University of Oxford

Department of Physics

3rd Year Project: Detailed Theoretical and Experimental Investigation of a Digital Clinometer for Cave Surveying

Lev S. Bishop

Supervised by: R. B. Nickerson

September 1999

Abstract

Cavers would like to have accurate digital angle-measuring instruments to use in producing cave surveys. A possible design for a digital clinometer (for measuring angles from the vertical) is presented, using the ADXL202 accelerometer chip. A simple-to-perform calibration method for such a device is derived and a software suite, written to simulate the performance of the clinometer is explained. The results of the simulation confirm the feasibility of the design and suggest suitable values for instrument parameters. A simple version of the clinometer was constructed to test the concept further, but problems with the circuit prevented the gathering of useful data.

TABLE OF CONTENTS

1 INTRODUCTION 4

2 GEOMETRY AND MATHEMATICAL DESCRIPTION 7

3 CALIBRATION 12
3.1 OVERVIEW 12
3.2 NONLINEAR LEAST-SQUARES ALGORITHMS 16
3.3 THE SOFTWARE SUITE 19
3.3.1 Overview and philosophy 19
3.3.2 Summary of Components 21
3.3.3 More Detailed Description of Software 24
3.4 RESULTS AND ANALYSIS 26

4 HARDWARE 33
4.1 OVERVIEW 33
4.2 CIRCUIT DESIGN 33
4.3 CONSTRUCTION 36
4.4 TESTING AND DEBUGGING 37
4.5 RESULTS AND ANALYSIS 38

5 CONCLUSIONS AND SUMMARY 39

A PROGRAM FILES 42
A.1 C PROGRAM FILES 42
A.1.1 add.c 42
A.1.2 aln.c 42
A.1.3 analyse.c 43
A.1.4 cal.c 43
A.1.5 dat.c 50
A.1.6 error.c 51
A.1.7 fnl.c 51
A.1.8 fpic.c 55
A.1.9 lpts.c 59
A.1.10 off.c 60
A.1.11 pic.c 60
A.1.12 pts.c 65
A.2 C HEADER FILES 66
A.2.1 drv_ax.h 66
A.2.2 drv_ay.h 66
A.2.3 drv_az.h 66
A.2.4 drv_by.h 67
A.2.5 drv_bz.h 67
A.2.6 drv_da.h 67
A.2.7 drv_db.h 68
A.2.8 matrix.h 68
A.3 SCRIPT FILES 68
A.3.1 doit 68
A.3.2 dolots 69
A.3.3 domany 69
A.3.4 convert.awk 69
A.3.5 rawdata.awk 70
A.3.6 makefile 70

B SAMPLE OUTPUT 71
B.1 RUN 22 OF NOISY SELF RECALIBRATION WITH 8 POINTS 71
B.1.1 nsr_8_22.aln 71
B.1.2 nsr_8_22.off 71
B.1.3 nsr_8_22.pts 71
B.1.4 nsr_8_22.lpt 71
B.1.5 nsr_8_22.dat 72
B.1.6 nsr_8_22.ldt 72

B.1.7 nsr_8_22.cal ... *72*
B.1.8 nsr_8_22.cof ... *72*
B.1.9 nsr_8_22.pof ... *73*
B.1.10 nsr_8_22.ppt ... *73*
B.1.11 nsr_8_22.pdt ... *73*
B.1.12 nsr_8_22.pcl ... *73*
B.1.13 nsr_8_22.bpt ... *73*
B.1.14 nsr_8_22.bdt ... *74*
B.1.15 nsr_8_22.fin ... *74*
B.1.16 Screen output of the cal program ... *74*
B.1.17 Screen output of the pic program ... *75*
B.2 VERTICAL LEGS ... 75
B.2.1 vertical.bpt ... *75*
B.2.2 vertical.fin ... *75*
B.3 SCALING ... 75
B.3.1 scale.bdt ... *75*
B.3.2 scale.fin ... *76*
C LMDER1 DOCUMENTATION ... 77
D DATA SHEETS ... 82
REFERENCES ... 124

List of Figures

Figure 2.1: Diagrammatic definition of some variables 10
Figure 3.1: A typical data-flow diagram 20
Figure 3.2: The effect on the errors of varying the number of calibration points 27
Figure 3.3: The effect on the errors of varying the sensor noise 28
Figure 3.4: The effect on the errors of varying the levelling error 28
Figure 3.5: The effect on the errors of varying the "ratio of errors" 30
Figure 3.6: The effect on the errors of varying the temperature 30
Figure 3.7: The effect on the errors of varying the number of points and the sensor noise during self-recalibration 31
Figure 4.1: Duty-cycle modulation 33
Figure 4.2: Digital circuit diagram 35
Figure 4.3: Temperature control circuit diagram 36
Figure 4.4: Layout for ADXL202 PCBs 36a
Figure 4.5: The completed circuit (from below) 36b
Figure 4.6: The completed circuit (from above) 36b
Figure 4.7: Oscillogram of switch-on voltage spikes from the old power supply 36c

1 Introduction

When cavers discover new cave passage, they will eventually want to survey it. The requirements for such a survey will vary, depending on the circumstances. It may be a survey of a small extension to a well-known local cave for publication in the caving press; a survey of a large, complex system undergoing active exploration to give the explorers a better feel for how the various parts fit together; or even a survey of many new discoveries on a foreign expedition. To describe the various types of surveys the British Cave Research Association (BCRA) has published a description of survey grades, numbered 1 to 6. Grade 5 is generally considered the most accurate that is justified or achievable, except in special circumstances such as archaeological sites. For all the survey grades where measurements are taken (as opposed to estimates) these are used to produce an accurately-known line along the passage, termed the *centreline*. Other passage details, such as dimensions, shape and character, can be estimated or measured in relation to the centreline. The description for the centreline of a grade 5 survey is:[1]

> Grade **5:** A magnetic survey. Horizontal and vertical angles **accurate** to ±1 degree. Distance **accurate** to ±10cm. Station position error less than 10cm.

There is a set of notes to accompany the table of specifications, but even so there has been some debate[2] as to the precise mathematical interpretation,[*] with most surveyors viewing the limits as "three sigma" limits, thus making the standard deviation of the angular measurements 0.33°.

The usual way grade 5 surveys are produced is with a fibreglass-reinforced measuring tape and sighting compass and clinometer. The measuring tape is cheap, and relatively easy to use in practice with the only difficulties arising in very muddy sections or large pitches (vertical sections). The only suitable instruments for measuring the angles are the compasses and clinometers made by Silva and by Suunto. These are sighting instruments, where the surveyor uses one eye to sight on the station and the other to read the scale, relying on the fact that both eyes point in nearly the same direction. There are a number of problems with such instruments:

[*] The guidelines were written for the benefit of surveyors, not mathematicians!

a) they are expensive (prices range from £90 to £150 for the different models of Suunto clinometer);

b) they are not designed for caving and do not cope with mud, water or impact well – on nearly every surveying trip misting up of the lenses is a problem;

c) they cannot be read out-of-plane, in the sense that the compass must be held level and the clinometer held vertical, which can be a problem on steeply sloping survey legs;

d) it is necessary for the surveyor to have his head behind the instrument to use it, which rules out many otherwise suitable choices for survey stations;

e) they are inaccurate – there is good evidence[3] that even experienced surveyors under favourable conditions cannot expect to come close to the grade 5 specification, with deviations of several degrees being typical, and there is evidence[4] that errors inherent to the instruments (especially older ones) are of the order of a degree;[*]

f) because of the sighting method, different surveyors can obtain different results using the same instrument and it is necessary to calibrate for each combination of surveyor and instrument;

g) the scales are difficult to read (especially in the presence of mud and condensation), and the numbers must be written on fragile waterproof paper – it is all too common for very large “blunder” errors to result.

An ideal solution to all these problems except perhaps that of cost, would be a fully sealed, no-moving-parts instrument. It would be possible to hold the instrument in any orientation, pointing it from one station to the next, and the sighting mechanism would be to use a light beam from a low power laser diode. The RMS measurement error would be no greater than 0.33° with readout via a digital display, with internal storage of all measurements for subsequent download and analysis. With the additional feature that the laser sight is also a rangefinder, this would be a *Total Station Surveyor* (TSS) and would enormously simplify the task of producing an accurate centreline survey.

Constructing a TSS would be too large a task for a project such as this, so this investigation is limited to just one of the three components (compass, clinometer and

[*] These references both deal only with compass error, but similar considerations ought to apply to clinometers.

rangefinder). Designing a rangefinder would involve complex high-speed optoelectronics, and seems unnecessary given the fact that measuring tapes work relatively well. A problem with making a compass is that it is not sufficient simply to measure the vector components of the earth's magnetic field. This information is only sufficient to constrain the instrument direction to lie on a cone[5] (except in the special case that the instrument axis is parallel or antiparallel to the magnetic field). In order to obtain a compass direction it is necessary either to assume that the instrument is level, thereby reintroducing problem (c), or to measure the vertical angle of the instrument, which is equivalent to building a clinometer. For these reasons, the aim of this project is to investigate the possibility of making a digital clinometer suitable for cave surveying. Once a suitable clinometer has been constructed, it should then be possible to add magnetic field sensors and a commercial laser-rangefinder module and achieve the "Holy Grail" of a TSS.

The rest of this document is organised as follows. In the next section we describe the overall layout, geometry and basic equations of such a clinometer. In Sec. 3 we investigate possible ways of solving these equations, thereby determining the inclination. In Sec. 4 we turn our attention to an actual hardware implementation of such a clinometer. Finally, in Sec. 5 we summarise the preceding sections and look towards the future.

2 Geometry and Mathematical Description

The design envisaged for the clinometer is an instrument containing a number of gravity sensors, known as *accelerometers* because they measure the acceleration due to gravity. These sensors are rigidly mounted in relation to each other and to a laser diode, which produces a narrow beam of light for sighting the instrument. The output from the sensors is processed to determine the angle of the laser beam with respect to the direction of gravity.

The sensors chosen for this project were the ADXL202 devices from Analog Devices. Each chip contains two accelerometers with their sensitive axes mounted at right angles to each other. These were chosen because they are cheap, low power, sufficiently accurate and very easy to interface to digital hardware because of the Duty Cycle Modulated (DCM) output. The full datasheet for this device is reproduced in Appendix D, but the most relevant quantities are:

Table 2.1: ADXL202 Specifications

Parameter	Conditions	Min	Typ	Max	Units
Alignment error	X Sensor to Y Sensor		±0.001		Degrees
Noise density	@ +25°C		500	1000	$10^{-6}.g^{a}/\sqrt{\text{Hz}}$
$0\,g^{a}$ offset vs. temperature	ΔT from +25°C		2.0		$10^{-3}.g^{a}/°\text{C}$
Duty cycle per g^{a}	T1/T2 @ +25°C	10	12.5	15	$\%/g^{a}$
Sensitivity temperature drift (worst case fractional change over full temp. range)	ΔT from +25°C -40°C<T<85°C		±0.5		%

[a]*g* is the acceleration due to gravity.

It is clear that at least three gravity axes are required in order to be able to give position information for all orientations of the instrument.[5] In order to supply these it is therefore necessary to use two ADXL202 devices, giving a total of four axes.

It was decided to arrange the sensors so they are all at an angle of π/4 from the main axis of the clinometer (the line of the laser sight). If we call this the *x*-axis then one ADXL202 is mounted in the *x*-*z* plane and one in the *x*-*y* plane. We can define the respective sensitive axes of the four sensors as

$$\mathbf{m}_1 \equiv \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\1\end{pmatrix},\quad \mathbf{m}_2 \equiv \frac{1}{\sqrt{2}}\begin{pmatrix}1\\0\\-1\end{pmatrix},\quad \mathbf{m}_3 \equiv \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\1\\0\end{pmatrix},\quad \mathbf{m}_4 \equiv \frac{1}{\sqrt{2}}\begin{pmatrix}-1\\-1\\0\end{pmatrix}, \qquad (1)$$

and the output q for any acceleration **a** should be the dot product of **a** with **m**

$$q \equiv \mathbf{a}^T\mathbf{m} \quad . \tag{2}$$

There is actually a scale factor, s, and a zero offset error, Δ, for each sensor, so the values, y, we actually obtain will be scaled and shifted as follows,

$$y \equiv sq + \Delta \quad . \tag{3}$$

It will not be possible to mount the chips to the clinometer to sub-degree accuracy (in fact the sensors are only aligned to the chip package to ±1° accuracy) so there will be a number of alignment errors present. In general, three angles are needed to specify an orientation in 3-dimensional space, such as the Euler angles, which would give us a total of six alignment errors. Since these angles are small, an easy way to specify them for each chip is as three consecutive rotations about the *x*-, *y*- and *z*-axes in turn.

The direction of a vector can be specified with two angles, such as the spherical polar angles (θ,ϕ). It makes sense to use spherical polars to specify the direction of the gravity vector *g* in the co-ordinate system of the clinometer, with the $\theta=0$ direction of the spherical polar system being along the measurement axis of the clinometer (the *x*-axis). In this way θ gives us the inclination angle, which is what we are measuring with the clinometer. On the other hand, we are not interested in the values of ϕ, so we may be able to reduce the number of alignment errors by one by defining one of the rotations about the *x*-axis to be zero. This means the zero of ϕ is fixed to the (unknown) orientation of one of the ADXL202 chips.

There is also an “X Sensor to Y Sensor” alignment error specified in the ADXL202 datasheet (Appendix D or Table 2.1) which is of much smaller magnitude. This can be represented by a rotation about the *y*-axis the for the ‘Y sensor’ of the first ADXL202 and a rotation about the *z*-axis of the ‘Y sensor’ for the second ADXL202.

Putting all this together we can write y_J^n, the output of the *J*-th sensor ($1 \le J \le 4$),[*] on the *n*-th measurement ($1 \le n \le N$), in units of *g*, as

[*] Note that henceforth we use the notation convention that upper-case roman suffices (e.g. *I*, *J*,…) run over the values 1,…,4, whereas lower case suffices (e.g. *i*, *j*,…) run over the values 1,2,3.

$$y_J^n \equiv \sum_{i=1}^{3} s_J M_{Ji} g_i^n\left(\theta^n, \phi^n\right) + \Delta_J \quad , \tag{4}$$

where s_J is a scale factor for each sensor; Δ_J is a zero offset for each sensor; g_i^n is the gravity vector in the clinometer axis system as defined above, as a function of the polar angles θ^n, ϕ^n,

$$g_i(\theta, \phi) \equiv \begin{pmatrix} \cos\theta \\ \sin\theta\cos\phi \\ \sin\theta\sin\phi \end{pmatrix} \quad ; \tag{5}$$

and M_{Ji} is a matrix whose rows are unit vectors which represent the sensitive axes of the sensors,

$$M_{Ji} \equiv \begin{pmatrix} \mathbf{m}_1^T\, \mathbf{R}\left(\alpha_x, \alpha_y, \alpha_z\right) \\ \mathbf{m}_2^T\, \mathbf{R}_y(\delta_a)\mathbf{R}\left(\alpha_x, \alpha_y, \alpha_z\right) \\ \mathbf{m}_3^T\, \mathbf{R}\left(0, \beta_y, \beta_z\right) \\ \mathbf{m}_4^T\, \mathbf{R}_z(\delta_b)\mathbf{R}\left(0, \beta_y, \beta_z\right) \end{pmatrix} \quad . \tag{6}$$

The **R** matrices are rotation matrices about the different axes,

$$\begin{gathered} \mathbf{R}(x, y, z) \equiv \mathbf{R}_x(x)\mathbf{R}_y(y)\mathbf{R}_z(z) \quad ; \\ \mathbf{R}_x \equiv \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\theta) & \sin(\theta) \\ 0 & -\sin(\theta) & \cos(\theta) \end{pmatrix} \quad , \\ \mathbf{R}_y \equiv \begin{pmatrix} \cos(\theta) & 0 & \sin(\theta) \\ 0 & 1 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) \end{pmatrix} \quad , \\ \mathbf{R}_z \equiv \begin{pmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad ; \end{gathered} \tag{7}$$

$\alpha_x, \alpha_y, \alpha_z, \beta_y$ and β_z are the mounting errors described above; and δ_a and δ_b are the "X Sensor to Y Sensor" alignment errors described above.

In practise there will also be noise, ε_J^n, associated with each value, y_J^n, resulting in a measured quantity Y_J^n,

$$Y_J^n = y_J^n + \varepsilon_J^n \quad , \tag{8}$$

where ε_J^n may be drawn from the normal distribution $N\left(0,\sigma_\varepsilon^2\right)$, if the noise can be treated as Gaussian white noise.

