GAUSSFIT—A SYSTEM FOR LEAST SQUARES AND ROBUST ESTIMATION

W. H. JEFFERYS, M. J. FITZPATRICK, and B. E. MCARTHUR

Department of Astronomy, The University of Texas at Austin, Austin TX 78712, U.S.A.

Abstract. GaussFit is a new computer program for solving least squares and robust estimation problems. It has a number of unique features, including a complete programming language designed especially to formulate estimation problems, a built-in compiler and interpreter to support the programming language, and a built-in algebraic manipulator for calculating the required partial derivatives analytically. These features make GaussFit very easy to use, so that even complex problems can be set up and solved with minimal effort. GaussFit can correctly handle many cases of practical interest: nonlinear models, exact constraints, correlated observations, and models where the equations of condition contain more than one observed quantity. An experimental robust estimation capability is built into GaussFit so that data sets contaminated by outliers can be handled simply and efficiently.

1. Introduction

Early in the Space Telescope program, it became evident that astrometry needed a very flexible least squares estimation program. It should be able to handle complex models, including general overlapping-plate models such as are frequently encountered in astrometry. The program should be very flexible—it should be easy to define new models or to change old ones, and to test the results of applying different models to the same data set. And finally, the program should incorporate the best available algorithms.

An early version of such a program, called REDUCE, was described by Jefferys and Feo (1986). However, REDUCE was not flexible enough and moreover because of its structure it could not be fully integrated into the Science Data Analysis Software (SDAS) system at the Space Telescope Science Institute. Therefore, the fundamental structure of REDUCE was changed, and a new, much more flexible program, called GaussFit, is the result. Whereas REDUCE required the user to build estimation models with an existing FORTRAN compiler, GaussFit contains its own computer language, which is especially designed to make it easy to specify complex reduction models. The GaussFit programming language provides an easy and natural way to formulate problems in nonlinear estimation, problems where an equation of condition can contain more than one observation, problems with correlated observations, problems involving exact constraints among the parameters, and problems in which the model can only be expressed algorithmically and not as a closed form expression. GaussFit uses orthogonal transformations

(Householder transformations) instead of normal equations to solve the basic least squares problem, and it also allows the user to specify a robust estimation method that is resistant to outliers in the data.

GaussFit is written in C, and is currently running on VAX/VMS, Berkeley UNIX v. 4.3, and Macintosh computers. A user's manual is available for distribution. Further information about GaussFit's availability can be obtained by contacting the first author.

2. The Structure of GaussFit

The basic structure of GaussFit is shown in Figure 1. The shaded area represents the GaussFit program itself, and the boxes outside the shaded area are supplied by the user. The user first provides a model, written in the GaussFit programming language (which is rather similar to C). GaussFit takes this program and converts it, via a built-in compiler, into an intermediate code which is actually the assembly language of an abstract computing machine. This intermediate code is similar in spirit to the P-code used in the well-known UCSD Pascal system.

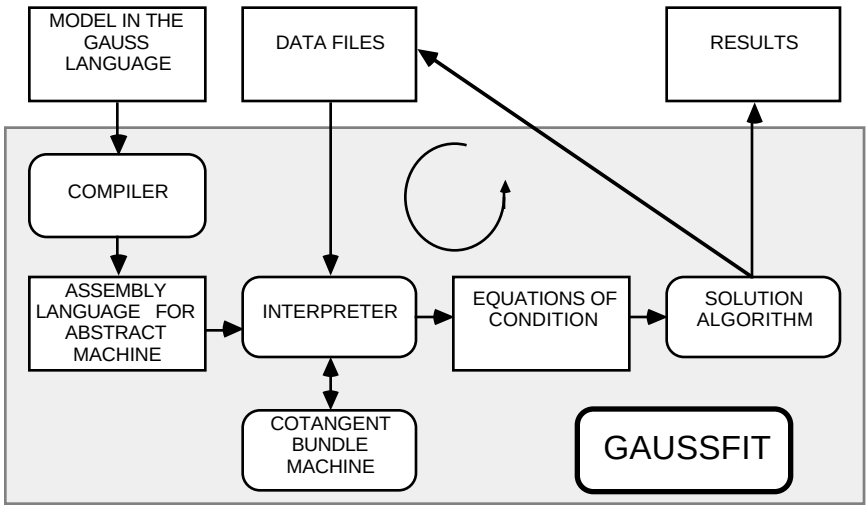

Fig. 1. The structure of GaussFit.

