init, Ns * obs_dim, M,
739:           s_obs, s_cond, s_jacob, s_weights, s_a, a_flag, algo_mode,
740:           ls_flag, chisq_target, covar, out);
741:
742:     /* if ls approximation failed, go to next subset  */
743:     if (err) continue;
744:
745:     fprintf( out, "\n#\n# subset Parameters:  ");
746:     for (j = 0; j < M; j++)
747:     {
748:       fprintf( out, "a%d=%16.12G, ", j + 1, s_a[j]);
749:     }
750:
751:     /* compute cost function based on all data points*/
752:     costs = 0; cnt_inliers = 0;
753:     for ( i = 0; i < N * obs_dim; i++)
754:     {
755:       /* decide between linear and nonlinear models */
756:       if (ls_flag->linear)
757:       {
758:         double model_val;
759:
760:         /* get calculated data points dependent on current
761:            parameters */
762:         model_val = 0.0;
763:         for (j = 0; j < M; j++)
764:         {
765:           model_val += s_a[j] * jacob[i][j];
766:         }
767:         deviates[i] = fabs( model_val - obs[i]);
768:       }
769:       else /* nonlinear */
770:       {
771:         deviates[i] = fabs( funct( i, cond, s_a) - obs[i]);
772:       }
773:       /* check, whether data point (from entire set) is with
774:        * in the limits or not
775:        */
776:       if (deviates[i] < threshold)
777:       {
778:         costs += deviates[i] * deviates[i];
779:         cnt_inliers++;  /* count inliers */
780:       }
781:       else
782:       {
783:         costs += threshold * threshold;
784:       }
785:     }
786:     fprintf( out, "  inliers: %d", cnt_inliers);
787:     /* evaluate hypothesis  */
788:     if (min_costs > costs)
789:     {
790:       fprintf( out, "  ####### best subset so far (%f)", costs);
791:       fflush( out);
792:       min_costs = costs;
793:       /* save current subset as best subset */
794:       for ( i = 0; i < Ns; i++)
795:       {
796:         s_idx_best[i] = s_idx[i];
797:       }
798:       /* save deviates of modell function based on this subset */
799:       for ( i = 0; i < N * obs_dim; i++)
800:       {
801:         deviates_best[i] = deviates[i];
802:       }
803:       for ( i = 0; i < M; i++)
804:       {
805:         s_a_best[i] = s_a[i];
806:       }
807:       cnt_inliers_best = cnt_inliers;
808:
809:       /* adapt required number of subsets and increase value
810:        * somewhat because we are not only in an outlier-free set
811:        * but in a set containing the maximum number of inliers
812:        */
813:       rin = (double)cnt_inliers_best / (N * obs_dim);
814:       rin *= 0.9; /* this increases the number of trials slightly */
815:       num_iterat = (int)(log(1-0.9999) / log(1-pow( rin, Ns)));
816:       fprintf( out, "\n# required trials set to: %d", num_iterat);
```

```
817:       fprintf( out, "\t already seen: %d", iter+1);
818:     }
819:   } /* for ( iter */
820:
821:   num_outlier = N * obs_dim - cnt_inliers_best;
822:   if (num_outlier > 0.4*N)
823:   {
824:     fprintf( out,
825:       "\n#\n# consensus set is too small: only %d out of %d!",
826:         N * obs_dim-num_outlier, N);
827:     fprintf( out,
828:       "\n#\n# reset to full data set with equal weights!");
829:     num_outlier = 0;
830:   }
831:   else
832:   {
833:     fprintf( out, "\n# number of trials: %d", iter);
834:     fprintf( out, "\n# number of outliers: %d\n#", num_outlier);
835:     /* for nonlinear models: copy best parameters as initial
836:      * vector for final approximation (in case of outliers)
837:      */
838:     if (!ls_flag->linear && num_outlier)
839:     {
840:       for ( i = 0; i < M; i++)
841:       {
842:         a[i] = s_a_best[i];
843:       }
844:     }
845:     /* mark outliers */
846:     for ( i = 0; i < N * obs_dim; i++)
847:     {
848:       if (deviates_best[i] >= threshold)
849:       {
850:         weights[i] = 0.;
851:       }
852:     }
853:   }
854:
855:
856:   free_vector( &dev_sort);
857:   free_vector( &deviates);
858:   free_vector( &deviates_best);
859:   free_vector( &s_obs);
860:   free_vector( &s_cond);
861:   free_vector( &s_weights);
862:   free_uivector( &s_idx);
863:   free_uivector( &s_idx_best);
864:
865:
866: #ifndef COMPTEST
867:   fprintf( out,
868:     "\n# -- %s - end  ------------------------------", rtn);
869: #endif
870:
871:   return num_outlier;
872: }
```

S.3 Model functions

```
  0: /*************************************************************
  1:  *
  2:  * File........: functions.c
  3:  * Function....: model functions and their derivatives
  4:  * Author......: Tilo Strutz
  5:  * last changes: 07.05.2008, 26.10.2009, 18.02.2010, 01.01.2011
  6:  *
  7:  * LICENCE DETAILS: see software manual
  8:  *  free academic use
  9:  *  cite source as
 10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 11:  *  2nd edition 2015"
 12:  *
 13: *************************************************************/
 14: #include <stdio.h>
 15: #include <stdlib.h>
 16: #include <string.h>
 17: #include <math.h>
 18: #include <float.h>
 19: #include <assert.h>
 20: #include "macros.h"
 21: #include "functions.h"
 22: #include "defines.h"
 23:
 24: #define DEG2RAD M_PI/180.
 25:
 26: /* for reasons of compatibility all derivative functions
 27:  * have the same parameter list
 28:  * xxx_deriv( double (*funct)(int,double*,double*),
 29:  *        int i, int j, int M, double *cond, double *a)
 30:  *
 31:  * However, the only function requiring (*funct) is the
 32:  * generic function f_deriv()
 33:  */
 34:
 35: /*-------------------------------------------------------------
 36:  *  fconstant_deriv()
 37:  *-------------------------------------------------------------*/
 38: double
 39: fconstant_deriv( double (*funct)(int,double*,double*),
 40:         int i, int j, int M, double *cond, double *a)
 41: {
 42:   return 1.0; /* derivation of a1 */
 43: }
 44:
 45: /*-------------------------------------------------------------
 46:  *  flin_deriv()
 47:  *  f(x|a) = a1 + SUM_j a_j * x_j
 48:  *-------------------------------------------------------------*/
 49: double
 50: flin_deriv( double (*funct)(int,double*,double*),
 51:           int i, int j, int M, double *cond, double *a)
 52: {
 53:   /* i ... number of current observation */
 54:   /* j = 0,1,2,...  ... number of parameter a_j*/
 55:   if (j == 0)
 56:     return 1.;  /* derivation of a1 */
 57:   else
 58:     /* all conditions x_j are stored in a single array cond[.] */
 59:     return cond[(M-1) * i + j - 1];
 60: }
 61:
 62: /*-------------------------------------------------------------
 63:  *  flin2_deriv()
 64:  *  f(x|a) = SUM_j a_j * x_j
 65:  *-------------------------------------------------------------*/
 66: double
 67: flin2_deriv( double (*funct)(int,double*,double*),
 68:           int i, int j, int M, double *cond, double *a)
 69: {
 70:     /* all conditions x_j are stored in a single array cond[.] */
 71:     return cond[M * i + j];
 72: }
 73:
 74: /*-------------------------------------------------------------
 75:  *  fcosine_deriv()
 76:  * f(x|a) = a1 + a2 * cos( x) + a3 * sin( x)
 77:  *-------------------------------------------------------------*/
 78: double
 79: fcosine_deriv( double (*funct)(int,double*,double*),
 80:           int i, int j, int M, double *cond, double *a)
 81: {
 82:   if (j == 0)
 83:     return 1.;  /* derivation of a1 */
 84:   else if (j == 1)
 85:     return (cos( cond[i] * DEG2RAD));
 86:   else
 87:     return (sin( cond[i] * DEG2RAD));
 88: }
 89:
 90: /*-------------------------------------------------------------
 91:  *  fcosine_nonlin()
 92:  * f(x|a) = a1 + a2 * cos( x - a3)
 93:  *-------------------------------------------------------------*/
```

```
double
fcosine_nonlin( int i, double *cond, double *a)
{
  return a[0] + a[1] * cos( (cond[i] - a[2]) * DEG2RAD);
}

/*---------------------------------------------------------------
 *  fcosine_nonlin_deriv()
 * f(x|a) = a1 + a2 * cos( x - a3)
 *---------------------------------------------------------------*/
double
fcosine_nonlin_deriv( double (*funct)(int,double*,double*),
          int i, int j, int M, double *cond, double *a)
{
  if (j == 0)
    return 1.;  /* derivation of a1 */
  else if (j == 1)
    return (cos( (cond[i] - a[2]) * DEG2RAD));
  else
    return (a[1] * sin( (cond[i] - a[2]) * DEG2RAD));
}

/*---------------------------------------------------------------
 *  fcosine_trend()
 * f(x|a) = a1 + a2 * x + a3 * cos( x - a4)
 *---------------------------------------------------------------*/
double
fcosine_trend( int i, double *cond, double *a)
{
  return a[0] + a[1] * cond[i] + a[2] * cos( cond[i] - a[3]);
}

/*---------------------------------------------------------------
 *  fcosine_trend_deriv()
 * f(x|a) = a1 + a2 * x + a3 * cos( x - a4)
 *---------------------------------------------------------------*/
double
fcosine_trend_deriv( double (*funct)(int,double*,double*),
          int i, int j, int M, double *cond, double *a)
{
  if (j == 0)
    return 1.;  /* derivation of a1 */
  else if (j == 1)
    return cond[i];
  else if (j == 2)
    return cos( cond[i] - a[3]);
  else
    return (a[2] * sin( cond[i] - a[3]));
}

/*---------------------------------------------------------------
 *  ftrigonometric1()
 * f(x|a) = a1 + a2*cos(a3*x-a4)
 *---------------------------------------------------------------*/
double
ftrigonometric1( int i, double *cond, double *a)
{
  return a[0] + a[1] * cos(   a[2]*cond[i] - a[3]);
}

/*---------------------------------------------------------------
 *  ftrigonometric2()
 * f(x|a) = a1 + a2*cos(a3*x-a4) + a5*cos(2*a3*x-a7)
 *---------------------------------------------------------------*/
double
ftrigonometric2( int i, double *cond, double *a)
{
  return a[0] + a[1] * cos(   a[2]*cond[i] - a[3])
              + a[4] * cos( 2*a[2]*cond[i] - a[5]);
}

/*---------------------------------------------------------------
 *  flogarithmic()
 * f(x|a) = log( a1 * x)
 *---------------------------------------------------------------*/
double
flogarithmic( int i, double *cond, double *a)
{
  assert( a[0] * cond[i] > 0.);
  return log( a[0] * cond[i]);
}

/*---------------------------------------------------------------
 *  flogarithmic_deriv()
 * f(x|a) = log( a1 * x)
 *---------------------------------------------------------------*/
double
flogarithmic_deriv( double (*funct)(int,double*,double*),
            int i, int j, int M, double *cond, double *a)
{
  /* correction of values, which are outside the domain of
     definition */
  /* if (a[0] <= 0.0) a[0] = 10 * DBL_EPSILON; Too small !! */
  /* y=log(a*x) = log(a) + log(x) ==> y'= 1/a */
  if (a[0] <= 0.0) a[0] = 1.0e-10; /* take a small value */
  return 1./a[0];
}

/*---------------------------------------------------------------
 *  flogarithmic_deriv2()
 * f(x|a) = log( a1 * x)
 *---------------------------------------------------------------*/
double
flogarithmic_deriv2( double (*funct)(int,double*,double*),
            int j, int k, int M, int i, double *cond, double *a)
{
  if (a[0] <= 0.0) a[0] = 1.0e-10; /* take a small value */
  /* y'= 1/a  ==> y'' = -1/(a*a)  */
  return -1./(a[0]*a[0]);
}

/*---------------------------------------------------------------
 *  fexponential()
 * f(x|a) = a1 + a2 * exp( a3 * x)
 *---------------------------------------------------------------*/
double
fexponential( int i, double *cond, double *a)
{
  return (a[0] + a[1] * exp( a[2] *cond[i]));
}

/*---------------------------------------------------------------
 *  fexponential_deriv()
 * f(x|a) = a1 + a2 * exp( a3 * x)
 *---------------------------------------------------------------*/
double
fexponential_deriv( double (*funct)(int,double*,double*),
            int i, int j, int M, double *cond, double *a)
{
  if (j == 0)
    return 1.;
  else if (j == 1)
    return (exp( a[2] * cond[i]));
  else
  {
    if (a[2] < 0) /* limitation of parameter space */
    {
      return (cond[i] * a[1] * exp( a[2] * cond[i]));
    }
    else
      return cond[i] * a[1]; /* set a[2] virtually to zero */
  }
}


/*---------------------------------------------------------------
 *  fpolynomial()
 * f(x|a) = sum_{j=1}^M aj * x^(j-1)
 *---------------------------------------------------------------*/
double fpolynomial( int i, double *cond, double *a)
{
  long j;
  double y, xj;
  /* since M ist not passed as a parameter, we assume maximal
   * number. oveflous a[j] must be zero !
   */
  y = a[0];
  xj = 1.;
  for ( j = 1; j < M_MAX; j++)
  {
    xj *= cond[i];
    y += a[j] * xj;
  }
  return y;
}

/*---------------------------------------------------------------
 *  fpolynomial_deriv()
 * f(x|a) = sum_{j=1}^M aj * x^(j-1)
 *---------------------------------------------------------------*/
double
fpolynomial_deriv( double (*funct)(int,double*,double*),
            int i, int j, int M, double *cond, double *a)
{
  return pow( cond[i], (double)j);
}

/*---------------------------------------------------------------
 *  fgen_laplace()
 * f(x|a) = a1 * exp( -|x|^a2 * a3)
 *---------------------------------------------------------------*/
double
fgen_laplace( int i, double *cond, double *a)
{
  return a[0] * exp( -pow( fabs( cond[i]), a[1]) * a[2]);
}

/*---------------------------------------------------------------
 *  fgen_laplace_deriv()
 * f(x|a) = a1 * exp( -|x|^a2 * a3)
 *---------------------------------------------------------------*/
double
```

```
286: fgen_laplace_deriv( double (*funct)(int,double*,double*),
287:              int i, int j, int M, double *cond, double *a)
288: {
289:   if (a[1] < 0) /* limitation of parameter space */
290:     a[1] = 1E-10;
291:   if (j == 0)
292:   {
293:     /*      y = a0 * exp( -|x|^a1 * a2)
294:      * dy/da0 =      exp( -a2 * |x|^a1)
295:      */
296:     return exp( -a[2] * pow( fabs( cond[i]), a[1]));
297:   }
298:   else if (j == 1)
299:   {
300:     /*      y = a0 * exp( -a2 * |x|^a1)
301:      * dy/da1 = a0 * exp( -a2 * |x|^a1) * -a2 * |x|^a1 * ln|x|
302:      */
303:     return -a[0] * exp( -a[2] * pow( fabs( cond[i]), a[1])) *
304:               a[2] *
305:               pow( fabs( cond[i]), a[1]) * log( fabs( cond[i]));
306:   }
307:   else
308:   {
309:     /*      y = a0 * exp( -a2 * |x|^a1)
310:      * dy/da2 = a0 * exp( -a2 * |x|^a1) * - |x|^a1
311:      */
312:     return -a[0] * exp( -a[2] * pow( fabs( cond[i]), a[1])) *
313:                         pow( fabs( cond[i]), a[1]);
314:   }
315: }
316:
317: /*-------------------------------------------------------------
318:  *  fexpon2()
319:  * f(x|a) = a1 * exp( a2 * x)
320:  *-------------------------------------------------------------*/
321: double
322: fexpon2( int i, double *cond, double *a)
323: {
324:   return (a[0] * exp( a[1] * cond[i]));
325: }
326:
327: /*-------------------------------------------------------------
328:  *  fexpon2_deriv()
329:  * f(x|a) = a1 * exp( a2 * x)
330:  *-------------------------------------------------------------*/
331: double
332: fexpon2_deriv( double (*funct)(int,double*,double*),
333:              int i, int j, int M, double *cond, double *a)
334: {
335:   if (j == 0)
336:   {
337:     if (a[1] < 0) /* limitation of parameter space */
338:     {
339:       return (exp( a[1] * cond[i]));
340:     }
341:     else
342:       return 1; /* set a[1] virtually to zero */
343:   }
344:   else
345:   {
346:     if (a[1] < 0) /* limitation of parameter space */
347:     {
348:       return (cond[i] * a[0] * exp( a[1] * cond[i]));
349:     }
350:     else
351:       return cond[i] * a[0]; /* set a[1] virtually to zero */
352:   }
353: }
354:
355: /*-------------------------------------------------------------
356:  *  fgauss2()
357:  *  f(x|a) = a1 * exp( a3 * (x-a2)^2) +
358:  *           a4 * exp( a6 * (x-a5)^2)
359:  *-------------------------------------------------------------*/
360: double
361: fgauss2( int i, double *cond, double *a)
362: {
363:   double tmp1 = cond[i] - a[1];
364:   double tmp2 = cond[i] - a[4];
365:   return (a[0] * exp( a[2] * tmp1 * tmp1) +
366:           a[3] * exp( a[5] * tmp2 * tmp2) );
367: }
368:
369: /*-------------------------------------------------------------
370:  *  fgauss1()
371:  *  f(x|a) = a1 * exp( a3 * (x-a2)^2)
372:  *-------------------------------------------------------------*/
373: double
374: fgauss1( int i, double *cond, double *a)
375: {
376:   double tmp1 = cond[i] - a[1];
377:   return a[0] * exp( a[2] * tmp1 * tmp1);
378: }
379:
380: /*-------------------------------------------------------------
381:  *  fgauss_deriv()
382:  *  f(x|a) = a1 * exp( a3 * (x-a2)^2)
383:  *-------------------------------------------------------------*/
384: double
385: fgauss_deriv( double (*funct)(int,double*,double*),
386:          int i, int j, int M, double *cond, double *a)
387: {
388:   double tmp1 = cond[i] - a[1];
389:
390:   if (a[1] < 0) /* limitation of parameter space */
391:     a[1] = 1E-10;
392:   if (a[2] > 0) /* limitation of parameter space */
393:     a[2] = -1E-10;
394:   if (j==0)
395:     /*     y = a1 * exp( a3 * (x-a2)^2)
396:      *dy/da1 =      exp( a3 * (x-a2)^2)
397:      */
398:     return exp( a[2] * tmp1 * tmp1);
399:   else  if (j==1)
400:     /*     y = a1 * exp( a3 * (x-a2)^2)
401:      *dy/da2 =-a1 * exp( a3 * (x-a2)^2) * a3 * 2 * (x-a2)
402:      */
403:     return -a[0] * exp( a[2] * tmp1 * tmp1) * a[2] *
404:              2 * tmp1;
405:   else
406:     /*     y = a1 * exp( a3 * (x-a2)^2)
407:      *dy/da3 = a1 * exp( a3 * (x-a2)^2) * (x-a2)^2
408:      */
409:     return  a[0] * exp( a[2] * tmp1 * tmp1) * tmp1*tmp1;
410: }
411:
412: /*-------------------------------------------------------------
413:  *  fgauss1_deriv2()
414:  *  f(x|a) = a1 * exp( a3 * (x-a2)^2)
415:  * see also http://www.mathetools.de/differenzieren/
416:  *-------------------------------------------------------------*/
417: double
418: fgauss_deriv2( double (*funct)(int,double*,double*),
419:          int j, int k, int M, int i, double *cond, double *a)
420: {
421:   double tmp1 = cond[i] - a[1];
422:
423:   if (a[2] > 0) /* limitation of parameter space */
424:     a[2] = -1E-10;
425:
426:   if (j==0)
427:   {
428:     if (k == 0)
429:       /*     y = a1 * exp( a3 * (x-a2)^2)
430:        *dy/da1 =      exp( a3 * (x-a2)^2)
431:        *d2y/d2a1 =      0
432:        */
433:       return 0;
434:     else if (k == 1)
435:       /*     y = a1 * exp( a3 * (x-a2)^2)
436:        *dy/da1 =      exp( a3 * (x-a2)^2)
437:        *d2y/da1da2 = -exp( a3 * (x-a2)^2) * a3 * 2 * (x-a2)
438:        */
439:       return -exp( a[2] * tmp1*tmp1) * a[2] * 2 * tmp1;
440:     else
441:       /*     y = a1 * exp( a3 * (x-a2)^2)
442:        *dy/da1 =      exp( a3 * (x-a2)^2)
443:        *d2y/da1da3 =  exp( a3 * (x-a2)^2) * (x-a2)^2
444:        */
445:       return  exp( a[2] * tmp1*tmp1) * tmp1* tmp1;
446:   }
447:   else  if (j==1)
448:   {
449:     if (k == 0)
450:       /*     y = a1 * exp( a3 * (x-a2)^2)
451:        *dy/da2 =-a1 * exp( a3 * (x-a2)^2) * a3 * 2 * (x-a2)
452:        *dy/da2da1 =-exp( a3 * (x-a2)^2) * a3 * 2 * (x-a2)
453:        */
454:       return -exp( a[2] * tmp1*tmp1) * a[2] * 2 * tmp1;
455:     else if (k == 1)
456:       /*     y = a1 * exp( a3 * (x-a2)^2)
457:        *dy/da2 =-a1 * exp( a3 * (x-a2)^2) * a3 * 2 * (x-a2)
458:        *dy/da2 =-2*a1*a3 * exp( a3 * (x-a2)^2)  *   (x-a2)
459:        *dy/d2a2=-2*a1*a3 * exp( a3 * (x-a2)^2)* (-1) +
460:                 2*a1*a3 * exp( a3 * (x-a2)^2)* a3* 2*(x-a2) * (x-a2)
461:        *dy/d2a2= 4*a1*a3^2 * exp( a3 * (x-a2)^2)*(x-a2)^2 +
462:        *          2*a1*a3 * exp( a3 * (x-a2)^2)
463:        */
464:       return 4*a[0]*a[2]*a[2] * exp( a[2] * tmp1*tmp1)*tmp1*tmp1 +
465:                2*a[0]*a[2] * exp( a[2] * tmp1*tmp1);
466:     else
467:       /*     y = a1 * exp( a3 * (x-a2)^2)
468:        *dy/da2 =-2*a1*a3 * exp( a3 * (x-a2)^2)  *   (x-a2)
469:        *dy/da2da3 =-2*a1*(x-a2)^3* a3 * exp( a3 * (x-a2)^2) -
470:        *            2 * a1* (x-a2)*exp( a3 * (x-a2)^2)
471:        */
472:       return 2*a[0] * tmp1*tmp1*tmp1 * a[2] * exp( a[2] * tmp1*tmp1)
473:                    -2 * a[0]* tmp1 * exp( a[2] * tmp1*tmp1);
474:   }
475:   else
476:   {
477:     if (k == 0)
```