We can use various minimisation procedures to determine θ from the above equations, but these procedures will usually need an initial estimate in order to ensure convergence. If we assume $\alpha_x = \alpha_y = \alpha_z = \beta_y = \beta_z = \delta_a = \delta_b = \Delta_J = s_J = \varepsilon_J^n = 0, \quad \forall J, n$ then we can define the following quantities,

$$l_1 \equiv \sqrt{Y_1^2 + Y_2^2}\ ; \quad a_1 \equiv \mathrm{Arctan}\left(\frac{Y_1}{Y_2}\right) \quad , \tag{9}$$

$$l_2 \equiv \sqrt{Y_3^2 + Y_4^2}\ ; \quad a_2 \equiv \mathrm{Arctan}\left(\frac{-Y_4}{-Y_3}\right) \quad , \tag{10}$$

as shown in Fig. 2.1, where we have dropped the implied *n* superscript for ease of notation. Here $\mathrm{Arctan}\left(\frac{b}{a}\right)$ is a function similar to $\arctan\left(\frac{b}{a}\right)$, but choosing the correct quadrant depending on the signs of the arguments *a* and *b*, rather than the principal range $\left(-\frac{\pi}{2},\frac{\pi}{2}\right)$ implied by $\arctan\left(\frac{b}{a}\right)$.

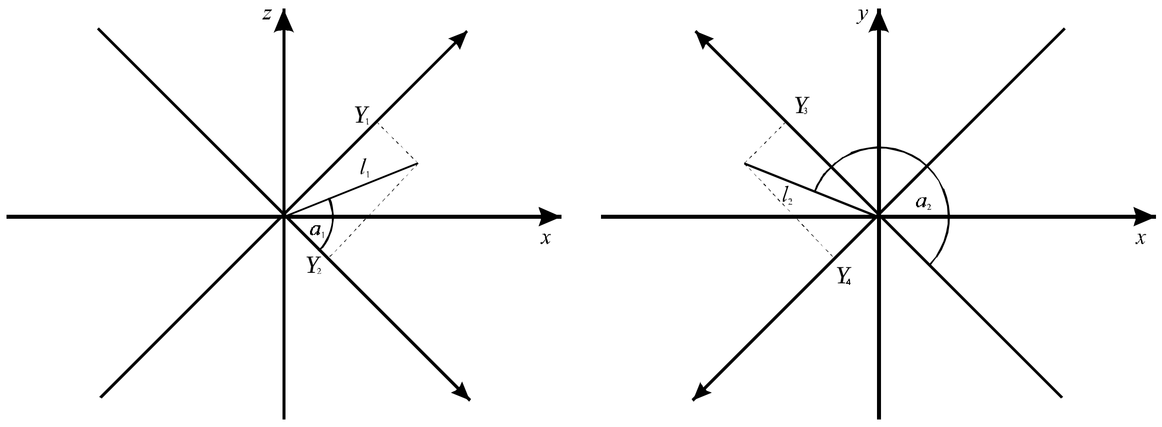

Figure 2.1: Diagrammatic definition of some variables

Using these values we can now calculate the Cartesian components of *g* in the clinometer basis. For the *y*- and *z*-components we have

$$g_z \equiv g_3 = l_1 \sin\left(a_1 - \tfrac{\pi}{4}\right); \quad g_y \equiv g_2 = l_2 \sin\left(a_2 - \tfrac{\pi}{4}\right) \quad . \tag{11}$$

Since we have two ways of calculating g_x we take their arithmetic mean,

$$g_x \equiv g_1 = \frac{l_1 \cos\left(a_1 - \frac{\pi}{4}\right) + l_2 \cos\left(a_2 - \frac{\pi}{4}\right)}{2} \quad . \tag{12}$$

Now that we have the Cartesian co-ordinates it is simple to transform to the polar co-ordinates

$$\phi = \mathrm{Arctan}\left(\frac{g_z}{g_y}\right); \quad \theta = \mathrm{Arctan}\left(\frac{g_x}{\sqrt{g_y^2 + g_z^2}}\right) \quad . \tag{13}$$

In the next section we attempt to solve the equations defined in this section, in order to determine values of the various constants, and hence obtain more accurate values of θ.

3 Calibration

3.1 Overview

In order to produce accurate measurements with the clinometer it is necessary to determine various calibration parameters. These are the scale factors and zero offsets for each sensor and the various angular misalignment errors defined in Sec. 2. A calibration procedure is necessary to determine these parameters.

One natural way of calibrating the clinometer would be to construct some kind of test jig, which could accurately position the clinometer in a number of known orientations. We could then attempt to find the parameters by performing the minimisation

$$\min_{\mathbf{b}} \sum_{n=1}^{N} \sum_{J=1}^{4} \left[Y_J^n - y_J^n\left(\theta^n, \phi^n; \mathbf{b}\right)\right]^2 \quad , \tag{14}$$

where **b** is a vector whose components are the parameters being determined. This is equivalent to a *maximum likelihood* calibration in the case that the errors, ε, can be regarded as Gaussian white noise. In other words, the vector **b** that solves Eq. (14) is the one for which the observed values Y are the most likely to have occurred. Equation (14) can be solved by methods of Nonlinear Least-Squares (NLS). Here nonlinearity refers to the nonlinear dependence of the function y on the parameters **b** and is unrelated to any nonlinearity with respect to the independent variables θ, ϕ. There are a number of numerical methods for solving NLS problems, which are discussed in Sec. 3.2.

In this scheme the co-ordinate axes of the clinometer are fixed, since the values of θ and ϕ are known. Hence we cannot use the trick described in Sec. 2 of removing one of the misalignment angles by allowing the zero of ϕ to vary. Therefore, the number of elements in **b** is 14 (4 zero offsets, 4 scale factors and 6 alignment errors), if we treat the "X Sensor to Y Sensor" alignment errors as zero, which is reasonable since they are so small. Each observation along a different direction gives us 4 pieces of data, or 4 of the bracketed terms in Eq. (14) (J=1,…,4). The minimum number of observations necessary in order to perform the calibration is thus 4, although more would probably be used in practise. The numerical problem of solving Eq. (14) will be quite easy because all the nonlinearity is from sines and cosines of the alignment errors. Since all these angles should be small (of the order of 5°), the problem will be almost linear. Also, the minimisation will be over 14 parameters and will be of a sum of around 14 terms so the problem is a small one.

The main disadvantage of this calibration scheme is the need for a special jig, which would need to be machined and levelled to sub-degree accuracy. For this reason, other calibration schemes, which do not require any kind of jig, were investigated. In the most promising scheme, a number of observations are taken with the clinometer in a variety of orientations, with θ and ϕ unknown. Since this is not sufficient to fix the zero of θ to be vertical, it is necessary also to have a number of observations of known θ (but the values of ϕ still need not be known). These measurements could most easily be obtained by taking a few shots across a known level surface, such as the surface of a calm pool of water.

At first sight, this might appear to be an ideal problem to tackle with a technique called implicit Orthogonal Distance Regression (ODR), which is capable of solving problems of the form

$$\min_{\mathbf{b}} \sum_{n=1}^{N} \sum_{j=1}^{4} \left(\varepsilon_j^n\right)^2 \quad , \tag{15}$$

with constraints

$$f_j^n\left(y^n;\mathbf{b}\right) = 0 \quad p = 1,2,\ldots,P \quad . \tag{16}$$

In the present case the constraints arise from eliminating θ and ϕ from Eqs. (4) – (7).

One way to derive these equations is to define 3-element vectors and matrices from the 4-element ones in Eq. (4),

$$y_j^{(1)} \equiv \begin{pmatrix} y_2 \\ y_3 \\ y_4 \end{pmatrix}, \quad y_j^{(2)} \equiv \begin{pmatrix} y_1 \\ y_3 \\ y_4 \end{pmatrix}, \quad y_j^{(3)} \equiv \begin{pmatrix} y_1 \\ y_2 \\ y_4 \end{pmatrix}, \quad y_j^{(4)} \equiv \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \qquad j = 1,2,3 \quad , \tag{17}$$

and similarly for $s_j^{(z)}$, $\Delta_j^{(z)}$ and $M_{ji}^{(z)}$. Illustrative examples are

$$s_j^{(3)} \equiv \begin{pmatrix} s_1 \\ s_2 \\ s_4 \end{pmatrix}, \quad \Delta_j^{(4)} \equiv \begin{pmatrix} \Delta_1 \\ \Delta_2 \\ \Delta_3 \end{pmatrix}, \quad M_{ji}^{(2)} \equiv \begin{pmatrix} M_{11} & M_{12} & M_{13} \\ M_{31} & M_{32} & M_{33} \\ M_{41} & M_{43} & M_{43} \end{pmatrix} \quad , \tag{18}$$

in which the bracketed superscript indicates the omitted component, and we continue to make use of the suffix notation defined in the footnote on p.8.

We can hence rewrite Eq. (4) as follows,

$$y_j^{(z)} = \sum_i s_j^{(z)} M_{ji}^{(z)} g_i(\theta,\phi) + \Delta_j^{(z)} \quad , \tag{19}$$

ignoring the *n* superscripts for now. These equations can be solved for *g* by inverting *M*,

$$g_i = \left(M^{(z)}\right)_{ij}^{-1} \frac{y_j^{(z)} - \Delta_j^{(z)}}{s_j^{(z)}} \quad . \tag{20}$$

So we can write four different expressions for *g*, one from each of the different values of *z*. Equating any two of these finally eliminates *g* (and thus θ and ϕ) from our equations. Thus, for example,

$$\left(M^{(1)}\right)_{ij}^{-1} \frac{y_j^{(1)} - \Delta_j^{(1)}}{s_j^{(1)}} = \left(M^{(2)}\right)_{ij}^{-1} \frac{y_j^{(2)} - \Delta_j^{(2)}}{s_j^{(2)}} \quad , \tag{21}$$

which actually gives us three constraints, but only two of these will be independent, since we have only eliminated two variables, θ and ϕ.

We must now include the observations that were taken across a level surface. For these values of *n* we know θ=π/2+δ, where δ is a measurement error, assumed to be distributed normally, $\delta^n \sim N\left(0,\sigma_\delta^2\right)$. If we define the first *k* measurements to be taken across the level surface then we can use the fact that g_1=cos (θ), from Eq. (5) to write the full equations as, for example,

$$\min_{\mathbf{b},\varepsilon,\delta}\left[\sum_{J=1}^{4}\sum_{n=1}^{N}\left(\frac{\varepsilon_J^n}{\sigma_\varepsilon}\right)^2 + \sum_{n=1}^{k}\left(\frac{\delta^n}{\sigma_\delta}\right)^2\right] \quad , \tag{22}$$

subject to the constraints

$$\begin{gathered}\sum_{j=1}^{3}\left[\left(M^{(1)}\right)_{ij}^{-1}\frac{Y_j^{(1)n} - \varepsilon_j^{(1)n} - \Delta_j^{(1)}}{s_j^{(1)}} - \left(M^{(2)}\right)_{ij}^{-1}\frac{Y_j^{(2)n} - \varepsilon_j^{(2)n} - \Delta_j^{(2)}}{s_j^{(2)}}\right] = 0\,; \quad i = 1,2 \quad , \quad n = 1,\dots,N \\ \sum_{j=1}^{3}\left(M^{(1)}\right)_{1j}^{-1}\frac{Y_j^{(1)n} - \varepsilon_j^{(1)n} - \Delta_j^{(1)}}{s_j^{(1)}} - \cos\left(\frac{\pi}{2} + \delta^n\right) = 0\,; \quad n = 1,\dots,k\,.\end{gathered} \tag{23}$$

The method generally used[*] for implicit ODR with *P* constraints in the form of Eqs. (15) and (16) is to solve

[*] See, e.g., Ref. 6.

$$\lim_{\xi\to\infty}\ \min_{\mathbf{b},\varepsilon}\sum_{n=1}^{N}\left[\xi\left(\sum_{p=1}^{P}\left[f_p^n\left(Y^n-\varepsilon^n;\mathbf{b}\right)\right]^2\right)+\sum_{j}^{4}\left(\varepsilon_j^n\right)^2\right] \tag{24}$$

using a standard NLS technique for the minimisation part.

Solving Eq. (24) is generally more difficult than solving Eq. (14), for a number of reasons:

a) the number of parameters will be much larger, since the minimisation is now over ε and δ as well as **b**. This means there will be 4*N*+13+*k* parameters as opposed to just 14 (since in this situation we *can* use the trick described in Sec. 2 to remove one of the misalignment variables);

b) the number of observations, *N*, will necessarily be larger, by a factor in the range (4/3,2). Each observation still produces 4 pieces of data, but there are only 3 (for levelled shots) or 2 (for shots with a general orientation) constraints on the y_J^n per observation in Eq. (24) as opposed to the 4 (implicit) constraints in Eq. (14);

c) the amount of nonlinearity may well be much greater in Eq. (24); and

d) the NLS minimisation must be carried out multiple times, for differing values of ξ.

All of these difficulties can be overcome with some effort, but there is a better way. Instead of using an implicit ODR method, we can retain the original NLS method, and make the θ^n and ϕ^n parameters to be fitted instead of independent variables, thereby solving the following equation,

$$\min_{\theta,\phi,\mathbf{b}}\sum_{n=1}^{N}\sum_{J=1}^{4}\left[\frac{Y_J^n-y_J^n\left(\theta^n,\phi^n;\mathbf{b}\right)}{\sigma_\varepsilon}\right]^2+\sum_{n=1}^{k}\left[\frac{\theta^n-\frac{\pi}{2}}{\sigma_\delta}\right]^2 \quad . \tag{25}$$

Equation (25) should be easier to solve than Eq. (24) because although point (b) also applies in this case, points (c) and (d) do not. As regards point (a), Eq. (25) only requires a minimisation in 2*N*+13 parameters as opposed to the 4*N*+13+*k* parameters of Eq. (24). In addition to these important advantages, Eq. (25) is simpler and easier to visualise than Eqs. (15) and (16), since it is expressed in terms of θ and ϕ. For these reasons Eq. (25) is our preferred way to extract the parameters from the calibration data.

3.2 Nonlinear Least-Squares Algorithms

Each of the various calibration procedures of Sec. 3.1 requires an NLS algorithm at some stage in the process. An NLS problem is one of the form

$$\min_{\mathbf{x}\in\mathbf{R}^n} f(\mathbf{x}), \quad f(\mathbf{x}) \equiv \tfrac{1}{2}\sum_{\alpha=1}^{m} r_\alpha^2(\mathbf{x}), \quad m \geq n \quad , \tag{26}$$

where each $r_\alpha(\mathbf{x})$ is a nonlinear function, called the *residual* at **x**.

We define the Jacobian of the residual vector, **r**(**x**), as

$$J(\mathbf{x})_{\alpha\beta} \equiv \frac{\partial r_\alpha(\mathbf{x})}{\partial x_\beta} \quad , \tag{27}$$

and the Hessian matrices of **r**(**x**)

$$G_\alpha(\mathbf{x}) \equiv \nabla^2 r_\alpha(\mathbf{x}), \quad G_\alpha(\mathbf{x})_{\beta\gamma} = \frac{\partial^2 r_\alpha(\mathbf{x})}{\partial x_\beta \partial x_\gamma} \quad . \tag{28}$$

The first and second derivatives of *f*(**x**) are then given by

$$\nabla f(\mathbf{x}) = J(\mathbf{x})^T \mathbf{r}(\mathbf{x}) \quad , \tag{29}$$

and

$$\nabla^2 f(\mathbf{x}) = J(\mathbf{x})^T J(\mathbf{x}) + Q(\mathbf{x}), \quad Q(\mathbf{x}) \equiv \sum_{\alpha=1}^{m} r_\alpha(\mathbf{x}) G_\alpha(\mathbf{x}) \quad . \tag{30}$$

The NLS problem (26) can be viewed as a special case of an optimisation problem, in which a quadratic model of *f* (**x**) is used,

$$\tilde{f}_c(\mathbf{x}_c + \Delta\mathbf{x}) = f(\mathbf{x}_c) + \nabla f(\mathbf{x}_c)^T \Delta\mathbf{x} + \tfrac{1}{2}\Delta\mathbf{x}^T \nabla^2 f(\mathbf{x}_c) \Delta\mathbf{x} \quad , \tag{31}$$

and iterating with $\mathbf{x}_{c+1} = \mathbf{x}_c + \Delta\mathbf{x}$. This is equivalent to *Newton's method*, for which the local convergence rate is usually quadratic (and linear problems are solved in a single step), and which takes no advantage of the special form of (26).

Quite often, however, $Q(\mathbf{x})$ can be ignored. This will be the case if either $\mathbf{r}(\mathbf{x})$ is only mildly nonlinear at $\mathbf{x}_c$ or the residuals $r_\alpha(\mathbf{x}_c)$ are small.* For the problems of Sec. 3.1 to which we will be applying NLS, these conditions should hold, and we will proceed to ignore $Q(\mathbf{x})$. This is equivalent to making a linear approximation to $\mathbf{r}(\mathbf{x})$ in the region of $\mathbf{x}_c$ and is desirable because frequently second-derivative information about $\mathbf{r}(\mathbf{x})$ is not easily available. In fact, it has been suggested[7] that inclusion of this term can be destabilising if the model fits badly or the data are contaminated by "outlier" points.