The intermediate code is then interpreted by an interpreter that emulates the abstract machine. Equations of condition are formed one-by-one for each line of data in the user's data files, and are then sent to the user-specified solution algorithm. In forming the equations of condition, partial derivatives with respect to the parameters and the data variables of the problem are required. To compute these, the abstract computer relies on an algebraic manipulator called the *cotangent bundle machine* that is also built into the program. The algebraic manipulator

actually performs all of the calculations required by the program, not on the set of real numbers, but instead on complex structures consisting of both values and derivatives. Mathematically, these objects are elements of the cotangent bundle of a manifold associated with the estimation problem. However, no knowledge of cotangent bundles is required to use GaussFit, as these calculations are invisible to the user.

When all the equations of condition and constraint have been calculated and sent to the solution algorithm, the latter performs the appropriate matrix calculations to obtain the corrections to the parameters and residuals. The corrected values are then entered into the user's data files to update the solution. If the solution has not converged, the process is iterated until convergence has been obtained or the number of iterations exceeds a user-defined maximum. Other results are printed to a results file, and then program execution is terminated.

Interpretation has both advantages and penalties. The advantages are that the derivative calculations are easily built in, and features can be included in the program to make it easier for the user to debug models. The price paid is that interpretation slows down the execution of the user's model by the usual factor of 10-30 that is common to interpreters. However, since the code is precompiled into a tokenized intermediate code, the interpreter can be relatively efficient, and since most of the program's time is spent in solving the equations of condition and not in forming them, the actual penalty is not so great as it might seem at first glance.

3. Models in GaussFit

The GaussFit programming language is a complete language, like FORTRAN, Pascal and C. Its structure is similar to that of the C programming language (Kernighan and Ritchie 1978). It possesses the usual complement of conditional and looping structures found in traditional computer languages, as well as a modern nested-statement structure. Both vector and scalar valued variables can be defined and used, and the user can define new functions and procedures. Thus there is no theoretical limit to the complexity of model that GaussFit can express.

Most least squares packages require the user to solve for the dependent variable (the observation) in terms of the independent variables (the parameters). However, when an equation of condition can contain more than one observation, this is no longer possible. Therefore, a method described by Deming (1938) has been used in which the equations of condition are expressed implicitly. For example, consider a fit of an exponentially decaying quantity y as a function of x. The quantity x is assumed error-free, whereas y has error. The quantities a and b are parameters to be solved for. Explicitly one would solve for y, writing

$$y = ae^{-bx}, \tag{1}$$

and this form would be required by most least squares packages. However, if both x and y were subject to error, it would no longer be possible to solve for the

observations in terms of the parameters *a* and *b*. So GaussFit allows the equation of condition to be expressed *implicitly*. For example, one may write

$$y - ae^{-bx} = 0, \tag{2}$$

or equivalently, one may write

$$\log y - \log a + bx = 0. \tag{3}$$

These two forms will give identical results when used on the same data in GaussFit.

Equation (3) would be programmed in the GaussFit programming language as follows:

```
parameter a,b;
data x;              /* x is error-free */
observation y;       /* y has error.    */

main() {
   while(import())
      export(log(y) - log(a) + b*x);
}
```

In this example, the variables `a` and `b` are unknown parameters to be determined by the fit; `x` is assumed known perfectly (without error), and the `observation` `y` has error. The program is a very short one. The function **`import`** reads the data files line by line, returning the value **`true`** as long as data are read successfully. Each time a data line has been successfully read in, the equation of condition is calculated, and then the function **`export`** sends its argument to the estimation routines. Notice that when the argument of **`export`** is evaluated, the cotangent bundle machine also calculates the partial derivatives of the expression with respect to `a`, `b`, and `y`. The differentiation is performed *analytically*, not numerically. Thus the full precision of the machine word is preserved (all calculations are done in double precision).

In this example, we made the usual assumption that all of the error is in the variable *y*. However, it is very simple to modify the model to handle the case where both *x* and *y* have errors. All that has to be done is to replace the **`data`** statement in the above program with the statement

```
observation x;       /* x has error.    */
```

When this is done, the program will automatically set up the equations of condition according to the correct prescription (Britt and Luecke 1973; Jefferys 1980, 1981).

To give another example, in astrometry we may desire to solve for both the *x* and *y* variables simultaneously. This means that each line of data will generate two independent equations of condition. A program to solve a four plate parameter model would take the form

```
parameter a,b,c,d;
observation x,y,xi,eta;

main() {
   while(import()) {
      export(xi  – ( a*x + b*y + c));
      export(eta – (–b*x + a*y + d));
   }
}
```

In this example, the variables `x` and `y` are the measured positions of a star on the plate, whereas the variables `xi` and `eta` are the catalog positions of the reference stars. Since both plate measurements and catalog positions have errors, they must all be declared in an **`observation`** statement. Not to do so could introduce an unwanted bias into the results. As Lybanon (1984) has documented, the correct formulation of the equations of condition to avoid bias, when more than one quantity is in error, is not widely known.