```
478:       /*     y = a1 * exp( a3 * (x-a2)^2)
479:        *dy/da3 = a1 * exp( a3 * (x-a2)^2) * (x-a2)^2
480:        *dy/da3da1 = exp( a3 * (x-a2)^2) * (x-a2)^2
481:        */
482:       return  exp( a[2] * tmp1*tmp1) * tmp1* tmp1;
483:     else if (k == 1)
484:       /*     y = a1 * exp( a3 * (x-a2)^2)
485:        *dy/da3 = a1 * exp( a3 * (x-a2)^2) * (x-a2)^2
486:        *dy/da2da3 =-2*a1*(x-a2)^3* a3 * exp( a3 * (x-a2)^2) -
487:        *            2 * a1* (x-a2)*exp( a3 * (x-a2)^2)
488:        */
489:       return 2*a[0] * tmp1*tmp1*tmp1 * a[2] * exp( a[2] * tmp1*tmp1)
490:                   -2 * a[0]* tmp1 * exp( a[2] * tmp1*tmp1);
491:     else
492:       /*     y = a1 * exp( a3 * (x-a2)^2)
493:        *dy/da3 = a1 * exp( a3 * (x-a2)^2) * (x-a2)^2
494:        *dy/d2a3= a1 * exp( a3 * (x-a2)^2) * (x-a2)^4
495:        */
496:       return a[0] * exp( a[2] * tmp1*tmp1) * tmp1*tmp1 * tmp1*tmp1;
497:   }
498: }
499:
500: /*-------------------------------------------------------------
501:  *  fcircleTLS()
502:  *  f(x|a) = 0 = (sqrt[(x1-a1)^2 + (x2-a2)^2^] - a3)^2
503:  *-------------------------------------------------------------*/
504: double
505: fcircleTLS( int i, double *cond, double *a)
506: {
507:   double tmp1, tmp2, d;
508:   /* two conditions per measurement */
509:   tmp1 = cond[2*i  ] - a[0];
510:   tmp2 = cond[2*i+1] - a[1];
511:   d = sqrt(tmp1*tmp1 + tmp2*tmp2) - a[2];
512:   return d;
513: }
514:
515: /*-------------------------------------------------------------
516:  *  fcircleTLS_deriv()
517:  *  f(x|a) = 0 = (sqrt[(x1-a1)^2 + (x2-a2)^2^] - a3)^2
518:  *-------------------------------------------------------------*/
519: double
520: fcircleTLS_deriv( double (*funct)(int,double*,double*),
521:         int i, int j, int M, double *cond, double *a)
522: {
523:   double b, tmp1, tmp2;
524:     tmp1 = a[0] - cond[2*i  ];
525:     tmp2 = a[1] - cond[2*i+1];
526:     b = sqrt(tmp1*tmp1 + tmp2*tmp2);
527:     if (j==0)
528:       return tmp1 / b;
529:     else  if (j==1)
530:       return tmp2 / b;
531:     else
532:       return  (-1);
533: }
534: /*-------------------------------------------------------------
535:  *  fcircle()
536:  *  f(x|a) = 0 = (x1-a1)^2 + (x2-a2)^2 - a3^2
537:  *-------------------------------------------------------------*/
538: double
539: fcircle( int i, double *cond, double *a)
540: {
541:   double tmp1, tmp2;
542:   /* two conditions per measurement */
543:   tmp1 = cond[2*i  ] - a[0];
544:   tmp2 = cond[2*i+1] - a[1];
545:
546:   return tmp1*tmp1 + tmp2*tmp2 - a[2]*a[2];
547: }
548:
549: /*-------------------------------------------------------------
550:  *  fcircle_deriv()
551:  * f(x|a) = 0 = (x-a1)^2 + (x2-a2)^2 - a3^2
552:  *-------------------------------------------------------------*/
553: double
554: fcircle_deriv( double (*funct)(int,double*,double*),
555:         int i, int j, int M, double *cond, double *a)
556: {
557:     if (j==0)
558:       return 2*(a[0] - cond[2*i]);
559:     else  if (j==1)
560:       return 2*(a[1] - cond[2*i+1]);
561:     else
562:       return  -2 * a[2];
563: }
564:
565: /*-------------------------------------------------------------
566:  *  fcirclelin_deriv()
567:  * f(x|b) = x1^2 + x2^2 = b1*x1 + b2*x2 - b3
568:  *-------------------------------------------------------------*/
569: double
570: fcirclelin_deriv( double (*funct)(int,double*,double*),
571:         int i, int j, int M, double *cond, double *a)
572: {
573:     if (j==0)
574:       return cond[2*i];
575:     else  if (j==1)
576:       return cond[2*i+1];
577:     else
578:       return  -1.;
579: }
580:
581: /*-------------------------------------------------------------
582:  *  frotation()
583:  * 21... x = f1(u|a) = a1 + cos(a3) * u - sin(a3) * v
584:  *       y = f2(u|a) = a2 + sin(a3) * u + cos(a3) * v
585:  *-------------------------------------------------------------*/
586: double
587: frotation( int i, double *cond, double *a)
588: {
589:   if (i%2 == 0)
590:   {
591:     /* equation for x */
592:     return a[0] + cos(a[2]) * cond[i] - sin(a[2]) * cond[i+1];
593:   }
594:   else
595:   {
596:     /* equation for y */
597:     return a[1] + sin(a[2]) * cond[i-1] + cos(a[2]) * cond[i];
598:   }
599: }
600: /*-------------------------------------------------------------
601:  *  frotation_deriv()
602:  * 21... x = f1(u|a) = a1 + cos(a3) * u - sin(a3) * v
603:  *       y = f2(u|a) = a2 + sin(a3) * u + cos(a3) * v
604:  *-------------------------------------------------------------*/
605: double
606: frotation_deriv( double (*funct)(int,double*,double*),
607:         int i, int j, int M, double *cond, double *a)
608: {
609:   if (i%2 == 0)
610:   {
611:     if (j==0)
612:       return 1;
613:     else  if (j==1)
614:       return 0;
615:     else
616:       return  - cond[i] * sin(a[2]) - cond[i+1] * cos(a[2]);
617:   }
618:   else
619:   {
620:     if (j==0)
621:       return 0;
622:     else  if (j==1)
623:       return 1;
624:     else
625:       return  cond[i-1] * cos(a[2]) - cond[i] * sin(a[2]);
626:   }
627: }
628:
629:
630: /*-------------------------------------------------------------
631:  *  fpolynom2_deriv()
632:  * f(x|a) = a1 + a2 * x + a3 * x*x
633:  *-------------------------------------------------------------*/
634: double
635: fpolynom2_deriv( double (*funct) (int,double*,double*),
636:             int i, int j, int M, double *cond, double *a)
637: {
638:   if (j == 0)
639:     return 1.;
640:   else if (j == 1)
641:   {
642:     return cond[i];
643:   }
644:   else
645:   {
646:     return cond[i] * cond[i];
647:   }
648: }
649:
650:
651: /*-------------------------------------------------------------
652:  *  fpolynom3_deriv()
653:  * f(x|a) = a1 + a2 * x + a3 * x*x + a4 * x*x*x
654:  *-------------------------------------------------------------*/
655: double
656: fpolynom3_deriv( double (*funct) (int,double*,double*),
657:             int i, int j, int M, double *cond, double *a)
658: {
659:   if (j == 0)
660:     return 1.;
661:   else if (j == 1)
662:   {
663:     return cond[i];
664:   }
665:   else if (j == 2)
666:   {
667:     return cond[i] * cond[i];
668:   }
669:   else
```

```
670:   {
671:     return cond[i] * cond[i] * cond[i];
672:   }
673: }
674:
675: /*--------------------------------------------------------------
676:  *  fquadsurface_deriv()
677:  * f(x|a) = a1 + a2*x1 + a3*x1^2 + a4*x2 + a5*x2^2
678:  *--------------------------------------------------------------*/
679: double
680: fquadsurface_deriv( double (*funct) (int,double*,double*),
681:             int i, int j, int M, double *cond, double *a)
682: {
683:   if (j == 0)
684:     return 1.;
685:   else if (j == 1)
686:   {
687:     return cond[2*i];
688:   }
689:   else if (j == 2)
690:   {
691:     return cond[2*i] * cond[2*i];
692:   }
693:   else if (j == 3)
694:   {
695:     return cond[2*i+1];
696:   }
697:   else
698:   {
699:     return cond[2*i+1] * cond[2*i+1];
700:   }
701: }
702:
703: /*--------------------------------------------------------------
704:  *  fNN_3_3()
705:  *--------------------------------------------------------------*/
706: double
707: fNN_3_3( int i, double *cond, double *a)
708: {
709:   double h1, h2, h3;
710:   double arg1, arg2, arg3;
711:
712:   /* 1st hidden neuron */
713:   arg1 = a[0] + a[1] * cond[3*i]
714:     + a[2] * cond[3*i+1]
715:     + a[3] * cond[3*i+2];
716:   h1 = tanh( arg1);
717:
718:   /* 2nd hidden neuron */
719:   arg2 = a[4] + a[5] * cond[3*i]
720:     + a[6] * cond[3*i+1]
721:     + a[7] * cond[3*i+2];
722:   h2 = tanh( arg2);
723:
724:   /* 3rd hidden neuron */
725:   arg3 = a[8] + a[9] * cond[3*i]
726:     + a[10] * cond[3*i+1]
727:     + a[11] * cond[3*i+2];
728:   h3 = tanh( arg3);
729:
730:   /* output neuron */
731:   return a[12] * h1 + a[13] * h2 + a[14] * h3;
732: }
733:
734: /*--------------------------------------------------------------
735:  *  fNN_3_3_sigmoid(), obsolete
736:  *--------------------------------------------------------------*/
737: double
738: fNN_3_3_sigmoid( int i, double *cond, double *a)
739: {
740:   double h1, h2, h3;
741:   double arg1, arg2, arg3;
742:
743:   /* 1st hidden neuron */
744:   arg1 = a[0] + a[1] * cond[3*i]
745:     + a[2] * cond[3*i+1]
746:     + a[3] * cond[3*i+2];
747:   h1 = 2. / (1. + exp( -arg1)) - 1;
748:
749:   /* 2nd hidden neuron */
750:   arg2 = a[4] + a[5] * cond[3*i]
751:     + a[6] * cond[3*i+1]
752:     + a[7] * cond[3*i+2];
753:   h2 = 2. / (1. + exp( -arg2)) - 1;
754:
755:   /* 3rd hidden neuron */
756:   arg3 = a[8] + a[9] * cond[3*i]
757:     + a[10] * cond[3*i+1]
758:     + a[11] * cond[3*i+2];
759:   h3 = 2. / (1. + exp( -arg3)) - 1;
760:
761:   /* output neuron */
762:   return a[12] * h1 + a[13] * h2 + a[14] * h3 + a[15];
763: }
764:
765: /*--------------------------------------------------------------
766:  *  fNN_3_2()
767:  * f(x|a) =
768:  *--------------------------------------------------------------*/
769: double
770: fNN_3_2( int i, double *cond, double *a)
771: {
772:   double h1, h2;
773:   double arg1, arg2;
774:
775:   /* 1st hidden neuron */
776:   arg1 = a[0] + a[1] * cond[3*i]
777:               + a[2] * cond[3*i+1]
778:               + a[3] * cond[3*i+2];
779:   if (arg1 < 0) arg1 = 0;
780:   h1 = 2. / (1. + exp( -arg1)) - 1;
781:
782:   /* 2nd hidden neuron */
783:   arg2 = a[4] + a[5] * cond[3*i]
784:               + a[6] * cond[3*i+1]
785:               + a[7] * cond[3*i+2];
786:   if (arg2 < 0) arg2 = 0;
787:   h2 = 2. / (1. + exp( -arg2)) - 1;
788:
789:   /* output neuron */
790:   return a[8] * h1 + a[9] * h2 + a[10];
791: }
792:
793: /*--------------------------------------------------------------
794:  *  fNN_2_2()
795:  * f(x|a) =
796:  *--------------------------------------------------------------*/
797: double
798: fNN_2_2( int i, double *cond, double *a)
799: {
800:   double h1, h2;
801:   double arg1, arg2;
802:
803:   /* 1st hidden neuron */
804:   arg1 = a[0] + a[1] * cond[2*i]
805:     + a[2] * cond[2*i+1];
806:   h1 = 2. / (1. + exp( -arg1)) - 1;
807:
808:   /* 2nd hidden neuron */
809:   arg2 = a[3] + a[4] * cond[2*i]
810:     + a[5] * cond[2*i+1];
811:   h2 = 2. / (1. + exp( -arg2)) - 1;
812:
813:   /* output neuron */
814:   return a[6] * h1 + a[7] * h2 + a[8];
815: }
816:
817: /*--------------------------------------------------------------
818:  *  fNN_1_2()
819:  * f(x|a) =
820:  *--------------------------------------------------------------*/
821: double
822: fNN_1_2( int i, double *cond, double *a)
823: {
824:   double h1, h2;
825:   double arg1, arg2;
826:
827:   /* 1st hidden neuron */
828:   arg1 = a[0] + a[1] * cond[i];
829:   /*h1 = 2. / (1. + exp( -arg1)) - 1;*/
830:   h1 = tanh( arg1);
831:
832:   /* 2nd hidden neuron */
833:   arg2 = a[2] + a[3] * cond[i];
834:   /* h2 = 2. / (1. + exp( -arg2)) - 1; */
835:   h2 = tanh( arg2);
836:
837:   /* output neuron */
838:   return a[4] * h1 + a[5] * h2 + a[6];
839: }
840:
841: /*--------------------------------------------------------------
842:  *  fNN_1_3()
843:  * f(x|a) =
844:  *--------------------------------------------------------------*/
845: double
846: fNN_1_3( int i, double *cond, double *a)
847: {
848:   double h1, h2, h3;
849:   double arg;
850:
851:   /* 1st hidden neuron */
852:   arg = a[0] + a[1] * cond[i];
853:   h1 = tanh( arg);
854:   /* h1 = 2. / (1. + exp( -arg)) - 1; */
855:
856:   /* 2nd hidden neuron */
857:   arg = a[2] + a[3] * cond[i];
858:   h2 = tanh( arg);
859:   /* h2 = 2. / (1. + exp( -arg)) - 1; */
860:
861:   /* 3rd hidden neuron */
```

```
862:   arg = a[4] + a[5] * cond[i];
863:   h3 = tanh( arg);
864:   /* h3 = 2. / (1. + exp( -arg)) - 1; */
865:
866:   /* output neuron */
867:   return a[6] * h1 + a[7] * h2 + a[8] * h2 + a[9];
868: }
869:
870: /*-------------------------------------------------------------
871:  *  f_deriv()
872:  *  numerical derivation, used for several model functions
873:  *-------------------------------------------------------------*/
874: double
875: f_deriv( double (*funct)(int,double*,double*),
876:       int i, int j, int M, double *cond, double *a)
877: {
878:   double tmp1, tmp2, atmp[M_MAX], ajp, ajm, C;
879:   double del;
880:
881:   /* inspect positions close to current one */
882:   if ( a[j] != 0.0)
883:   {
884:     C = 1000000;
885:     del = a[j]/C;
886:   }
887:   else
888:   {
889:     del = 0.001;
890:   }
891:   ajp = a[j] + del; /* plus a bit of current parameter value*/
892:   ajm = a[j] - del; /* minus % */
893:   /* create modified parameter vector,
894:    * copy maximum number of parameters
895:    */
896:   /* copy all possible parameters, needed for POLYNOMIAL_REG  */
897:   memcpy( atmp, a, sizeof(double) * M_MAX);
898:   atmp[j] = ajp;
899:   /* look, what result is at modified position */
900:   tmp1 = funct( i, cond, atmp);
901:   atmp[j] = ajm;
902:   tmp2 = funct( i, cond, atmp);
903:
904:   /* compute gradient */
905:   return tmp1 / (2*del) - tmp2 / (2*del);
906: }
907:
```

```
  0: /*****************************************************************
  1:  *
  2:  * File........: functions.h
  3:  * Function....: proto typing for functions.c
  4:  * Author......: Tilo Strutz
  5:  * last changes: 25.09.2009, 06.11.2009, 18.02.2010, 03.01.2011
  6:  *
  7:  * LICENCE DETAILS: see software manual
  8:  *  free academic use
  9:  *  cite source as
 10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 11:  *  2nd edition 2015"
 12:  *
 13:  *****************************************************************/
 14:
 15: #ifndef FUNCT_H
 16: #define FUNCT_H
 17:
 18: /* linear functions */
 19: double fconstant_deriv( double (*funct)(int,double*,double*),
 20:               int i, int j, int M, double *cond,  double *a);
 21: double flin_deriv( double (*funct)(int,double*,double*),
 22:               int i, int j, int M, double *cond, double *a);
 23: double flin2_deriv( double (*funct)(int,double*,double*),
 24:               int i, int j, int M, double *cond, double *a);
 25: double fcosine_deriv( double (*funct)(int,double*,double*),
 26:               int i, int j, int M, double *cond,  double *a);
 27: double fprediction_deriv( double (*funct)(int,double*,double*),
 28:               int i, int j, int M, double *cond, double *a);
 29: double fpolynom2_deriv( double (*funct)(int,double*,double*),
 30:               int i, int j, int M, double *cond, double *a);
 31: double fpolynom3_deriv( double (*funct)(int,double*,double*),
 32:               int i, int j, int M, double *cond, double *a);
 33: double fpolynomial_deriv( double (*funct)(int,double*,double*),
 34:               int i, int j, int M, double *cond, double *a);
 35: double fquadsurface_deriv( double (*funct)(int,double*,double*),
 36:               int i, int j, int M, double *cond, double *a);
 37:
 38: /* nonlinear functions */
 39: double fpolynomial( int i, double *cond, double *a);
 40: int init_polynomial( int N, double *obs, double *cond,
 41:               double *a, unsigned char *a_flag, FILE *logfile);
 42:
 43: double fcosine_nonlin( int i, double *cond, double *a);
 44: double fcosine_nonlin_deriv( double (*funct)(int,double*,double*),
 45:           int i, int j, int M, double *cond, double *a);
 46: int init_cosine_nonlin( int N, double *obs, double *cond,
 47:               double *a, unsigned char *a_flag, FILE *logfile);
 48:
 49: double fcosine_trend( int i, double *cond, double *a);
 50: double fcosine_trend_deriv( double (*funct)(int,double*,double*),
 51:           int i, int j, int M, double *cond, double *a);
 52: int init_cosine_trend( int N, double *obs, double *cond,
 53:            double *a, unsigned char *a_flag, FILE *logfile);
 54:
 55: double ftrigonometric1( int i, double *cond, double *a);
 56: int init_trigonometric1( int N, double *obs, double *cond,
 57:            double *a, unsigned char *a_flag, FILE *logfile);
 58:
 59: double ftrigonometric2( int i, double *cond, double *a);
 60: int init_trigonometric2( int N, double *obs, double *cond,
 61:            double *a, unsigned char *a_flag, FILE *logfile);
 62:
 63: double flogarithmic( int i, double *cond, double *a);
 64: double flogarithmic_deriv( double (*funct)(int,double*,double*),
 65:             int i, int j, int M, double *cond, double *a);
 66: double flogarithmic_deriv2( double (*funct)(int,double*,double*),
 67:             int j, int k, int M, int i, double *cond, double *a);
 68: int init_logarithmic( int N, double *obs, double *cond,
 69:            double *a, unsigned char *a_flag, FILE *logfile);
 70:
 71: double fexponential( int i, double *cond, double *a);
 72: double fexponential_deriv( double (*funct)(int,double*,double*),
 73:              int i, int j, int M, double *cond, double *a);
 74: int init_exponential( int N, double *obs, double *cond,
 75:             double *a, unsigned char *a_flag, FILE *out);
 76:
 77: double fexpon2( int i, double *cond, double *a);
 78: int init_expon2( int N, double *obs, double *cond,
 79:             double *a, unsigned char *a_flag, FILE *out);
 80: double fgen_laplace( int i, double *cond, double *a);
 81: double fgen_laplace_deriv( double (*funct)(int,double*,double*),
 82:             int i, int j, int M, double *cond, double *a);
 83: int init_gen_laplace( int N, double *obs, double *cond,
 84:             double *a, unsigned char *a_flag, FILE *out);
 85: double fexpon2_deriv( double (*funct)(int,double*,double*),
 86:               int i, int j, int M, double *cond, double *a);
 87:
 88: double fgauss2( int i, double *cond, double *a);
 89: double fgauss1( int i, double *cond, double *a);
 90:
 91: int init_gauss2( int N, double *obs, double *cond,
 92:             double *a, unsigned char *a_flag, FILE *out);
 93: int init_gauss1( int N, double *obs, double *cond,double *a,
 94:           unsigned char *a_flag, int peak_flag, FILE *out);
 95:
 96: double fgauss_deriv( double (*funct)(int,double*,double*),
 97:         int i, int j, int M, double *cond, double *a);
 98: double fgauss_deriv2( double (*funct)(int,double*,double*),
 99:         int j, int k, int M, int i, double *cond, double *a);
100: int init_gauss( int N, double *obs, double *cond,double *a,
101:           unsigned char *a_flag, FILE *out);
102:
103: double frotation( int i, double *cond, double *a);
104: double
105: frotation_deriv( double (*funct)(int,double*,double*),
106:         int i, int j, int M, double *cond, double *a);
107: int init_rotation( int N, double *obs, double *cond,
108:               double *a, unsigned char *a_flag, FILE *logfile);
109:
110: double fcircleTLS( int i, double *cond, double *a);
111: double fcircleTLS_deriv( double (*funct)(int,double*,double*),
112:         int i, int j, int M, double *cond, double *a);
113: double fcircle( int i, double *cond, double *a);
114: double fcircle_deriv( double (*funct)(int,double*,double*),
115:              int i, int j, int M, double *cond, double *a);
116: int init_circle( int N, double *obs, double *cond,
117:                 double *a, unsigned char *a_flag, FILE *logfile);
118:
119: double fcirclelin_deriv( double (*funct)(int,double*,double*),
120:         int i, int j, int M, double *cond, double *a);
121: int init_circlelin( int N, double *obs, double *cond,
122:   double *a, unsigned char *a_flag, FILE *logfile);
123:
124: double fNN_3_3( int i, double *cond, double *a);
125: int init_NN3x3x1( int N, double *obs, double *cond,
126:             double *a, unsigned char *a_flag, FILE *logfile);
127: int init_NN( int N, double *obs, double *cond,
128:             double *a, unsigned char *a_flag, FILE *logfile);
129: double fNN_3_2( int i, double *cond, double *a);
130: double fNN_2_2( int i, double *cond, double *a);
131: double fNN_1_2( int i, double *cond, double *a);
132: double fNN_1_3( int i, double *cond, double *a);
133: int init_NN1x3x1( int N, double *obs, double *cond,
134:               double *a, unsigned char *a_flag, FILE *logfile);
135:
136: /* common numerical derivation function for all
137:  * nonlinear problems
138:  */
139: double f_deriv( double (*funct)(int,double*,double*),
140:             int i, int j, int M, double *cond, double *a);
141:
142: #endif
```

```
 0:
 1: /****************************************************************
 2:  *
 3:  * File....: functions.c
 4:  * Function: model functions and their derivatives
 5:  * Author..: Tilo Strutz
 6:  * Date....: 23.09.2007
 7:  *
 8:  * LICENCE DETAILS: see software manual
 9:  *  free academic use
10:  *  cite source as
11:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
12:  *  2nd edition 2015"
13:  *
14:  ****************************************************************/
15: #include <stdio.h>
16: #include <stdlib.h>
17: #include <string.h>
18: #include <math.h>
19: #include "macros.h"
20: #include "functions.h"
21:
22: #define DEG2RAD M_PI/180.
23:
24: /*-------------------------------------------------------------
25:  *  fNIST_Eckerle4()
26:  * f(x|a) = a1 / a2 * exp(-0.5*((x -a3)/ a2)^2)
27:  *-------------------------------------------------------------*/
28: double
29: fNIST_Eckerle4( int i, double *cond, double *a)
30: {
31:   double x;
32:   x = (cond[i] - a[2]) / a[1];
33:   return (a[0] / a[1]) * exp( -0.5*x*x);
34: }
35:
36: /*-------------------------------------------------------------
37:  *  fNIST_Eckerle4_deriv()
38:  * f(x|a) = a1 / a2 * exp(-0.5*((x -a3)/ a2)^2)
39:  *-------------------------------------------------------------*/
40: double
41: fNIST_Eckerle4_deriv( double (*funct)(int,double*,double*),
42:            int i, int j, int M, double *cond, double *a)
43: {
44:   double x;
45:   x = (cond[i] - a[2]) / a[1];
46:   if (j == 0)
47:     /* y      = a0 / a1 * exp(-0.5*((x -a2)/ a1)^2) */
48:     return   exp(-0.5*x*x) / a[1];
49:   else if (j == 1)
50:     return a[0] * exp(-0.5*x*x) / (a[1]*a[1]) * (x*x - 1);
51:   else
52:     return a[0] * exp(-0.5*x*x) * (cond[i] - a[2]) /
53:                       (a[1]*a[1]*a[1]);
54: }
55:
56: /*-------------------------------------------------------------
57:  *  fNIST_Rat43()
58:  * f(x|a) = a1 / [1 + exp(a2 - a3*x)]^(1/a4)
59:  *-------------------------------------------------------------*/
60: double
61: fNIST_Rat43( int i, double *cond, double *a)
62: {
63:   double x;
64:   x =  exp( a[1] - a[2]*cond[i]);
65:   return a[0] / pow(1 + x, 1/a[3]);
66: }
67:
68: /*-------------------------------------------------------------
69:  *  fNIST_Rat43_deriv()
70:  * f(x|a) = a1 / [1 + exp(a2 - a3*x)]^(1/a4)
71:  *-------------------------------------------------------------*/
72: double
73: fNIST_Rat43_deriv( double (*funct)(int,double*,double*),
74:            int i, int j, int M, double *cond, double *a)
75: {
76:   double x;
77:   x =  exp( a[1] - a[2]*cond[i]);
78:   /*  y = a0 / [1 + exp( a1-a2*x)]^(1/a3) */
79:   if (j == 0)
80:     return 1. / pow(1 + x, 1/a[3]);
81:   else if (j == 1)
82:     return -a[0] * x * pow( (x+1), -1/a[3]-1 ) / a[3];
83:   else if (j == 2)
84:     return a[0]*cond[i] * x * pow( (x+1), -1/a[3]-1 ) / a[3];
85:   else
86:     return a[0] * log( x+1) / (pow( (x+1), 1/a[3]) *a[3]*a[3]);
87: }
88:
89: /*-------------------------------------------------------------
90:  *  fNIST_Rat42()
91:  * f(x|a) = a1 / (1 + exp(a2 - a3*x))
92:  *-------------------------------------------------------------*/
93: double
94: fNIST_Rat42( int i, double *cond, double *a)
95: {
96:   return a[0] / (1 + exp( a[1] - a[2]*cond[i]) );
97: }
98:
99: /*-------------------------------------------------------------
100:  *  fNIST_Rat42_deriv()
101:  * f(x|a) = a1 / (1 + exp(a2 - a3*x))
102:  *-------------------------------------------------------------*/
103: double
104: fNIST_Rat42_deriv( double (*funct)(int,double*,double*),
105:            int i, int j, int M, double *cond, double *a)
106: {
107:   /*  y = a0 / (1 + exp( a1 - a2*x)) */
108:   if (j == 0)
109:       return 1. / (1 + exp( a[1] - a[2]*cond[i]) );
110:   else if (j == 1)
111:       return -a[0] * exp( a[1] - a[2]*cond[i]) /
112:           pow( (1 + exp( a[1] - a[2]*cond[i])), 2. );
113:   else
114:       return a[0] * cond[i] * exp( a[1] - a[2]*cond[i]) /
115:           pow( (1 + exp( a[1] - a[2]*cond[i])), 2. );
116: }
117:
118: /*-------------------------------------------------------------
119:  *  fNIST_thurber()
120:  * f(x|a) =(a1 + a2*x + a3*x**2 + a4*x**3) /
121:  *          (1 + a5*x + a6*x**2 + a7*x**3)
122:  *-------------------------------------------------------------*/
123: double
124: fNIST_thurber( int i, double *cond, double *a)
125: {
126:   double x2, x3;
127:   x2 = cond[i]*cond[i];
128:   x3 = x2*cond[i];
129:   return (a[0] + a[1]*cond[i] + a[2]*x2 + a[3]*x3) /
130:            (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3);
131: }
132:
133: /*-------------------------------------------------------------
134:  *  fNIST_thurber_deriv()
135:  *   f(x|a) = a1 * exp( a2 / (x+a3))
136:  *-------------------------------------------------------------*/
137: double
138: fNIST_thurber_deriv( double (*funct)(int,double*,double*),
139:            int i, int j, int M, double *cond, double *a)
140: {
141:   double x2, x3;
142:   x2 = cond[i]*cond[i];
143:   x3 = x2*cond[i];
144:   if (j == 0)
145:     /* y      = (a0 + a1*x + a2*x**2 + a3*x**3) /
146:                   (1 + a4*x + a5*x**2 + a6*x**3) */
147:     return 1. / (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3);
148:   else if (j == 1)
149:     return cond[i] / (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3);
150:   else if (j == 2)
151:     return      x2 / (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3);
152:   else if (j == 3)
153:     return      x3 / (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3);
154:   else if (j == 4)
155:     return -cond[i] * (a[0] + a[1]*cond[i] + a[2]*x2 + a[3]*x3) /
156:                  pow( (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3), 2.);
157:   else if (j == 5)
158:     return      -x2 * (a[0] + a[1]*cond[i] + a[2]*x2 + a[3]*x3) /
159:                  pow( (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3), 2.);
160:   else
161:     return      -x3 * (a[0] + a[1]*cond[i] + a[2]*x2 + a[3]*x3) /
162:                  pow( (1 + a[4]*cond[i] + a[5]*x2 + a[6]*x3), 2.);
163: }
164:
165: /*-------------------------------------------------------------
166:  *  fNIST_MGH09()
167:  *   f(x|a) =a1 * (x**2 + a2*x) / (x*x + a3*x + a4)
168:  *-------------------------------------------------------------*/
169: double
170: fNIST_MGH09( int i, double *cond, double *a)
171: {
172:   double x2;
173:   x2 = cond[i]*cond[i];
174:   return a[0] * (x2 + a[1]*cond[i]) /
175:                 (x2 + a[2]*cond[i] + a[3]);
176: }
177:
178: /*-------------------------------------------------------------
179:  *  fNIST_MGH09_deriv()
180:  *   f(x|a) =a1 * (x**2 + a2*x) / (x*x + a3*x + a4)
181:  *-------------------------------------------------------------*/
182: double
183: fNIST_MGH09_deriv( double (*funct)(int,double*,double*),
184:            int i, int j, int M, double *cond, double *a)
185: {
186:   double x2;
187:   x2 = cond[i]*cond[i];
188:   if (j == 0)
189:     /* y      = a0 * (x**2 + a1*x) / (x*x + a2*x + a3)  */
190:     /* dy/da0 = (x**2 + a1*x) / (x*x + a2*x + a3) */
191:     return (x2 + a[1]*cond[i]) / (x2 + a[2]*cond[i] + a[3]);
```