Solving the iteration that results from making this change, namely

$$\min_{\mathbf{x}_{c+1}} \left\| \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)(\mathbf{x}_{c+1} - \mathbf{x}_c) \right\| \tag{32}$$

yields the *Gauss-Newton* method. This has fast convergence on mildly nonlinear, small-residual problems, but may fail to be even locally convergent on problems which fail to satisfy these conditions. It also has problems when $J(\mathbf{x})$ can be rank-deficient, indicating that some of the parameters x_α are not independent. In the case that we are using this technique to solve Eq. (25) this would occur if, for any n, the polar angle θ^n was equal to 0 or $\pi/2$, as in this case the corresponding angle ϕ^n becomes completely unconstrained. A modification of the Gauss-Newton method, which solves many of its deficiencies, is the *Levenberg-Marquardt* method, in which the iteration (32) is replaced by

$$\min_{\mathbf{x}_{c+1}} \left(\left\| \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)(\mathbf{x}_{c+1} - \mathbf{x}_c) \right\|^2 + \mu_c \left\| \mathbf{x}_{c+1} - \mathbf{x}_c \right\|^2 \right) \quad , \tag{33}$$

where $\mu_c \geq 0$ is the parameter that limits the size of $\Delta\mathbf{x}_c = \mathbf{x}_{c+1} - \mathbf{x}_c$. Now $\Delta\mathbf{x}_c$ is well defined by Eq. (33) with $\mu_c \neq 0$ even when $J(\mathbf{x}_c)$ is rank-deficient. As $\mu_c \to \infty$, $\|\Delta\mathbf{x}_c\| \to 0$ and the direction $\Delta\mathbf{x}_c$ becomes parallel to the steepest-descent direction $J(\mathbf{x}_c)\mathbf{r}(\mathbf{x}_c)$.

It can be shown that Eq. (33) is equivalent to the least-squares problem with quadratic constraint

---

* Strictly, "small" in the sense that $\|\mathbf{r}(\mathbf{x}_c)\|$ is small compared with the *smallest* eigenvalues of $J^T(\mathbf{x}_c)J(\mathbf{x}_c)$.

$$\min_{\Delta\mathbf{x}_c} \left\| \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)\Delta\mathbf{x}_c \right\| \quad , \qquad \left\| \Delta\mathbf{x}_c \right\| \le \delta_c \quad , \tag{34}$$

for some value of δ_c related to μ_c. If the constraint is not binding then $\mu_c = 0$, otherwise $\mu_c > 0$. The constraint can be thought of as providing a region of trust for the linear model

$$\mathbf{r}(\mathbf{x}) \approx \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)(\mathbf{x} - \mathbf{x}_c) \quad , \tag{35}$$

and for this reason this type of method is termed a *model trust region* method.

An implementation of the Levenberg-Marquardt method as a model trust region algorithm has been given by Moré[8] and is contained in the software package **MINPACK**. Moré's iteration is of the following form:

1. Determine $\Delta\mathbf{x}_c$ as a solution to

$$\min_{\Delta\mathbf{x}_c} \left\| \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)\Delta\mathbf{x}_c \right\| \quad , \qquad \left\| D_c \Delta\mathbf{x}_c \right\| \le \delta_c \, .$$

2. Compute the model prediction of the reduction in $f(\mathbf{x})$ as

$$\Delta f^{\text{pred}} = \tfrac{1}{2}\left(\left\| \mathbf{r}(\mathbf{x}_c) \right\|^2 - \left\| \mathbf{r}(\mathbf{x}_c) + J(\mathbf{x}_c)\Delta\mathbf{x}_c \right\|^2 \right) \quad ,$$

and the actual reduction as

$$\Delta f^{\text{real}} = \tfrac{1}{2}\left(\left\| \mathbf{r}(\mathbf{x}_c) \right\|^2 - \left\| \mathbf{r}(\mathbf{x}_c + \Delta\mathbf{x}_c) \right\|^2 \right) \quad .$$

3. Compute the ratio $\rho_c = \Delta f^{\text{real}} / \Delta f^{\text{pred}}$. If $\rho_c > \beta$ then set $\mathbf{x}_{c+1} = \mathbf{x}_c + \Delta\mathbf{x}_c$, otherwise set $\mathbf{x}_{c+1} = \mathbf{x}_c$.

4. Update D_c and δ_c.

Here, D_c is a diagonal *scaling matrix*. Moré chooses the scaling such that the algorithm is scale-invariant, in the sense that the same iterations occur for $\mathbf{r}(K\mathbf{x})$ for any nonsingular diagonal matrix *K*. The constant β is in the range (0,1). An iteration with $\rho_c > \beta$ is considered successful; after an unsuccessful iteration δ_c is reduced. There are several other techniques used by Moré to control the size of δ_c in order to minimise the number of function evaluations needed for convergence. The general idea is to increase δ_c whenever the quadratic model is performing well, and reduce δ_c when the model is performing badly. Step 1 of the algorithm is usually performed by solving the equivalent form of Eq. (33), and searching for the correct value of μ. Much care is needed when implementing this algorithm as a program for a real computer to avoid losing more precision than is necessary when representing numbers to only finite precision.

The initial values $\mathbf{x}_0, D_0$ and δ_0 are given to the algorithm as inputs, as is β.

Moré has proven that under rather mild given conditions the algorithm will always converge. The algorithm works very well in practise, particularly in the very carefully-written form of the software package **MINPACK**. It was chosen by this author, as the best algorithm to use in the current context of the problem of calibrating a clinometer, for a number of reasons. In particular, the author attempted to consult a selection of modern textbooks[7,9,10] on numerical optimisation methods. Under the assumption that the Levenberg-Marquardt technique is appropriate for the problem at hand these all concurred that **MINPACK** was a tried-and-tested all-purpose implementation. Furthermore, a search through the online archives of numerical software routines (especially **netlib**) for a routine turned up only **MINPACK** and some other more complex Levenberg-Marquardt derivatives, thereby validating our assumption about the appropriateness of the Levenberg-Marquardt technique. Finally, the more complex derivatives were found under closer inspection to offer no advantages to the problem at hand.

The next section describes the software suite that was written around **MINPACK** for the purpose of calibrating a clinometer.

3.3 The Software Suite

3.3.1 Overview and philosophy

The general philosophy was to split the problem into as small sections as was practicable, to write simple programs to solve these small sections, and then to stitch those small programs together using shell scripting. This approach brings the usual advantages of modularity over writing large monolithic programs:

a) there are many different ways the programs can be stitched together in order to solve different problems – with one large program it would be necessary to write new code for each new problem;

b) the programming and debugging of each individual section is much easier;

c) the data are available from each intermediate stage for analysis and modification. This makes testing much easier and allows more flexibility, with the possibility to edit data by hand in special cases, such as those in Appendices B.2 and B.3; and

d) the same programs are used to solve the different problems and hence if they have been tested well in one situation, such as with computer-generated data, they can be expected to perform reliably in another where testing might be harder, such as with physically-measured data.

The software was all written in the ‘C’ language, which was chosen primarily for reasons of familiarity.

The package is capable of performing two types of calibration using the Levenberg-Marquardt algorithm of Sec. 3.2, as implemented in **MINPACK**, to solve the NLS problem in Eq. (25). It can attempt either to calibrate $\alpha_x, \alpha_y, \alpha_z, \beta_y, \beta_z$, s_J and Δ_J, or only to calibrate s_J and Δ_J. For the former **cal** is used, and for the latter either **pic** or **fpic**. In addition, it can perform Monte Carlo simulations of the clinometer hardware under various circumstances, generating numbers to represent all the sources of error described in Sec. 2. The package is quite flexible and the parts can be joined together in some complex ways. A typical example of a data-flow diagram is in Fig. 3.1, which describes a situation like the one used to create Fig. 3.7. In this diagram the solid lines represent data-flow which is essential for the programs to run, and the dashed lines represent “optional” data, which is only used by the programs to compare the calculated results against the original numbers.

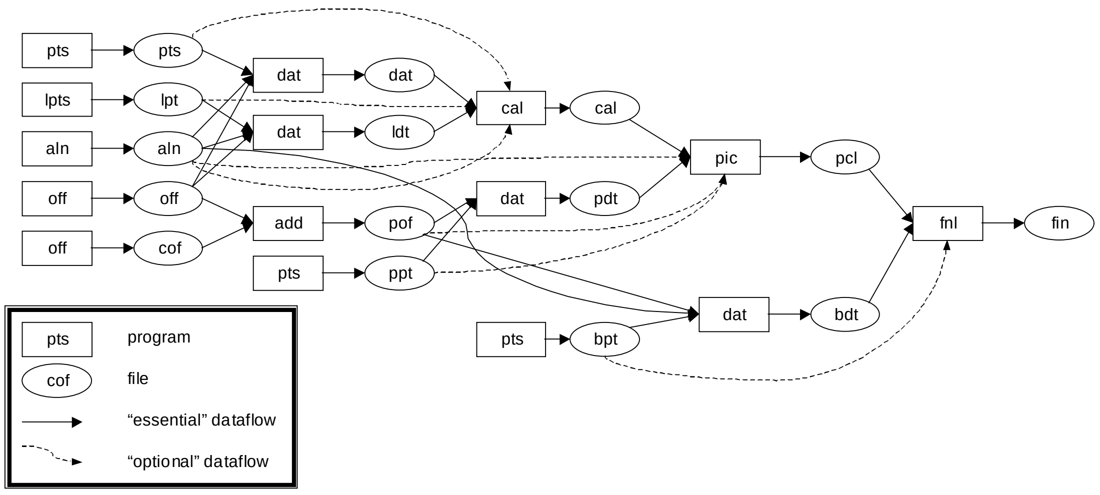

Figure 3.1: A typical data-flow diagram

3.3.2 Summary of Components

aln

generates alignment errors. It takes two arguments, the standard deviation (s.d.) of the chip package mounting error and the sensor-to-sensor misalignment (within the same chip package). It produces angles $\alpha_x, \alpha_y, \alpha_z$, β_y, β_z, δ_a, δ_b as defined in Sec. 2.

off

generates a zero offset and scale factor error for each of the 4 sensors (s_J and Δ_J of Sec. 2). It takes two arguments, the s.d. of the offsets (in units of *g*), and the s.d. of the scale factors (in percent).

pts

generates polar angles which are evenly distributed over the sphere. It takes one argument, the number of such angles to produce. This uses the same output format as **lpts**.

lpts

generates polar angles which are close to the horizontal. It takes two arguments, the number of angles to produce, and the s.d. of the differences from the horizontal. This uses the same output format as **pts**.

dat

simulates the sensor output. It takes four arguments, the name of a file containing alignment errors (produced by **aln**), the name of a file containing offsets and scale factors (produced by **off**), the name of a file containing angles (produced by **pts** or **lpts**), and a value for sensor RMS noise in units of *g*/1000.

cal

attempts to calibrate the clinometer. It takes either three or seven arguments, a file of sensor data (produced by **dat**), a file of sensor data nominally taken at horizontal angles (produced from **dat**, usually acting on a file from **lpts**), an optional file of alignment data (from **aln**), an optional file of offset and scale factor data (from **off**), two optional files of the angular data which was fed to **dat** to produce the first two files. The final argument is the factor $\sigma_\varepsilon/\sigma_\delta$ using the notation of Sec. 3.1 (and measuring σ_ε in units of *g* and σ_δ in radians). The standard output of this program contains the calibrated values for the alignment errors, offsets and scale factors. The screen output (standard error) contains additional information about the calibration process. If the optional arguments are used (obviously the files would not exist for a real calibration) then the calculated values are compared against the actual values.

add

sums offset and scale factor errors, which is useful for simulating effects like temperature drift. It takes two arguments, specifying the names of two files containing offset and scale factor information (as produced by **off**). The output is the element-by-element sum of these files, in the same format as used by **off**.

pic

is similar in function to **cal**, but it does not attempt to calibrate the angular misalignment errors, instead using those from a previous run of **cal**. It takes either two or four arguments, a file of calibration data (as produced by **cal**) from which the angular calibration is taken, along with the initial guess for the offsets and scale factors, a file of sensor data (as produced by **dat**), an optional file of offset and scale factor data (as produced by **off** or **add**), and an optional file of the angular data which was fed to **dat** to produce the second file (as produced by **pts** or **lpts**). The output is a set of calibration constants, in the same format as is produced by **cal**, with the angular constants fed straight through. The screen output (standard error) contains additional information about the calibration process. If the optional arguments are used then the calculated values are compared against the actual values.

fpic

performs exactly the same function as **pic**, only using single-precision floating-point arithmetic instead of double-precision.

fnl

calculates the polar angles for a set of data. It takes either two or three arguments, a file of sensor data (as produced by **dat**), a file of calibration data (as produced by **cal**, **pic** or

fpic) and an optional file of the angular data which was fed to **dat** to produce the first file (as produced by **pts** or **lpts**). The output is the set of polar angles. If the optional argument is given then various statistics are also output, concerning the level of agreement between the calculated values and the original values.

analyse

is a filter which takes the results of a number of runs of **fnl** and produces various summary information.

doit

is a shell script, which automates tying the previous programs together in various different ways. The version in Appendix A.3.1 is configured for looking at the effect of temperature drift.

dolots

is a shell script which automates running **doit** 100 times (saving the output each time), and running **analyse** on the results to produce a summary.

domany

is a shell script which automates running **dolots** a number of different times (saving the output each time), with different values of its second parameter, and summarising the results. This is useful for investigating the effect of varying one of the parameters of the experiment, and producing graphs like those in Sec. 3.4.

convert.awk

is a program for the **awk** utility which converts the human-friendly output of **domany** into a more machine-friendly form, for importing into spreadsheets, and so on.

rawdata.awk

is a program for the **awk** utility which automates converting data from pairs of 8-digit counter readings (as recorded during actual experiments), into floating-point numbers in the range (-1,1) by calculating $x = 1 + 2(x_{\max} - x_{\min})\left(\frac{a}{a+b} - x_{\min}\right)$, where a, b are the two numbers and x is the result.

makefile

is a program for the **make** utility, which automates compiling only those parts of the suite which need recompiling at any given time.

3.3.3 More Detailed Description of Software

Most of the code is fairly straightforward, and space does not permit too much detail, so this is a brief description of the more interesting aspects. The code itself is reproduced in Appendix A.

Most of the programs produce as their first line of output a summary of their input parameters. This line is appended to any subsequent summaries based on the output of other programs. This makes it slightly easier to follow the sometimes complex dataflow which is possible (see Fig. 3.1).

All the programs use degrees as their unit of angle externally, in user input and data files, but convert angles into radians for internal use.

The files **aln.c**, **off.c**, **pts.c**, **lpts.c** and **dat.c** are all involved in generating random datasets as part of the Monte Carlo simulations. The only interesting aspect to them is that they frequently require random numbers with a Gaussian (normal) distribution, where the standard 'C' `rand()`function gives a uniformly distributed number from 0 to `RAND_MAX`. The algorithm used is the "polar method for normal deviates", originally described by Box *et al.*[11] This is Knuth's version of the algorithm,[12] implemented in function `error()`of **error.c**.

1. [Get uniform variables.] Generate two independent random variables V_1, V_2, uniformly distributed between –1 and +1.

2. [Compute S]. Set $S \leftarrow V_1^2 + V_2^2$.

3. [Is $S \geq 1$?] If $S \geq 1$ return to step **1**.

4. [Compute X_1, X_2.] Set $X_1 = V_1\sqrt{\frac{-2\ln S}{S}}$, $X_2 = V_2\sqrt{\frac{-2\ln S}{S}}$. These are normally distributed variables with zero mean and unity variance.

The files **cal.c**, **pic.c**, **fpic.c** and **fnl.c** are all very similar. They all call upon routines from the **MINPACK** package to perform the least-squares parameter fitting. They have a number of compile-time options that modify their behaviour for testing purposes. These are:

- `VERBOSE`: Causes the program to output values of the parameters to be fitted both before and after the fitting procedure has been called, and various other debugging information;
- `CHECKJAC`: Causes the program to call the `chkder()` routine of **MINPACK** in order to check that the Jacobian calculated by the program is consistent with the function calculated by the program; and
- `NUMDIFF`: Causes the program to use the `lmdif1()` routine instead of the `lmder1()` routine of **MINPACK** to perform the fit. This means that the Jacobian is calculated by a numerical method rather than analytically.

The header files (ending in '**.h**') contain expressions for the matrix M_{Ji} of Sec. 2 and its various derivatives. These were calculated using the computer algebra package **Derive**.

The parameter `TOL` to the least-squares routine (`lmder1()` or `lmdif1()`), which sets the tolerance condition on terminating the least-squares procedure, is set to zero, which is interpreted as meaning "use the machine precision to set the tolerance". This is probably inefficient, causing more iterations than is strictly necessary, but ensures the software returns the most precise values it is capable of producing.

The parameters to be fitted by the least-squares routine are stored in the array `X[]` in **cal.c** (which takes the place of $\mathbf{x}_c$ in Sec. 3.2) in the following order:

$$\texttt{X[]}=[\,\alpha_x,\alpha_y,\alpha_z,\beta_y,\beta_z,\, s_1,\ldots,s_4,\, \Delta_1,\ldots,\Delta_4,\, \theta^1,\phi^1,\ldots,\theta^N,\phi^N\,] \quad .$$

The files **pic.c**, **fpic.c** and **fnl.c** contain similar arrays `X[]`, but omitting the relevant parameters which are not being fitted in each case.