GaussFit allows the parameters to be subscripted. This is useful, for example, when the model has some parameters associated with particular groups of data. For example, in an overlapping-plate astrometric solution each plate has its own plate constants, and each star has its own star constants. In addition, there could be parameters that are common to all plates and stars, for example, those describing the distortion of the telescope.

An interesting example of a model involving subscripted parameters is the one used by Barnes, Moffett, Jefferys and Hawley (1987) to reduce photometric data on Cepheid variable stars. The model involved 131 parameters (65 stars $\times$ 2 parameters per star + 1 global parameter), and approximately 3500 equations of condition were generated. The model in the GaussFit programming language was only 5 lines long:

```
parameter b, alfa[star], rz[star];
observation v, r, vmr;

main() {
   while(import())
      export(alfa + b*vmr + 5*log10(rz + r) – v);
}
```

In this example, the subscript `star` is used to index the 70 different values of the parameters `alfa` and `rz`, while the single global parameter `b`, which is what we are really interested in, is common to all stars.

Another capability of GaussFit is to enforce exact constraints on a solution. For example, in an overlapping-plate model with no external reference star positions, the plate parameters of one of the plates would normally be constrained to have particular values, making it the "reference" plate. Another example would be if we wanted to "force-fit" a curve so that it passes through a particular point. For example, one way to force the exponentially decaying curve of the first example through the point x=0, y=1 is by constraining a–1=0. In GaussFit this could be accomplished by executing the **`exportconstraint`** statement. Thus, the statement

```
exportconstraint(a - 1);
```

would accomplish the force-fitting just described.

4. File Formats

GaussFit assumes that the data are provided to it in a tabular format. At the present time, this is implemented with a simple text file data interchange standard that is also used by a number of popular microcomputer programs (spreadsheets, database managers and the like). The files are ASCII files arranged in rows and columns. Each row is terminated by an ASCII carriage return character, and the columns in each row are separated by white space. The first row contains the names of the columns. A portion of a data file for the Cepheid variable problem is shown in Figure 2. Data files also contain information on the variance-covariance matrix of the observations (if relevant). This information is provided in additional columns of the table. For example, the last column of Figure 2 contains the covariance between `v` and `vmr`.

star	phase	vmr	r	v	v___vmr_
2	32	0.65	-23.56	5.54	0.0001
2	33	0.62	-26.22	5.49	0.0001
2	34	0.64	-23.03	5.42	0.0001
4	1	0.33	-1.04	3.24	0.0002
4	**2**	**0.34**	**-0.95**	**3.26**	**0.0002**
4	3	0.39	0.04	3.37	0.0002
4	4	0.39	0.15	3.36	0.0002

Fig. 2. Sample data file. The observation line that is being processed is shown in boldface.

The program will read each data file row by row, and will associate variables that have been declared of type **`data`** or **`observation`** with the columns headed by the corresponding name. As the statements in the program are executed, the required values of the data are fetched from the current line of the data file and used to form the equation of condition.

Parameters are found in a separate *parameter file*, which has the same general format as the data files. Each parameter is found in the column which is headed by the name of the corresponding variable. Columns for unsubscripted parameters contain only a single number, found in the first available row of the column. Figure 3 shows a sample data file for the Cepheid variable problem discussed earlier. The global variable `b` that appears in each equation of condition has the value 3.349. In the case of subscripted variables, several values are listed in each column, one for each possible value of the subscript. Figure 3 demonstrates how the values for a subscripted parameter are looked up. In the figure, it is assumed that the current star is star 4 (see boldfaced line in Figure 2). By looking in the column headed `star` and finding the value 4, and then reading across the table to the `alfa` column, the program finds the value `alfa[4]=10.49` to be used in forming the equation of condition.

star	alfa	rz	b
3	12.95	108.22	***3.349***
5	8.99	69.10	
4	***10.49***	**43.50**	
2	14.48	203.66	

Fig. 3. Sample parameter file. The value of `b` is 3.349, and the value of `alfa[4]` is 10.49.

The initial values of the parameters must be supplied by the user. Unfortunately, the range of models (linear, nonlinear, simple, complex) allowed by GaussFit is so wide that no general procedure for estimating starting values for the iteration can be given. This task, therefore, is left to the experience of the user who is, of course, most familiar with the data and the problem.