```
  else  if (j == 1)
    return a[0] * cond[i] / (x2 + a[2]*cond[i] + a[3]);
  else  if (j == 2)
    return -a[0] * (x2 + a[1]*cond[i])*cond[i] /
                pow( (x2 + a[2]*cond[i] + a[3]), 2.);
  else
    return -a[0] * (x2 + a[1]*cond[i]) /
                pow( (x2 + a[2]*cond[i] + a[3]), 2.);
}

/*-------------------------------------------------------------
 *  fNIST_MGH10()
 *   f(x|a) = a1 * exp( a2 / (x+a3))
 *-------------------------------------------------------------*/
double
fNIST_MGH10( int i, double *cond, double *a)
{
  return a[0] * exp( a[1] / (cond[i] + a[2]));
}

/*-------------------------------------------------------------
 *  fNIST_MGH10_deriv()
 *   f(x|a) = a1 * exp( a2 / (x+a3))
 *-------------------------------------------------------------*/
double
fNIST_MGH10_deriv( double (*funct)(int,double*,double*),
            int i, int j, int M, double *cond, double *a)
{
  if (j == 0)
    /* y      = a0 * exp( a1 / (x+a2))  */
    /* dy/da0 =      exp( a1 / (x+a2))  */
    return exp( a[1] / (cond[i] + a[2]));
  else  if (j == 1)
    /* y      = a0 * exp( a1 / (x+a2))  */
    /* dy/da1 = a0 * exp( a1 / (x+a2)) / (x+a2) */
    return a[0] * exp( a[1] / (cond[i] + a[2])) / (cond[i]+ a[2]);
  else
    /* y      = a0 * exp( a1 / (x+a2))  */
    /* dy/da2 =-a0*a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
    return -a[0] * a[1] * exp( a[1] / (cond[i] + a[2])) /
          ((cond[i]+ a[2])*(cond[i]+ a[2]));
}

/*-------------------------------------------------------------
 *  fNIST_MGH10_deriv2()
 *   f(x|a) = a1 * exp( a2 / (x+a3))
 *-------------------------------------------------------------*/
double
fNIST_MGH10_deriv2( double (*funct)(int,double*,double*),
           int j, int k, int M, int i, double *cond, double *a)
{
  /* i ... number of current observation */
  /* j ... derivation with respect to a_j */
  /* k ... derivation with respect to a_k */
  if (j == 0)
  {
    if ( k == 0)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da0 =      exp( a1 / (x+a2))  */
      return 0;
    else if ( k == 1)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da0 =      exp( a1 / (x+a2))  */
      /* d2y/da0da1 =  exp( a1 / (x+a2)) / (x+a2) */
      return exp( a[1] / (cond[i] + a[2])) /
                          (cond[i] + a[2]);
    else
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da0 =      exp( a1 / (x+a2))  */
      /* d2y/da0da2 = -a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
      return  -a[1] * exp( a[1] / (cond[i] + a[2])) /
                  ((cond[i] + a[2])*(cond[i] + a[2]));
  }
  else  if (j == 1)
  {
    if ( k == 0)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da1 = a0 * exp( a1 / (x+a2)) / (x+a2) */
      /* d2y/da1da0 = exp( a1 / (x+a2)) / (x+a2)  */
      return exp( a[1] / (cond[i] + a[2])) /
                          (cond[i] + a[2]);
    else if ( k == 1)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da1 = a0 * exp( a1 / (x+a2)) / (x+a2) */
      /* d2y/d2a1 = a0 * exp( a1 / (x+a2)) / (x+a2)^2 */
      return a[0] * exp( a[1] / (cond[i] + a[2])) /
             ((cond[i] + a[2])*(cond[i] + a[2]));
    else
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da1 = a0 * exp( a1 / (x+a2)) / (x+a2) */
      /* d2y/da1da2 =-a0    * exp( a1 / (x+a2)) / (x+a2)^2 -
                       a0*a1 * exp( a1 / (x+a2)) / (x+a2)^3  */
      /* d2y/da1da2 =-a0 * exp( a1 / (x+a2)) / (x+a2)^2 *
                       (1 + a1 / (x+a2))  */
      return -a[0] * exp( a[1] / (cond[i] + a[2])) /
               ((cond[i] + a[2])*(cond[i] + a[2])) *
          (1 + a[1]/(cond[i]+ a[2]));
  }
  else
  {
    if ( k == 0)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da2 =-a0*a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
      /* d2y/da0da2 = -a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
      return  -a[1] * exp( a[1] / (cond[i] + a[2])) /
                  ((cond[i] + a[2])*(cond[i] + a[2]));
    else if ( k == 1)
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da2 =-a0*a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
      /* d2y/da1da2 =-a0    * exp( a1 / (x+a2)) / (x+a2)^2 -
                       a0*a1 * exp( a1 / (x+a2)) / (x+a2)^3  */
      /* d2y/da1da2 =-a0 * exp( a1 / (x+a2)) / (x+a2)^2 *
                       (1 + a1 / (x+a2))  */
      return -a[0] * exp( a[1] / (cond[i] + a[2])) /
               ((cond[i] + a[2])*(cond[i] + a[2])) *
          (1 + a[1]/(cond[i]+ a[2]));
    else
      /* y      = a0 * exp( a1 / (x+a2))  */
      /* dy/da2 =-a0*a1 * exp( a1 / (x+a2)) / (x+a2)^2  */
      /* d2y/da2da2 =2*a0*a1  * exp( a1 / (x+a2)) / (x+a2)^3 +
                       a0*a1^2 * exp( a1 / (x+a2)) / (x+a2)^4  */
      /* d2y/da2da2 = a0*a1  * exp( a1 / (x+a2)) / (x+a2)^3 *
                       (2 + a1 / (x+a2) */
      return a[0] * a[1] * exp( a[1] / (cond[i] + a[2])) /
          ((cond[i]+ a[2])*(cond[i]+ a[2])*(cond[i]+ a[2])) *
          (2 + a[1] /(cond[i]+ a[2]));
  }
}

/*-------------------------------------------------------------
 *  fNIST_Bennett5()
 *   f(x|a) = a1 * (x+a2)^(-1/a3)
 *-------------------------------------------------------------*/
double
fNIST_Bennett5( int i, double *cond, double *a)
{
  return a[0] * pow( (cond[i] + a[1]), -1./a[2]);
}

/*-------------------------------------------------------------
 *  fNIST_Bennett5_deriv()
 *   f(x|a) = a1 * (x+a2)^(-1/a3)
 *-------------------------------------------------------------*/
double
fNIST_Bennett5_deriv( double (*funct)(int,double*,double*),
            int i, int j, int M, double *cond, double *a)
{
  /* i ... number of current observation */
  if (j == 0)
    return pow( (cond[i] + a[1]), -1./a[2]);  /* derivation of a1 */
  else  if (j == 1)
    return a[0] * pow( (cond[i] + a[1]), -1./a[2] - 1.)*(-1./a[2]);
              /* derivation of a2 */
  else
    return a[0] * pow( (cond[i] + a[1]), -1./a[2]) *
             log( a[1] + cond[i]) / (a[2]*a[2]);
}

/*-------------------------------------------------------------
 *  fNIST_BoxBOD()
 *  f(x|a) = a1 * (1 - exp( -a2 * x) )
 *-------------------------------------------------------------*/
double
fNIST_BoxBOD( int i, double *cond, double *a)
{
  return ( a[0] * (1 - exp( -a[1] *cond[i])) );
}

/*-------------------------------------------------------------
 *  fNIST_BoxBOD_deriv()
 *  f(x|a) = a1 * (1 - exp( -a2 * x) )
 *-------------------------------------------------------------*/
double
fNIST_BoxBOD_deriv( double (*funct)(), int i, int j, int M,
                    double *cond, double *a)
{
  /*  y = a0 * (1 - exp( -a1 * x) ) */
  if ( a[1] < 0 ) a[1] = 0;
  if (j == 0)
  {
      return (1 - exp( -a[1] *cond[i]));
  }
  else
  {
      return ( a[0] * cond[i] * exp( -a[1]*cond[i]) );
  }
}
```

```
/*************************************************************
 *
 * File....: functions_NIST.h
```

S.4 Initialisation of nonlinear models

```
 3:  * Function: proto typing for functions_NIST.c
 4:  * Author..: Tilo Strutz
 5:  * Date....: 23.09.2009
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  ****************************************************************/
14:
15: #ifndef FUNCT_NIST_H
16: #define FUNCT_NIST_H
17:
18:
19: /* linear functions */
20:
21: /* nonlinear functions */
22:
23: double fNIST_BoxBOD( int i, double *cond, double *a);
24: double fNIST_BoxBOD_deriv( double (*funct)(int,double*,double*),
25:             int i, int j, int M, double *cond, double *a);
26: int init_NIST_BoxBOD( int N, double *obs,
27:             double *cond, double *a, unsigned char *a_flag, FILE *out);
28: double fNIST_MGH09( int i, double *cond, double *a);
29: double fNIST_MGH09_deriv( double (*funct)(int,double*,double*),
30:           int i, int j, int M, double *cond, double *a);
31: int init_NIST_MGH09( int N, double *obs, double *cond,
32:           double *a, unsigned char *a_flag, FILE *logfile);
33:
34: double fNIST_thurber( int i, double *cond, double *a);
35: double fNIST_thurber_deriv( double (*funct)(int,double*,double*),
36:           int i, int j, int M, double *cond, double *a);
37: int init_NIST_thurber( int N, double *obs,
38:             double *cond, double *a, unsigned char *a_flag, FILE *out);
39:
40: double fNIST_Rat42( int i, double *cond, double *a);
41: double fNIST_Rat42_deriv( double (*funct)(int,double*,double*),
42:           int i, int j, int M, double *cond, double *a);
43: int init_NIST_Rat42( int N, double *obs,
44:             double *cond, double *a, unsigned char *a_flag, FILE *out);
45:
46: double fNIST_Rat43( int i, double *cond, double *a);
47: double fNIST_Rat43_deriv( double (*funct)(int,double*,double*),
48:           int i, int j, int M, double *cond, double *a);
49: int init_NIST_Rat43( int N, double *obs,
50:             double *cond, double *a, unsigned char *a_flag, FILE *out);
51:
52: double fNIST_Eckerle4( int i, double *cond, double *a);
53: double fNIST_Eckerle4_deriv( double (*funct)(int,double*,double*),
54:           int i, int j, int M, double *cond, double *a);
55: int init_NIST_Eckerle4( int N, double *obs,
56:             double *cond, double *a, unsigned char *a_flag, FILE *out);
57:
58: double fNIST_MGH10( int i, double *cond, double *a);
59: int init_NIST_MGH10( int N, double *obs,
60:             double *cond, double *a, unsigned char *a_flag, FILE *out);
61: double fNIST_MGH10_deriv( double (*funct)(int,double*,double*),
62:           int i, int j, int M, double *cond, double *a);
63: double fNIST_MGH10_deriv2( double (*funct)(int,double*,double*),
64:           int i, int j, int M, int n, double *cond, double *a);
65:
66: double fNIST_Bennett5( int i, double *cond, double *a);
67: int init_NIST_Bennett5( int N, double *obs,
68:             double *cond, double *a, unsigned char *a_flag, FILE *out);
69: double fNIST_Bennett5_deriv( double (*funct)(int,double*,double*),
70:           int i, int j, int M, double *cond, double *a);
71:
72: #endif
```

```
 0: /****************************************************************
 1:  *
 2:  * File........:  init_collection.c
 3:  * Function....:  parameter initialisation for
 4:  *                different functions
 5:  * Author......:  Tilo Strutz
 6:  * last changes:  02.07.2009, 30.09.2009, 08.01.2010, 18.02.2010
 7:  *
 8:  * LICENCE DETAILS: see software manual
 9:  *  free academic use
10:  *  cite source as
11:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
12:  *  2nd edition 2015"
13:  *
14:  ****************************************************************/
15: #include <stdio.h>
16: #include <stdlib.h>
17: #include <string.h>
18: #include <math.h>
19: #include "functions.h"
20: #include "macros.h"
21: #include "defines.h"
22: #include "prototypes.h"
23:
24: #ifndef WIN32
25: #include <sys/time.h>
26: #else
27: #include <time.h>
28: #define random rand
29: #endif
30:
31: /*-------------------------------------------------------------
32:  *  init_polynomial()
33:  * f(x|a) = sum_{j=1}^M aj * x^(j-1)
34:  *-------------------------------------------------------------*/
35: int init_polynomial( int N, double *obs, double *cond,
36:              double *a, unsigned char *a_flag, FILE *logfile)
37: {
38:   long j;
39:
40:   if (!a_flag[0])
41:       a[0] = ((double)rand()/ RAND_MAX - 0.5) * 10.01;
42:   if (!a_flag[1])
43:       a[1] = ((double)rand()/ RAND_MAX - 0.5) * 10.01;
44:
45:   /* assume maximal number of parameters */
46:   for ( j = 2; j < M_MAX; j++)
47:   {
48:     if (!a_flag[j])
49:       a[j] = ((double)rand()/ RAND_MAX - 0.5) * 10.01;
50:   }
51:   return 0;
52: }
53:
54: /*-------------------------------------------------------------
55:  *  init_cosine_nonlin()
56:  * 5: f(x|a) = a1 + a2 * cos( x - a3)  (omega = 2*pi)
57:  *-------------------------------------------------------------*/
58: int
59: init_cosine_nonlin( int N, double *obs, double *cond,
60:           double *a, unsigned char *a_flag, FILE *logfile)
61: {
62:   int i;
63:   double mean;
64:
65:   /* mean value for a[0] */
66:   if (!a_flag[0])
67:   {
68:     mean = 0;
69:     for (i = 0; i < N; i++)
70:           mean += obs[i];
71:     a[0] = mean / N;
72:   }
73:
74:   /* estimation of a2 = radius */
75:   {
76:     double max_obs, min_obs;
77:     max_obs = min_obs = obs[0];
78:     for (i = 1; i < N; i++)
79:     {
80:       if (max_obs < obs[i]) max_obs = obs[i];
81:       if (min_obs > obs[i]) min_obs = obs[i];
82:     }
83:     if (!a_flag[1]) /* if not set on command line */
84:     {
85:       a[1] = 0.5 * ( max_obs - min_obs);
86:     }
87:   }
88:
89:   /* estimation of a3 = phase shift */
90:   if (!a_flag[2])
```

```
  {
    a[2] = 1.;  /* dummy */
  }

  return 0;
}

/*-------------------------------------------------------------
 *  init_cosine_trend()
 * 12: f(x|a) = a1 + a2 * x + a3 * cos( x - a4)
 *-------------------------------------------------------------*/
int
init_cosine_trend( int N, double *obs, double *cond,
           double *a, unsigned char *a_flag, FILE *logfile)
{
  int i;
  double mean;

  /* mean value for a[0] */
  if (!a_flag[0])
  {
    mean = 0;
    for (i = 0; i < N; i++)
          mean += obs[i];
    a[0] = mean / N;
  }

  /* estimation of a3 = linear trend */
  if (!a_flag[1])
  {
    a[1] = 0.;  /* dummy */
  }

  /* estimation of a3 = radius */
  if (!a_flag[2]) /* if not set on command line */
  {
    mean = 0;
    for (i = 0; i < N; i++)
          mean += obs[i] * sqrt( 2.);
    a[2] = mean / N;
  }

  /* estimation of a4 = phase shift */
  if (!a_flag[3])
  {
    a[3] = 0.;  /* dummy */
  }

  return 0;
}

/*-------------------------------------------------------------
 *  init_trigonometric1()
 * f(x|a) = a1 + a2*cos(a3*x-a4)
 *-------------------------------------------------------------*/
int
init_trigonometric1( int N, double *obs, double *cond,
           double *a, unsigned char *a_flag, FILE *logfile)
{
  int i;
  double mean;

  /* mean value for a[0] */
  if (!a_flag[0])
  {
    mean = 0;
    for (i = 0; i < N; i++)
          mean += obs[i];
    a[0] = mean / N;
  }

  /* estimation of a3 = period */
  {
    double max_x, min_x;
    max_x = min_x = cond[0];
    for (i = 1; i < N; i++)
    {
      if (max_x < cond[i]) max_x = cond[i];
      if (min_x > cond[i]) min_x = cond[i];
    }
    if (!a_flag[2])
      a[2] = 2*3.141 / (2 * ( max_x - min_x));
  }

  /* estimation of a2 = amplitude */
  {
    double max_obs, min_obs;
    max_obs = min_obs = obs[0];
    for (i = 1; i < N; i++)
    {
      if (max_obs < obs[i]) max_obs = obs[i];
      if (min_obs > obs[i]) min_obs = obs[i];
    }
    if (!a_flag[1]) /* if not set on command line */
    {
      a[1] = 0.5 * ( max_obs - min_obs);
    }
  }

  /* estimation of a4 = phase shift */
  if (!a_flag[3])
  {
    a[3] = 0.;  /* dummy */
  }

  return 0;
}

/*-------------------------------------------------------------
 *  init_trigonometric2()
 * f(x|a) = a1 + a2*cos(a3*x-a4) + a5*cos(2*a3*x-a6)
 *-------------------------------------------------------------*/
int
init_trigonometric2( int N, double *obs, double *cond,
           double *a, unsigned char *a_flag, FILE *logfile)
{
  int i;
  double mean;

  /* mean value for a[0] */
  if (!a_flag[0])
  {
    mean = 0;
    for (i = 0; i < N; i++)
          mean += obs[i];
    a[0] = mean / N;
  }

  /* estimation of a3 = period */
  {
    double max_x, min_x;
    max_x = min_x = cond[0];
    for (i = 1; i < N; i++)
    {
      if (max_x < cond[i]) max_x = cond[i];
      if (min_x > cond[i]) min_x = cond[i];
    }
    if (!a_flag[2])
      a[2] = 2*3.141 / (2 * ( max_x - min_x));
  }

  /* estimation of a2, a5  = amplitude */
  {
    double max_obs, min_obs;
    max_obs = min_obs = obs[0];
    for (i = 1; i < N; i++)
    {
      if (max_obs < obs[i]) max_obs = obs[i];
      if (min_obs > obs[i]) min_obs = obs[i];
    }
    if (!a_flag[1]) /* if not set on command line */
    {
      a[1] = 0.5 * ( max_obs - min_obs);
    }
    if (!a_flag[4]) /* if not set on command line */
    {
      a[4] = rand() * 0.5 * ( max_obs - min_obs);
    }
  }

  /* estimation of a4,a6 = phase shift */
  if (!a_flag[3])
  {
    a[3] = 0.;  /* dummy */
  }
  if (!a_flag[5])
  {
    a[5] = 0.;  /* dummy */
  }
  return 0;
}

/*-------------------------------------------------------------
 *  init_logarithmic()
 *  f(x|a) = log( a1 * x)
 *-------------------------------------------------------------*/
int
init_logarithmic( int N, double *obs, double *cond,
           double *a, unsigned char *a_flag, FILE *logfile)
{
  if (!a_flag[0])
    a[0] = 5;
  return 0;
}
/*-------------------------------------------------------------
 *  init_exponential()
 *  f(x|a) = a1 + a2 * exp( a3 * x)
 *-------------------------------------------------------------*/
int
init_exponential( int N, double *obs, double *cond,
           double *a, unsigned char *a_flag, FILE *logfile)
{
```

```
283:   int err = 0;     /* return value */
284:   int i, itmp;
285:   double mean;
286:
287:   /* number of conditions to be inspected */
288:   itmp = MAX( 0, MIN( 5, N));
289:
290:   /* estimation of a1 = tail of graph */
291:   if (!a_flag[0])
292:   {
293:     mean = 0;
294:     for (i = N - 1; i > N - itmp; i--)
295:       mean += obs[i];
296:     a[0] = mean / itmp;
297:   }
298:
299:   /* estimation of a2 = head of function */
300:   if (!a_flag[1])
301:   {
302:     mean = 0;
303:     for (i = 0; i < itmp; i++)
304:       mean += obs[i];
305:
306:     /* assumes conditions starting close to zero
307:      * y(x=0) = a1 + a2 * exp( a3 * 0) = a1 + a2
308:      */
309:     a[1] = mean / itmp;
310:   }
311:
312:   /* estimation of a3 = gradient at head of function */
313:   if (!a_flag[2])
314:   {
315:     mean = 0;
316:     for (i = 1; i < itmp; i++)
317:       mean += (obs[i] - obs[i - 1]) / (cond[i] - cond[i - 1]);
318:     a[2] = mean / (itmp * 0.5);
319:   }
320:
321:   /* if not decaying */
322:   if (a[0] > a[1])
323:   {
324:     fprintf( stderr, "\n a1 > a2 !");
325:     fprintf( stderr, "\n flip signs of a2 and a3 !\n");
326:     a[1] = -a[1];
327:     a[2] = -a[2];
328:   }
329:
330:   return err;
331: }
332:
333: /*-------------------------------------------------------------
334:  *  init_expon2()
335:  * f(x|a) = a1 * exp( a2 * x)
336:  *-------------------------------------------------------------*/
337: int
338: init_expon2( int N, double *obs, double *cond,
339:   double *a, unsigned char *a_flag, FILE *out)
340: {
341:   int i, itmp;
342:   double mean;
343:
344:   /* number of conditions to be inspected */
345:   itmp = MAX( 0, MIN( 5, N));
346:
347:   /* estimation of a1 = head of function */
348:   if (!a_flag[0])
349:   {
350:     mean = 0;
351:     for (i = 0; i < itmp; i++)
352:       mean += obs[i];
353:
354:     /* assumes conditions starting close to zero
355:      * y(x=0) = a1 * exp( a2 * 0) = a1
356:      */
357:     a[0] = mean / itmp;
358:   }
359:
360:   /* estimation of a2 = gradient at head of function
361:    * a2 = f'(0)/a1
362:    */
363:   if (!a_flag[1])
364:   {
365:     mean = 0;
366:     for (i = 1; i <= itmp; i++)
367:       mean += (obs[i] - obs[i - 1]) / (cond[i] - cond[i - 1]);
368:     a[1] = mean / (itmp * a[0]);
369:   }
370:   return 0;
371: }
372:
373: /*-------------------------------------------------------------
374:  *  init_gauss()
375:  * f(x|a) = a1 * exp( a2 * (x-a3)^2) +
376:  *-------------------------------------------------------------*/
377: int
378: init_gauss( int N, double *obs, double *cond,
379:             double *a, unsigned char *a_flag, FILE *out)
380: {
381:   int err = 0;     /* return value */
382:   int i;
383:   int i_mean = 0, i_max, i_min;
384:   double max_val, min_val, condmin=0., condmax=0.;
385:   double mean, var, sum, sigma, tmp;
386:
387:   /*
388:    * get starting point
389:    * assuming that one Gaussian is good enough to fit the data
390:    */
391:
392:   /* get peak of curve */
393:   max_val = min_val = obs[1];
394:   i_max = i_min = 1;
395:   condmax = cond[1];
396:   condmin = cond[1];
397:   for (i = 2; i < N-1; i++) /* let 1 sample border */
398:   {
399:     if (max_val < obs[i])
400:     {
401:       max_val = obs[i];
402:       i_max = i;  /* peak position index */
403:       condmax = cond[i];  /* peak position */
404:     }
405:     if (min_val > obs[i])
406:     {
407:       min_val = obs[i];
408:       i_min = i;  /* peak position */
409:       condmin = cond[i];
410:     }
411:   }
412:   if (max_val == min_val)
413:   {
414:     fprintf( out, "\n\n Nothing to fit !!");
415:     a[0] = 0.;
416:     a[2] = -50000000.0;
417:     a[1] = 0.;
418:     err = 8;
419:     goto endfunc;
420:   }
421:
422:   mean = sum = var = 0.;
423:   /* take only that part which has the highest peak */
424:   if (fabs(max_val) > fabs(min_val))
425:   {
426:     /* positive amplitude */
427:     for (i = 0; i < N; i++)
428:     {
429:       if (obs[i] > 0.)
430:       {
431:         /* mean and variance of condition
432:          * observed value is like probability
433:          */
434:         tmp = cond[i] * obs[i];
435:         mean += tmp;
436:         var += cond[i] * tmp;
437:         sum += obs[i];
438:       }
439:     }
440:     if (sum > 0.)
441:     {
442:       mean /= sum;  /* average along cond[i] */
443:       var = var/sum - mean*mean;
444:     }
445:   }
446:   else
447:   {
448:     /* negative amplitude */
449:     for (i = 0; i < N; i++)
450:     {
451:       if (obs[i] < 0.)
452:       {
453:         tmp = - cond[i] * obs[i];
454:         mean += tmp;
455:         var += cond[i] * tmp;
456:         sum -= obs[i];
457:       }
458:     }
459:     if (sum > 0.)
460:     {
461:       mean /= sum;
462:       var = var/sum - mean*mean;
463:     }
464:   }
465:
466:   /* if only one data point, then sigma is zero */
467:   if (var > 0.) sigma = sqrt( var);  /* deviation of Gaussian */
468:   else
469:    sigma = 0.0000001;
470:
471:
472:   /* get index of mean position */
473:   for (i = 1; i < N; i++)
474:   {
```