It is necessary to supply an initial estimate of the array `X[]` to the least-squares routine. The misalignment angles are estimated as zero, in **cal.c**. In **cal.c** the offsets and scale factors are initialised to zero and one, whereas in **pic.c** and **fpic.c** they are initialised to their values taken from the calibration file which is fed to the program. In all cases the θ^n and ϕ^n are initialised to their values estimated using Eqs. (9) – (13).

The remaining files and scripts are straightforward and should be self-explanatory.

3.4 Results and Analysis

The first problem to which the software suite was applied was to determine how well the calibration procedures developed in the previous sections could be expected to work, and what were optimal values for the various variables, such as the measurement noise and the number of measurements made with the clinometer levelled. In order to answer this question, the software was set up to generate a set of percentage scale factors distributed as $N(0,25^2)$, offset errors distributed as $N(0,0.2^2)$ in units of g, ADXL202 misalignment errors distributed as $N(0,5^2)$ degrees and sensor-to-sensor alignment errors distributed as $N(0,0.001^2)$ degrees. It then generated a number, n_l, of nominally level points with levelling error distributed as $N(0,\sigma_\delta{}^2)$, and a number, n_p, of randomly distributed points. These points were converted to simulated clinometer raw data with a noise distributed as $N(0,\sigma_\varepsilon{}^2)$. The calibration routine was performed, with the ratio of errors set to *r*. Finally, another dataset of 100 points was generated, using the same parameters, in order to check the performance of the clinometer. This whole process was repeated 100 times and the overall standard error was calculated, as well as the maximum standard error over all 100 runs, and the maximum error of any point in the whole process. These values of the variances for the various quantities were taken from the ADXL202 data sheet or from practical considerations.

By some trial and error and common sense, it was determined that suitable values for the parameters might be:

$$n_l = 12$$
$$n_p = 8$$
$$\sigma_\varepsilon = 0.001g$$
$$\sigma_\delta = 0.5°$$
$$r = 0.001$$

Figure 3.2 shows the effect of varying n_l and n_p from the above values, showing that little improvement results from increasing these any further than 12 and 8 respectively. In addition, the total number of points is 20, which is few enough for calibration to remain a relatively speedy process, from the point of view of taking the measurements.

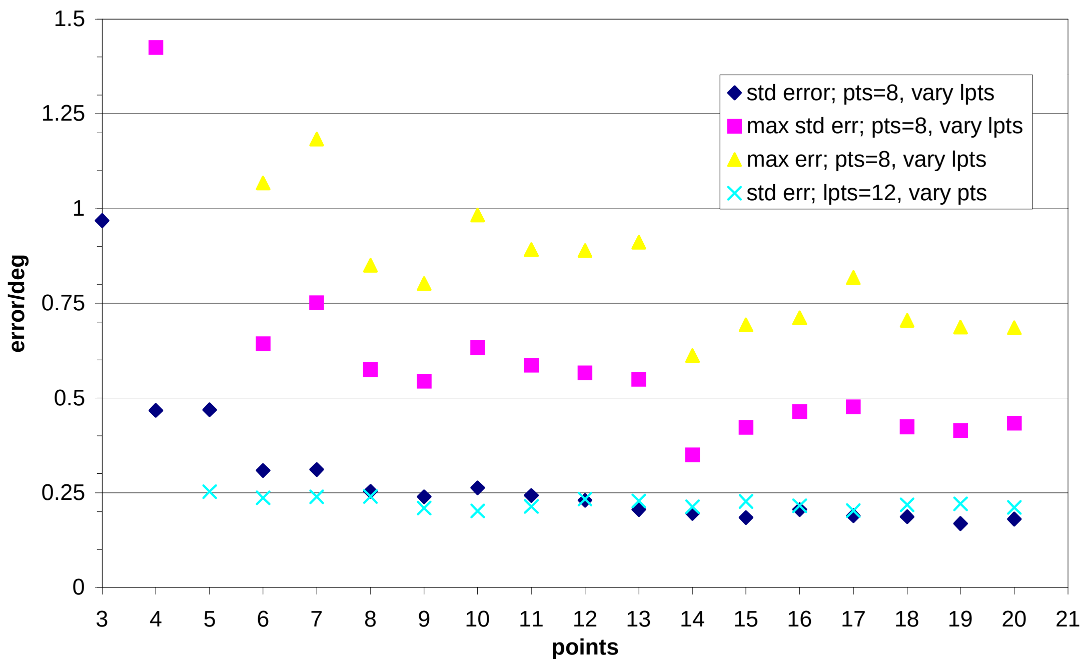

Figure 3.2: The effect on the errors of varying the number of calibration points

The next parameter to investigate is the sensor noise, σ_ε. The first two plots in Fig. 3.3 show there is not much point in reducing this to less than around $0.001g$. A noise of $0.001g$ is achievable for calibration purposes, but for taking measurements in the field it would be desirable to be able to use a larger value of the noise, since attaining a $0.001g$ noise means using a bandwidth of 1 Hz. There is no problem with using such a small bandwidth for measurements on the surface, where the clinometer can be placed on something solid, but holding it still by hand underground for 1 s would be difficult. Therefore, a new parameter was introduced, σ'_ε, which was used in place of σ_ε, for the sensor noise on the set of 100 points used for evaluating the performance of the clinometer. The second two plots of Fig. 3.3 show the results of using this new "quiet calibration" method, fixing $\sigma_\varepsilon = 0.001g$ and varying σ'_ε. From the graph, a value of $0.0045g$ for σ'_ε seems appropriate, as this is high enough to allow a measurement to be taken in 0.05 s.

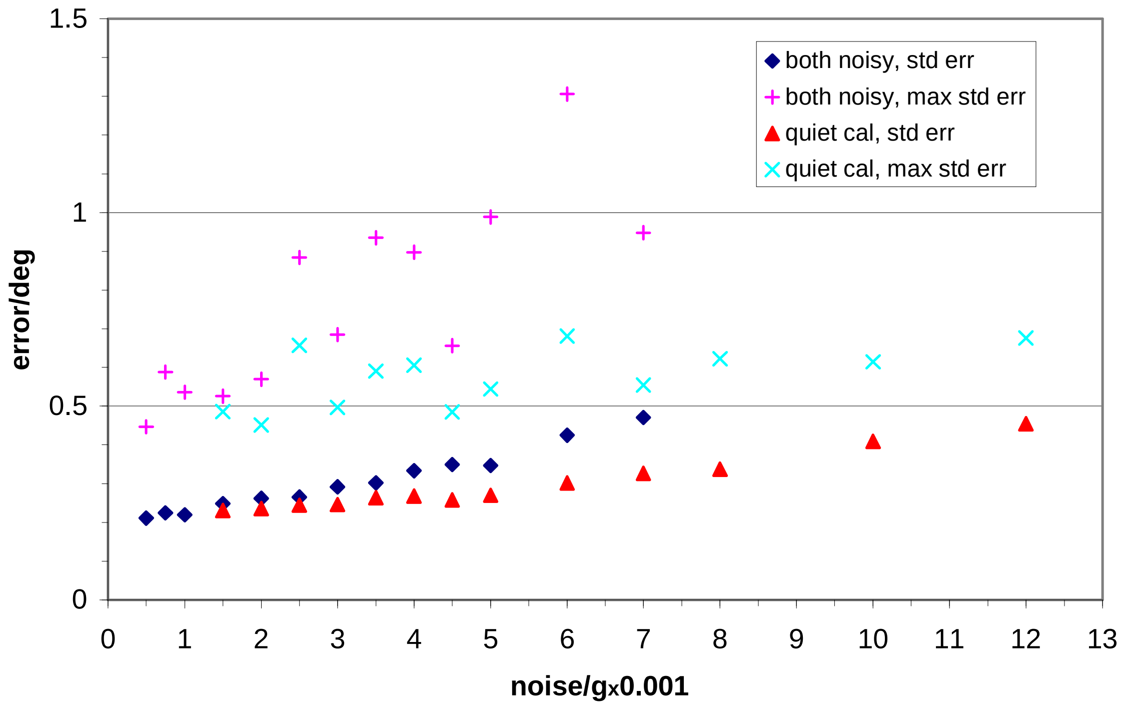

Figure 3.3: The effect on the errors of varying the sensor noise

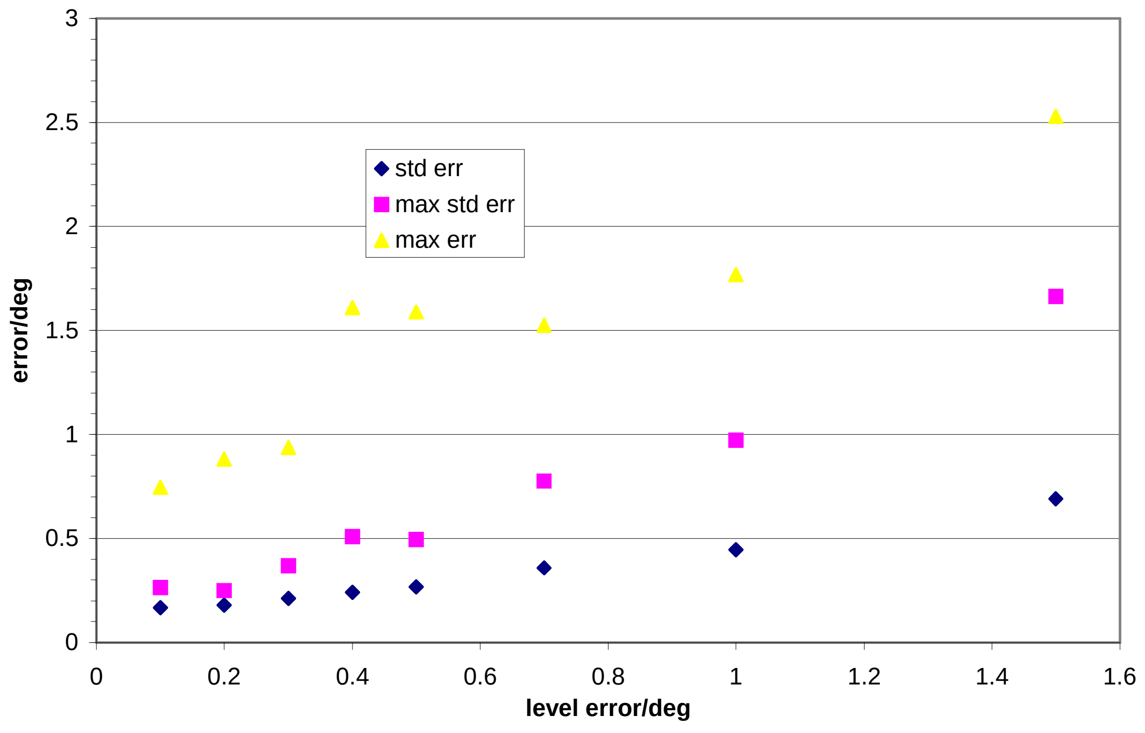

Figure 3.4: The effect on the errors of varying the levelling error

Figure 3.4 shows the effect of varying the levelling error, σ_δ. From the graph it seems there is little to be gained by improving the levelling error beyond the easily attainable value of 0.5°.

Finally, it is necessary to investigate the effect of varying the ratio of errors, *r*. The theory of Sec. 3.2 states that this should be equal to $\sigma_\varepsilon/\sigma_\delta$ (measuring σ_δ in radians because of the way the software uses radians internally). For the above values of the errors, we would therefore expect the best results for $r = 0.11$. Figure 3.5 shows the effects of varying *r* over a wide range. It seems that there is indeed a shallow minimum near this value of *r*. However, the performance is much worse for *r* any larger than this value and only very slightly worse for *r* any lower than this. Therefore, it was decided to use $r = 0.001$ in order to stay well within the region of good performance.

One potential problem with the clinometer is that although the scale factors and offsets can be calibrated out on the surface, these parameters will change with temperature. In order to investigate this effect, between running the calibration routine and generating the 100 points for performance evaluation, the scale factors were changed by an amount distributed as $N\left(0,(0.015\Delta T)^2\right)$ percent and the offsets changed by an amount distributed as $N\left(0,(2\Delta T)^2\right)$ in units of *g*/1000. The results of doing this for different values of Δ*T*, the difference between the temperature during calibration and the temperature when measurements are being taken, in °C, is shown in Fig. 3.6.

From Fig. 3.6 it is clear that in order to achieve the 0.33° standard error required for a grade 5 survey, the temperature of the clinometer cannot vary by more than 4°C. This would not be possible in practice since the ambient temperature within caves can be as low as 0°C, but the surveyor handling the instrument will be at around 37°C.

One possible way to solve the problem of temperature dependence would be to allow the clinometer to recalibrate itself in the field. This would be possible if the clinometer contained a relatively intelligent microprocessor, which would in any case be necessary in order to interface with the ADXL202s, convert the raw data into angles, handle user interaction and so on. Therefore, after simulating the temperature-induced change in the scale factors and offsets, a further set of n_r points was generated, with sensor noise of σ_r.

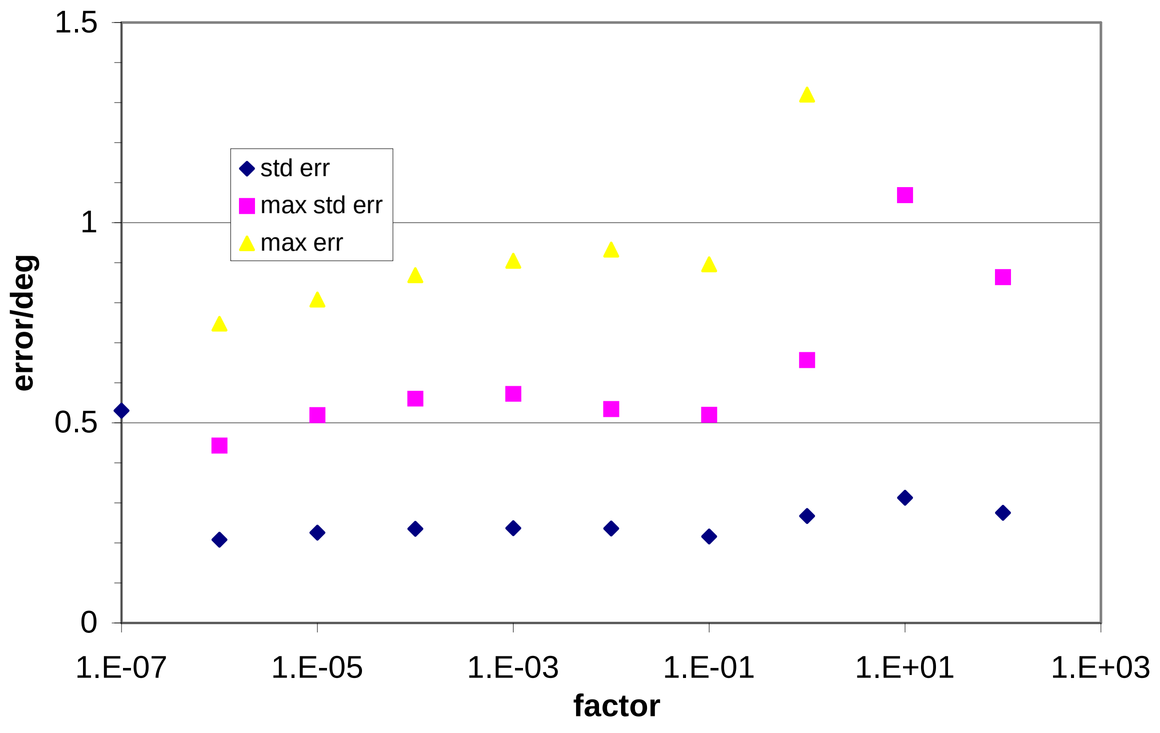

Figure 3.5: The effect on the errors of varying the "ratio of errors"

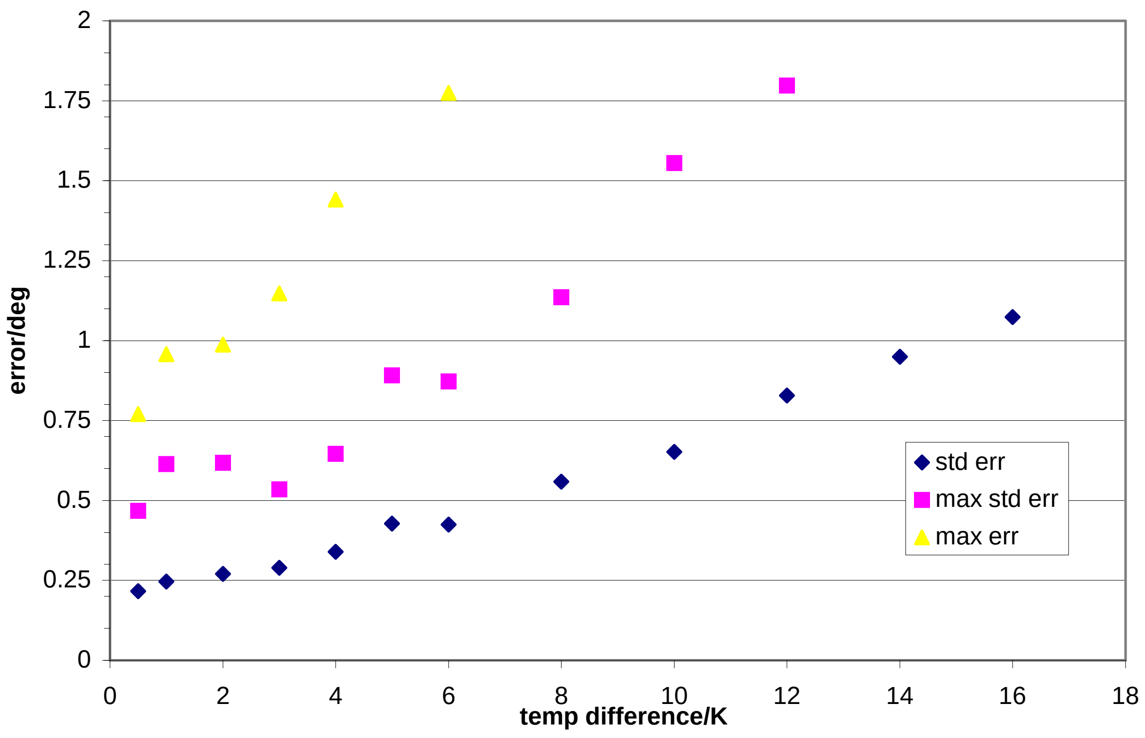

Figure 3.6: The effect on the errors of varying the temperature

The first two plots in Fig. 3.7 show the effect of varying n_r with σ_r held at $0.0045g$. The second two plots show the same thing, but performing all the calculations using single- as opposed to double-precision arithmetic. The reason for doing this is that single-precision numbers are quicker to work with and require less storage space, an important consideration since it is intended that this calculation will be performed on a small microcontroller. The final plot on Fig. 3.7 shows the effect of reducing σ_r to just $0.001g$, which would necessitate resting the clinometer on something immobile, rather than holding it by hand.