The table format described here is very simple, yet quite flexible. Because data in this format are interchangeable with many microcomputer programs, it is easy to use these programs to prepare input data and also to obtain plotted output (for example, display of the residuals). Indeed, we find that modern integrated spreadsheets such as Microsoft EXCEL® are so flexible and powerful that it is actually easier to use them to examine the results from GaussFit than it would be to program a special application to accomplish this task.

5. Derivatives and the Cotangent Bundle Machine

As was mentioned earlier, the objects manipulated by GaussFit (e.g., the quantities declared in **`data`**, **`observation`**, and **`parameter`** statements) are actually complex structures containing the value of an expression *plus* all of the relevant partial derivatives. All arithmetic in GaussFit is accomplished by a special section of code (the "cotangent bundle machine") that can handle general objects of this type. It is important to stress that the derivatives are calculated using the analytic formulas, and not by numerical differentiation.

In the notation of differential geometry, derivative information is carried explicitly by the differential of a function. The objects we manipulate are therefore elements of the cotangent bundle of a manifold. For example, if a function $f(a,b,...,c)$ is expanded in Taylor's series about the point $(a_0,b_0,...,c_0)$, we can write

$$f = f(a_0,b_0,...,c_0) + \mathrm{d}f = f_0 + \mathrm{d}f, \tag{4}$$

where the differential $\mathrm{d}f$ carries all of the derivative information:

$$\mathrm{d}f = \frac{\partial f}{\partial a}\,\mathrm{d}a + \frac{\partial f}{\partial b}\,\mathrm{d}b + ... + \frac{\partial f}{\partial c}\,\mathrm{d}c\ . \tag{5}$$

The objects manipulated by GaussFit are of the form specified in Eqs. (4-5). The algebra of such objects is well known. Thus to multiply f and g, we use the identity

$$h = fg = (f_0 + \mathrm{d}f)(g_0 + \mathrm{d}g) = f_0g_0 + (f_0\mathrm{d}g + g_0\mathrm{d}f) = h_0 + \mathrm{d}h. \tag{6}$$

Similarly, a function of such an object can be calculated. For example, we can take the logarithm of f provided that $f_0 \neq 0$:

$$\log f = \log f_0 + \frac{\mathrm{d}f}{f_0}\ . \tag{7}$$

The beauty of this scheme is that the user need no longer worry about derivatives. When any quantity is calculated, the derivatives are automatically "carried along for the ride". This shows itself to great advantage when the model is too complicated to be written down as a simple equation. An example is a model we wrote to fit double star data. In this example, each observation has to be fitted to the projected Keplerian ellipse. To calculate the position in the orbit, Kepler's equation has to be solved, and it is well known that this does not have a closed-form solution in elementary functions. Therefore, an iterative solution is required. Thus (for $e<0.1$, say) we might write the following function in the GaussFit programming language:

```
kepler(e,M)  {
   variable E,n;

   E = M;
   for(n=0;n<5;n=n+1)  /* Loop 5 times */
      E = M + e*sin(E);
   return E;
}
```

This function will be called from elsewhere in our double star model to compute the solution of Kepler's equation when it is needed. Notice that the eccentricity is one of the parameters we are solving for, so that the object sent to `kepler` is actually of the form $e = e_0 + de$. Similarly, the mean anomaly M sent to `kepler` will depend on the period and the time of perihelion passage in the orbit, and will include derivatives with respect to these parameters. Thus the objects being sent to `kepler` are actually elements of the cotangent bundle, and the eccentric anomaly E returned by it will also be such an object and will include partial derivatives with respect to the eccentricity, the period, and the time of perihelion passage. Yet to the user it appears no more complicated than programming a straightforward numerical solution of Kepler's equation.

6. A Word on Robust Estimation

The standard reduction method in GaussFit is the generalized least squares algorithm of Jefferys (1980, 1981), which allows correlated observations, nonlinear equations, constraints, and more than one observation per equation of condition. For numerical stability, the solution is obtained by Householder transformations instead of normal equations, using a new algorithm (to be described elsewhere) especially designed for efficiency in overlapping-plate problems.

An experimental robust estimation facility has also been implemented. This algorithm generalizes the textbook methods of Huber (1981) to implicitly specified nonlinear equations, exact constraints, and equations of condition containing more than one observation, as well as partially to the case of correlated observations. This algorithm is also a new one that will be described fully elsewhere. Two solution methods are provided: the method of iteratively reweighted least squares (IRLS) and Newton's method. Both use Householder transformations to solve the relevant matrix equations.