```
475:     if (cond[i-1] <= mean && mean <= cond[i])
476:     {
477:       i_mean = i; /* mean position */
478:       break;
479:     }
480:   }
481:
482:   /* make values more robust by averaging */
483:   max_val = (max_val + obs[i_max-1] + obs[i_max+1]) /3;
484:   min_val = (min_val + obs[i_min-1] + obs[i_min+1]) /3;
485:   if (obs[i_mean] > 0.)
486:   {
487:     {
488:       /* select highest peak, when there are 2 or more */
489:       if (!a_flag[0]) a[0] = max_val;
490:       if (!a_flag[1]) a[1] = condmax;
491:       /* reduce deviation accordingly */
492:       if (sigma > fabs(mean - condmax))
493:           sigma -= fabs(mean - condmax);
494:     }
495:   }
496:   else
497:   {
498:     {
499:       if (!a_flag[0]) a[0] = min_val;
500:       if (!a_flag[1]) a[1] = condmin;
501:       if (sigma > fabs(mean-condmin)) sigma -= fabs(mean-condmin);
502:     }
503:   }
504:   /* transcode deviation */
505:   if (!a_flag[2]) a[2] = -0.5 / (sigma*sigma);
506:
507: endfunc:
508:   return err;
509: }
510:
511: /*-------------------------------------------------------------
512:  *  init_gen_laplace()
513:  * f(x|a) = a1 * exp( -|x|^a2 * a3)
514:  *-------------------------------------------------------------*/
515: int
516: init_gen_laplace( int N, double *obs, double *cond,
517:   double *a, unsigned char *a_flag, FILE *out)
518: {
519:   /* assumes conditions starting close to zero
520:    * y(x=0) = a1 * exp( 0) = a1
521:    */
522:   if (!a_flag[0])
523:   {
524:     a[0] = obs[0];
525:   }
526:   if (!a_flag[1])
527:   {
528:     a[1] = 1.0;
529:   }
530:   if (!a_flag[2])
531:   {
532:     a[2] = 0.8;
533:   }
534:
535:   return 0;
536: }
537:
538: /*-------------------------------------------------------------
539:  *  init_circlelin()
540:  *  f(x|a) = 0 = (x1-a1)^2 + (x2-a2)^2 - a3^2
541:  *-------------------------------------------------------------*/
542: int
543: init_circlelin( int N, double *obs, double *cond,
544:   double *a, unsigned char *a_flag, FILE *logfile)
545: {
546:   int err = 0;     /* return value */
547:   double b1, b2, b3;
548:
549:   /* get estimates of centre coordinates and radius */
550:   err = init_circle( N, obs, cond, a, a_flag, logfile);
551:
552:   /* convert into vector b  */
553:   b1 = 2 * a[0];
554:   b2 = 2 * a[1];
555:   b3 = a[0]*a[0] + a[1]*a[1] - a[2]*a[2];
556:
557:   /* put back to a[]  */
558:   a[0] = b1;
559:   a[1] = b2;
560:   a[2] = b3;
561:
562:   return err;
563: }
564:
565: /*-------------------------------------------------------------
566:  *  init_circle()
567:  *  f(x|a) = 0 = (x1-a1)^2 + (x2-a2)^2 - a3^2
568:  *-------------------------------------------------------------*/
569: int
570: init_circle( int N, double *obs, double *cond,
571:   double *a, unsigned char *a_flag, FILE *logfile)
572: {
573:   int err = 0;     /* return value */
574:   int i;
575:   double sum_x, sum_y, rad2, diff1, diff2;
576:
577:   fprintf( logfile, "\n#\n# init_circle()");
578:
579:   /*
580:    * determine circle centre
581:    */
582:
583:   /* compute centroids of conditions */
584:   sum_x = sum_y = 0.;
585:   /* two conditions */
586:   for (i = 0; i < 2*N; i+=2)
587:   {
588:     sum_x += cond[i];
589:     sum_y += cond[i+1];
590:   }
591:   sum_x /= (double)N;
592:   sum_y /= (double)N;
593:
594:   rad2 = 0;
595:   for (i = 0; i < 2*N; i+=2)
596:   {
597:     diff1 = cond[i] - sum_x;
598:     diff2 = cond[i+1] - sum_y;
599:     rad2 += sqrt( diff1*diff1 + diff2*diff2);
600:   }
601:   rad2 = rad2 / (double)N;
602:   fprintf( logfile, "\n#\n# mean of condition coordinates");
603:   fprintf( logfile, "\n#   mean(x)= %f", sum_x);
604:   fprintf( logfile, "\n#   mean(y)= %f", sum_y);
605:   fprintf( logfile, "\n#   radius = %f", rad2);
606:
607:   if (!a_flag[0]) a[0] = sum_x;
608:   if (!a_flag[1]) a[1] = sum_y;
609:   if (!a_flag[2]) a[2] = rad2;
610:
611:   fprintf( logfile,
612:            "\n#  f(x|a) =0= (x1-%f)**2 + (x2-%f)**2 - %f**2",
613:            a[0], a[1], a[2]);
614:
615:   return err;
616: }
617:
618: /*-------------------------------------------------------------
619:  *  init_rotation()
620:  * 21... f1(x|a) = a1 + cos(a3) * x1 - sin(a3) * x2
621:  *       f2(x|a) = a2 + sin(a3) * x1 + cos(a3) * x2
622:  *-------------------------------------------------------------*/
623: int
624: init_rotation( int N, double *obs, double *cond,
625:   double *a, unsigned char *a_flag, FILE *logfile)
626: {
627:   int err = 0;     /* return value */
628:   int i;
629:   double sum_x, sum_y, sum_u, sum_v;
630:
631:   fprintf( logfile, "\n#\n# init_rotation()");
632:
633:   /*
634:    * determine rough translation
635:    */
636:
637:   /* compute centroids of conditions and observations */
638:   sum_x = sum_y = 0;
639:   sum_u = sum_v = 0;
640:   /* assume double observations and conditions */
641:   for (i = 0; i < N * 2; i+=2)
642:   {
643:     sum_x += obs[i];
644:     sum_y += obs[i+1];
645:     sum_u += cond[i];
646:     sum_v += cond[i+1];
647:   }
648:   sum_x /= (double)N;
649:   sum_y /= (double)N;
650:   sum_u /= (double)N;
651:   sum_v /= (double)N;
652:   fprintf( logfile, "\n#\n# mean of condition coordinates");
653:   fprintf( logfile, "\n#   mean(u)= %f", sum_u);
654:   fprintf( logfile, "\n#   mean(v)= %f", sum_v);
655:   fprintf( logfile, "\n# mean of observed coordinates");
656:   fprintf( logfile, "\n#   mean(x)= %f", sum_x);
657:   fprintf( logfile, "\n#   mean(y)= %f", sum_y);
658:
659:   if (!a_flag[0]) a[0] = sum_x - sum_u;
660:   if (!a_flag[1]) a[1] = sum_y - sum_v;
661:   if (!a_flag[2]) a[2] = 0;   /* assume no rotation */
662:
663:
664:   fprintf( logfile,
665:            "\n#  f1(u,v) = %f + cos(%f) * u - sin(%f) * v",
666:            a[0], a[2], a[2]);
```

```
667:   fprintf( logfile,
668:           "\n#  f2(u,v) = %f + sin(%f) * u + cos(%f) * v",
669:           a[1], a[2], a[2]);
670:
671:   return err;
672: }
673:
674: /*-------------------------------------------------------------
675:  *  init_NN3x3x1()
676:  * 3x3x1
677:  *-------------------------------------------------------------*/
678: int
679: init_NN3x3x1( int N, double *obs, double *cond,
680:               double *a, unsigned char *a_flag, FILE *logfile)
681: {
682:   int i, j;
683:   double minval, maxval;
684: #ifndef WIN32
685:   struct timeval tv;
686:   struct timezone tz;
687:
688:   gettimeofday( &tv, &tz);
689:   srandom( tv.tv_sec);
690: #else
691:    /* Seed the random-number generator with current time so that
692:     * the numbers will be different every time we run.
693:     */
694:    srand( (unsigned)time( NULL ) );
695: #endif
696:
697:   /* give parameters random values */
698:   for ( j = 0; j < M_MAX; j++)
699:   {
700:     if (!a_flag[j])
701:       a[j] = 2. * (float)random()/ (float)RAND_MAX - 1.;
702:   }
703:
704:   /* make random numbers in a range that |cond x param| < 5 */
705:   minval = maxval = cond[0];
706:   for ( i = 1; i < N; i++)
707:   {
708:     if (minval > cond[3*i]) minval = cond[3*i];
709:     if (maxval < cond[3*i]) maxval = cond[3*i];
710:   }
711:   /* weights from 1st input */
712:   if (!a_flag[1]) a[1] = a[1] / (maxval-minval);
713:   if (!a_flag[5]) a[5] = a[5] / (maxval-minval);
714:   if (!a_flag[9]) a[9] = a[9] / (maxval-minval);
715:
716:   minval = maxval = cond[1];
717:   for ( i = 1; i < N; i++)
718:   {
719:     if (minval > cond[3*i+1]) minval = cond[3*i+1];
720:     if (maxval < cond[3*i+1]) maxval = cond[3*i+1];
721:   }
722:   /* weights from 2nd input */
723:   if (!a_flag[2]) a[2] = a[2] / (maxval-minval);
724:   if (!a_flag[6]) a[6] = a[6] / (maxval-minval);
725:   if (!a_flag[10])a[10]= a[10] / (maxval-minval);
726:
727:   minval = maxval = cond[2];
728:   for ( i = 1; i < N; i++)
729:   {
730:     if (minval > cond[3*i+2]) minval = cond[3*i+2];
731:     if (maxval < cond[3*i+2]) maxval = cond[3*i+2];
732:   }
733:   /* weights from 3rd input */
734:   if (!a_flag[3]) a[3] = a[3] / (maxval-minval);
735:   if (!a_flag[7]) a[7] = a[7] / (maxval-minval);
736:   if (!a_flag[11])a[11]= a[11]/ (maxval-minval);
737:
738:   return 0;
739: }
740:
741: /*-------------------------------------------------------------
742:  *  init_NN1x3x1()
743:  * 1x3x1
744:  *-------------------------------------------------------------*/
745: int
746: init_NN1x3x1( int N, double *obs, double *cond,
747:               double *a, unsigned char *a_flag, FILE *logfile)
748: {
749:   int i, j;
750:   double minval, maxval;
751: #ifndef WIN32
752:   struct timeval tv;
753:   struct timezone tz;
754:
755:   gettimeofday( &tv, &tz);
756:   srandom( tv.tv_sec);
757: #else
758:    /* Seed the random-number generator with current time so that
759:     * the numbers will be different every time we run.
760:     */
761:    srand( (unsigned)time( NULL ) );
762: #endif
763:
764:   /* give parameters random values */
765:   for ( j = 0; j < M_MAX; j++)
766:   {
767:     if (!a_flag[j])
768:     {
769:       a[j] = 2. * (float)random() / (float)RAND_MAX - 1.;
770:     }
771:   }
772:
773:   /* make random numbers in a range that |cond x param| < 5 */
774:   minval = maxval = cond[0];
775:   for ( i = 1; i < N; i++)
776:   {
777:     if (minval > cond[i]) minval = cond[i];
778:     if (maxval < cond[i]) maxval = cond[i];
779:   }
780:   /* weights from 1st input */
781:   if (!a_flag[1]) a[1] = a[1] / (maxval-minval);
782:   if (!a_flag[3]) a[3] = a[3] / (maxval-minval);
783:   if (!a_flag[5]) a[5] = a[5] / (maxval-minval);
784:
785:   return 0;
786: }
787:
788: /*-------------------------------------------------------------
789:  *  init_NN()
790:  * 3x...
791:  *-------------------------------------------------------------*/
792: int
793: init_NN( int N, double *obs, double *cond,
794:          double *a, unsigned char *a_flag, FILE *logfile)
795: {
796:   int j;
797: #ifndef WIN32
798:   struct timeval tv;
799:   struct timezone tz;
800:
801:   gettimeofday( &tv, &tz);
802:   srandom( tv.tv_sec);
803: #else
804:    /* Seed the random-number generator with current time so that
805:     * the numbers will be different every time we run.
806:     */
807:    srand( (unsigned)time( NULL ) );
808: #endif
809:
810:   for ( j = 0; j < M_MAX; j++)
811:   {
812:     if (!a_flag[j])
813:     {
814:       a[j] = 1. * (float)random() / (float)RAND_MAX - 0.5;
815:     }
816:   }
817:   return 0;
818: }
```

```
  0: /*************************************************************
  1:  *
  2:  * File........: init_gauss2.c
  3:  * Function....: parameter initialisation for
  4:  *                 data fitting with 2 Gaussians
  5:  * Author......: Tilo Strutz
  6:  * last changes: 02.07.2009
  7:  *
  8:  * LICENCE DETAILS: see software manual
  9:  *  free academic use
 10:  *  cite source as
 11:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 12:  *  2nd edition 2015"
 13:  *
 14:  *************************************************************/
 15: #include <stdio.h>
 16: #include <stdlib.h>
 17: #include <string.h>
 18: #include <math.h>
 19: #include "matrix_utils.h"
 20: #include "functions.h"
 21:
 22: /*-------------------------------------------------------------
 23:  *  init_gauss1()
 24:  * f(x|a) = a1 * exp( (x-a2)^2 * a3)
 25:  *-------------------------------------------------------------*/
 26: int
 27: init_gauss1( int N, double *obs, double *cond,
 28:              double *a, unsigned char *a_flag,
 29:              int peak_flag, FILE *logfile)
 30: {
 31:   int err = 0;     /* return value */
 32:   int i;
 33:   int i_mean = 0, i_max, i_min;
 34:   double max_val, min_val, condmin=0., condmax=0.;
 35:   double mean, var, sum, sigma, tmp;
 36:
 37:   /*
```

```
38:    * get starting point
39:    * assuming that one Gaussian is good enough to fit the data
40:    */
41:
42:   /* get peak of curve */
43: /*
44:   max_val = min_val = obs[1];
45:   i_max = i_min = 1;
46:   condmax = cond[1];
47:   condmin = cond[1];
48:   for (i = 2; i < N-1; i++) /* let 1 sample border */
49:   max_val = min_val = obs[0];
50:   i_max = i_min = 0;
51:   condmax = cond[0];
52:   condmin = cond[0];
53:   for (i = 1; i < N; i++) /* let 1 sample border */
54:   {
55:     if (max_val < obs[i])
56:     {
57:       max_val = obs[i];
58:       i_max = i;  /* peak position index */
59:       condmax = cond[i];  /* peak position */
60:     }
61:     if (min_val > obs[i])
62:     {
63:       min_val = obs[i];
64:       i_min = i;  /* peak position */
65:       condmin = cond[i];
66:     }
67:   }
68:   if (max_val == min_val)
69:   {
70:     fprintf( logfile, "\n\n Nothing to fit !!");
71:     a[0] = 0.;
72:     a[2] = -50000000.0;
73:     a[1] = 0.;
74:     err = 8;
75:     goto endfunc;
76:   }
77:
78:   mean = sum = var = 0.;
79:   /* take only that part which has the highest peak */
80:   if (fabs(max_val) > fabs(min_val))
81:   {
82:     /* positive amplitude */
83:     for (i = 0; i < N; i++)
84:     {
85:       if (obs[i] > 0.)
86:       {
87:         /* mean and variance of condition
88:          * observed value is like probability
89:          */
90:         tmp = cond[i] * obs[i];
91:         mean += tmp;
92:         var += cond[i] * tmp;
93:         sum += obs[i];
94:       }
95:     }
96:     if (sum > 0.)
97:     {
98:       mean /= sum;  /* average along cond[i] */
99:       var = var/sum - mean*mean;
100:    }
101:  }
102:  else
103:  {
104:    /* negative amplitude */
105:    for (i = 0; i < N; i++)
106:    {
107:      if (obs[i] < 0.)
108:      {
109:        tmp = - cond[i] * obs[i];
110:        mean += tmp;
111:        var += cond[i] * tmp;
112:        sum -= obs[i];
113:      }
114:    }
115:    if (sum > 0.)
116:    {
117:      mean /= sum;
118:      var = var/sum - mean*mean;
119:    }
120:  }
121:  /* if only one data point, then sigma is zero */
122:  if (var > 0.) sigma = sqrt( var);  /* deviation of Gaussian */
123:  else
124:   sigma = 0.0000001;
125:
126:
127:  /* get index of mean position */
128:  for (i = 1; i < N; i++)
129:  {
130:    if (cond[i-1] <= mean && mean <= cond[i])
131:    {
132:      i_mean = i; /* mean position */
133:      break;
134:    }
135:  }
136:
137:  /* make values more robust by averaging */
138:  max_val = (max_val + obs[i_max-1] + obs[i_max+1]) /3;
139:  min_val = (min_val + obs[i_min-1] + obs[i_min+1]) /3;
140:  if (obs[i_mean] > 0.)
141:  {
142:    if (peak_flag)
143:    {
144:      /* select highest peak, when there are 2 or more */
145:      if (!a_flag[0]) a[0] = max_val;
146:      if (!a_flag[1]) a[1] = condmax;
147:      /* reduce deviation accordingly */
148:      if (sigma > fabs(mean - condmax))
149:          sigma -= fabs(mean - condmax);
150:    }
151:    else
152:    {
153:      /* increase by value dependent on obs at mean position */
154:      if (!a_flag[0])
155:      {
156:        a[0] = max_val + (max_val - obs[i_mean]) * 0.5;
157:      }
158:      if (!a_flag[1]) a[1] = mean;
159:    }
160:  }
161:  else
162:  {
163:    if (peak_flag)
164:    {
165:      if (!a_flag[0]) a[0] = min_val;
166:      if (!a_flag[1]) a[1] = condmin;
167:      if (sigma > fabs(mean-condmin)) sigma -= fabs(mean-condmin);
168:    }
169:    else
170:    {
171:      if (!a_flag[0])
172:      {
173:        a[0] = max_val + (max_val - obs[i_mean]) * 0.5;
174:      }
175:      if (!a_flag[1]) a[1] = mean;
176:    }
177:  }
178:  /* transcode deviation */
179:  if (!a_flag[2]) a[2] = -0.5 / (sigma*sigma);
180:
181:  if (a[0] > 100)
182:  {
183:    i = i;
184:  }
185: endfunc:
186:  return err;
187: }
188:
189: /*-------------------------------------------------------------
190:  *  init_gauss2()
191:  * f(x|a) = a1 * exp( a2 * (x-a3)^2) +
192:  *          a4 * exp( a5 * (x-a6)^2)
193:  *-------------------------------------------------------------*/
194: int
195: init_gauss2( int N, double *obs, double *cond,
196:              double *a, unsigned char *a_flag, FILE *out)
197: {
198:   int err = 0;     /* return value */
199:   int i, M;
200:   double sigma1, sigma2;
201:   double *obs_cpy=NULL;
202:
203:   M = 6;  /* fixed number of parameters */
204:   obs_cpy = vector( N);
205:
206:   /*
207:    * get starting point
208:    * assuming that one Gaussian is good enough to fit the data
209:    */
210:   /* initialises a[0], a[1], a[2]; select highest peak */
211:   err = init_gauss1( N, obs, cond, a, a_flag, 1, out);
212:   if (err) goto endfunc;
213:
214:   sigma2 = sqrt( -0.5 / a[2]); /* get deviation back */
215:
216:   fprintf( out, "\n# Initial parameter: first Gaussian");
217:   fprintf( out, "\n# amplitude: %f, mean: %f, deviation: %f",
218:            a[0], a[1], sigma2);
219:   /*
220:    * compute residuals between observation and model
221:    */
222:   for ( i = 0; i < N; i++)
223:   {
224:       obs_cpy[i] = obs[i] - fgauss1( i, cond, a);
225:   }
226:   /* prevent overwriting of first three parameters */
227:   a[3] = a[0];
228:   a[4] = a[1];
229:   a[5] = a[2];
```

```
230:
231:   /* initialises a[0], a[1], a[2]; fitting of residual */
232:   init_gauss1( N, obs_cpy, cond, a, a_flag, 1, out);
233:   sigma1 = sqrt( -0.5 / a[2]);
234:   fprintf( out, "\n# Initial parameter: second Gaussian");
235:   fprintf( out, "\n# amplitude: %f, mean: %f, deviation: %f",
236:            a[0], a[1], sigma1);
237:
238:   fprintf( out, "\n#\n# f(x) = %f * exp( ( x- %f)**2 * %f)",
239:            a[0],a[1],a[2]);
240:   fprintf( out, "+ %f * exp( ( x- %f)**2 * %f)",
241:            a[3],a[4],a[5]);
242: endfunc:
243:   free_vector( &obs_cpy);
244:   return err;
245: }
```

```
  0:
  1: /*****************************************************************
  2:  *
  3:  * File........: init_NIST.c
  4:  * Function....: parameter initialisation for
  5:  *                  different functions
  6:  * Author......: Tilo Strutz
  7:  * last changes: 28.09.2009, 20.01.2010
  8:  *
  9:  * LICENCE DETAILS: see software manual
 10:  *  free academic use
 11:  *  cite source as
 12:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 13:  *  2nd edition 2015"
 14:  *
 15:  *****************************************************************/
 16: #include <stdio.h>
 17: #include <stdlib.h>
 18: #include <string.h>
 19: #include <math.h>
 20: #include "functions.h"
 21: #include "macros.h"
 22: #ifndef WIN32
 23: #include <sys/time.h>
 24: #else
 25: #include <time.h>
 26: #define random rand
 27: #endif
 28:
 29: /*-----------------------------------------------------------------
 30:  *  init_NIST_Eckerle4()
 31:  * f(x|a) = a1 / a2 * exp(-0.5*((x -a3)/ a2)^2)
 32:  *----------------------------------------------------------------*/
 33: int
 34: init_NIST_Eckerle4( int N, double *obs, double *cond,
 35:            double *a, unsigned char *a_flag, FILE *logfile)
 36: {
 37:   int i, maxpos;
 38:   double maxval;
 39:
 40:   /* estimated parameter of a Gaussian bell */
 41:
 42:   /* get mean value at maximum point */
 43:   if (!a_flag[2])
 44:   {
 45:     maxval = obs[0];
 46:     maxpos = 0;
 47:     a[2] = cond[0];
 48:     for ( i = 1; i < N; i++)
 49:     {
 50:         if (maxval < obs[i])
 51:         {
 52:           maxval = obs[i];
 53:           maxpos = i;
 54:           a[2] = cond[i];
 55:         }
 56:     }
 57:   }
 58:   /* get sigma  */
 59:   if (!a_flag[1])
 60:   {
 61:     for ( i = maxpos+1; i < N; i++)
 62:     {
 63:         if (obs[i] < maxval/2)
 64:         {
 65:           a[1] = i - maxpos;
 66:           break;
 67:         }
 68:     }
 69:   }
 70:   /* get magnification */
 71:   if (!a_flag[0])
 72:   {
 73:     a[0] = maxval * a[1];
 74:   }
 75:
 76:   return 0;
 77: }
```