The plots show no significant difference between the single- and double-precision versions of the calculations and suggest that setting $\sigma_r = 0.0045\,g$ and $n_r = 8$ is sufficient to obtain acceptable results. The **MINPACK** documentation (see Appendix C) explains that the single-precision version of `lmder1()` will require 976 single-precision storage locations if $n_r = 8$. Assuming a single-precision number requires 4 bytes of storage, then the RAM required for the self-calibration will be of the order of 4kbytes, which is reasonable for a low-cost microcontroller.

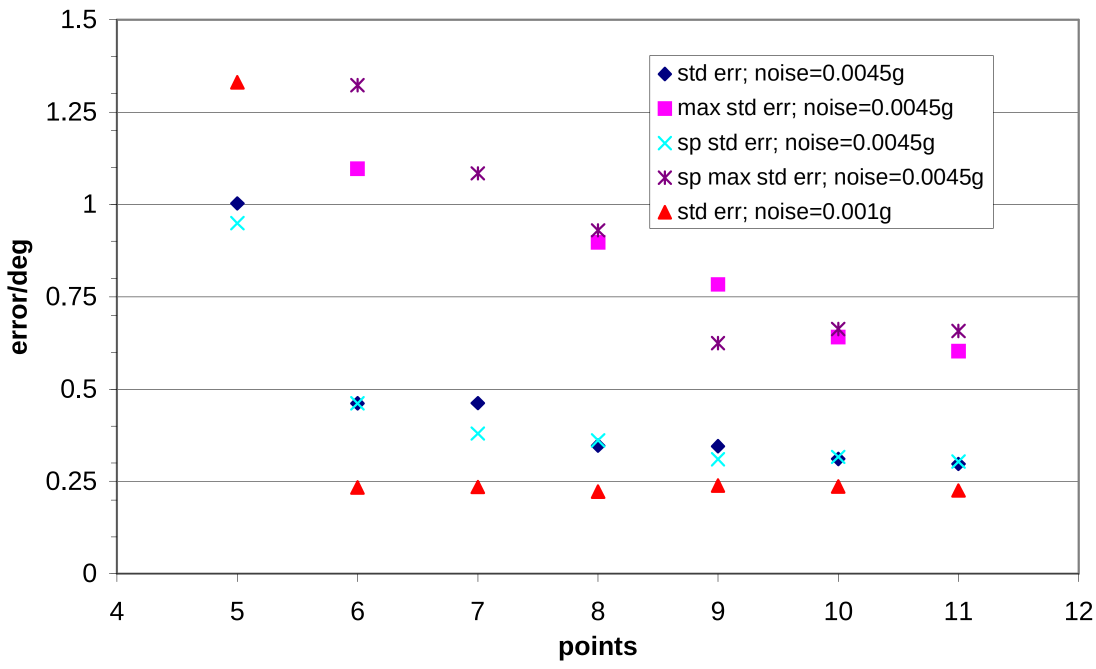

Figure 3.7: The effect on the errors of varying the number of points and the sensor noise during self-recalibration

Some sample output from the software suite is included in Appendix B.1 for the 22nd run for the double-precision version of the routine with $n_r = 8$ and $\sigma_r = 0.0045g$. Some more program output is in Appendix B.2, for which the set of points was modified by hand in order to produce several vertical or near-vertical legs, to check the performance of the software in this situation. From these data it is clear that the algorithms cope easily with this situation.

Obviously, the clinometer will need to be held still in order for the data produced by it to represent a meaningful representation of the inclination. If the device is accelerated as the measurement is taken then the magnitude of the acceleration it experiences will be other than $1g$. If this is the case then it should be possible for the clinometer to recognise this and display an error condition, rather than an inaccurate angle. In order to investigate this the data in Appendix B.3 were produced, where the raw data were scaled by hand by a factor of 1.05, corresponding to an acceleration of $0.05g$ in an upwards direction or an acceleration of $0.3g$ in a horizontal direction. A $0.3g$ acceleration in a horizontal direction would produce an angular error of up to 19°. The output in Appendix B.3.2 shows the sum of squares, q, the number that is minimised by the Levenberg-Marquardt algorithm, as a quality-of-fit indicator. The data show that q is larger by around a factor of 10 for the scaled data compared with the unscaled data, showing that this approach could indeed be used to eliminate gross errors caused by accidental movement of the clinometer during the measurement process.

Overall, the software suite performed very well indeed, solving all the problems required of it and even displaying sufficient flexibility to answer a number of questions that were not conceived at the time of its programming. It was also adequately fast. For example, a set of 100 runs using 100 points (enough to produce a single data point in Fig. 3.7) took around 80 seconds on a modestly powerful desktop computer (AMD K6-II 350MHz). A possible criticism of the software is its tendency to produce a large number of files, around 30000 for a typical run. However, this did not prove to be a problem and retaining all this data was useful in understanding apparent anomalies.

In the next section we take a more practical approach to clinometer design, and describe an attempt to create hardware implementation of such a device.

4 Hardware

4.1 Overview

The ADXL202 produces a Duty Cycle Modulated (DCM) output, which is ideal for easy interfacing to digital devices, such as microprocessors. If a clinometer were constructed for caving use based on the ADXL202, then it would almost certainly include a small single-chip microcontroller which would handle taking measurements from the accelerometers, converting the raw data into an angle (taking into account calibration constants determined by techniques described in Sec. 3), and storing the data for later download to a PC. However, designing such a system seemed rather ambitious for a project such as this one, so instead a rather simpler circuit was designed, to allow gathering of data in a laboratory context. It was hoped that this would prove the concept of using the ADXL202 in a clinometer for cave surveying.

4.2 Circuit Design

One of the DCM outputs of an ADXL202 is shown diagrammatically in Fig. 4.1. The relevant quantity is the duty cycle, T_1/T_2 a dimensionless number in the range (0,1) which varies linearly with measured acceleration. Typically a change of 0.125 in T_1/T_2 corresponds to an acceleration of 1 g, so in order to measure acceleration accurate to $0.001\,g$, it is necessary to measure T_1 and T_2 accurate to around one part in 10000. The largest value of T_2 achievable is 10 ms. Therefore, T_1 and T_2 must be measured accurate to around 1 µs.

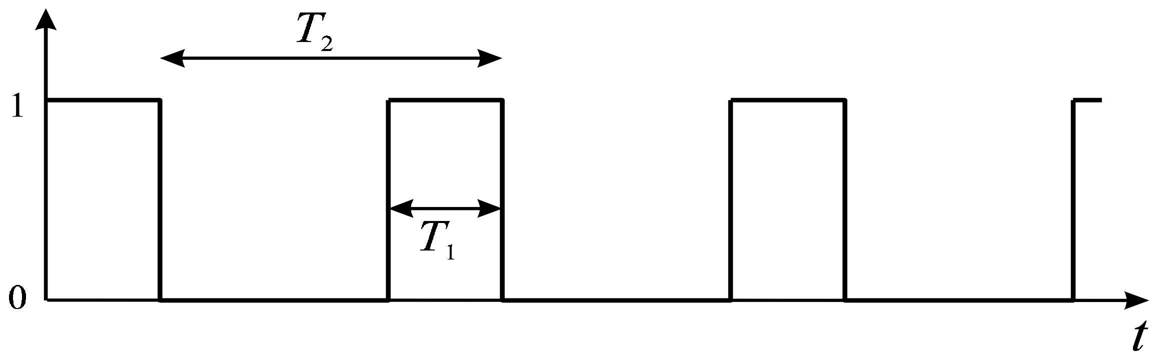

Figure 4.1: Duty-cycle modulation

One way to measure T_1 and T_2 is to use the ADXL202 output to gate a square-wave oscillator with a frequency of around 1 MHz (giving a period of 1 µs). Denoting the

ADXL202 output as *D* and the oscillator output as *E*, we can form the Boolean combinations $A = DE$ and $B = \overline{D}E$. Counting the number of rising edges of *A* and *B* over a single period allows determination of the duty cycle, which will be given by

$$\frac{C_A}{C_A + C_B} \quad ,$$

where C_A and C_B denote the counts for *A* and *B* respectively. If the measurement occurs over an integer number of periods, *n*, then the expression will be modified to

$$\frac{C_A}{C_A + C_B} = \frac{nC_A^1}{nC_A^1 + nC_B^1} = \frac{C_A^1}{C_A^1 + C_B^1} \quad ,$$

where C_A^1 denotes the count for *A* over a single period. If instead we have a non-integer number of periods, $n' = n + \varepsilon, 0 \le \varepsilon \le 1$, as will occur if we start and stop the measurement by hand, then we have

$$\frac{C_A}{C_A + C_B} = \frac{n + \delta}{n'} \frac{C_A^1}{C_A^1 + C_B^1}, \quad 0 \le \delta \le 1 \quad ,$$

which is no longer equal to the duty cycle, but is no more different to it than $1/n'$. Therefore, in order to measure the duty cycle to one part in 10000, it is necessary to measure for 10000 periods, or around 100 s, with a period of 10 ms.

Figure 4.2 shows a circuit to produce *A* and *B* outputs for the 4 sensors. In this circuit IC1–3 are 74AC00s, chosen because they can operate with a 5 V supply at 1 MHz and are able to drive a 50 Ω transmission line directly, for interface to counters and oscilloscopes. The power supply bypass capacitors are the value recommended in the datasheet. (All semiconductor datasheets are reproduced in Appendix D). IC4 and IC5 are the ADXL202s and the component values are taken directly from the datasheet. The ‘In’ connection is for connection to the external oscillator, and the discrete components ensure that the transmission line is correctly terminated and that the voltages applied to the inputs of IC1–2 are not outside the power supply range. The switch allows starting and stopping of the measurement period.

The ADXL202 datasheet states that the 0 *g* offset of each sensor varies with temperature at a rate of 0.002 *g*/°C. Therefore, in order to make measurements of the acceleration accurate to 0.001 *g*, it is necessary to regulate the temperature of the ADXL202 to better than 0.5 °C**.** This could be done by measuring the ADXL202 temperature and applying a calibration based on this, an approach that would be appropriate for a finished product. Another approach is to keep the ADXL202s in temperature-controlled ovens, maintaining a constant temperature for each. This would probably be inappropriate for a finished product, which would need to minimise energy usage due to only small reserves being available in a battery, but is ideal for this project, where no such restriction applies.

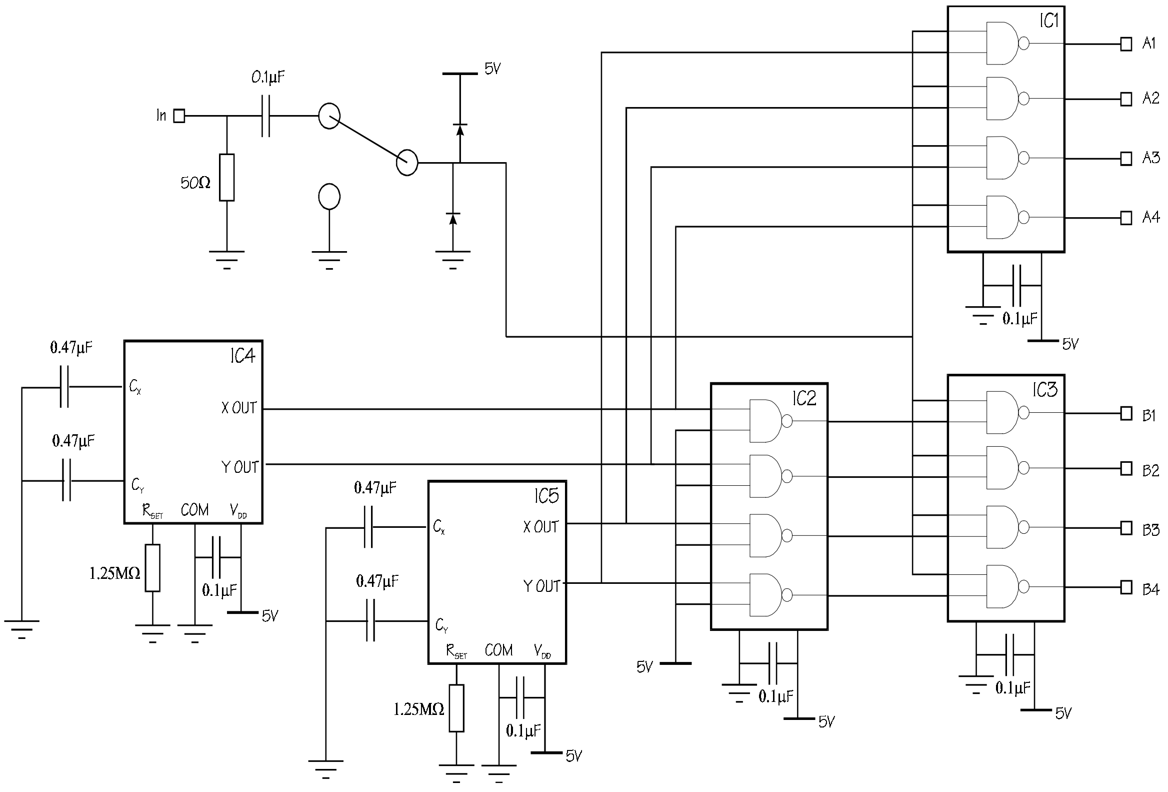

Figure 4.2: Digital circuit diagram

Expanded polystyrene typically has a thermal conductivity of around $0.04\,\mathrm{Wm^{-1}K^{-1}}$, so enclosing the ADXL202 in 2 cm of polystyrene insulation would mean that around 20 mW of power would need to be supplied in order to maintain a 20°C temperature difference from the environment.

Figure 4.3 shows the circuit that was used to maintain the ADXL202s at constant temperature. It is a negative-feedback circuit in which IC6 is a LT1013 dual op-amp, which was chosen because of its ability to operate from a single-ended 5V supply. IC7 and IC8 are AD22100 temperature sensors, which produce an output voltage directly proportional to the supply voltage and to the temperature. H1 and H2 are resistors, used in this context as heaters. The transistors allow more current to be passed through the heaters than IC6 can supply. The values of the resistors in the potential divider make the set temperature a nominal 40°C.

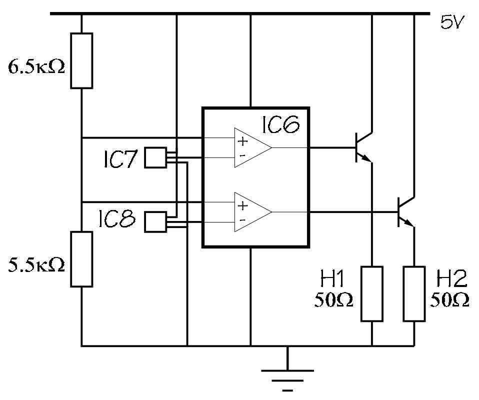

Figure 4.3: Temperature control circuit diagram

4.3 Construction

The ADXL202s are only available in QC-14 packages, intended for surface mounting, so it was necessary to design and etch PCBs for them. This was complicated by the need to mount them at 45° to the clinometer axis and at right angles to each other. The final arrangement involved etching two small square PCBs with all the tracks at 45° to the sides of the square. The PCB layout is shown in Fig. 4.4. Each PCB was then mounted on a length of aluminium angle-iron and a lecture-theatre laser-pointer was clamped into the groove of the angle-iron and was used for sighting the instrument. The rest of the circuit involved many cross-connections and would have required at least a double-sided PCB, so instead it was constructed on a piece of stripboard using a mixture of soldering and wire-wrap techniques. The external connections were brought out to a row of 4 mm sockets on another piece of aluminium angle iron.