Robust estimation is useful when the data are likely to be corrupted by "outliers". In such cases, the method of least squares may be unreliable. Although least squares is known to be the most efficient estimator *if the data are normally distributed,* it rapidly becomes inefficient if outliers contaminate the data. In such cases, robust estimators, based on noneuclidean metrics, retain their efficiency while the least squares estimator quickly deteriorates.

Many robust metrics have been proposed, and for simplicity we have chosen to implement a particular one that is a good overall compromise, called "fair" (Rey 1983). We minimize the sum over all residuals u of

$$\rho(u) = c^2\left[\frac{|u|}{c} - \log\left(1 + \frac{|u|}{c}\right)\right] \tag{8}$$

where c is a parameter that is adjusted to a value that separates "small" residuals from "large" ones. The function $\rho(u)$ behaves like the least squares metric when u is small, whereas for large u it grows only linearly with u so that the effect of a large residual on the solution is small. A very large residual will hardly affect the solution at all. The metric "fair" has many excellent properties, including insensitivity of the results to the exact value of c. See Holland and Welsch (1977) for a comparison of several robust metrics, including "fair".

Actually, c is an adjustable parameter, and varying it produces a family of metrics of differing *asymptotic relative efficiency* (ARE). This is a measure of how efficient the particular estimator is, when it is compared to the least squares estimator on identical normally distributed data. The ARE is less than or at most equal to 1, and we get good results with an ARE of 0.9 to 0.95. In GaussFit, the ARE is a user-specified input parameter, and the value of c is calculated by the program from the ARE and the actual distribution of the residuals.

In addition to this method, we have included a robust method based on minimizing the sum of the absolute value of the residuals. This method is less efficient than "fair" (its ARE is approximately 0.7) but it is very robust. It has been implemented using the algorithm of Barrodale and Roberts (1978), which is based on an improvement of the Simplex algorithm of linear programming. The primary purpose of including this method is to give the user a simple way to examine the residuals so that outliers can be identified and removed prior to a standard least squares adjustment. It is most useful in the linear case, although it appears to work for some nonlinear models as well.

7. Conclusions

GaussFit is a new and powerful way to solve problems in least squares and robust estimation. By including a computer language especially designed to formulate estimation models efficiently, GaussFit relieves the user of many of the niggling details that formerly required attention, so that he can concentrate on the broader and ultimately more interesting tasks of finding the best model for the problem and obtaining useful scientific results. Also, since the algorithms implemented in GaussFit are powerful and general, the program has a wide range of application and should be able to handle the great majority of problems that are encountered in day-to-day practice.

Acknowledgements

The authors thank all those who have provided ideas and criticism during the development of this program. We especially thank our colleagues on the Hubble Space Telescope Astrometry Team: G. F. Benedict, R. L. Duncombe, O. G. Franz, L. W. Fredrick, P. D. Hemenway, P. J. Shelus, and W. F. van Altena. We also thank Professor David Hinkley for his assistance with robust estimation techniques. Many of the ideas incorporated in this program developed out of research conducted with John Feo early in the Space Telescope project, and GaussFit could not have been written without this experience. This software was developed under NASA Contract NAS8–32906, the support of which is gratefully acknowledged.

References

Barrodale, I. and Roberts, F. D. K.: 1978, *SIAM J. Numer. Anal.* **15**, 603.

Barnes, T. G., Moffett, T. J., Jefferys, W. H. and Hawley, S. L.: 1987, "The Surface-Brightness Relation and the Slope of the P-L Relation," Paper presented at the Vancouver Meeting of the American Astronomical Society.

Britt, H. I. and Luecke, R. H.: 1973, *Technometrics* **15**, 233

Deming, W. E.: 1938, *Statistical Adjustment of Data*, John Wiley and Sons, New York .

Holland, Paul W. and Welsch, Roy E.: 1977, *Commun. Statist.-Theor. Meth.* **A6**. 813 .

Huber, P. J.: 1981, *Robust Statistics*, John Wiley and Sons, New York.

Jefferys, W. H.: 1980, *Astron. J.* **85**, 177.

Jefferys, W. H.: 1981, *Astron. J.* **86**, 149.

Jefferys, W. H. and Feo, J.: 1986, in *Astrometric Techniques*, H. K. Eichhorn and R. J. Leacock (eds.), 637-641.

Kernighan, B. W. and Ritchie, D. M.: 1978, *The C Programming Language*, Prentice–Hall, Englewood Cliffs, NJ.

Lybanon, Matthew: 1984, *Am. J. Physics* **52**, 22.

Rey, William J. J.: 1983, *Introduction to Robust and Quasi-Robust Statistical Methods*, Springer-Verlag, New York.