```
 78:
 79: /*-----------------------------------------------------------------
 80:  *  init_NIST_Rat43()
 81:  * f(x|a) = a1 / (1 + exp(a2 - a3*x)^(1/a4))
 82:  *----------------------------------------------------------------*/
 83: int
 84: init_NIST_Rat43( int N, double *obs, double *cond,
 85:            double *a, unsigned char *a_flag, FILE *logfile)
 86: {
 87:   /* set 1 */
 88:   if (!a_flag[0]) a[0] = 100;
 89:   if (!a_flag[1]) a[1] = 10;
 90:   if (!a_flag[2]) a[2] = 1;
 91:   if (!a_flag[3]) a[3] = 1;
 92:
 93:   return 0;
 94: }
 95:
 96: /*-----------------------------------------------------------------
 97:  *  init_NIST_Rat42()
 98:  * f(x|a) = a1 / (1 + exp(a2 - a3*x))
 99:  *----------------------------------------------------------------*/
100: int
101: init_NIST_Rat42( int N, double *obs, double *cond,
102:            double *a, unsigned char *a_flag, FILE *logfile)
103: {
104:   /* set 1 */
105:   if (!a_flag[0]) a[0] = 100;
106:   if (!a_flag[1]) a[1] = 10;
107:   if (!a_flag[2]) a[2] = 0.1;
108:
109:   return 0;
110: }
111:
112: /*-----------------------------------------------------------------
113:  *  init_NIST_thurber()
114:  * f(x|a) =(a1 + a2*x + a3*x**2 + a4*x**3) /
115:  *          (1 + a5*x + a6*x**2 + a7*x**3)
116:  *----------------------------------------------------------------*/
117: int
118: init_NIST_thurber( int N, double *obs, double *cond,
119:            double *a, unsigned char *a_flag, FILE *logfile)
120: {
121:   /* set 1 */
122:   if (!a_flag[0]) a[0] = 1000;
123:   if (!a_flag[1]) a[1] = 1000;
124:   if (!a_flag[2]) a[2] = 400;
125:   if (!a_flag[3]) a[3] = 40;
126:   if (!a_flag[4]) a[4] = 0.7;
127:   if (!a_flag[5]) a[5] = 0.3;
128:   if (!a_flag[6]) a[6] = 0.03;
129:
130:   return 0;
131: }
132:
133: /*-----------------------------------------------------------------
134:  *  init_NIST_MGH09()
135:  *   f(x|a) = a1 * (x**2 + a2*x) / (x*x + a3*x + a4)
136:  *----------------------------------------------------------------*/
137: int
138: init_NIST_MGH09( int N, double *obs, double *cond,
139:            double *a, unsigned char *a_flag, FILE *logfile)
140: {
141:   /* set 1 */
142:   if (!a_flag[0]) a[0] = 25;
143:   if (!a_flag[1]) a[1] = 39;
144:   if (!a_flag[2]) a[2] = 41.5;
145:   if (!a_flag[3]) a[3] = 39;
146:
147:   return 0;
148: }
149:
150: /*-----------------------------------------------------------------
151:  *  init_NIST_MGH10()
152:  *   f(x|a) = a1 * exp( a2 / (x+a3))
153:  *----------------------------------------------------------------*/
154: int
155: init_NIST_MGH10( int N, double *obs,
156:             double *cond, double *a, unsigned char *a_flag, FILE *out)
157: {
158:   /* set 1 */
159:   if (!a_flag[0]) a[0] = 2;
160:   if (!a_flag[1]) a[1] = 400000;
161:   if (!a_flag[2]) a[2] = 25000;
162:
163:   return 0;
164: }
165: /*-----------------------------------------------------------------
166:  *  init_NIST_Bennett5()
167:  *   f(x|a) = a1 * (x+a2)^(-1/a3)
168:  *----------------------------------------------------------------*/
169: int
170: init_NIST_Bennett5( int N, double *obs,
171:             double *cond, double *a, unsigned char *a_flag, FILE *out)
172: {
173:   /* set 1 */
```

```
174:   if (!a_flag[0]) a[0] = -2000;
175:   if (!a_flag[1]) a[1] = 50;
176:   if (!a_flag[2]) a[2] = 0.8;
177:
178:   return 0;
179: }
180:
181: /*-------------------------------------------------------------
182:  *  init_NIST_BoxBOD()
183:  *  f(x|a) = a1 * (1 - exp( -a2 * x) )
184:  *-------------------------------------------------------------*/
185: int
186: init_NIST_BoxBOD( int N, double *obs, double *cond,
187:           double *a, unsigned char *a_flag, FILE *logfile)
188: {
189:   int i, itmp;
190:   double mean, maxval;
191:
192:   itmp = MAX( 0, MIN( 5, N));
193:
194:   /* estimation of a1 = head of function */
195:   if (!a_flag[0])
196:   {
197:     maxval = obs[0];
198:     for (i = 1; i < itmp; i++)
199:     {
200:       if (maxval < obs[i]) maxval = obs[i];
201:     }
202:     a[0] = maxval;
203:   }
204:   /* estimation of a1*a2 = gradient at head of function */
205:   if (!a_flag[1])
206:   {
207:     mean = 0;
208:     for (i = 1; i < itmp; i++)
209:       mean += (obs[i] - obs[i - 1]) / (cond[i] - cond[i - 1]);
210:     a[1] = mean / (itmp) / a[0];
211:     /* use moderate value */
212:     if (a[1] > 2.) a[1] = 2.;
213:   }
214:
215:   return 0;
216: }
```

S.5 Matrix processing

S.5.1 Utils

```
 0: /*************************************************************
 1:  *
 2:  * File........:  decomp_LU.c
 3:  * Function....:  LU decomposition
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  20.10.2007
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *************************************************************/
14: #include <stdio.h>
15: #include <stdlib.h>
16: #include <math.h>
17: #include "matrix_utils.h"
18: #include "errmsg.h"
19: #define LITTLEBIT 1.0e-20;
20:
21: /*-------------------------------------------------------------
22:  *  decomp_LU()
23:  *-------------------------------------------------------------*/
24: int
25: decomp_LU( double **normal, int M, int *indx, int *s)
26: {
27:   char *rtn = "decomp_LU";
28:   int err = 0, i, imax = 0, j, k;
29:   double max_element, val, sum, tmp;
30:   double *row_scale = NULL; /* scaling of each row */
31:
32:   /* allocate vector */
33:   row_scale = vector( M);
34:
35:   *s = 1;
36:   /* examine input matrix */
37:   for (i = 0; i < M; i++)
38:   {
39:     max_element = 0.0;
40:     for (j = 0; j < M; j++)
41:     {
42:       if (( tmp = fabs( normal[i][j])) > max_element)
43:         max_element = tmp;
44:     }
45:     if (max_element == 0.0)
46:     {
47:       err = errmsg( ERR_IS_ZERO, rtn, "'max_element'", 0);
48:       goto endfunc;
49:     }
50:     row_scale[i] = 1.0 / max_element;
51:   }
52:   /* loop over columns of Crout's method */
53:   for (j = 0; j < M; j++)
54:   {
55:     for (i = 0; i < j; i++)
56:     {
57:       sum = normal[i][j];
58:       for (k = 0; k < i; k++)
59:         sum -= normal[i][k] * normal[k][j];
60:       normal[i][j] = sum;
61:     }
62:     max_element = 0.0;
63:     for (i = j; i < M; i++)
64:     {
65:       sum = normal[i][j];
66:       for (k = 0; k < j; k++)
67:       {
68:         sum -= normal[i][k] * normal[k][j];
69:       }
70:       normal[i][j] = sum;
71:       /* is new pivot better than current best ? */
72:       if (( val = row_scale[i] * fabs( sum)) >= max_element)
73:       {
74:         max_element = val;
75:         imax = i;
76:       }
77:     }
78:     if (j != imax)
79:     {
80:       /* interchange of rows */
81:       for (k = 0; k < M; k++)
82:       {
83:         val = normal[imax][k];
84:         normal[imax][k] = normal[j][k];
85:         normal[j][k] = val;
86:       }
87:       *s = -( *s);
88:       /* interschange scale factors */
89:       row_scale[imax] = row_scale[j];
90:     }
```

```
    indx[j] = imax;
    if (normal[j][j] == 0.0)
      normal[j][j] = LITTLEBIT;
    if (j != (M - 1))
    {
      /* divide by the pivot element */
      val = 1.0 / (normal[j][j]);
      for (i = j + 1; i < M; i++)
        normal[i][j] *= val;
    }
    /* next column in reduction */
  }

endfunc:
  free_vector( &row_scale);
  return err;
}

/*-------------------------------------------------------------
 *  backsub_LU()
 *-------------------------------------------------------------*/
void
backsub_LU( double **lu, int N, int *indx, double back[])
{
  int i, ii, idx, j;
  double sum;

  ii = -1;
  for (i = 0; i < N; i++)
  {
    idx = indx[i];
    sum = back[idx];
    back[idx] = back[i];
    if (ii >= 0.)
    {
      for (j = ii; j <= i - 1; j++)
        sum -= lu[i][j] * back[j];
    }
    else if (sum)
      ii = i;
    back[i] = sum;
  }
  for (i = N - 1; i >= 0; i--)
  {
    sum = back[i];
    for (j = i + 1; j < N; j++)
      sum -= lu[i][j] * back[j];
    back[i] = sum / lu[i][i];
  }
}
```

```
/*************************************************************
 *
 * File........:  svd_inversion.c
 * Function....:  matrix inversion via SVD
 * Author......:  Tilo Strutz
 * last changes:  18.01.2010
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
 *************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <math.h>
#include "errmsg.h"
#include "matrix_utils.h"
#include "macros.h"
#include "prototypes.h"
#include "functions.h"

/*-------------------------------------------------------------
 *  svd_inversion()
 *
 *-------------------------------------------------------------*/
int
svd_inversion( int M, double **normal, double **normal_i, FILE *out)
{
  char *rtn = "svd_inversion";
  int i, j, err = 0;
  double thresh, smax;
  double **tmpmat = NULL; /* temporary matrix */
  double *s = NULL; /* singular values */
  double **V = NULL;  /* V matrix */

    V = matrix( M, M);  /* V matrix for SVD */
    s = vector( M); /* singular values for SVD */
    tmpmat = matrix( M, M); /* temporary matrix */

    err = singvaldec( normal, M, M, s, V);
    if (err)
    {
        free_matrix( &V);
        free_vector( &s);
        free_matrix( &tmpmat);
        goto endfunc;
    }

    smax = 0.0;
    for (j = 0; j < M; j++)
    {
      if (s[j] > smax)          smax = s[j];
    }
    if (smax < TOL_S2)
    {
      fprintf( stderr,
        "\n###\n###     singular matrix, smax = %f",smax);
      fprintf( out,
        "\n###\n###     singular matrix, smax = %f",smax);

      err = 1;
      goto endfunc;
    }
    else if (smax > 1.e+31)
    {
      fprintf( stderr,
        "\n###\n###     degraded matrix, smax = huge");
      fprintf( out,
        "\n###\n###     degraded matrix, smax = huge");
      err = 1;
      goto endfunc;
    }

    thresh = MIN( TOL_S * smax, TOL_S);

    /* invert singular values */
    for (j = 0; j < M; j++)
    {
      /* <= in case of smax =0 */
      if (s[j] <= thresh)
          s[j] = 0.0;
      else
          s[j] = 1. / s[j];
    }

    /* V * [diag(1/s[j])] */
    for (i = 0; i < M; i++)
    {
      for (j = 0; j < M; j++)
      {
        tmpmat[i][j] = V[i][j] * s[j];
      }
    }
    /* get inverse of normal by multiplication of tmpmat with
       transposed of normal */
    multmatsqT( M, normal_i, tmpmat, normal);


endfunc:
  free_vector( &s);
  free_matrix( &V);
  free_matrix( &tmpmat);

  return err;
}
```

```
/*************************************************************
 *
 * File........:  singvaldec.c
 * Function....:  singular value decomposition
 * Author......:  Tilo Strutz
 * last changes:  20.10.2007
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
 *************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <assert.h>
#include "matrix_utils.h"
#include "macros.h"

#define SIGN(a,b) ((b) >= 0.0 ? fabs(a) : -fabs(a))


/*-------------------------------------------------------------
 *  euclid_dist()
 *-------------------------------------------------------------*/
double
euclid_dist( double a, double b)
```

```
29: {
30:   double abs_a, abs_b, val, dval;
31:
32:   abs_a = fabs( a);
33:   abs_b = fabs( b);
34:   if (abs_a > abs_b)
35:   {
36:     dval = abs_b / abs_a;
37:     val = abs_a * sqrt( 1.0 + dval * dval);
38:     return val;
39:   }
40:   else
41:   {
42:     if (abs_b == 0.0)
43:       return 0.0;
44:     else
45:     {
46:       dval = abs_a / abs_b;
47:       val = abs_b * sqrt( 1.0 + dval * dval);
48:       return val;
49:     }
50:   }
51: }
52:
53: /*---------------------------------------------------------------
54:  *  singvaldec()
55:  *  singular value decomposition
56:  * translation from http://www.pdas.com/programs/fmm.f90
57:  *--------------------------------------------------------------*/
58: int
59: singvaldec( double **a, /* matrix to be decomposed */
60:             int N, /* number of lines */
61:             int M, /* number of columns  */
62:             double w[], double **v)
63: {
64:   char *rtn = "singvaldec";
65:   int err = 0;
66:   int flag, i, its, j, jj, k, l = 0, nm;
67:   double anorm, c, f, g, h, s, scale, x, y, z, *rv1;
68:
69:   rv1 = vector( M);
70:
71:   /*
72:    * housholder reduction to bidiagonal form
73:    */
74:   g = scale = anorm = 0.0;
75:   for (i = 0; i < M; i++)
76:   {
77:     l = i + 1;
78:     assert( i >= 0);
79:     rv1[i] = scale * g;
80:     g = s = scale = 0.0;
81:     if (i < N)
82:     {
83:       for (k = i; k < N; k++)
84:       {
85:         scale += fabs( a[k][i]);
86:       }
87:       if (scale)
88:       {
89:         for (k = i; k < N; k++)
90:         {
91:           a[k][i] /= scale;
92:           s += a[k][i] * a[k][i];
93:         }
94:         f = a[i][i];
95:         assert( s>= 0);
96:         g = -SIGN( sqrt( s), f);
97:         h = f * g - s;
98:         a[i][i] = f - g;
99:         for (j = l; j < M; j++)
100:        {
101:          s = 0.0;
102:          for (k = i; k < N; k++)
103:          {
104:            s += a[k][i] * a[k][j];
105:          }
106:          f = s / h;
107:          for (k = i; k < N; k++)
108:          {
109:            a[k][j] += f * a[k][i];
110:          }
111:        }
112:        for (k = i; k < N; k++)
113:          a[k][i] *= scale;
114:      }
115:    }
116:    w[i] = scale * g;
117:    g = s = scale = 0.0;
118:    if (i < N && i != M - 1)
119:    {
120:      for (k = l; k < M; k++)
121:      {
122:        scale += fabs( a[i][k]);
123:      }
124:      if (scale)
125:      {
126:        for (k = l; k < M; k++)
127:        {
128:          a[i][k] /= scale;
129:          s += a[i][k] * a[i][k];
130:        }
131:        f = a[i][l];
132:        assert( s>= 0);
133:        g = -SIGN( sqrt( s), f);
134:        h = f * g - s;
135:        a[i][l] = f - g;
136:        for (k = l; k < M; k++)
137:        {
138:          assert( k >= 0);
139:          rv1[k] = a[i][k] / h;
140:        }
141:        for (j = l; j < N; j++)
142:        {
143:          for (s = 0.0, k = l; k < M; k++)
144:            s += a[j][k] * a[i][k];
145:          for (k = l; k < M; k++)
146:            a[j][k] += s * rv1[k];
147:        }
148:        for (k = l; k < M; k++)
149:          a[i][k] *= scale;
150:      }
151:    }
152:    anorm = MAX( anorm, (fabs( w[i]) + fabs( rv1[i])));
153:  }
154:
155:  /*
156:   * accumulation of right-hand transformations
157:   */
158:  for (i = M - 1; i >= 0; i--)
159:  {
160:    if (i < M - 1)
161:    {
162:      if (g)
163:      {
164:        for (j = l; j < M; j++)
165:        {
166:          v[j][i] = (a[i][j] / a[i][l]) / g;
167:        }
168:        for (j = l; j < M; j++)
169:        {
170:          for (s = 0.0, k = l; k < M; k++)
171:            s += a[i][k] * v[k][j];
172:          for (k = l; k < M; k++)
173:            v[k][j] += s * v[k][i];
174:        }
175:      }
176:      for (j = l; j < M; j++)
177:        v[i][j] = v[j][i] = 0.0;
178:    }
179:    v[i][i] = 1.0;
180:    assert( i >= 0);
181:    g = rv1[i];
182:    l = i;
183:  }
184:
185:  /*
186:   * accumulation of left-hand transformations
187:   */
188:  for (i = MIN( N, M) - 1; i >= 0; i--)
189:  {
190:    l = i + 1;
191:    g = w[i];
192:    for (j = l; j < M; j++)
193:      a[i][j] = 0.0;
194:    if (g)
195:    {
196:      g = 1.0 / g;
197:      for (j = l; j < M; j++)
198:      {
199:        for (s = 0.0, k = l; k < N; k++)
200:          s += a[k][i] * a[k][j];
201:        f = (s / a[i][i]) * g;
202:        for (k = i; k < N; k++)
203:          a[k][j] += f * a[k][i];
204:      }
205:      for (j = i; j < N; j++)
206:        a[j][i] *= g;
207:    }
208:    else
209:    {
210:      for (j = i; j < N; j++)
211:        a[j][i] = 0.0;
212:    }
213:    ++a[i][i];
214:  }
215:
216:  /*
217:   * diagonalization of the bidiagonal form
218:   */
219:  for (k = M - 1; k >= 0; k--)  /* loop over singular values */
220:  {
```

```
221:     assert( k >= 0);
222:     for (its = 0; its < 30; its++)  /* loop over allowed
223:                                        iterations */
224:     {
225:       flag = 1;
226:       for (l = k; l >= 0; l--)  /* test for splitting */
227:       {
228:         nm = l - 1; /* note that rv1[0] is always zero */
229:         if (( double)( fabs( rv1[l]) + anorm) == anorm)
230:         {
231:           flag = 0;
232:           break;
233:         }
234:         if (( double)( fabs( w[nm]) + anorm) == anorm)
235:           break;
236:       }
237:       if (flag)
238:       {
239:         assert( l >= 0);
240:         c = 0.0;  /* cancellation of rv1[l] if l greater than 1 */
241:         s = 1.0;
242:         for (i = l; i < k; i++)
243:         {
244:           f = s * rv1[i];
245:           rv1[i] = c * rv1[i];
246:           if (( double)( fabs( f) + anorm) == anorm)
247:             break;
248:           g = w[i];
249:           h = euclid_dist( f, g);
250:           w[i] = h;
251:           h = 1.0 / h;
252:           c = g * h;
253:           s = -f * h;
254:           for (j = 0; j < N; j++)
255:           {
256:             y = a[j][nm];
257:             z = a[j][i];
258:             a[j][nm] = y * c + z * s;
259:             a[j][i] = z * c - y * s;
260:           }
261:         }
262:       }
263:       /* test for convergence */
264:       z = w[k];
265:       if (l == k)
266:       {
267:         if (z < 0.0)  /* singular value is made non-negative */
268:         {
269:           w[k] = -z;
270:           for (j = 0; j < M; j++)
271:             v[j][k] = -v[j][k];
272:         }
273:         break;
274:       }
275:       if (its == 30)
276:       {
277:         fprintf( stderr,
278:           "\n%s: No convergence after 30 iterations(SVD)\n", rtn);
279:         err = 57;
280:         goto endfunc;
281:       }
282:       /* shift from bottom 2 by 2 minor */
283:       x = w[l];
284:       nm = k - 1;
285:       y = w[nm];
286:       g = rv1[nm];
287:         assert( k >= 0);
288:       h = rv1[k];
289:       f =
290:         ( (y - z) * (y + z) + (g - h) * (g +
291:         h)) / (2.0 * h * y);
292:       g = euclid_dist( f, 1.0);
293:       f =
294:         ( (x - z) * (x + z) + h * (( y / (f + SIGN( g,
295:         f))) - h)) / x;
296:
297:       /* next qr transformation */
298:       c = s = 1.0;
299:       for (j = l; j <= nm; j++)
300:       {
301:         i = j + 1;
302:         assert( i >= 0);
303:         g = rv1[i];
304:         y = w[i];
305:         h = s * g;
306:         g = c * g;
307:         z = euclid_dist( f, h);
308:         rv1[j] = z;
309:         c = f / z;
310:         s = h / z;
311:         f = x * c + g * s;
312:         g = g * c - x * s;
313:         h = y * s;
314:         y *= c;
315:         for (jj = 0; jj < M; jj++)
316:         {
317:           x = v[jj][j];
318:           z = v[jj][i];
319:           v[jj][j] = x * c + z * s;
320:           v[jj][i] = z * c - x * s;
321:         }
322:         z = euclid_dist( f, h);
323:         w[j] = z;
324:         if (z)  /* rotation can be arbitrary if z is zero */
325:         {
326:           z = 1.0 / z;
327:           c = f * z;
328:           s = h * z;
329:         }
330:         f = c * g + s * y;
331:         x = c * y - s * g;
332:         for (jj = 0; jj < N; jj++)
333:         {
334:           y = a[jj][j];
335:           z = a[jj][i];
336:           a[jj][j] = y * c + z * s;
337:           a[jj][i] = z * c - y * s;
338:         }
339:       }
340:       assert( l >= 0);
341:       assert( k >= 0);
342:       rv1[l] = 0.0;
343:       rv1[k] = f;
344:       w[k] = x;
345:     }
346:   }
347:
348: endfunc:
349:   free_vector( &rv1);
350:   return err;
351: }
```

S.5.2 Allocation and matrix handling

```
  0: /*************************************************************
  1:  *
  2:  * File........:  matrix_utils.c
  3:  * Function....:  special functions for matrices
  4:  * Author......:  Tilo Strutz
  5:  * last changes:  20.10.2009, 01.01.2011, 29.3.2011
  6:  *
  7:  * LICENCE DETAILS: see software manual
  8:  *  free academic use
  9:  *  cite source as
 10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 11:  *  2nd edition 2015"
 12:  *
 13:  *************************************************************/
 14: #include <stdio.h>
 15: #include <stdlib.h>
 16: #include <math.h>
 17: #include "errmsg.h"
 18:
 19: /*-------------------------------------------------------------
 20:  *  vector()
 21:  *  create a vector with subscript range v[0..N-1]
 22:  *-------------------------------------------------------------*/
 23: double *vector( long N)
 24: {
 25:   int err = 0;
 26:   double *v;
 27:
 28:   v = (double*)calloc( N, sizeof(double));
 29:   if (v == NULL)
 30:   {
 31:     err = errmsg( ERR_ALLOCATE, "vector", " ", 0);
 32:     exit( err);
 33:   }
 34:   return v;
 35: }
 36:
 37: /*-------------------------------------------------------------
 38:  *  fvector()
 39:  *  create a vector with subscript range v[0..N-1]
 40:  *-------------------------------------------------------------*/
 41: float *fvector( long N)
 42: {
 43:   int err = 0;
 44:   float *v;
 45:
 46:   v = (float*)calloc( N, sizeof(float));
 47:   if (v == NULL)
 48:   {
 49:     err = errmsg( ERR_ALLOCATE, "vector", " ", 0);
 50:     exit( err);
 51:   }
 52:   return v;
 53: }
 54:
```