Each ADXL202 and the corresponding heating resistor and temperature sensor were kept in physical and thermal contact by a thermally conductive glue (datasheet is in Appendix D). This assembly was then enclosed in a piece of expanded polystyrene cut from a piece originally used for packaging, and glued to the PCB using silicone rubber sealant, which was chosen because unlike other adhesives it contained no solvents which would dissolve the polystyrene.

Photos of the completed circuit are in Figs. 4.5 and 4.6.

The 1 MHz oscillation was supplied from a very large RF oscillator made by Marconi Instruments, chosen because it happened to be in the laboratory. This took several minutes to warm up, but was subsequently more than sufficiently stable.

The circuit outputs were fed into Topward universal counters, model 1212. These were able to count at sufficiently high frequency and overflowed every 10^9 counts, allowing several minutes' averaging to take place for each measurement.

For the level surface that is required by the calibration procedure, a 930 mm long Perspex trough (originally used for a wave propagation experiment) was half-filled with water. Once a suitable arrangement of clamps had been constructed for the clinometer, it proved straightforward to sight across the surface of the water, accurate to about 5 mm, or about 0.3°.

4.4 Testing and Debugging

The temperature-control loop was tested by monitoring the output from the temperature sensors IC7 and IC8. The high gain of the feedback loop resulted in the current to the heaters mostly being at the maximum and minimum and rarely at an intermediate value. This behaviour led to small oscillations in the temperature as measured at the sensors, with a period of around 1 minute. However, the amplitude of these oscillations was small, of the order of 0.2 °C, and this was deemed acceptable. The drift of the temperature with time was smaller than these oscillations, even when switching off the circuit overnight and starting it up again the next morning. Therefore, the temperature-control circuit worked as well as was required.

More problems were apparent with the digital part of the circuit. Some of these were traced to bad termination of faulty 50 Ω coaxial cables, but strange behaviour was still observed. The circuit would seemingly work perfectly for some minutes and then some of the digital gates seemed to stop working altogether. Eventually, the problem was traced to the power supply, an aged Farnell unit. Although it produced a very stable and noise-free output in general, this would put out large voltage spikes, of the order of 30 V, at switch-on. These were very successful at destroying the sensitive 74AC00 chips. Figure 4.7 shows an oscillogram of such a spike. Diagnosis of this fault was further complicated by the fact that the power supply only exhibited this behaviour when connected to a relatively low impedance load. Dealing with this problem occupied quite a large part of the project time, both in diagnosis and because new ICs had to be obtained.

With new ICs and a more appropriate power supply the circuit seemed to be working much better. Even turning the apparatus off overnight and back on the next morning, the agreement between two sets of measurements suggested a measurement noise of 0.001 – 0.002 g, which should have been sufficiently small to allow use of the software described in Sec. 3.3. With this in mind, 12 sets of measurements were obtained with the clinometer levelled, and a further 8 with it in arbitrary orientations (a process which took 2 days). As a consistency check at the end of this, the last set of measurements was repeated and, unfortunately, it was discovered that while the sets of measurements from the sensors in IC4 agreed to the same accuracy as before, those from the sensors in IC5 only agreed to within around 0.1g! Much effort was expended in trying to discover the cause of this, but this was unsuccessful. It is possible that repeated exposure to the 30 V spikes produced by the old power supply finally caused damage to the ADXL202 itself, although no anomalies were visible on an oscillogram of the output from IC5.

4.5 Results and Analysis

The problems described in Sec. 4.4 meant that the data that was collected were of very dubious quality. In an effort to salvage the situation, the data were fed to the software with a larger and larger set of observations deleted, to try to include only measurements that were taken before the problems began. This approach did not prove successful, and throwing away no particular subset of the data produced a significant improvement in the quality of the fit. Therefore, the main conclusion from this part of the investigation is that further research is needed to verify that the behaviour of the clinometer is consistent with the software model constructed in Sec. 3.3.

The problems with the hardware were made much more difficult to diagnose by the fact that getting one piece of data took around 2 minutes, and taking a full set of 4 around 10 minutes. To see if each circuit modification improved the performance therefore required at least 20 minutes. It could be argued that perhaps it would have been better to design a microprocessor-based system in the first place, as then a set of 4 measurements could be completed in less than a second. Another advantage of doing this would be that many subsequent changes could be achieved with software-only modifications, which should be significantly easier than making hardware modifications.

5 Conclusions and Summary

The original aim of this project was to design and test a digital clinometer for caving use, which could hopefully be used as a component part of an eventual Total Survey Station instrument. The simulations performed in Sec. 3 suggest construction of this type of clinometer ought to be feasible, and suggest possible values for instrument parameters. The Levenberg-Marquardt algorithm, and in particular the implementation in **MINPACK** was shown to be ideally suited to the task of calibrating a clinometer, and a flexible software suite was built around this package.

It was hoped that the construction of the prototype clinometer described in Sec. 4 would further prove the concept of a digital clinometer in general, and the specific design developed in this document in particular. Unfortunately, problems with the hardware, described in Sec. 4.4, prevented any meaningful results from being obtained. However, useful lessons were learnt for the future: if further research is to be performed then it would certainly be wise to consider seriously a microcontroller-based circuit, as opposed to a less sophisticated design implemented in discrete logic. Not only would this accelerate the process of taking measurements by several orders of magnitude, the resulting circuit would almost certainly be more compact, and much closer to a realistic design for a finished product. It may also simplify the task of modifying the design, since many modifications could be implemented in software.

Overall, the amount of time spent on the various parts of the problem can be summarised roughly as follows:

- 5 days researching cave surveying in general, the results of which are in Sec.1;
- 10 days deriving the mathematical description of a clinometer in Sec. 2 and making a preliminary investigation into the calibration of clinometers;
- 10 days searching the available literature for the numerical methods of Secs. 3.1 and 3.2.
- 18 days working on the software suite of Sec. 3.3, of which around 3 were used for generating the data for the graphs in Sec. 3.4;
- 5 days designing the circuits in Sec. 4.2; and

- 20 days in the laboratory, of which 8 were spent constructing the clinometer hardware, 2 were spent gathering data and the remaining 10 were used in attempts to solve the problems described in Sec. 4.4, either taking measurements or in waiting for new ICs to be obtained.

Looking towards the future, the obvious next step on the road towards a TSS is the addition of some magnetic field sensors to the clinometer. This would produce a combined compass and clinometer, a device that has the potential to make a significant impact on the way caves are surveyed. The calibration techniques of Sec. 3 would remain appropriate for this new device, and the mathematical models of Sec. 2 should only need minor modifications, in order to incorporate any peculiarities of the magnetic field sensors used and to include a third angular variable in the description of the orientation of the instrument.

APPENDICES

A PROGRAM FILES 42
A.1 C PROGRAM FILES 42
A.2 C HEADER FILES 66
A.3 SCRIPT FILES 68
B SAMPLE OUTPUT 71
B.1 RUN 22 OF NOISY SELF RECALIBRATION WITH 8 POINTS 71
B.2 VERTICAL LEGS 75
B.3 SCALING 75
C LMDER1 DOCUMENTATION 77
D DATA SHEETS 82

A Program Files

A.1 C Program Files

A.1.1 add.c

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

int main(int argc,char**argv)
{
      char tmp[256];
      int n;
      FILE*off1,*off2;
      double o1,o2,s1,s2;
      if(argc!=3)
      {
            fprintf(stderr,"Usage: add <off_file1> <off_file2>\n\n");
            exit(1);
      }
      if(!(off1=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      if(!(off2=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      fscanf(off1,"%[^\n]",tmp);printf("( %s ) : ",tmp);
      fscanf(off2,"%[^\n]",tmp);printf("( %s )\n",tmp);
      for(n=0;n<4;n++)
      {
            fscanf(off1,"%lg %lg",&o1,&s1);
            fscanf(off2,"%lg %lg",&o2,&s2);
            printf("% .5f\t% .5f\n",o1+o2,s1+s2);
      }
}
```

A.1.2 aln.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

double error(double);

int main(int argc,char**argv)
{
      double aln,aln_xy;
      srand(time(0)+getpid());
      if(argc!=3)
      {
            fprintf(stderr,
                  "Usage: aln <aln> <aln_xy>\n\tvalues in degrees\n\n");
            exit(1);
      }
      sscanf(argv[1],"%lg",&aln);
      sscanf(argv[2],"%lg",&aln_xy);
      printf("aln=%g aln_xy=%g\n",aln,aln_xy);
      printf("%.4f %.4f\n%.4f %.4f %.4f\n\t%.4f %.4f\n",
            error(aln_xy),error(aln_xy),error(aln),error(aln),error(aln),
            error(aln),error(aln));
      return 0;
}
```

A.1.3 analyse.c

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

int main(void)
{
      int n=0;
      double m,s,mxs=0,max=0,sum=0;
      while(1)
      {
            scanf(" %*[^M]Max error: %lf\nStd. error:%lf",&m,&s);
            if(feof(stdin))break;
            if(m>max)max=m;
            if(s>mxs)mxs=s;
            sum+=s*s;
            n++;
      }
      printf("\n%d runs\nMax error: %f\nMax std error: %f\n"
             "Overall std error:%f\n\n",n,max,mxs,sqrt(sum/n));
      return 0;
}
```

A.1.4 cal.c

```
/* Possible defines are:
   VERBOSE - gives listing of each data point before and after lmder
   CHECKJAC - runs chckder to check the jacobian
   NUMDIFF - determines he jacobian numerically rather than analytically
*/

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

void mtrx(double,double,double,double,double,double,double,double(*)[3]);
void d_da(double,double,double,double,double,double,double,double(*)[3]);
void d_db(double,double,double,double,double,double,double,double(*)[3]);
void d_ax(double,double,double,double,double,double,double,double(*)[3]);
void d_ay(double,double,double,double,double,double,double,double(*)[3]);
void d_az(double,double,double,double,double,double,double,double(*)[3]);
void d_by(double,double,double,double,double,double,double,double(*)[3]);
void d_bz(double,double,double,double,double,double,double,double(*)[3]);
int fcn(int*,int*,double*,double*,double*,int*,int*);
int fcn2(int*,int*,double*,double*,int*);
int lmder1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*fjac,
          int*ldfjac,double*tol,int*info,int*ipvt,double*wa,int*lwa);
```

```
int
lmdif1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*tol,int*info,
          int*iwa,double*wa,int*lwa);
int chkder_(int*m,int*n,double*x,double*fvec,double*fjac,int*ldfjac,
          double*xp,double*fvecp,int*mode,double*err);
double d_lg10(double*);
enum parameters{AX=0,AY,AZ,BY,BZ,S1,S2,S3,S4,D1,D2,D3,D4,ANGLES};
enum angles{THETA=0,PHI};

double pi,*data=0,ratio=5;
int nump,numl;
double d_lg10(double*a)
{
      return log10(*a);
}

int main(int argc,char**argv)
{
      int known,numrl=0,numrp=0,n,a;
      FILE*ldt,*dat,*off,*aln,*pts,*lpts;
      double*X,*FVEC,*FJAC,*WA,TOL,l1,l2,a1,a2,hz,hy,v,*d,real_x[ANGLES],
            *real_ang=0,max,s,sum,sumsq;
      int N,M,LWA,LDFJAC,INFO,*IPVT,*IWA;
      char tmp[256];
#ifdef CHECKJAC
      double *ERR,*XP,*FVECP;
      int MODE;
#endif /* CHECKJAC */
      pi=4*atan(1);
      if(argc!=4&&argc!=8)
      {
            fprintf(stderr,"Usage: cal <dat_file> <ldt_file> [<aln_file> "
                   "<off_file> <pts_file> <lpts_file>] ratio\n\n");
            exit(1);
      }
      sscanf(argv[argc-1],"%lf",&ratio);
      printf("ratio %f : ",ratio);
      if(!(dat=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      if(!(ldt=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      fscanf(dat,"%[^\n]",tmp);printf("%s\n",tmp);
      fscanf(ldt,"%[^\n]",tmp);printf("%s\nXXX\n",tmp);
      if((known=(argc==8)))
      {
            if(!(aln=fopen(argv[3],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[3]);exit(4);}
            if(!(off=fopen(argv[4],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[4]);exit(5);}
            if(!(pts=fopen(argv[5],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[5]);exit(6);}
            if(!(lpts=fopen(argv[6],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[6]);exit(7);}
            fscanf(aln,"%*[^\n] %*g %*g");
            fscanf(off,"%*[^\n]");
            fscanf(pts,"%*[^\n]");
            fscanf(lpts,"%*[^\n]");
            for(n=0;n<S1;n++)fscanf(aln,"%lg",real_x+n);
            for(n=0;n<4;n++)fscanf(off,"%lg %lg",real_x+D1+n,real_x+S1+n);
      }
      while(!feof(ldt))
      {
            if(!(data=realloc(data,sizeof(double)*4*++numl)))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(8);
            }
            for(n=0;n<4;n++)fscanf(ldt,"%lg ",data+4*numl+n-4);
      }
      while(!feof(dat))
      {
            if(!(data=realloc(data,sizeof(double)*4*(numl+ ++nump))))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(9);
            }
            for(n=0;n<4;n++)fscanf(dat,"%lg ",data+4*(numl+nump-1)+n);
      }
      if(known)
      {
            if(!(real_ang=malloc(sizeof(double)*2*(numl+nump))))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(10);
            }
            do for(n=0;n<2;n++)fscanf(lpts,"%lg ",real_ang+2*numrl+n);
            while(!feof(lpts)&&++numrl<=numl);
            do for(n=0;n<2;n++)
                  fscanf(pts,"%lg ",real_ang+2*(numrl+numrp+1)+n);
            while(!feof(pts)&&++numrp<=nump);
            if(nump!=numrp+1||numl!=numrl+1)
            {
                  fprintf(stderr,
                        "Different number of data points and angles\n");
                  exit(12);
            }
      }
      M=4*nump+5*numl;
      N=ANGLES+2*(nump+numl);
      LWA=5*N+M
#ifdef NUMDIFF
            +M*N; // last term only for lmdif1
#endif
      ;
      LDFJAC=M;
      if(!(X=malloc(sizeof(double)*N))||
         !(FVEC=malloc(sizeof(double)*M))||
         !(FJAC=malloc(sizeof(double)*LDFJAC*N))||
         !(WA=malloc(sizeof(double)*LWA))||
         !(IPVT=malloc(sizeof(int)*N))||
         !(IWA=malloc(sizeof(int)*N))
#ifdef CHECKJAC
         ||!(XP=malloc(sizeof(double)*N))||
         !(FVECP=malloc(sizeof(double)*M))||
         !(ERR=malloc(sizeof(double)*M))
#endif /* CHECKJAC */
            )
      {
            fprintf(stderr,"Out of memory allocating arrays!");
            exit(13);
      }
      TOL=0.00000;
      for(n=0;n<=D4;n++)X[n]=0;
      for(n=S1;n<=S4;n++)X[n]=1;
      for(n=0;n<numl+nump;n++)
      {
            d=data+4*n;
            l1=sqrt(d[0]*d[0]+d[1]*d[1]);
            l2=sqrt(d[2]*d[2]+d[3]*d[3]);
            a1=atan2(d[1],d[0]);
            a2=atan2(d[3],d[2]);
            hz=l1*sin(pi/4-a1);
            hy=l2*sin(pi/4-a2);
            v=(l1*cos(pi/4-a1)-l2*cos(pi/4-a2))/2;
            X[ANGLES+2*n+PHI]=atan2(hz,hy);
            if(X[ANGLES+2*n+PHI]<0)X[ANGLES+2*n+PHI]+=2*pi;
            X[ANGLES+2*n+THETA]=atan2(sqrt(hy*hy+hz*hz),v);
#ifdef VERBOSE
            fprintf(stderr,"\n%3f %3f",X[ANGLES+2*n+THETA]/pi*180,
                   X[ANGLES+2*n+PHI]/pi*180);
            if(known)fprintf(stderr," - %3f %3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
#endif /* VERBOSE */
      }
#ifndef CHECKJAC
#ifndef NUMDIFF
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
#else /* NUMDIFF */
      lmdif1_(fcn2,&M,&N,X,FVEC,&TOL,&INFO,IWA,WA,&LWA);
#endif /* NUMDIFF */
#else  /* CHECKJAC */
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
      MODE=1;
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      fcn(&M,&N,XP,FVECP,FJAC,&LDFJAC,&MODE);
      MODE=2;
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      for(n=0;n<M;n++){
            printf("%d %f\n",n,ERR[n]);
//          for(a=0;a<N;a++)printf("%.2f ",FJAC[n+a*LDFJAC]);printf("\n");
      }
      exit(0);
#endif /* CHECKJAC */
      fprintf(stderr,"\nlmder1 exit code: %d\n",INFO);
      for(n=0;n<numl
#ifdef VERBOSE
                +nump
#endif
                ;n++)
      {
            fprintf(stderr,"\n % 3f % 3f",
                   X[ANGLES+2*n+THETA]*180/pi,X[ANGLES+2*n+PHI]*180/pi);
            if(known)fprintf(stderr," -  % 3f % 3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
      }
#if 1
      fprintf(stderr,"\nax: % 3f\tay: % 3f\taz: % 3f\n",
             X[AX]*180/pi,X[AY]*180/pi,X[AZ]*180/pi);
      if(known)fprintf(stderr,"    % 3f\t    % 3f\t    % 3f\n",
                  real_x[AX],real_x[AY],real_x[AZ]);
      fprintf(stderr,"\t\tby: % 3f\tbz: %
3f\n",X[BY]/pi*180,X[BZ]/pi*180);
      if(known)fprintf(stderr,
                   "\t\t    % 3f\t    % 3f\n",real_x[BY],real_x[BZ]);
      for(n=0;n<4;n++){
            fprintf(stderr,"d%d: % 3f\ts%d: % 3f\n",
                  n,X[D1+n],n,(X[S1+n]-1)*100);
            if(known)fprintf(stderr,"    % 3f\t    % 3f\n",
                        real_x[D1+n],real_x[S1+n]);
      }
      if(known)
      {
            for(sum=sumsq=max=n=0;n<nump;n++) {
                  s=fabs(X[ANGLES+2*n+numl*2+THETA]/pi*180);
                  if(s>180)s=360-s;
                  s-=real_ang[2*(n+numl)+THETA];
                  sum+=s;
                  sumsq+=s*s;
                  if(fabs(s)>max)max=fabs(s);
            }
            fprintf(stderr,"Max error: %f\nStd. error:%f\n",
                   max,sqrt(sumsq/nump));
      }
#endif
      printf("% .5f\t% .5f\t% .5f\n\t\t% .5f\t% .5f\n",
             X[AX],X[AY],X[AZ],X[BY],X[BZ]);
      for(n=0;n<4;n++)printf("% .5f\t% .5f\n",X[D1+n],X[S1+n]);
      return 0;
}