```
55: /*-------------------------------------------------------------
56:  *  ivector()
57:  *  create a vector with subscript range v[0..N-1]
58:  *-------------------------------------------------------------*/
59: int *
60: ivector( long N)
61: {
62:   int err = 0;
63:   int *v;
64:
65:   v = (int*)calloc( N, sizeof(int));
66:   if (v == NULL)
67:   {
68:     err = errmsg( ERR_ALLOCATE, "vector", " ", 0);
69:     exit( err);
70:   }
71:   return v;
72: }
73: /*-------------------------------------------------------------
74:  *  uivector()
75:  *  create a vector with subscript range v[0..N-1]
76:  *-------------------------------------------------------------*/
77: unsigned int *
78: uivector( long N)
79: {
80:   int err = 0;
81:   unsigned int *v;
82:
83:   v = (unsigned int*)calloc( N, sizeof(unsigned int));
84:   if (v == NULL)
85:   {
86:     err = errmsg( ERR_ALLOCATE, "vector", " ", 0);
87:     exit( err);
88:   }
89:   return v;
90: }
91:
92: /*-------------------------------------------------------------
93:  *  matrix()
94:  *  create a matrix with subscript range v[0..M-1][0..N-1]
95:  *-------------------------------------------------------------*/
96: double **
97: matrix( long N, long M)
98: {
99:   int err = 0;
100:   long i;
101:   double **m;
102:
103:   /* allocate pointers to rows */
104:   m = (double **)malloc( N * sizeof(double*));
105:   if (m == NULL)
106:   {
107:     err = errmsg( ERR_ALLOCATE, "matrix", " ", 0);
108:     exit( err);
109:   }
110:
111:   /* allocate rows and set pointers to them */
112:   m[0] = (double*)calloc( M * N, sizeof(double));
113:   if (m[0] == NULL)
114:   {
115:     err = errmsg( ERR_ALLOCATE, "matrix", " ", 0);
116:     exit( err);
117:   }
118:
119:   for (i = 1; i < N; i++)
120:   {
121:     m[i] = m[i - 1] + M;
122:   }
123:
124:   /* return pointer to array of pointers to rows */
125:   return m;
126: }
127: /*-------------------------------------------------------------
128:  *  fmatrix()
129:  *  create a matrix with subscript range v[0..M-1][0..N-1]
130:  *-------------------------------------------------------------*/
131: float **
132: fmatrix( long N, long M)
133: {
134:   int err = 0;
135:   long i;
136:   float **m;
137:
138:   /* allocate pointers to rows */
139:   m = (float **)malloc( N * sizeof(float*));
140:   if (m == NULL)
141:   {
142:     err = errmsg( ERR_ALLOCATE, "matrix", " ", 0);
143:     exit( err);
144:   }
145:
146:   /* allocate rows and set pointers to them */
147:   m[0] = (float*)calloc( M * N, sizeof(float));
148:   if (m[0] == NULL)
149:   {
150:     err = errmsg( ERR_ALLOCATE, "matrix", " ", 0);
151:     exit( err);
152:   }
153:
154:   for (i = 1; i < N; i++)
155:   {
156:     m[i] = m[i - 1] + M;
157:   }
158:
159:   /* return pointer to array of pointers to rows */
160:   return m;
161: }
162:
163: /*-------------------------------------------------------------
164:  * free a vector allocated by vector()
165:  *-------------------------------------------------------------*/
166: void
167: free_vector( double *v[])
168: {
169:   if (*v != NULL)
170:     free( *v);
171:   *v = NULL;
172: }
173:
174: /*-------------------------------------------------------------
175:  * free a vector allocated by ivector()
176:  *-------------------------------------------------------------*/
177: void
178: free_ivector( int *v[])
179: {
180:   if (*v != NULL)
181:     free( *v);
182:   *v = NULL;
183: }
184: /*-------------------------------------------------------------
185:  * free a vector allocated by uivector()
186:  *-------------------------------------------------------------*/
187: void
188: free_uivector( unsigned int *v[])
189: {
190:   if (*v != NULL)
191:     free( *v);
192:   *v = NULL;
193: }
194:
195: /*-------------------------------------------------------------
196:  * free a matrix allocated by matrix()
197:  *-------------------------------------------------------------*/
198: void
199: free_matrix( double **m[])
200: {
201:   if (*m != NULL)
202:   {
203:     if (*m[0] != NULL)
204:       free( *m[0]);
205:     free( *m);
206:   }
207:   *m = NULL;
208: }
209:
210: /*-------------------------------------------------------------
211:  * free a matrix allocated by fmatrix()
212:  *-------------------------------------------------------------*/
213: void
214: free_fmatrix( float **m[])
215: {
216:   if (*m != NULL)
217:   {
218:     if (*m[0] != NULL)
219:       free( *m[0]);
220:     free( *m);
221:   }
222:   *m = NULL;
223: }
224:
225: /*-------------------------------------------------------------
226:  *  determinant_2x2()
227:  *-------------------------------------------------------------*/
228: double
229: determinant_2x2( double **a)
230: {
231:   return a[0][0] * a[1][1] - a[0][1] * a[1][0];
232: }
233:
234: /*-------------------------------------------------------------
235:  *  determinant_3x3()
236:  *-------------------------------------------------------------*/
237: double
238: determinant_3x3( double **a)
239: {
240:   /* The numerical stability depends on the order of operations.
241:    * The discrimination below works for the mentioned data sets
242:    * in Release mode, but should evaluated in more detail
243:    */
244:   if (fabs(a[0][0]) < 1)
245:     return
246:       /* better performance for Eckerle4.dat  */
```

```
      (
      a[0][0] * (a[1][1] * a[2][2] - a[2][1] * a[1][2]) -
      a[0][1] * (a[1][0] * a[2][2] + a[2][0] * a[1][2])
      ) +
      a[0][2] * (a[1][0] * a[2][1] - a[2][0] * a[1][1]);
  else
    return
      /* better performance for MGH10.dat*/
      a[0][0] * a[1][1] * a[2][2] - a[0][0] * a[2][1] * a[1][2] -
      a[0][1] * a[1][0] * a[2][2] + a[0][1] * a[2][0] * a[1][2] +
      a[0][2] * a[1][0] * a[2][1] - a[0][2] * a[2][0] * a[1][1];
}

/*-----------------------------------------------------------------
 *  inverse_5x5()
 * get the inverse of a square 5x5 matrix
 * returns determinant
 *-----------------------------------------------------------------*/
double
inverse_5x5( double **a, double **b)
{
  double det;

  det =
    a[0][0] * (
      a[1][1] * (+a[2][2]*a[3][3]*a[4][4] - a[2][2]*a[4][3]*a[3][4]
                 -a[3][2]*a[2][3]*a[4][4] - a[3][3]*a[4][2]*a[2][4]
                 +a[4][2]*a[2][3]*a[3][4] + a[4][3]*a[3][2]*a[2][4] ) +
      a[2][2] * (-a[3][1]*a[1][3]*a[4][4] - a[3][3]*a[4][1]*a[1][4]
                 +a[4][1]*a[1][3]*a[3][4] + a[4][3]*a[3][1]*a[1][4] ) +
      a[3][1] * (+a[1][2]*a[2][3]*a[4][4] + a[1][3]*a[4][2]*a[2][4]
      a[3][2] * (+a[2][1]*a[1][3]*a[4][4] + a[2][3]*a[4][1]*a[1][4]
      a[3][3] * (-a[2][1]*a[1][2]*a[4][4] + a[4][1]*a[1][2]*a[2][4]
                 +a[4][2]*a[2][1]*a[1][4] ) +
      a[4][1] * (-a[1][2]*a[2][3]*a[3][4] - a[1][3]*a[3][2]*a[2][4]
      a[4][2] * (-a[2][1]*a[1][3]*a[3][4] - a[2][3]*a[3][1]*a[1][4]
      a[4][3] * (+a[2][1]*a[1][2]*a[3][4] - a[3][1]*a[1][2]*a[2][4]
                 -a[3][2]*a[2][1]*a[1][4] )
             ) +
    a[1][1] * (
      a[2][2] * (-a[3][0]*a[0][3]*a[4][4] - a[3][3]*a[4][0]*a[0][4]
                 +a[4][0]*a[0][3]*a[3][4] + a[4][3]*a[3][0]*a[0][4] )
      a[3][0] * (+a[0][2]*a[2][3]*a[4][4] + a[0][3]*a[4][2]*a[2][4]
      a[3][2] * (+a[2][0]*a[0][3]*a[4][4] + a[2][3]*a[4][0]*a[0][4]
      a[3][3] * (-a[2][0]*a[0][2]*a[4][4] + a[4][0]*a[0][2]*a[2][4]
                 +a[4][2]*a[2][0]*a[0][4] ) +
      a[4][0] * (-a[0][2]*a[2][3]*a[3][4] - a[0][3]*a[3][2]*a[2][4]
      a[4][2] * (-a[2][0]*a[0][3]*a[3][4] - a[2][3]*a[3][0]*a[0][4]
      a[4][3] * (+a[2][0]*a[0][2]*a[3][4] - a[3][0]*a[0][2]*a[2][4]
                 -a[3][2]*a[2][0]*a[0][4] )
           ) +
    a[2][2] * (
      a[3][0] * (+a[0][1]*a[1][3]*a[4][4] + a[0][3]*a[4][1]*a[1][4]
      a[3][1] * (+a[1][0]*a[0][3]*a[4][4] + a[1][3]*a[4][0]*a[0][4]
      a[3][3] * (-a[1][0]*a[0][1]*a[4][4] + a[4][0]*a[0][1]*a[1][4]
                 +a[4][1]*a[1][0]*a[0][4] ) +
      a[4][0] * (-a[0][1]*a[1][3]*a[3][4] - a[0][3]*a[3][1]*a[1][4]
      a[4][1] * (-a[1][0]*a[0][3]*a[3][4] - a[1][3]*a[3][0]*a[0][4]
      a[4][3] * (+a[1][0]*a[0][1]*a[3][4] - a[3][0]*a[0][1]*a[1][4]
                 -a[3][1]*a[1][0]*a[0][4] )
           ) +
    a[3][0] * (
      a[0][1] * (-a[1][2]*a[2][3]*a[4][4] - a[1][3]*a[4][2]*a[2][4]
      a[0][2] * (-a[2][1]*a[1][3]*a[4][4] - a[2][3]*a[4][1]*a[1][4]
      a[0][3] * (+a[2][1]*a[1][2]*a[4][4] - a[4][1]*a[1][2]*a[2][4]
                 -a[4][2]*a[2][1]*a[1][4] )
           ) +
    a[3][1] * (
      a[1][0] * (-a[0][2]*a[2][3]*a[4][4] - a[0][3]*a[4][2]*a[2][4]
      a[1][2] * (-a[2][0]*a[0][3]*a[4][4] - a[2][3]*a[4][0]*a[0][4]
      a[1][3] * (+a[2][0]*a[0][2]*a[4][4] - a[4][0]*a[0][2]*a[2][4]
                 -a[4][2]*a[2][0]*a[0][4] )
           ) +
    a[3][2] * (
      a[2][0] * (-a[0][1]*a[1][3]*a[4][4] - a[0][3]*a[4][1]*a[1][4]
      a[2][1] * (-a[1][0]*a[0][3]*a[4][4] - a[1][3]*a[4][0]*a[0][4]
      a[2][3] * (+a[1][0]*a[0][1]*a[4][4] - a[4][0]*a[0][1]*a[1][4]
                 -a[4][1]*a[1][0]*a[0][4] )
           ) +
    a[3][3] * (
      a[2][0] * (+a[0][1]*a[1][2]*a[4][4] + a[0][2]*a[4][1]*a[1][4]
      a[2][1] * (+a[1][0]*a[0][2]*a[4][4] + a[1][2]*a[4][0]*a[0][4]
      a[4][0] * (-a[0][1]*a[1][2]*a[2][4] - a[0][2]*a[2][1]*a[1][4]
      a[4][1] * (-a[1][0]*a[0][2]*a[2][4] - a[1][2]*a[2][0]*a[0][4]
      a[4][2] * (+a[1][0]*a[0][1]*a[2][4] - a[2][0]*a[0][1]*a[1][4]
                 -a[2][1]*a[1][0]*a[0][4] )
           ) +
    a[4][0] * (
      a[0][1] * (+a[1][2]*a[2][3]*a[3][4] + a[1][3]*a[3][2]*a[2][4]
      a[0][2] * (+a[2][1]*a[1][3]*a[3][4] + a[2][3]*a[3][1]*a[1][4]
      a[0][3] * (-a[2][1]*a[1][2]*a[3][4] + a[3][1]*a[1][2]*a[2][4]
                 +a[3][2]*a[2][1]*a[1][4] )
           ) +
    a[4][1] * (
      a[1][0] * (+a[0][2]*a[2][3]*a[3][4] + a[0][3]*a[3][2]*a[2][4]
      a[1][2] * (+a[2][0]*a[0][3]*a[3][4] + a[2][3]*a[3][0]*a[0][4]
      a[1][3] * (-a[2][0]*a[0][2]*a[3][4] - a[3][0]*a[0][2]*a[2][4]
                 -a[3][2]*a[2][0]*a[0][4] )
           ) +
    a[4][2] * (
      a[1][0] * (-a[0][2]*a[2][3]*a[3][4] - a[0][3]*a[3][2]*a[2][4]
      a[1][2] * (-a[2][0]*a[0][3]*a[3][4] - a[2][3]*a[3][0]*a[0][4]
      a[1][3] * (+a[2][0]*a[0][2]*a[3][4] + a[3][0]*a[0][2]*a[2][4]
                 +a[3][2]*a[2][0]*a[0][4] )
           ) +
    a[4][3] * (
      a[1][0] * (+a[0][2]*a[2][3]*a[3][4] + a[0][3]*a[3][2]*a[2][4]
      a[1][2] * (+a[2][0]*a[0][3]*a[3][4] + a[2][3]*a[3][0]*a[0][4]
      a[1][3] * (-a[2][0]*a[0][2]*a[3][4] - a[3][0]*a[0][2]*a[2][4]
                 +a[3][2]*a[2][0]*a[0][4] )
           ) +
    a[4][4] * (
      a[1][0] * (+a[0][2]*a[2][3]*a[3][4] + a[0][3]*a[3][2]*a[2][4]
      a[1][2] * (+a[2][0]*a[0][3]*a[3][4] + a[2][3]*a[3][0]*a[0][4]
      a[1][3] * (-a[2][0]*a[0][2]*a[3][4] - a[3][0]*a[0][2]*a[2][4]
                 +a[3][2]*a[2][0]*a[0][4] )
           );

  b[0][0] = -(
    a[1][1] * (
              a[2][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
              a[2][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
              a[2][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
              )+
    a[1][2] * (
              a[2][1] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
              a[2][3] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
              a[2][4] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1])
              )+
    a[1][3] * (
              a[2][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
              a[2][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
              a[2][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
              )+
    a[1][4] * (
              a[2][1] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
              a[2][2] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
              a[2][3] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1])
              ));

  b[0][1] =
    a[0][1] * (
              a[2][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
              a[2][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
              a[2][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
              )+
    a[0][2] * (
              a[2][1] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
              a[2][3] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
              a[2][4] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1])
              )+
    a[0][3] * (
              a[2][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
              a[2][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
              a[2][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
              )+
    a[0][4] * (
              a[2][1] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
              a[2][2] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
              a[2][3] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1])
              );

  b[0][2] =
    a[1][1] * (
              a[0][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
              a[0][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
              a[0][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
              )+
    a[1][2] * (
              a[0][1] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
              a[0][3] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
              a[0][4] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1])
              )+
    a[1][3] * (
              a[0][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
              a[0][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
              a[0][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
              )+
    a[1][4] * (
              a[0][1] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
              a[0][2] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
              a[0][3] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1])
              );

  b[0][3] = +(
    a[1][1] * (
              a[2][2] * (-a[0][3]*a[4][4] + a[0][4]*a[4][3]) +
              a[2][3] * (+a[0][2]*a[4][4] - a[0][4]*a[4][2]) +
              a[2][4] * (-a[0][2]*a[4][3] + a[0][3]*a[4][2])
              )+
    a[1][2] * (
              a[2][1] * (+a[0][3]*a[4][4] - a[0][4]*a[4][3]) +
              a[2][3] * (-a[0][1]*a[4][4] + a[0][4]*a[4][1]) +
              a[2][4] * (+a[0][1]*a[4][3] - a[0][3]*a[4][1])
              )+
    a[1][3] * (
```

```
439:                a[2][1] * (-a[0][2]*a[4][4] + a[0][4]*a[4][2]) +
440:                a[2][2] * (+a[0][1]*a[4][4] - a[0][4]*a[4][1]) +
441:                a[2][4] * (-a[0][1]*a[4][2] + a[0][2]*a[4][1])
442:                )+
443:     a[1][4] * (
444:                a[2][1] * (+a[0][2]*a[4][3] - a[0][3]*a[4][2]) +
445:                a[2][2] * (-a[0][1]*a[4][3] + a[0][3]*a[4][1]) +
446:                a[2][3] * (+a[0][1]*a[4][2] - a[0][2]*a[4][1])
447:                ));
448:
449:    b[0][4] =
450:      a[1][1] * (
451:                a[2][2] * (-a[3][3]*a[0][4] + a[3][4]*a[0][3]) +
452:                a[2][3] * (+a[3][2]*a[0][4] - a[3][4]*a[0][2]) +
453:                a[2][4] * (-a[3][2]*a[0][3] + a[3][3]*a[0][2])
454:                )+
455:      a[1][2] * (
456:                a[2][1] * (+a[3][3]*a[0][4] - a[3][4]*a[0][3]) +
457:                a[2][3] * (-a[3][1]*a[0][4] + a[3][4]*a[0][1]) +
458:                a[2][4] * (+a[3][1]*a[0][3] - a[3][3]*a[0][1])
459:                )+
460:      a[1][3] * (
461:                a[2][1] * (-a[3][2]*a[0][4] + a[3][4]*a[0][2]) +
462:                a[2][2] * (+a[3][1]*a[0][4] - a[3][4]*a[0][1]) +
463:                a[2][4] * (-a[3][1]*a[0][2] + a[3][2]*a[0][1])
464:                )+
465:      a[1][4] * (
466:                a[2][1] * (+a[3][2]*a[0][3] - a[3][3]*a[0][2]) +
467:                a[2][2] * (-a[3][1]*a[0][3] + a[3][3]*a[0][1]) +
468:                a[2][3] * (+a[3][1]*a[0][2] - a[3][2]*a[0][1])
469:                );
470:   /* second row */
471:    b[1][0] =
472:      a[1][0] * (
473:                a[2][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
474:                a[2][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
475:                a[2][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
476:                )+
477:      a[1][2] * (
478:                a[2][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
479:                a[2][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
480:                a[2][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
481:                )+
482:      a[1][3] * (
483:                a[2][0] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
484:                a[2][2] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
485:                a[2][4] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0])
486:                )+
487:      a[1][4] * (
488:                a[2][0] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
489:                a[2][2] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
490:                a[2][3] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
491:                );
492:
493:    b[1][1] = -(
494:      a[0][0] * (
495:                a[2][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
496:                a[2][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
497:                a[2][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
498:                )+
499:      a[0][2] * (
500:                a[2][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
501:                a[2][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
502:                a[2][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
503:                )+
504:      a[0][3] * (
505:                a[2][0] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
506:                a[2][2] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
507:                a[2][4] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0])
508:                )+
509:      a[0][4] * (
510:                a[2][0] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
511:                a[2][2] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
512:                a[2][3] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
513:                ));
514:
515:    b[1][2] = +(
516:      a[0][0] * (
517:                a[1][2] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
518:                a[1][3] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
519:                a[1][4] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
520:                )+
521:      a[0][2] * (
522:                a[1][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
523:                a[1][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
524:                a[1][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
525:                )+
526:      a[0][3] * (
527:                a[1][0] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
528:                a[1][2] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
529:                a[1][4] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0])
530:                )+
531:      a[0][4] * (
532:                a[1][0] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
533:                a[1][2] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
534:                a[1][3] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
535:                ));
536:
537:    b[1][3] = -(
538:      a[0][0] * (
539:                a[1][2] * (-a[2][3]*a[4][4] + a[2][4]*a[4][3]) +
540:                a[1][3] * (+a[2][2]*a[4][4] - a[2][4]*a[4][2]) +
541:                a[1][4] * (-a[2][2]*a[4][3] + a[2][3]*a[4][2])
542:                )+
543:      a[0][2] * (
544:                a[1][0] * (+a[2][3]*a[4][4] - a[2][4]*a[4][3]) +
545:                a[1][3] * (-a[2][0]*a[4][4] + a[2][4]*a[4][0]) +
546:                a[1][4] * (+a[2][0]*a[4][3] - a[2][3]*a[4][0])
547:                )+
548:      a[0][3] * (
549:                a[1][0] * (-a[2][2]*a[4][4] + a[2][4]*a[4][2]) +
550:                a[1][2] * (+a[2][0]*a[4][4] - a[2][4]*a[4][0]) +
551:                a[1][4] * (-a[2][0]*a[4][2] + a[2][2]*a[4][0])
552:                )+
553:      a[0][4] * (
554:                a[1][0] * (+a[2][2]*a[4][3] - a[2][3]*a[4][2]) +
555:                a[1][2] * (-a[2][0]*a[4][3] + a[2][3]*a[4][0]) +
556:                a[1][3] * (+a[2][0]*a[4][2] - a[2][2]*a[4][0])
557:                ));
558:
559:    b[1][4] = +(
560:      a[0][0] * (
561:                a[1][2] * (-a[2][3]*a[3][4] + a[2][4]*a[3][3]) +
562:                a[1][3] * (+a[2][2]*a[3][4] - a[2][4]*a[3][2]) +
563:                a[1][4] * (-a[2][2]*a[3][3] + a[2][3]*a[3][2])
564:                )+
565:      a[0][2] * (
566:                a[1][0] * (+a[2][3]*a[3][4] - a[2][4]*a[3][3]) +
567:                a[1][3] * (-a[2][0]*a[3][4] + a[2][4]*a[3][0]) +
568:                a[1][4] * (+a[2][0]*a[3][3] - a[2][3]*a[3][0])
569:                )+
570:      a[0][3] * (
571:                a[1][0] * (-a[2][2]*a[3][4] + a[2][4]*a[3][2]) +
572:                a[1][2] * (+a[2][0]*a[3][4] - a[2][4]*a[3][0]) +
573:                a[1][4] * (-a[2][0]*a[3][2] + a[2][2]*a[3][0])
574:                )+
575:      a[0][4] * (
576:                a[1][0] * (+a[2][2]*a[3][3] - a[2][3]*a[3][2]) +
577:                a[1][2] * (-a[2][0]*a[3][3] + a[2][3]*a[3][0]) +
578:                a[1][3] * (+a[2][0]*a[3][2] - a[2][2]*a[3][0])
579:                ));
580:   /* third row  */
581:    b[2][0] = -(
582:      a[1][0] * (
583:                a[2][1] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
584:                a[2][3] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
585:                a[2][4] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1])
586:                )+
587:      a[1][1] * (
588:                a[2][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
589:                a[2][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
590:                a[2][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
591:                )+
592:      a[1][3] * (
593:                a[2][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
594:                a[2][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
595:                a[2][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
596:                )+
597:      a[1][4] * (
598:                a[2][0] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1]) +
599:                a[2][1] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
600:                a[2][3] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
601:                ));
602:
603:    b[2][1] = +(
604:      a[0][0] * (
605:                a[2][1] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
606:                a[2][3] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
607:                a[2][4] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1])
608:                )+
609:      a[0][1] * (
610:                a[2][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
611:                a[2][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
612:                a[2][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
613:                )+
614:      a[0][3] * (
615:                a[2][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
616:                a[2][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
617:                a[2][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
618:                )+
619:      a[0][4] * (
620:                a[2][0] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1]) +
621:                a[2][1] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
622:                a[2][3] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
623:                ));
624:
625:    b[2][2] = -(
626:      a[0][0] * (
627:                a[1][1] * (-a[3][3]*a[4][4] + a[3][4]*a[4][3]) +
628:                a[1][3] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
629:                a[1][4] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1])
630:                )+
```

```
631:      a[0][1] * (
632:                  a[1][0] * (+a[3][3]*a[4][4] - a[3][4]*a[4][3]) +
633:                  a[1][3] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
634:                  a[1][4] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0])
635:                  )+
636:      a[0][3] * (
637:                  a[1][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
638:                  a[1][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
639:                  a[1][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
640:                  )+
641:      a[0][4] * (
642:                  a[1][0] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1]) +
643:                  a[1][1] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
644:                  a[1][3] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
645:                  ));
646:
647:    b[2][3] = +(
648:      a[0][0] * (
649:                  a[1][1] * (-a[2][3]*a[4][4] + a[2][4]*a[4][3]) +
650:                  a[1][3] * (+a[2][1]*a[4][4] - a[2][4]*a[4][1]) +
651:                  a[1][4] * (-a[2][1]*a[4][3] + a[2][3]*a[4][1])
652:                  )+
653:      a[0][1] * (
654:                  a[1][0] * (+a[2][3]*a[4][4] - a[2][4]*a[4][3]) +
655:                  a[1][3] * (-a[2][0]*a[4][4] + a[2][4]*a[4][0]) +
656:                  a[1][4] * (+a[2][0]*a[4][3] - a[2][3]*a[4][0])
657:                  )+
658:      a[0][3] * (
659:                  a[1][0] * (-a[2][1]*a[4][4] + a[2][4]*a[4][1]) +
660:                  a[1][1] * (+a[2][0]*a[4][4] - a[2][4]*a[4][0]) +
661:                  a[1][4] * (-a[2][0]*a[4][1] + a[2][1]*a[4][0])
662:                  )+
663:      a[0][4] * (
664:                  a[1][0] * (+a[2][1]*a[4][3] - a[2][3]*a[4][1]) +
665:                  a[1][1] * (-a[2][0]*a[4][3] + a[2][3]*a[4][0]) +
666:                  a[1][3] * (+a[2][0]*a[4][1] - a[2][1]*a[4][0])
667:                  ));
668:
669:    b[2][4] = -(
670:      a[0][0] * (
671:                  a[1][1] * (-a[2][3]*a[3][4] + a[2][4]*a[3][3]) +
672:                  a[1][3] * (+a[2][1]*a[3][4] - a[2][4]*a[3][1]) +
673:                  a[1][4] * (-a[2][1]*a[3][3] + a[2][3]*a[3][1])
674:                  )+
675:      a[0][1] * (
676:                  a[1][0] * (+a[2][3]*a[3][4] - a[2][4]*a[3][3]) +
677:                  a[1][3] * (-a[2][0]*a[3][4] + a[2][4]*a[3][0]) +
678:                  a[1][4] * (+a[2][0]*a[3][3] - a[2][3]*a[3][0])
679:                  )+
680:      a[0][3] * (
681:                  a[1][0] * (-a[2][1]*a[3][4] + a[2][4]*a[3][1]) +
682:                  a[1][1] * (+a[2][0]*a[3][4] - a[2][4]*a[3][0]) +
683:                  a[1][4] * (-a[2][0]*a[3][1] + a[2][1]*a[3][0])
684:                  )+
685:      a[0][4] * (
686:                  a[1][0] * (+a[2][1]*a[3][3] - a[2][3]*a[3][1]) +
687:                  a[1][1] * (-a[2][0]*a[3][3] + a[2][3]*a[3][0]) +
688:                  a[1][3] * (+a[2][0]*a[3][1] - a[2][1]*a[3][0])
689:                  ));
690:
691:   /* fourth row */
692:    b[3][0] = +(
693:      a[1][0] * (
694:                  a[2][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
695:                  a[2][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
696:                  a[2][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
697:                  )+
698:      a[1][1] * (
699:                  a[2][0] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
700:                  a[2][2] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
701:                  a[2][4] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
702:                  )+
703:      a[1][2] * (
704:                  a[2][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
705:                  a[2][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
706:                  a[2][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
707:                  )+
708:      a[1][4] * (
709:                  a[2][0] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1]) +
710:                  a[2][1] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0]) +
711:                  a[2][2] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
712:                  ));
713:
714:    b[3][1] = -(
715:      a[0][0] * (
716:                  a[2][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
717:                  a[2][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
718:                  a[2][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
719:                  )+
720:      a[0][1] * (
721:                  a[2][0] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
722:                  a[2][2] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
723:                  a[2][4] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
724:                  )+
725:      a[0][2] * (
726:                  a[2][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
727:                  a[2][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
728:                  a[2][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
729:                  )+
730:      a[0][4] * (
731:                  a[2][0] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1]) +
732:                  a[2][1] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0]) +
733:                  a[2][2] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
734:                  ));
735:
736:    b[3][2] = +(
737:      a[0][0] * (
738:                  a[1][1] * (-a[3][2]*a[4][4] + a[3][4]*a[4][2]) +
739:                  a[1][2] * (+a[3][1]*a[4][4] - a[3][4]*a[4][1]) +
740:                  a[1][4] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
741:                  )+
742:      a[0][1] * (
743:                  a[1][0] * (+a[3][2]*a[4][4] - a[3][4]*a[4][2]) +
744:                  a[1][2] * (-a[3][0]*a[4][4] + a[3][4]*a[4][0]) +
745:                  a[1][4] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
746:                  )+
747:      a[0][2] * (
748:                  a[1][0] * (-a[3][1]*a[4][4] + a[3][4]*a[4][1]) +
749:                  a[1][1] * (+a[3][0]*a[4][4] - a[3][4]*a[4][0]) +
750:                  a[1][4] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
751:                  )+
752:      a[0][4] * (
753:                  a[1][0] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1]) +
754:                  a[1][1] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0]) +
755:                  a[1][2] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
756:                  ));
757:
758:    b[3][3] = -(
759:      a[0][0] * (
760:                  a[1][1] * (-a[2][2]*a[4][4] + a[2][4]*a[4][2]) +
761:                  a[1][2] * (+a[2][1]*a[4][4] - a[2][4]*a[4][1]) +
762:                  a[1][4] * (-a[2][1]*a[4][2] + a[2][2]*a[4][1])
763:                  )+
764:      a[0][1] * (
765:                  a[1][0] * (+a[2][2]*a[4][4] - a[2][4]*a[4][2]) +
766:                  a[1][2] * (-a[2][0]*a[4][4] + a[2][4]*a[4][0]) +
767:                  a[1][4] * (+a[2][0]*a[4][2] - a[2][2]*a[4][0])
768:                  )+
769:      a[0][2] * (
770:                  a[1][0] * (-a[2][1]*a[4][4] + a[2][4]*a[4][1]) +
771:                  a[1][1] * (+a[2][0]*a[4][4] - a[2][4]*a[4][0]) +
772:                  a[1][4] * (-a[2][0]*a[4][1] + a[2][1]*a[4][0])
773:                  )+
774:      a[0][4] * (
775:                  a[1][0] * (+a[2][1]*a[4][2] - a[2][2]*a[4][1]) +
776:                  a[1][1] * (-a[2][0]*a[4][2] + a[2][2]*a[4][0]) +
777:                  a[1][2] * (+a[2][0]*a[4][1] - a[2][1]*a[4][0])
778:                  ));
779:
780:    b[3][4] = +(
781:      a[0][0] * (
782:                  a[1][1] * (-a[2][2]*a[3][4] + a[2][4]*a[3][2]) +
783:                  a[1][2] * (+a[2][1]*a[3][4] - a[2][4]*a[3][1]) +
784:                  a[1][4] * (-a[2][1]*a[3][2] + a[2][2]*a[3][1])
785:                  )+
786:      a[0][1] * (
787:                  a[1][0] * (+a[2][2]*a[3][4] - a[2][4]*a[3][2]) +
788:                  a[1][2] * (-a[2][0]*a[3][4] + a[2][4]*a[3][0]) +
789:                  a[1][4] * (+a[2][0]*a[3][2] - a[2][2]*a[3][0])
790:                  )+
791:      a[0][2] * (
792:                  a[1][0] * (-a[2][1]*a[3][4] + a[2][4]*a[3][1]) +
793:                  a[1][1] * (+a[2][0]*a[3][4] - a[2][4]*a[3][0]) +
794:                  a[1][4] * (-a[2][0]*a[3][1] + a[2][1]*a[3][0])
795:                  )+
796:      a[0][4] * (
797:                  a[1][0] * (+a[2][1]*a[3][2] - a[2][2]*a[3][1]) +
798:                  a[1][1] * (-a[2][0]*a[3][2] + a[2][2]*a[3][0]) +
799:                  a[1][2] * (+a[2][0]*a[3][1] - a[2][1]*a[3][0])
800:                  ));
801:
802:   /* fifth row  */
803:    b[4][0] = +(
804:      a[1][1] * (
805:                  a[2][2] * (-a[3][3]*a[4][0] + a[3][0]*a[4][3]) +
806:                  a[2][3] * (+a[3][2]*a[4][0] - a[3][0]*a[4][2]) +
807:                  a[2][0] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2])
808:                  )+
809:      a[1][2] * (
810:                  a[2][1] * (+a[3][3]*a[4][0] - a[3][0]*a[4][3]) +
811:                  a[2][3] * (-a[3][1]*a[4][0] + a[3][0]*a[4][1]) +
812:                  a[2][0] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1])
813:                  )+
814:      a[1][3] * (
815:                  a[2][1] * (-a[3][2]*a[4][0] + a[3][0]*a[4][2]) +
816:                  a[2][2] * (+a[3][1]*a[4][0] - a[3][0]*a[4][1]) +
817:                  a[2][0] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
818:                  )+
819:      a[1][0] * (
820:                  a[2][1] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
821:                  a[2][2] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
822:                  a[2][3] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1])
```

```
823:                 ));
824:
825:     b[4][1] = +(
826:       a[0][0] * (
827:                 a[2][1] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2]) +
828:                 a[2][2] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1]) +
829:                 a[2][3] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
830:                 )+
831:       a[0][1] * (
832:                 a[2][0] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
833:                 a[2][2] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
834:                 a[2][3] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
835:                 )+
836:       a[0][2] * (
837:                 a[2][0] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
838:                 a[2][1] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0]) +
839:                 a[2][3] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
840:                 )+
841:       a[0][3] * (
842:                 a[2][0] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1]) +
843:                 a[2][1] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0]) +
844:                 a[2][2] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
845:                 ));
846:
847:     b[4][2] = -(
848:       a[0][0] * (
849:                 a[1][1] * (-a[3][2]*a[4][3] + a[3][3]*a[4][2]) +
850:                 a[1][2] * (+a[3][1]*a[4][3] - a[3][3]*a[4][1]) +
851:                 a[1][3] * (-a[3][1]*a[4][2] + a[3][2]*a[4][1])
852:                 )+
853:       a[0][1] * (
854:                 a[1][0] * (+a[3][2]*a[4][3] - a[3][3]*a[4][2]) +
855:                 a[1][2] * (-a[3][0]*a[4][3] + a[3][3]*a[4][0]) +
856:                 a[1][3] * (+a[3][0]*a[4][2] - a[3][2]*a[4][0])
857:                 )+
858:       a[0][2] * (
859:                 a[1][0] * (-a[3][1]*a[4][3] + a[3][3]*a[4][1]) +
860:                 a[1][1] * (+a[3][0]*a[4][3] - a[3][3]*a[4][0]) +
861:                 a[1][3] * (-a[3][0]*a[4][1] + a[3][1]*a[4][0])
862:                 )+
863:       a[0][3] * (
864:                 a[1][0] * (+a[3][1]*a[4][2] - a[3][2]*a[4][1]) +
865:                 a[1][1] * (-a[3][0]*a[4][2] + a[3][2]*a[4][0]) +
866:                 a[1][2] * (+a[3][0]*a[4][1] - a[3][1]*a[4][0])
867:                 ));
868:
869:     b[4][3] = +(
870:       a[0][0] * (
871:                 a[1][1] * (-a[2][2]*a[4][3] + a[2][3]*a[4][2]) +
872:                 a[1][2] * (+a[2][1]*a[4][3] - a[2][3]*a[4][1]) +
873:                 a[1][3] * (-a[2][1]*a[4][2] + a[2][2]*a[4][1])
874:                 )+
875:       a[0][1] * (
876:                 a[1][0] * (+a[2][2]*a[4][3] - a[2][3]*a[4][2]) +
877:                 a[1][2] * (-a[2][0]*a[4][3] + a[2][3]*a[4][0]) +
878:                 a[1][3] * (+a[2][0]*a[4][2] - a[2][2]*a[4][0])
879:                 )+
880:       a[0][2] * (
881:                 a[1][0] * (-a[2][1]*a[4][3] + a[2][3]*a[4][1]) +
882:                 a[1][1] * (+a[2][0]*a[4][3] - a[2][3]*a[4][0]) +
883:                 a[1][3] * (-a[2][0]*a[4][1] + a[2][1]*a[4][0])
884:                 )+
885:       a[0][3] * (
886:                 a[1][0] * (+a[2][1]*a[4][2] - a[2][2]*a[4][1]) +
887:                 a[1][1] * (-a[2][0]*a[4][2] + a[2][2]*a[4][0]) +
888:                 a[1][2] * (+a[2][0]*a[4][1] - a[2][1]*a[4][0])
889:                 ));
890:
891:     b[4][4] = -(
892:       a[0][0] * (
893:                 a[1][1] * (-a[2][2]*a[3][3] + a[2][3]*a[3][2]) +
894:                 a[1][2] * (+a[2][1]*a[3][3] - a[2][3]*a[3][1]) +
895:                 a[1][3] * (-a[2][1]*a[3][2] + a[2][2]*a[3][1])
896:                 )+
897:       a[0][1] * (
898:                 a[1][0] * (+a[2][2]*a[3][3] - a[2][3]*a[3][2]) +
899:                 a[1][2] * (-a[2][0]*a[3][3] + a[2][3]*a[3][0]) +
900:                 a[1][3] * (+a[2][0]*a[3][2] - a[2][2]*a[3][0])
901:                 )+
902:       a[0][2] * (
903:                 a[1][0] * (-a[2][1]*a[3][3] + a[2][3]*a[3][1]) +
904:                 a[1][1] * (+a[2][0]*a[3][3] - a[2][3]*a[3][0]) +
905:                 a[1][3] * (-a[2][0]*a[3][1] + a[2][1]*a[3][0])
906:                 )+
907:       a[0][3] * (
908:                 a[1][0] * (+a[2][1]*a[3][2] - a[2][2]*a[3][1]) +
909:                 a[1][1] * (-a[2][0]*a[3][2] + a[2][2]*a[3][0]) +
910:                 a[1][2] * (+a[2][0]*a[3][1] - a[2][1]*a[3][0])
911:                 ));
912:
913:    return det;
914:
915: }
916:
917: /*---------------------------------------------------------------
918:  *  inverse_4x4()
919:  * get the inverse of a square 4x4 matrix
920:  * returns determinant
921:  *--------------------------------------------------------------*/
922: double
923: inverse_4x4( double **a, double **b)
924: {
925:   double det;
926:
927:   det =
928:     - a[0][0] * a[1][1] * a[2][2] * a[3][3]
929:     + a[0][0] * a[1][1] * a[2][3] * a[3][2]
930:     + a[0][0] * a[2][1] * a[1][2] * a[3][3]
931:     - a[0][0] * a[2][1] * a[1][3] * a[3][2]
932:     - a[0][0] * a[3][1] * a[1][2] * a[2][3]
933:     + a[0][0] * a[3][1] * a[1][3] * a[2][2]
934:
935:     + a[1][0] * a[0][1] * a[2][2] * a[3][3]
936:     - a[1][0] * a[0][1] * a[2][3] * a[3][2]
937:     - a[1][0] * a[2][1] * a[0][2] * a[3][3]
938:     + a[1][0] * a[2][1] * a[0][3] * a[3][2]
939:     + a[1][0] * a[3][1] * a[0][2] * a[2][3]
940:     - a[1][0] * a[3][1] * a[0][3] * a[2][2]
941:
942:     - a[2][0] * a[0][1] * a[1][2] * a[3][3]
943:     + a[2][0] * a[0][1] * a[1][3] * a[3][2]
944:     + a[2][0] * a[1][1] * a[0][2] * a[3][3]
945:     - a[2][0] * a[1][1] * a[0][3] * a[3][2]
946:     - a[2][0] * a[3][1] * a[0][2] * a[1][3]
947:     + a[2][0] * a[3][1] * a[0][3] * a[1][2]
948:
949:     + a[3][0] * a[0][1] * a[1][2] * a[2][3]
950:     - a[3][0] * a[0][1] * a[1][3] * a[2][2]
951:     - a[3][0] * a[1][1] * a[0][2] * a[2][3]
952:     + a[3][0] * a[1][1] * a[0][3] * a[2][2]
953:     + a[3][0] * a[2][1] * a[0][2] * a[1][3]
954:     - a[3][0] * a[2][1] * a[0][3] * a[1][2] ;
955:
956:
957:   b[0][0] = - a[1][1] * (a[2][2]*a[3][3] - a[2][3] * a[3][2])
958:             + a[2][1] * (a[1][2]*a[3][3] - a[1][3] * a[3][2])
959:             - a[3][1] * (a[1][2]*a[2][3] - a[1][3] * a[2][2]);
960:
961:   b[0][1] = + a[0][1] * (a[2][2]*a[3][3] - a[2][3] * a[3][2])
962:             - a[2][1] * (a[0][2]*a[3][3] - a[0][3] * a[3][2])
963:             + a[3][1] * (a[0][2]*a[2][3] - a[0][3] * a[2][2]);
964:
965:   b[0][2] = - a[0][1] * (a[1][2]*a[3][3] - a[1][3] * a[3][2])
966:             + a[1][1] * (a[0][2]*a[3][3] - a[0][3] * a[3][2])
967:             - a[3][1] * (a[0][2]*a[1][3] - a[0][3] * a[1][2]);
968:
969:   b[0][3] = + a[0][1] * (a[1][2]*a[2][3] - a[1][3] * a[2][2])
970:             - a[1][1] * (a[0][2]*a[2][3] - a[0][3] * a[2][2])
971:             + a[2][1] * (a[0][2]*a[1][3] - a[0][3] * a[1][2]);
972:
973:
974:   b[1][0] = + a[1][0] * (a[2][2]*a[3][3] - a[2][3] * a[3][2])
975:             - a[2][0] * (a[1][2]*a[3][3] - a[1][3] * a[3][2])
976:             + a[3][0] * (a[1][2]*a[2][3] - a[1][3] * a[2][2]);
977:
978:   b[1][1] = - a[0][0] * (a[2][2]*a[3][3] - a[2][3] * a[3][2])
979:             + a[2][0] * (a[0][2]*a[3][3] - a[0][3] * a[3][2])
980:             - a[3][0] * (a[0][2]*a[2][3] - a[0][3] * a[2][2]);
981:
982:   b[1][2] = + a[0][0] * (a[1][2]*a[3][3] - a[1][3] * a[3][2])
983:             - a[1][0] * (a[0][2]*a[3][3] - a[0][3] * a[3][2])
984:             + a[3][0] * (a[0][2]*a[1][3] - a[0][3] * a[1][2]);
985:
986:   b[1][3] = - a[0][0] * (a[1][2]*a[2][3] - a[1][3] * a[2][2])
987:             + a[1][0] * (a[0][2]*a[2][3] - a[0][3] * a[2][2])
988:             - a[2][0] * (a[0][2]*a[1][3] - a[0][3] * a[1][2]);
989:
990:
991:   b[2][0] = - a[1][0] * (a[2][1]*a[3][3] - a[2][3] * a[3][1])
992:             + a[2][0] * (a[1][1]*a[3][3] - a[1][3] * a[3][1])
993:             - a[3][0] * (a[1][1]*a[2][3] - a[1][3] * a[2][1]);
994:
995:   b[2][1] = + a[0][0] * (a[2][1]*a[3][3] - a[2][3] * a[3][1])
996:             - a[2][0] * (a[0][1]*a[3][3] - a[0][3] * a[3][1])
997:             + a[3][0] * (a[0][1]*a[2][3] - a[0][3] * a[2][1]);
998:
999:   b[2][2] = - a[0][0] * (a[1][1]*a[3][3] - a[1][3] * a[3][1])
1000:             + a[1][0] * (a[0][1]*a[3][3] - a[0][3] * a[3][1])
1001:             - a[3][0] * (a[0][1]*a[1][3] - a[0][3] * a[1][1]);
1002:
1003:   b[2][3] = + a[0][0] * (a[1][1]*a[2][3] - a[1][3] * a[2][1])
1004:             - a[1][0] * (a[0][1]*a[2][3] - a[0][3] * a[2][1])
1005:             + a[2][0] * (a[0][1]*a[1][3] - a[0][3] * a[1][1]);
1006:
1007:
1008:   b[3][0] = + a[1][0] * (a[2][1]*a[3][2] - a[2][2] * a[3][1])
1009:             - a[2][0] * (a[1][1]*a[3][2] - a[1][2] * a[3][1])
1010:             + a[3][0] * (a[1][1]*a[2][2] - a[1][2] * a[2][1]);
1011:
1012:   b[3][1] = - a[0][0] * (a[2][1]*a[3][2] - a[2][2] * a[3][1])
1013:             + a[2][0] * (a[0][1]*a[3][2] - a[0][2] * a[3][1])
1014:             - a[3][0] * (a[0][1]*a[2][2] - a[0][2] * a[2][1]);
```

```
1015:
1016:   b[3][2] = + a[0][0] * (a[1][1]*a[3][2] - a[1][2] * a[3][1])
1017:             - a[1][0] * (a[0][1]*a[3][2] - a[0][2] * a[3][1])
1018:             + a[3][0] * (a[0][1]*a[1][2] - a[0][2] * a[1][1]);
1019:
1020:   b[3][3] = - a[0][0] * (a[1][1]*a[2][2] - a[1][2] * a[2][1])
1021:             + a[1][0] * (a[0][1]*a[2][2] - a[0][2] * a[2][1])
1022:             - a[2][0] * (a[0][1]*a[1][2] - a[0][2] * a[1][1]);
1023:
1024:   return det;
1025: }
1026:
1027: /*---------------------------------------------------------------
1028:  *  coFactor_3x3()
1029:  * Find the coFactor matrix of a square matrix
1030:  *---------------------------------------------------------------*/
1031: void
1032: coFactor_3x3( double **a, double **b)
1033: {
1034:   b[0][0] = +( a[1][1] * a[2][2] - a[2][1] * a[1][2]);
1035:   b[1][0] = -( a[1][0] * a[2][2] - a[1][2] * a[2][0]);
1036:   b[2][0] = +( a[1][0] * a[2][1] - a[1][1] * a[2][0]);
1037:   b[0][1] = -( a[0][1] * a[2][2] - a[0][2] * a[2][1]);
1038:   b[1][1] = +( a[0][0] * a[2][2] - a[2][0] * a[0][2]);
1039:   b[2][1] = -( a[0][0] * a[2][1] - a[0][1] * a[2][0]);
1040:   b[0][2] = +( a[0][1] * a[1][2] - a[0][2] * a[1][1]);
1041:   b[1][2] = -( a[0][0] * a[1][2] - a[0][2] * a[1][0]);
1042:   b[2][2] = +( a[0][0] * a[1][1] - a[0][1] * a[1][0]);
1043: }
1044:
1045: /*---------------------------------------------------------------
1046:  *  coFactor_2x2()
1047:  * Find the coFactor matrix of a square matrix
1048:  *---------------------------------------------------------------*/
1049: void
1050: coFactor_2x2( double **a, double **b)
1051: {
1052:   b[0][0] = +a[1][1];
1053:   b[1][0] = -a[0][1];
1054:   b[0][1] = -a[1][0];
1055:   b[1][1] = +a[0][0];
1056: }
1057:
1058: /*---------------------------------------------------------------
1059:  *  multiplication of squared matrices
1060:  * A = B * C
1061:  *---------------------------------------------------------------*/
1062: void
1063: multmatsq( int M, double **a, double **b, double **c)
1064: {
1065:   int i, j, n;
1066:   for (i = 0; i < M; i++)
1067:   {
1068:     for (j = 0; j < M; j++)
1069:     {
1070:       a[i][j] = 0;
1071:       for (n = 0; n < M; n++)
1072:       {
1073:         a[i][j] += b[i][n] * c[n][j];
1074:       }
1075:     }
1076:   }
1077: }
1078:
1079: /*---------------------------------------------------------------
1080:  *  multiplication of squared matrices
1081:  * A = B * C^T
1082:  *---------------------------------------------------------------*/
1083: void
1084: multmatsqT( int M, double **a, double **b, double **c)
1085: {
1086:   int i, j, n;
1087:   for (i = 0; i < M; i++)
1088:   {
1089:     for (j = 0; j < M; j++)
1090:     {
1091:       a[i][j] = 0;
1092:       for (n = 0; n < M; n++)
1093:       {
1094:         a[i][j] += b[i][n] * c[j][n];
1095:       }
1096:     }
1097:   }
1098: }
```

```
 0: /*****************************************************************
 1:  *
 2:  * File........:  matrix_utils.h
 3:  * Function....:  special functions (prototyping)
 4:  * Author......:  Tilo Strutz
 5:  * last changes: 20.10.2007, 29.3.2011
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *****************************************************************/
14:
15: #ifndef MATRIX_UTILS_H
16: #define MATRIX_UTILS_H
17:
18: int *ivector( long N);
19: unsigned int *uivector( long N);
20: float *fvector( long N);
21: double *vector( long N);
22: double **matrix( long M, long N);
23: float **fmatrix( long N, long M);
24:
25: void free_ivector( int *v[]);
26: void free_uivector( unsigned int *v[]);
27: void free_vector( double *v[]);
28: void free_matrix( double **m[]);
29: void free_fmatrix( float **m[]);
30:
31:
32: double determinant_2x2( double **a);
33: double determinant_3x3( double **a);
34: double inverse_4x4( double **a, double **b);
35: double inverse_5x5( double **a, double **b);
36: void coFactor_2x2( double **a, double **b);
37: void coFactor_3x3( double **a, double **b);
38:
39: void multmatsq( int M, double **a, double **b, double **c);
40: void multmatsqT( int N, double **a, double **b, double **c);
41:
42: #endif
```

S.6 Command-line parsing

```
 0: /*****************************************************************
 1:  *
 2:  * File........:  get_option.c
 3:  * Function....:  reading and analysing of
 4:  *                command-line parameters/options
 5:  * Author......:  Tilo Strutz
 6:  * last changes:  15.08.2006
 7:  *
 8:  * LICENCE DETAILS: see software manual
 9:  *  free academic use
10:  *  cite source as
11:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
12:  *  2nd edition 2015"
13:  *
14:  *****************************************************************/
15: #include <stdio.h>
16: #include <stdlib.h>
17: #include <string.h>
18: #include "get_option.h"
19:
20: /* contains argument of option, if OptArg != NULL */
21: char *OptArg = NULL;
22: char CheckStr[256];
23: /* CheckStr[] will be initialised with NEEDEDOPTIONS
24:  * all used optionen are deleted; if atleast one option remains,
25:  * an error message is output */
26: char *optstr;
27: int opt_num = 1;
28:
29: /*---------------------------------------------------------------
30:  *  check_opt()
31:  *---------------------------------------------------------------*/
32: int
33: check_opt( const char *name)
34: {
35:   char *ptr;
36:   int i, len, err = 0;
37:
38:   len = strlen( CheckStr);
39:   for (i = 0; i < len; i++)
40:   {
41:     if (( CheckStr[i] != ':') && (CheckStr[i] != ' '))
42:     {
43:       ptr = &CheckStr[i];
44:       ptr = (char*)strpbrk( ptr, ";:");
45:       ptr[0] = '\0';
46:       err = 1;
47:       break;
48:     }
49:   }
50:   if (err)
51:   {
52:     fprintf( stderr, "\n Missing Option for (-%s)!", &CheckStr[i]);
53:     usage( name);
54:   }
55:   return err;
```