int fcn2(m,n,x,fvec,iflag)
int*m,*n,*iflag;
double *x,*fvec;
{
      int if2=1;
      fcn(m,n,x,fvec,0,0,&if2);return 0;
}
int fcn(m,n,x,fvec,fjac,ldfjac,iflag)
int*m,*n;
double*x,*fvec,*fjac;
int*ldfjac,*iflag;
{
      int i,j,k,d;
      double mat[4][3],g[3],tmp,dmat[5][4][3],q=0;
      switch(*iflag)
      {
      case 1:
            mtrx(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],mat);
            for(i=0;i<nump+numl;i++)
            {
                  g[0]=cos(x[ANGLES+2*i+THETA]);
                  g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=x[D1+j];
                        for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                        tmp-=data[4*i+j];
                        fvec[4*i+j]=tmp;
                        q+=tmp*tmp;
                  }
            }
            for(i=0;i<numl;i++)fvec[4*(numl+nump)+i]=
                                    (x[ANGLES+2*i+THETA]-pi/2)*ratio;
//          for(i=0;i<*m;i++)printf("%d %f\n",i,fvec[i]);
//          printf("%f\n",q);
```

```
            return 0;

        case 2:
            memset(fjac,0,*ldfjac**n*sizeof(double));
            mtrx(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],mat);
            for(i=0;i<numl+nump;i++)
            {
                g[0]=-sin(x[ANGLES+2*i+THETA]);
                g[1]=cos(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                g[2]=cos(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                for(j=0;j<4;j++)
                {
                    tmp=0;
                    for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                    fjac[4*i+j+*ldfjac*(ANGLES+2*i+THETA)]=tmp;
                }
                g[0]=0;
                g[1]=sin(x[ANGLES+2*i+THETA])*-sin(x[ANGLES+2*i+PHI]);
                g[2]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                for(j=0;j<4;j++)
                {
                    tmp=0;
                    for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                    fjac[4*i+j+*ldfjac*(ANGLES+2*i+PHI)]=tmp;
                }
                g[0]=cos(x[ANGLES+2*i+THETA]);
                g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                d_ax(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],dmat[AX]);
                d_ay(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],dmat[AY]);
                d_az(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],dmat[AZ]);
                d_by(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],dmat[BY]);
                d_bz(0,0,x[AX],x[AY],x[AZ],x[BY],x[BZ],dmat[BZ]);
                for(d=AX;d<=BZ;d++)for(j=0;j<4;j++)
                {
                    tmp=0;
                    for(k=0;k<3;k++)
                        tmp+=x[S1+j]*dmat[d][j][k]*g[k];
                    fjac[4*i+j+*ldfjac*d]=tmp;
                }
                for(j=0;j<4;j++)
                {
                    tmp=0;
                    for(k=0;k<3;k++)tmp+=mat[j][k]*g[k];
                    fjac[4*i+j+*ldfjac*(S1+j)]=tmp;
                    fjac[4*i+j+*ldfjac*(D1+j)]=1;
                }
            }
            for(i=0;i<numl;i++)
                fjac[4*(numl+nump)+i+*ldfjac*(ANGLES+i*2+THETA)]=ratio;
            return 0;

        default:
            fprintf(stderr,"Iflag=%d\n",*iflag);
            exit(14);
        }
}

void mtrx(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
        double T[4][3]=
#include "matrix.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_da(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_da.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_db(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_db.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_ax(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_ax.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_ay(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_ay.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_az(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_az.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_by(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_by.h"
      ;
      memcpy(a,T,sizeof(T));
}
void d_bz(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "drv_bz.h"
      ;
      memcpy(a,T,sizeof(T));
}
```

A.1.5 dat.c

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <time.h>
#include <unistd.h>

double error(double);
void mtrx(double da,double db,double ax, double ay,double az,double by,
      double bz,double(*F)[3]);

int main(int argc,char**argv)
{
      double
noise,da,db,ax,ay,az,by,bz,offs[4],scl[4],M[4][3],th,ph,k,g[3],
            pi=4*atan(1);
      char tmp[256];
      int n,m;
      FILE*aln,*off,*pts;
      srand(time(0)+getpid());
      if(argc!=5)
      {
            fprintf(stderr,"Usage: dat <aln_file> <off_file> <pts_file>"
                  "<noise>\n\t<noise> in mg\n");
            exit(1);
      }
      if(!(aln=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      if(!(off=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      if(!(pts=fopen(argv[3],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[3]);exit(4);}
      sscanf(argv[4],"%lg",&noise);
      fscanf(aln,"%[^\n]",tmp);printf("( %s : ",tmp);
      fscanf(off,"%[^\n]",tmp);printf("%s : ",tmp);
      fscanf(pts,"%[^\n]",tmp);printf("%s ) noise %.3f\n",tmp,noise);
      fscanf(aln,"%lg %lg %lg %lg %lg %lg
%lg",&da,&db,&ax,&ay,&az,&by,&bz);
      for(n=0;n<4;n++)fscanf(off,"%lg %lg",offs+n,scl+n);
      mtrx(da/180*pi,db/180*pi,ax/180*pi,ay/180*pi,az/180*pi,
           by/180*pi,bz/180*pi,M);
      while(1)
      {
            fscanf(pts,"%lg %lg",&th,&ph);
            if(feof(pts))break;
            th/=180/pi;
            ph/=180/pi;
            g[0]=cos(th);
            g[1]=sin(th)*cos(ph);
            g[2]=sin(th)*sin(ph);
            for(n=0;n<4;n++)
            {
                  for(k=m=0;m<3;m++)k+=M[n][m]*(1+scl[n]/100)*g[m];
                  printf("% .5f ",k+offs[n]+error(noise/1000));
            }
            printf("\n");
      }
      return 0;
}
void mtrx(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
```

```
{
      double T[4][3]=
#include "matrix.h"
      ;
      memcpy(a,T,sizeof(T));
}
```

A.1.6 error.c

```
#include <stdlib.h>
#include <math.h>

double error(double std)  /* Polar method for normal deviates (Knuth 2
p117) */
{
        double v1,v2,s;
        do
        {
                v1=(double)rand()/RAND_MAX*2-1;
                v2=(double)rand()/RAND_MAX*2-1;
        }while((s=v1*v1+v2*v2)>=1);
        return(v1*sqrt(-2*log(s)/s)*std);
}
```

A.1.7 fnl.c

```
/* Possible defines are:
   VERBOSE - gives listing of each data point before and after lmder
   CHECKJAC - runs chckder to check the jacobian
   NUMDIFF - determines he jacobian numerically rather than analytically
*/

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

void mtrx(double,double,double,double,double,double,double,double(*)[3]);
int fcn(int*,int*,double*,double*,double*,int*,int*);
int fcn2(int*,int*,double*,double*,int*);
int lmder1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*fjac,
          int*ldfjac,double*tol,int*info,int*ipvt,double*wa,int*lwa);
int
lmdif1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*tol,int*info,
          int*iwa,double*wa,int*lwa);
int chkder_(int*m,int*n,double*x,double*fvec,double*fjac,int*ldfjac,
          double*xp,double*fvecp,int*mode,double*err);
double d_lg10(double*);
double pi,data[4],mat[4][3],scl[4],offset[4],q;
enum {THETA,PHI};
double d_lg10(double*a)
{
      return log10(*a);
}

int main(int argc,char**argv)
{
      int known,n,num=0;
      FILE*dat,*cal,*pts;
      int N=2,M=4,LWA=5*2+4
#ifdef NUMDIFF
            +2*4,IWA[2] // last term only for lmdif1
#endif
      ,LDFJAC=4,INFO,IPVT[2];
      double X[2],FVEC[4],FJAC[4*2],WA[LWA],TOL=0.001,l1,l2,a1,a2,hz,hy,v,
            real_ang[2],max=0,s,sum=0,sumsq=0,ax,ay,az,by,bz,d[4];
      char tmp[256];
#ifdef CHECKJAC
      double ERR[4],XP[2],FVECP[4];
      int MODE;
#endif /* CHECKJAC */
      pi=4*atan(1);
      if(argc!=3&&argc!=4)
      {
            fprintf(stderr,"Usage: fnl <dat_file> <cal_file>"
                  " [<pts_file>]\n\n");
            exit(1);
      }
      if(!(dat=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      fscanf(dat,"%[^\n]",tmp);
      if(!(cal=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      fscanf(cal,"%*[^\n] %*[^\n] %*[^\n] %lg %lg %lg %lg %lg",
             &ax,&ay,&az,&by,&bz);
      mtrx(0,0,ax,ay,az,by,bz,mat);
      for(n=0;n<4;n++)fscanf(cal,"%lg %lg",offset+n,scl+n);
      if((known=(argc==4)))
      {
            if(!(pts=fopen(argv[3],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[3]);exit(4);}
            fscanf(pts,"%*[^\n]");
      }
      while(1)
      {
            for(n=0;n<4;n++)
            {
                  fscanf(dat,"%lf",data+n);
                  d[n]=(data[n]-offset[n])/scl[n];
            }
            if(feof(dat))break;
            if(known)fscanf(pts,"%lf %lf",real_ang,real_ang+1);
            l1=sqrt(d[0]*d[0]+d[1]*d[1]);
            l2=sqrt(d[2]*d[2]+d[3]*d[3]);
            a1=atan2(d[1],d[0]);
            a2=atan2(d[3],d[2]);
            hz=l1*sin(pi/4-a1);
            hy=l2*sin(pi/4-a2);
            v=(l1*cos(pi/4-a1)-l2*cos(pi/4-a2))/2;
            X[PHI]=atan2(hz,hy);
            if(X[PHI]<0)X[PHI]+=2*pi;
            X[THETA]=atan2(sqrt(hy*hy+hz*hz),v);
#ifdef VERBOSE
            fprintf(stderr,"\n%3f %3f",X[THETA]/pi*180,
                  X[PHI]/pi*180);
            if(known)fprintf(stderr," - %3f %3f",
                          real_ang[THETA],real_ang[PHI]);
#endif /* VERBOSE */
#ifndef CHECKJAC
#ifndef NUMDIFF
```

```
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
#else /* NUMDIFF */
            lmdif1_(fcn2,&M,&N,X,FVEC,&TOL,&INFO,IWA,WA,&LWA);
#endif /* NUMDIFF */
#else  /* CHECKJAC */

      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
            MODE=1;
            chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
            fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
            fcn(&M,&N,XP,FVECP,FJAC,&LDFJAC,&MODE);
            MODE=2;
            fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
            chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
            for(n=0;n<M;n++){
                  printf("%d %f\n",n,ERR[n]);
            }
            exit(0);
#endif /* CHECKJAC */
#ifdef VERBOSE
            printf("\n % 3f % 3f",X[THETA]*180/pi,X[PHI]*180/pi);
            if(known)printf(" -  % 3f %
3f",real_ang[THETA],real_ang[PHI]);
            printf("\nq=%g",q);
#endif
            if(known)
            {
                  num++;
                  s=fabs(X[THETA]/pi*180);
                  if(s>180)s=360-s;
                  s-=real_ang[THETA];
                  if(fabs(s)>5)fprintf(stderr,"!! %f %f - %f %f",
                              X[THETA],X[PHI],real_ang[THETA],
                              real_ang[PHI]);
                  sum+=s;
                  sumsq+=s*s;
                  if(fabs(s)>max)max=fabs(s);
            }
      }
      if(known)
      {
            fprintf(stderr,"Max error: %f\nStd. error:%f\n",
                  max,sqrt(sumsq/num));
      }
      return 0;
}

int fcn2(m,n,x,fvec,iflag)
int*m,*n,*iflag;
double *x,*fvec;
{
      int if2=1;
      fcn(m,n,x,fvec,0,0,&if2);return 0;
}
int fcn(m,n,x,fvec,fjac,ldfjac,iflag)
int*m,*n;
double*x,*fvec,*fjac;
int*ldfjac,*iflag;
{
      int j,k;
      double g[3],tmp;
      switch(*iflag)
      {
      case 1:
            g[0]=cos(x[THETA]);
            g[1]=sin(x[THETA])*cos(x[PHI]);
            g[2]=sin(x[THETA])*sin(x[PHI]);
            q=0;
            for(j=0;j<4;j++)
            {
                  tmp=offset[j];
                  for(k=0;k<3;k++)tmp+=scl[j]*mat[j][k]*g[k];
                  tmp-=data[j];
                  q+=tmp*tmp;
                  fvec[j]=tmp;
            }
            return 0;

      case 2:
            memset(fjac,0,*ldfjac**n*sizeof(double));
            g[0]=-sin(x[THETA]);
            g[1]=cos(x[THETA])*cos(x[PHI]);
            g[2]=cos(x[THETA])*sin(x[PHI]);
            for(j=0;j<4;j++)
            {
                  tmp=0;
                  for(k=0;k<3;k++)tmp+=scl[j]*mat[j][k]*g[k];
                  fjac[j+*ldfjac*THETA]=tmp;
            }
            g[0]=0;
            g[1]=sin(x[THETA])*-sin(x[PHI]);
            g[2]=sin(x[THETA])*cos(x[PHI]);
            for(j=0;j<4;j++)
            {
                  tmp=0;
                  for(k=0;k<3;k++)tmp+=scl[j]*mat[j][k]*g[k];
                  fjac[j+*ldfjac*PHI]=tmp;
            }
            return 0;

      default:
            fprintf(stderr,"Iflag=%d\n",*iflag);
            exit(14);
      }
}

void mtrx(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "matrix.h"
            ;
      memcpy(a,T,sizeof(T));
}
```