```
 56: }
 57:
 58: /*---------------------------------------------------------------
 59:  *  get_option()
 60:  *
 61:  *  opt_num is number of option to be read
 62:  *  result: option string
 63:  *  required: global string containing all options
 64:  *  at first call opt_num must be equal to 1 !
 65:  *
 66:  *---------------------------------------------------------------*/
 67: char *
 68: get_option( int argc, const char *argv[])
 69: {
 70:   char optstring[256], *ptr, c, d, string[256];
 71:   char *gerrstr="#";
 72:   int len, i, num;
 73:
 74:   if (opt_num == 1)
 75:   {
 76:     strcpy( CheckStr, NEEDEDOPTIONS);
 77:   }
 78:   if (opt_num > (argc - 1))
 79:   {
 80:     return (NULL);
 81:   }
 82:   else if (argv[opt_num][0] == '+')
 83:   {
 84:     /* + signals end of parameter list */
 85:     opt_num++;
 86:     return (NULL);
 87:   }
 88:   else if (argv[opt_num][0] != '-')
 89:   {
 90:     fprintf( stderr, "\n Option-Error !!  ********** ");
 91:     fprintf( stderr,
 92:       "\n every Option must start with '-' (%s)!", argv[opt_num]);
 93:     /* usage( argv[0]); */
 94:     return gerrstr;
 95:   }
 96:
 97:   /***** copy without '-' *****/
 98:   num = opt_num;
 99:   strcpy( optstring, argv[num]);
100:   strcpy( string, &optstring[1]);
101:
102:   len = strlen( string);
103:   if (len == 0)
104:   {
105:     /*  single '-'   */
106:     fprintf( stderr, "\n Option-Error !!  ********** ");
107:     fprintf( stderr, "\n lonely dash !");
108:     /* usage( argv[0]); */
109:     return gerrstr;
110:   }
111:   ptr = OPTIONSTRING;
112:   do
113:   {
114:     /* search option string in OPTIONSTRING */
115:     ptr = (char*)strstr( ptr, string);
116:     if (ptr == NULL)
117:     {
118:       fprintf( stderr, "\n Option-Error !!  ********** ");
119:       fprintf( stderr, "\n Unknown Option (%s)!", optstring);
120:       /* usage( argv[0]); */
121:       return gerrstr;
122:     }
123:     c = ptr[len]; /* remember subsequent character */
124:     d = ptr[-1];  /* remember predecessor */
125:
126:     /* skip this entry by searching for next ':' or ';' */
127:     ptr = (char*)strpbrk( ptr, ";:.");
128:   } while (( (c != ':') && (c != ';') && (c != '.')) || (( d != ':')
129:     && (d != ';')&& (d != '.')));
130:
131:   if (c == ';') /* info, whether argument follows */
132:   {
133:     OptArg = NULL;
134:     opt_num++;
135:   }
136:   else
137:   {
138:     opt_num++;
139:     if (opt_num > (argc - 1))
140:     {
141:       fprintf( stderr, "\n Option-Error !!  ********** ");
142:       fprintf( stderr, "\n Missing Argument for (%s)!", optstring);
143:       /* usage( argv[0]); */
144:       return gerrstr;
145:     }
146:     else if (argv[opt_num][0] == '-'  && c == ':')
147:     {
148:       fprintf( stderr, "\n Option-Error !!  ********** ");
149:       fprintf( stderr, "\n Missing Argument for (%s)!", optstring);
150:       /* usage( argv[0]); */
151:       return gerrstr;
152:     }
153:     else
154:     {
155:         /* if c == '.' then negativ parameter are allowed */
156:     }
157:     OptArg = (char*)argv[opt_num];
158:     opt_num++;
159:   }
160:   strcpy( string, ":");
161:   strcat( string, &optstring[1]);
162:   strcat( string, ":");
163:   ptr = (char*)strstr( CheckStr, string);
164:   if (ptr != NULL)
165:     for (i = 0; i < len; i++)
166:       ptr[i + 1] = ' ';
167:
168:   return (( char*)argv[num]);
169: }
```

```
 0: /*************************************************************
 1:  *
 2:  * File........:  get_option.h
 3:  * Function....:  prototyping etc. for get_option.c
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  30.03.2006
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *************************************************************/
14:
15: #ifndef GETOPT_H
16: #define GETOPT_H
17:
18: /* defined in get_option.c */
19: extern char *OptArg;
20: extern char CheckStr[256];
21: /* CheckStr will be initialised with NEEDEDOPTIONS,
22:  * all used optionen are deleted;
23:  * if at least one option remains, an error message is output
24:  */
25: extern char *optstr;
26: /* can be used in main file to point to filenames */
27: extern int opt_num; /* is defined in get_option.c */
28:
29: /* defined in usage.c */
30: extern char *title;
31: extern char *OPTIONSTRING;
32: extern char *NEEDEDOPTIONS;
33:
34: /* Prototyping  */
35: void usage( const char *name);
36: int check_opt( const char *name);
37: char *get_option( int argc, const char **argv);
38:
39: #endif
```

```
 0: /*************************************************************
 1:  *
 2:  * File....: usage.c
 3:  * Function: parameters for Fitting
 4:  * Author..: Tilo Strutz
 5:  * Date... : 07.05.2008, 01.10.2009, 6.11.2009, 08.01.2010
 6:  *                18.02.2010, 10.03.2010
 7:  * changes:
 8:  * 28.01.2014 new option cw
 9:  *
10:  * LICENCE DETAILS: see software manual
11:  *  free academic use
12:  *  cite source as
13:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
14:  *  2nd edition 2015"
15:  *
16:  *************************************************************/
17: #include <stdio.h>
18: #include <stdlib.h>
19: #include <string.h>
20:
21: /* allowed options:  must start with ':'  !!  */
22: char *OPTIONSTRING =
23: {":a:a1.a2.a3.a4.a5.a6.a7.a8.a9.b:c;cc:co:cw:f;H;i:I:L;m:M:n;o:s;t:w:x:"};
24: char *NEEDEDOPTIONS = { ":i:o:m:" };  /* required options */
25:
26: char *title = { "Fitting WLS version 1.7c (12/2014)" };
27:
28: /*---------------------------------------------------------------
29:  *  usage()
30:  *---------------------------------------------------------------*/
31: void
32: usage( char *name)
33: {
```

```
34:     fprintf( stderr, "\n\n %s\n", title);
35:     fprintf( stderr, "\n\
36: Usage: %s [options]\n\n\
37:  Legal Options: \n\
38:     -i %%s  ... input data file (compulsory)\n\
39:     -o %%s  ... output file (compulsory)\n\
40:     -m  %%d ... model function (compulsory)\n\
41:        0 ... constant\n\
42:        1 ... f(x|a) = a1 + SUM_j=2^M a_j * x_(j-1)\n\
43:        2 ... f(x|a) = a1 + a2 * cos( x) + a3 * sin( x) [in degrees]\n\
44:        3 ... f(x|a) =      SUM_j=2^M a_j * x_(j-1)\n\
45:        5 ... f(x|a) = a1 + a2 * cos( x - a3) [in degrees]\n\
46:        6 ... f(x|a) = a1 + a2 * exp( a3 * x)\n\
47:        7 ... f(x|a) = log( a1 * x)\n\
48:        8 ... f(x|a) = a1 * exp( (x-a2)^2 * a3) + \n\
49:                       a4 * exp( (x-a5)^2 * a6)\n\
50:        9 ... f(x|a) = a1 * exp( a2 * x)\n\
51:       10 ... ln(f(x|a)) = ln(a1) + a2 * x\n\
52:       11 ... f(x|a) = a1 * exp( -|x|^a2 * a3)\n\
53:       12 ... f(x|a) = a1 + a2 * x + a3 * cos( x - a4) [in radians]\n\
54:       13 ... f(x|a) = a1 * exp( (x-a2)^2 * a3) + \n\
55:       16 ... f(x|a) = a1 + a2 * x + a3 * x^2\n\
56:       17 ... f(x|a) = a1 + a2 * x + a3 * x^2 + a4 * x^3\n\
57:       18 ... f(x|a) = sum_{j=1}^M aj * x^(j-1) (linear)\n\
58:       19 ... f(x|a) = sum_{j=1}^M aj * x^(j-1) (nonlinear)\n\
59:       20 ... f(x|a) = a1 + a2*x1 + a3*x1^2 + a4*x2 +a5*x2^2\n\
60:       21 ... f1(x|a) = a1 + cos(a3) * x1 - sin(a3) * x2 [in radians]\n\
61:              f2(x|a) = a2 + sin(a3) * x1 + cos(a3) * x2 [in radians]\n\
62:       22 ... f(x|a) = a1 + a2*cos(a3*x-a4)         [in radians]\n\
63:       23 ... f(x|a) = a1 + a2*cos(a3*x-a4) + a5*cos(2*a3*x-a6)\n\
64:       24 ... f(x|a) = 0 = (x1 - a1)^2 + (x2 - a2)^2 - a3*a3 (circle)\n\
65:       25 ... f(x|a) = x1^2 + x2^2 = a1*x1 + a2*x2 - a3\n\
66:                                (circle, linear)\n\
67:       26 ... f(x|a) = 0 = (sqrt[(x1 - a1)^2 + (x2 - a2)^2] - a3)^2\n\
68:                                       (circle, TLS)\n\
69:       30 ... neural network 3x3x1, feed forward\n\
70:       31 ... neural network 3x2x1\n\
71:       32 ... neural network 1x2x1\n\
72:       33 ... neural network 2x2x1\n\
73:       34 ... neural network 1x3x1\n\
74:       40 ... f(x|a) =(a1 + a2*x + a3*x*x + a4*x*x*x) /\n\
75:                      (1 + a5*x + a6*x*x + a7*x*x*x) NIST_THURBER\n\
76:       41 ... f(x|a) = a1 * (x**2 + a2*x) /\n\
77:                      (x*x + a3*x + a4) NIST_MGH09\n\
78:       42 ... f(x|a) = a1 / (1 + exp(a2 - a3*x))         NIST_Rat42\n\
79:       43 ... f(x|a) = a1 / [1 + exp(a2 - a3*x)]^(1/a4)  NIST_Rat43\n\
80:       44 ... f(x|a) = a1 / a2 * exp(-0.5*((x -a3)/ a2)^2) NIST_Eckerle4\n\
81:       45 ... f(x|a) = a1 * exp( a2 / (x+a3))  NIST_MGH10\n\
82:       46 ... f(x|a) = a1 * (x+a2)^(-1/a3)     NIST_Bennett5\n\
83:       47 ... f(x|a) = a1 *(1 - exp( -a2 * x)  NIST_BoxBOD\n\
84:     -a  %%d ... inversion algorithm (default: 1)\n\
85:           0 - cofactor method\n\
86:           1 - singular value decomposition\n\
87:           2 - LU decomposition\n\
88:     -a[j]  %%f ... provides initial value for  a_j (j=1,2,..,9)\n\
89:     -b  %%d ... observations per bin, for '-w 2' (default: 50)\n\
90:     -c      ... enable scaling of conditions \n\
91:     -cc %%s ... comma-separated list of column(s) containing \n\
92:                 conditions x (default: 1,2,...)\n\
93:     -co %%d ... column containing observations y (default: 2)\n\
94:     -cw %%d ... column containing weights\n\
95:     -f      ... forget weights after outlier removal\n\
96:     -H      ... enable true Hessian matrix\n\
97:     -I %%d   ... maximum number of iterations (default: 2000)\n\
98:     -M %%d   ... number of parameters (for '-m 1' only)\n\
99:     -n      ... force usage of numerical derivation\n\
100:    -L      ... use Gaus-Newton instead of Levenberg-Marquardt  \n\
101:    -s      ... disable special SVD function for solving linear model\n\
102:    -t %%f  ... target value for chisq (maximum error)\n\
103:                default: iteration until convergence\n\
104:    -w %%d   ... weighting  (default: 0)\n\
105:          0 ... no weighting \n\
106:          1 ... based on deviates \n\
107:          2 ... binning \n\
108:    -x %%d   ... outlier removal (default: 0)\n\
109:          0 ... no outlier removal\n\
110:          1 ... z-score + Chauvenet's criterion\n\
111:          2 ... cluster criterion (ClubOD)\n\
112:          3 ... M-score + Chauvenet's criterion \n\
113:          4 ... RANSAC \n\
114:     \n\
115:  ", name);
116: }
```

S.7 Error Handling

```
  0: /*************************************************************
  1:  *
  2:  * File........:  errmsg.c
  3:  * Function....:  error messages
  4:  * Author......:  Tilo Strutz
  5:  * last changes:  20.10.2007, 1.4.2011
  6:  *
  7:  * LICENCE DETAILS: see software manual
  8:  *  free academic use
  9:  *  cite source as
 10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 11:  *  2nd edition 2015"
 12:  *
 13:  *************************************************************/
 14: #include <stdio.h>
 15: #include <stdlib.h>
 16: #include <float.h> /* for DBL_MAX */
 17: #include "errmsg.h"
 18:
 19: int
 20: errmsg( int err, char *rtn, char *text, int value)
 21: {
 22:   switch (err)
 23:   {
 24:     case ERR_CALL:
 25:       fprintf( stderr, ERR_CALL_MSG, rtn, text);
 26:       break;
 27:     case ERR_OPEN_READ:
 28:       fprintf( stderr, ERR_OPEN_READ_MSG, rtn, text);
 29:       perror( "\nReason");
 30:       break;
 31:     case ERR_OPEN_WRITE:
 32:       fprintf( stderr, ERR_OPEN_WRITE_MSG, rtn, text);
 33:       perror( "\nReason");
 34:       break;
 35:     case ERR_ALLOCATE:
 36:       fprintf( stderr, ERR_ALLOCATE_MSG, rtn, text);
 37:       perror( "\nReason");
 38:       break;
 39:     case ERR_NOT_DEFINED:
 40:       fprintf( stderr, ERR_NOT_DEFINED_MSG, rtn, value, text);
 41:       break;
 42:     case ERR_IS_INFINITE:
 43:       fprintf( stderr, ERR_IS_INFINITE_MSG, rtn, text);
 44:       break;
 45:     case ERR_IS_ZERO:
 46:       fprintf( stderr, ERR_IS_ZERO_MSG, rtn, text);
 47:       break;
 48:     case ERR_IS_SINGULAR:
 49:       fprintf( stderr, ERR_IS_SINGULAR_MSG, rtn, text);
 50:       break;
 51:     default:
 52:       fprintf( stderr, "\nerrmsg: error %d is not defined\n", err);
 53:       break;
 54:   }
 55:   return err;
 56: }
 57:
 58: /*-------------------------------------------------------------
 59:  *  testing values()
 60:  * from http://www.johndcook.com/IEEE_exceptions_in_cpp.html
 61:  *-------------------------------------------------------------*/
 62: int IsNumber(double x)
 63: {
 64:     // This looks like it should always be true,
 65:     // but it's false if x is a NaN.
 66:     return (x == x);
 67: }
 68: int IsFiniteNumber(double x)
 69: {
 70:     return (x <= DBL_MAX && x >= -DBL_MAX);
 71: }
```

```
  0: /*************************************************************
  1:  *
  2:  * File........:  errmsg.h
  3:  * Function....:  error messages
  4:  * Author......:  Tilo Strutz
  5:  * last changes:  25.01.2010, 1.4.2011
  6:  *
  7:  * LICENCE DETAILS: see software manual
  8:  *  free academic use
  9:  *  cite source as
 10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 11:  *  2nd edition 2015"
 12:  *
 13:  *************************************************************/
 14: #ifndef ERRMSG_H
 15: #define ERRMSG_H
 16:
 17: #define ERR_CALL  1
 18: #define ERR_CALL_MSG \
 19:   "\n### %s: Wrong command-line parameters. \n  %s\n"
```

```
#define ERR_OPEN_READ 2
#define ERR_OPEN_READ_MSG \
  "\n### %s: Cannot open %s for reading\n"
#define ERR_OPEN_WRITE    3
#define ERR_OPEN_WRITE_MSG  \
  "\n### %s: Cannot open %s for writing\n"
#define ERR_ALLOCATE    4
#define ERR_ALLOCATE_MSG  \
  "\n### %s: Cannot allocate %s\n"
#define ERR_NOT_DEFINED 5
#define ERR_NOT_DEFINED_MSG \
  "\n### %s: Value %d for %s is not defined\n"
#define ERR_IS_ZERO 6
#define ERR_IS_ZERO_MSG \
  "\n### %s: Value for %s is zero\n"
#define ERR_IS_SINGULAR 7
#define ERR_IS_SINGULAR_MSG \
  "\n### %s: Matrix %s is singular\n"
#define ERR_IS_INFINITE 8
#define ERR_IS_INFINITE_MSG \
  "\n### %s: Variable %s is infinite\n"

int errmsg( int err, char *rtn, char *text, int value);

#endif
```

S.8 Other

```
/*************************************************************
 *
 * File........:  heap_sort.c
 * Function....:  sorting of values
 * Author......:  Tilo Strutz
 * last changes:  20.10.2007
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
 *************************************************************/

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

/*-----------------------------------------------------------
 *  heap_sort_d()
 * sorting of values in values[0...N-1] in ascending order
 * values[] is replaced on output by sorted values
 *-----------------------------------------------------------*/
void
heap_sort_d( unsigned long N, double values[])
{
  unsigned long i, ir, j, l;
  double rvalues;

  if (N < 2)
    return;

  l = (N >> 1);
  ir = N - 1;

  for (;;)
  {
    if (l > 0)
    {
      l--;
      rvalues = values[l];
    }
    else
    {
      rvalues = values[ir];
      values[ir] = values[0];
      ir--;
      if (ir == 0)
      {
        values[0] = rvalues;
        break;
      }
    }
    i = l;

    j = l + l + 2;
    while (j <= ir + 1)
    {
      if (j < ir + 1 && values[j - 1] < values[j])
      {
        j++;
      }
      if (rvalues < values[j - 1])
      {
        values[i] = values[j - 1];
        i = j - 1;
        j <<= 1;
      }
      else
      {
        j = ir + 2; /* terminate while-loop */
      }
    }
    values[i] = rvalues;
  }
}

/*-----------------------------------------------------------
 *  heap_sort_d_()
 * sorting of values in values[0...N-1] in ascending order
 * values[] is replaced on output by sorted values
 *-----------------------------------------------------------*/
void
heap_sort_d_( unsigned long N, double values[], long idx[])
{
  unsigned long i, ir, j, l;
  double rvalues;
  int iidx;

  if (N < 2)
    return;

  for ( i = 0; i < N; i++) idx[i] = i;

  l = (N >> 1);
  ir = N - 1;

  for (;;)
  {
    if (l > 0)
    {
      l--;
      rvalues = values[l]; iidx = idx[l];
    }
    else
    {
      rvalues = values[ir];     iidx = idx[ir];
      values[ir] = values[0];   idx[ir] = idx[0];
      ir--;
      if (ir == 0)
      {
        values[0] = rvalues;    idx[0] = iidx;
        break;
      }
    }
    i = l;

    j = l + l + 2;
    while (j <= ir + 1)
    {
      if (j < ir + 1 && values[j - 1] < values[j])
      {
        j++;
      }
      if (rvalues < values[j - 1])
      {
        values[i] = values[j - 1];  idx[i] = idx[j - 1];
        i = j - 1;
        j <<= 1;
      }
      else
      {
        j = ir + 1 + 1; /* terminate while-loop */
      }
    }
    values[i] = rvalues;    idx[i] = iidx;
  }
}
```

```
/*************************************************************
 * File........:  erf.c
 * Function....:  error functions
 * Author......:  Tilo Strutz
 * last changes:  05.02.2008
 *
 * LICENCE DETAILS: see software manual
 *  free academic use
 *  cite source as
 * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
 *  2nd edition 2015"
 *
 *************************************************************/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "erf.h"

```

```
18: static const double rel_error= 1E-12;
19: /* calculate 12 significant figures
20:  * you can adjust rel_error to trade off between accuracy and
21:  * speed, but don't ask for > 15 figures
22:  *(assuming usual 52 bit mantissa in a double)
23:  */
24:
25: /*--------------------------------------------------------------
26:  *  erf()
27:  *
28:  *  erf(x) = 2/sqrt(pi)*integral(exp(-t^2),t,0,x)
29:  *         = 2/sqrt(pi)*[x - x^3/3 + x^5/5*2! - x^7/7*3! + ...]
30:  *         = 1-erfc(x)
31:  *
32:  *------------------------------------------------------------*/
33: double erf(double x)
34: {
35:   /* 2/sqrt(pi)*/
36:   static const double two_sqrtpi = 1.128379167095512574;
37:   double sum, term, xsqr;
38:   int j= 1;
39:
40:     sum = x;
41:     term = x;
42:     xsqr = x * x;
43:
44:     if (fabs(x) > 2.2)
45:     {
46:       /*use continued fraction when fabs(x) > 2.2 */
47:       return 1.0 - erfc(x);
48:     }
49:     do
50:     {
51:       term *= xsqr / j;
52:       sum -= term /(2*j+1);
53:       j++;
54:       term *= xsqr / j;
55:       sum += term /(2*j+1);
56:       j++;
57:     } while (fabs(term) / sum > rel_error);
58:     return two_sqrtpi * sum;
59: }
60:
61: /*--------------------------------------------------------------
62:  *  erfc()
63:  *
64:  *  erfc(x) = 2/sqrt(pi)*integral(exp(-t^2),t,x,inf)
65:  *          = exp(-x^2)/sqrt(pi) * [1/x + (1/2)/x + (2/2)/x +
66:  *                                  (3/2)/x+ (4/2)/x+ ...]
67:  *          = 1-erf(x)
68:  *  expression inside [] is a continued fraction
69:  *  so '+' means add to denominator only
70:  *------------------------------------------------------------*/
71: double erfc(double x)
72: {
73:   /* 1/sqrt(pi)*/
74:   static const double one_sqrtpi = 0.564189583547756287;
75:   double a = 1, b;        /* last two convergent numerators */
76:   double c, d;  /* last two convergent denominators */
77:   double q1, q2;     /* last two convergents (a/c and b/d)  */
78:   double n = 1.0, t;
79:
80:     b = c = x;
81:     d = x * x + 0.5;
82:     q2 = b / d;
83:
84:     if (fabs(x) < 2.2)
85:     {
86:       return 1.0 - erf(x);   /* use series when fabs(x) < 2.2 */
87:     }
88:     if (x > 0)
89:     {               /* continued fraction only valid for x>0 */
90:       return 2.0 - erfc(-x);
91:     }
92:     do
93:     {
94:       t = a*n + b*x;
95:       a = b;
96:       b = t;
97:       t = c*n + d*x;
98:       c = d;
99:       d = t;
100:      n += 0.5;
101:      q1 = q2;
102:      q2 = b / d;
103:    } while (fabs(q1-q2) / q2 > rel_error);
104:    return one_sqrtpi * exp(-x*x) * q2;
105: }
106:
107: /*--------------------------------------------------------------
108:  *  erfinv()
109:  *
110:  *------------------------------------------------------------*/
111: int erfinv( double y, double *res )
112: {
113:   static double a[] = {0,  0.886226899, -1.645349621,
114:                           0.914624893, -0.140543331 };
115:   static double b[] = {0, -2.118377725,  1.442710462,
116:                          -0.329097515,  0.012229801 };
117:   static double c[] = {0, -1.970840454, -1.624906493,
118:                           3.429567803,  1.641345311 };
119:   static double d[] = {0,  3.543889200,  1.637067800 };
120:   int err = 0;
121:   double x, z;
122:
123:   if ( y > -1. )
124:   {
125:     if ( y >= -.7 )
126:     {
127:       if ( y <= .7 )
128:       {
129:         z = y*y;
130:         x = y * (((a[4]*z+a[3])*z+a[2])*z+a[1]) /
131:                (((b[4]*z+b[3])*z+b[2])*z+b[1])*z+1);
132:       }
133:       else if ( y < 1 )
134:       {
135:         z = sqrt(-log((1-y)/2));
136:         x = (((c[4]*z+c[3])*z+c[2])*z+c[1]) / ((d[2]*z+d[1])*z+1);
137:       }
138:       else
139:       {
140:         err = 3;
141:         x = 0;
142:       }
143:     }
144:     else
145:     {
146:       z = sqrt(-log((1+y)/2));
147:       x = -(((c[4]*z+c[3])*z+c[2])*z+c[1]) / ((d[2]*z+d[1])*z+1);
148:     }
149:   }
150:   else
151:   {
152:     err = 3;
153:     x = 0;
154:   }
155:   *res = x;
156:   return err;
157: }
```

```
 0: /***************************
 1:  * File........:  erf.h
 2:  * Function....:  prototypes for erf.c
 3:  * Author......:  Tilo Strutz
 4:  * last changes:  05.02.2008
 5:  *
 6:  * LICENCE DETAILS: see software manual
 7:  *  free academic use
 8:  *  cite source as
 9:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
10:  *  2nd edition 2015"
11:  *
12:  ***************************/
13: double erf(double x);
14: double erfc(double x);
15: int erfinv( double y, double *res );
```

```
 0: /*************************************************************
 1:  *
 2:  * File........:  macros.h
 3:  * Function....:  general purpose macros
 4:  * Author......:  Tilo Strutz
 5:  * last changes:  25.01.2010
 6:  *
 7:  * LICENCE DETAILS: see software manual
 8:  *  free academic use
 9:  *  cite source as
10:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
11:  *  2nd edition 2015"
12:  *
13:  *************************************************************/
14:
15: #ifndef MACROS_H
16: #define MACROS_H
17:
18: #define MAX(a,b) ((a) > (b) ?   (a) : (b))
19: #define MIN(a,b) ((a) < (b) ?   (a) : (b))
20:
21: #define M_PI        3.14159265358979323846       /* pi */
22: #define TOL 1.0e-10   /* for convergence criterion */
23: #define TOL_S2 1.0e-14    /* for test of singular values */
24: #define TOL_S 1.0e-12   /* for test of singular values */
25:   /* this small value is required, for example,
26:       for the Bennett5 data set */
27:
28: #endif
```

```
 0: /*********************************************************
```

```
 1:  * File........:  defines.h
 2:  * Function....:  definition of globally used values
 3:  * Author......:  Tilo Strutz
 4:  * last changes:  03.07.2009
 5:  *
 6:  * LICENCE DETAILS: see software manual
 7:  *  free academic use
 8:  *  cite source as
 9:  * "Strutz, T.: Data Fitting and Uncertainty. Springer Fachmedien,
10:  *  2nd edition 2015"
11:  *
12:  ************************************************************/
13:
14: #ifndef DEFINES_H
15: #define DEFINES_H
16:
17: /* for est_weights.c and outlier_detetction.c */
18:
19: #define MAX_LINES_W 500
20:
21:
22: /* for fitting.c  */
23:
24: /* maximal number of conditions per model function */
25: /* see functions.h  */
26: #define MAX_CONDITIONS (M_MAX-1)
27: /* output of maximal 100 lines of resulting values as feedback */
28: #define MAX_LINES 100
29: /* input file may contain maximal 512 characters per line */
30: #define MAXLINELENGTH 512
31:
32:
33: /* for functions.c  */
34:
35: /* maximal 15 parameters per model function (see f_deriv()) */
36: #define M_MAX 20
37:
38: /* for ls.c */
39: extern long ITERAT_MAX;
40:
41: #endif
```