A.1.8 fpic.c

```
/* Possible defines are:
   VERBOSE - gives listing of each data point before and after lmder
   CHECKJAC - runs chckder to check the jacobian
   NUMDIFF - determines he jacobian numerically rather than analytically
*/

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

void mtrx(float,float,float,float,float,float,float,float(*)[3]);
int fcn(int*,int*,float*,float*,float*,int*,int*);
int fcn2(int*,int*,float*,float*,int*);
int lmder1_(int(*fcn)(),int*m,int*n,float*x,float*fvec,float*fjac,
          int*ldfjac,float*tol,int*info,int*ipvt,float*wa,int*lwa);
int lmdif1_(int(*fcn)(),int*m,int*n,float*x,float*fvec,float*tol,
          int*info,int*iwa,float*wa,int*lwa);
int chkder_(int*m,int*n,float*x,float*fvec,float*fjac,int*ldfjac,
          float*xp,float*fvecp,int*mode,float*err);
float fabs_(float);
enum parameters{S1=0,S2,S3,S4,D1,D2,D3,D4,ANGLES};
enum angles{THETA=0,PHI};

float pi,*data=0,mat[4][3];
int nump=0,fev=0,jev=0;

float fabs_(float f)
{
      return fabs(f);
}
int main(int argc,char**argv)
{
      int known,numrp=0,n,a;
      FILE*dat,*off,*aln,*pts,*cal;
      float*X,*FVEC,*FJAC,*WA,TOL,l1,l2,a1,a2,hz,hy,v,*e,d[4],
            real_x[ANGLES],*real_ang,max,s,sum,sumsq,ax,ay,az,by,bz;
      int N,M,LWA,LDFJAC,INFO,*IPVT,*IWA;
#ifdef CHECKJAC
      float *ERR,*XP,*FVECP;
      int MODE;
#endif /* CHECKJAC */
      char tmp1[256],tmp2[256];
      pi=4*atan(1);
      if(argc!=3&&argc!=5)
      {
            fprintf(stderr,"Usage: fpic <cal_file> <dat_file>  ["
                   "<off_file> <pts_file>]\n\n");
            exit(1);
      }
      if(!(cal=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      fscanf(cal,"%[^\n] %[^\n] %*[^\n] %f %f %f %f %f",
             tmp1,tmp2,&ax,&ay,&az,&by,&bz);
      printf("{ %s\n  %s }\n",tmp1,tmp2);
      mtrx(0,0,ax,ay,az,by,bz,mat);
      if(!(dat=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      fscanf(dat,"%[^\n]",tmp1);printf("%s\n",tmp1);
      if((known=(argc==5)))
      {
            if(!(off=fopen(argv[3],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[3]);exit(4);}
            if(!(pts=fopen(argv[4],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[4]);exit(5);}
            fscanf(off,"%*[^\n]");
            fscanf(pts,"%*[^\n]");
            for(n=0;n<4;n++)fscanf(off,"%g %g",real_x+D1+n,real_x+S1+n);
      }
      while(!feof(dat))
      {
            if(!(data=realloc(data,sizeof(float)*4*++nump)))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(9);
            }
            for(n=0;n<4;n++)fscanf(dat,"%g ",data+4*(nump-1)+n);
      }
      if(known)
      {
            if(!(real_ang=malloc(sizeof(float)*2*nump)))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(10);
            }
            do for(n=0;n<2;n++)
                  fscanf(pts,"%g ",real_ang+2*(numrp)+n);
            while(!feof(pts)&&++numrp<nump);
            if(nump!=numrp+1)
            {
                  fprintf(stderr,
                        "Different number of data points and angles\n");
                  exit(12);
            }
      }
      M=4*nump;
      N=ANGLES+2*nump;
      LWA=5*N+M
#ifdef NUMDIFF
            +M*N; // last term only for lmdif1
#endif
      ;
      LDFJAC=M;
      if(!(X=malloc(sizeof(float)*N))||
         !(FVEC=malloc(sizeof(float)*M))||
         !(FJAC=malloc(sizeof(float)*LDFJAC*N))||
         !(WA=malloc(sizeof(float)*LWA))||
         !(IPVT=malloc(sizeof(int)*N))||
         !(IWA=malloc(sizeof(int)*N))
#ifdef CHECKJAC
         ||!(XP=malloc(sizeof(float)*N))||
         !(FVECP=malloc(sizeof(float)*M))||
         !(ERR=malloc(sizeof(float)*M))
#endif /* CHECKJAC */
            )
      {
            fprintf(stderr,"Out of memory allocating arrays!");
            exit(13);
      }
      TOL=0.00000;
      for(n=0;n<4;n++)fscanf(cal,"%f %f",X+D1+n,X+S1+n);
      for(n=0;n<nump;n++)
      {
            e=data+4*n;
            for(a=0;a<4;a++)d[a]=(e[a]-X[D1+a])/X[S1+a];
            l1=sqrt(d[0]*d[0]+d[1]*d[1]);
            l2=sqrt(d[2]*d[2]+d[3]*d[3]);
            a1=atan2(d[1],d[0]);
            a2=atan2(d[3],d[2]);
            hz=l1*sin(pi/4-a1);
            hy=l2*sin(pi/4-a2);
            v=(l1*cos(pi/4-a1)-l2*cos(pi/4-a2))/2;
            X[ANGLES+2*n+PHI]=atan2(hz,hy);
            if(X[ANGLES+2*n+PHI]<0)X[ANGLES+2*n+PHI]+=2*pi;
            X[ANGLES+2*n+THETA]=atan2(sqrt(hy*hy+hz*hz),v);
#ifdef VERBOSE
            fprintf(stderr,"\n%3f %3f",X[ANGLES+2*n+THETA]/pi*180,
                   X[ANGLES+2*n+PHI]/pi*180);
            if(known)fprintf(stderr," - %3f %3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
#endif /* VERBOSE */
      }
#ifndef CHECKJAC
#ifndef NUMDIFF
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
#else /* NUMDIFF */
      lmdif1_(fcn2,&M,&N,X,FVEC,&TOL,&INFO,IWA,WA,&LWA);
#endif /* NUMDIFF */
#else  /* CHECKJAC */
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
      MODE=1;
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      fcn(&M,&N,XP,FVECP,FJAC,&LDFJAC,&MODE);
      MODE=2;
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      for(n=0;n<M;n++){
            printf("%d %f\n",n,ERR[n]);
//          for(a=0;a<N;a++)printf("%.2f ",FJAC[n+a*LDFJAC]);printf("\n");
      }
      exit(0);
#endif /* CHECKJAC */
      fprintf(stderr,"\nlmder1 exit code: %d\n",INFO);
#ifdef VERBOSE
      for(n=0;n<nump;n++)
      {
            fprintf(stderr,"\n % 3f % 3f",
                   X[ANGLES+2*n+THETA]*180/pi,X[ANGLES+2*n+PHI]*180/pi);
            if(known)fprintf(stderr," -  % 3f % 3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
      }
#endif /* VERBOSE */
#if 1
      for(n=0;n<4;n++){
            fprintf(stderr,"d%d: % 3f\ts%d: % 3f\n",
                  n,X[D1+n],n,(X[S1+n]-1)*100);
            if(known)fprintf(stderr,"    % 3f\t    % 3f\n",
                         real_x[D1+n],real_x[S1+n]);
      }
      if(known)
      {
            for(sum=sumsq=max=n=0;n<nump;n++) {
                  s=fabs(X[ANGLES+2*n+THETA]/pi*180);
                  if(s>180)s=360-s;
                  s-=real_ang[2*n+THETA];
                  sum+=s;
                  sumsq+=s*s;
                  if(fabs(s)>max)max=fabs(s);
            }
            fprintf(stderr,"Max error: %f\nStd. error:%f\nFn.
evaluations:"
                  "%d\nJacobian evaluations %d\n\n",max,
                  sqrt(sumsq/nump),fev,jev);
      }
#endif
      printf("% .5f\t% .5f\t% .5f\n\t\t% .5f\t% .5f\n",
             ax,ay,az,by,bz);
      for(n=0;n<4;n++)printf("% .5f\t% .5f\n",X[D1+n],X[S1+n]);
      return 0;
}
int fcn2(m,n,x,fvec,iflag)
int*m,*n,*iflag;
float *x,*fvec;
{
      int if2=1;
      fcn(m,n,x,fvec,0,0,&if2);return 0;
}
int fcn(m,n,x,fvec,fjac,ldfjac,iflag)
int*m,*n;
float*x,*fvec,*fjac;
int*ldfjac,*iflag;
{
      int i,j,k;
      float g[3],tmp,q=0;
      switch(*iflag)
      {
      case 1:
            fev++;
            for(i=0;i<nump;i++)
            {
                  g[0]=cos(x[ANGLES+2*i+THETA]);
                  g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=x[D1+j];
                        for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                        tmp-=data[4*i+j];
                        fvec[4*i+j]=tmp;
                        q+=tmp*tmp;
                  }
            }
//          for(i=0;i<*m;i++)printf("%d %f\n",i,fvec[i]);
//          printf("%f\n",q);
            return 0;

      case 2:
            jev++;

            memset(fjac,0,*ldfjac**n*sizeof(float));
            for(i=0;i<nump;i++)
            {
                  g[0]=-sin(x[ANGLES+2*i+THETA]);
                  g[1]=cos(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=cos(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
```

```
				for(j=0;j<4;j++)
				{
					tmp=0;
					for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
					fjac[4*i+j+*ldfjac*(ANGLES+2*i+THETA)]=tmp;
				}
				g[0]=0;
				g[1]=sin(x[ANGLES+2*i+THETA])*-sin(x[ANGLES+2*i+PHI]);
				g[2]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
				for(j=0;j<4;j++)
				{
					tmp=0;
					for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
					fjac[4*i+j+*ldfjac*(ANGLES+2*i+PHI)]=tmp;
				}
				g[0]=cos(x[ANGLES+2*i+THETA]);
				g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
				g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
				for(j=0;j<4;j++)
				{
					tmp=0;
					for(k=0;k<3;k++)tmp+=mat[j][k]*g[k];
					fjac[4*i+j+*ldfjac*(S1+j)]=tmp;
					fjac[4*i+j+*ldfjac*(D1+j)]=1;
				}
			}
			return 0;

	default:
			fprintf(stderr,"Iflag=%d\n",*iflag);
			exit(14);
	}
}

void mtrx(da,db,ax,ay,az,by,bz,a)
float da,db,ax,ay,az,by,bz,(*a)[3];
{
	float T[4][3]=
#include "matrix.h"
	;
	memcpy(a,T,sizeof(T));
}
```

A.1.9 lpts.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

double error(double);

int main(int argc,char**argv)
{
	double lerr;
	int n,m;
	srand(time(0)+getpid());
```

```
      if(argc!=3)
      {
            fprintf(stderr,
                  "Usage: lpts <lpts> <lerr>\n\t<lerr> in degrees\n\n");
            exit(1);
      }
      sscanf(argv[1],"%d",&n);
      sscanf(argv[2],"%lg",&lerr);
      printf("lpts=%d lerr=%g\n",n,lerr);
      for(m=0;m<n;m++)
            printf("%.3f %.3f\n",
                   90+error(lerr),(double)rand()/RAND_MAX*360);
      return 0;
}
```

A.1.10 off.c

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <unistd.h>

double error(double);

int main(int argc,char**argv)
{
      double off,scl;
      int n;
      srand(time(0)+getpid());
      if(argc!=3)
      {
            fprintf(stderr,"Usage: off <off> <scale>\n\t<off> in g\n"
                  "\t<scale> in percent\n\n");
            exit(1);
      }
      sscanf(argv[1],"%lg",&off);
      sscanf(argv[2],"%lg",&scl);
      printf("off=%g scl=%g\n",off,scl);
      for(n=0;n<4;n++)printf("%.5f %.5f\n",error(off),error(scl));
      return 0;
}
```

A.1.11 pic.c

```
/* Possible defines are:
   VERBOSE - gives listing of each data point before and after lmder
   CHECKJAC - runs chckder to check the jacobian
   NUMDIFF - determines he jacobian numerically rather than analytically
*/

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

void mtrx(double,double,double,double,double,double,double,double(*)[3]);
int fcn(int*,int*,double*,double*,double*,int*,int*);
int fcn2(int*,int*,double*,double*,int*);
int lmder1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*fjac,
          int*ldfjac,double*tol,int*info,int*ipvt,double*wa,int*lwa);
int lmdif1_(int(*fcn)(),int*m,int*n,double*x,double*fvec,double*tol,
          int*info,int*iwa,double*wa,int*lwa);
int chkder_(int*m,int*n,double*x,double*fvec,double*fjac,int*ldfjac,
          double*xp,double*fvecp,int*mode,double*err);
double d_lg10(double*);

enum parameters{S1=0,S2,S3,S4,D1,D2,D3,D4,ANGLES};
enum angles{THETA=0,PHI};

double pi,*data=0,mat[4][3];
int nump=0,fev=0,jev=0;
double d_lg10(double*a)
{
      return log10(*a);
}

int main(int argc,char**argv)
{
      int known,numrp=0,n,a;
      FILE*dat,*off,*pts,*cal;
      double*X,*FVEC,*FJAC,*WA,TOL,l1,l2,a1,a2,hz,hy,v,*e,d[4],
            real_x[ANGLES],*real_ang,max,s,sum,sumsq,ax,ay,az,by,bz;
      int N,M,LWA,LDFJAC,INFO,*IPVT,*IWA;
#ifdef CHECKJAC
      double *ERR,*XP,*FVECP;
      int MODE;
#endif /* CHECKJAC */
      char tmp1[256],tmp2[256];
      pi=4*atan(1);
      if(argc!=3&&argc!=5)
      {
            fprintf(stderr,"Usage: pic <cal_file> <dat_file>  ["
                   "<off_file> <pts_file>]\n\n");
            exit(1);
      }
      if(!(cal=fopen(argv[1],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[1]);exit(2);}
      fscanf(cal,"%[^\n] %[^\n] %*[^\n] %lf %lf %lf %lf %lf",
             tmp1,tmp2,&ax,&ay,&az,&by,&bz);
      printf("{ %s\n  %s }\n",tmp1,tmp2);
      mtrx(0,0,ax,ay,az,by,bz,mat);
      if(!(dat=fopen(argv[2],"r")))
      {fprintf(stderr,"Can't open %s\n",argv[2]);exit(3);}
      fscanf(dat,"%[^\n]",tmp1);printf("%s\n",tmp1);
      if((known=(argc==5)))
      {
            if(!(off=fopen(argv[3],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[3]);exit(4);}
            if(!(pts=fopen(argv[4],"r")))
            {fprintf(stderr,"Can't open %s\n",argv[4]);exit(5);}
            fscanf(off,"%*[^\n]");
            fscanf(pts,"%*[^\n]");
            for(n=0;n<4;n++)fscanf(off,"%lg %lg",real_x+D1+n,real_x+S1+n);
      }
      while(!feof(dat))
      {
            if(!(data=realloc(data,sizeof(double)*4*++nump)))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(9);
            }
            for(n=0;n<4;n++)fscanf(dat,"%lg ",data+4*(nump-1)+n);
      }
      if(known)
      {
            if(!(real_ang=malloc(sizeof(double)*2*nump)))
            {
                  fprintf(stderr,"Out of memory!\n");
                  exit(10);
            }
            do for(n=0;n<2;n++)
                  fscanf(pts,"%lg ",real_ang+2*(numrp)+n);
            while(!feof(pts)&&++numrp<nump);
            if(nump!=numrp+1)
            {
                  fprintf(stderr,
                        "Different number of data points and angles\n");
                  exit(12);
            }
      }
      M=4*nump;
      N=ANGLES+2*nump;
      LWA=5*N+M
#ifdef NUMDIFF
            +M*N; // last term only for lmdif1
#endif
      ;
      LDFJAC=M;
      if(!(X=malloc(sizeof(double)*N))||
         !(FVEC=malloc(sizeof(double)*M))||
         !(FJAC=malloc(sizeof(double)*LDFJAC*N))||
         !(WA=malloc(sizeof(double)*LWA))||
         !(IPVT=malloc(sizeof(int)*N))||
         !(IWA=malloc(sizeof(int)*N))
#ifdef CHECKJAC
         ||!(XP=malloc(sizeof(double)*N))||
         !(FVECP=malloc(sizeof(double)*M))||
         !(ERR=malloc(sizeof(double)*M))
#endif /* CHECKJAC */
            )
      {
            fprintf(stderr,"Out of memory allocating arrays!");
            exit(13);
      }
      TOL=0.00000;
      for(n=0;n<4;n++)fscanf(cal,"%lf %lf",X+D1+n,X+S1+n);
      for(n=0;n<nump;n++)
      {
            e=data+4*n;
            for(a=0;a<4;a++)d[a]=(e[a]-X[D1+a])/X[S1+a];
            l1=sqrt(d[0]*d[0]+d[1]*d[1]);
            l2=sqrt(d[2]*d[2]+d[3]*d[3]);
            a1=atan2(d[1],d[0]);
            a2=atan2(d[3],d[2]);
            hz=l1*sin(pi/4-a1);
            hy=l2*sin(pi/4-a2);
            v=(l1*cos(pi/4-a1)-l2*cos(pi/4-a2))/2;
            X[ANGLES+2*n+PHI]=atan2(hz,hy);
            if(X[ANGLES+2*n+PHI]<0)X[ANGLES+2*n+PHI]+=2*pi;
            X[ANGLES+2*n+THETA]=atan2(sqrt(hy*hy+hz*hz),v);
#ifdef VERBOSE
            fprintf(stderr,"\n%3f %3f",X[ANGLES+2*n+THETA]/pi*180,
                   X[ANGLES+2*n+PHI]/pi*180);
            if(known)fprintf(stderr," - %3f %3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
#endif /* VERBOSE */
      }
#ifndef CHECKJAC
#ifndef NUMDIFF
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
#else /* NUMDIFF */
      lmdif1_(fcn2,&M,&N,X,FVEC,&TOL,&INFO,IWA,WA,&LWA);
#endif /* NUMDIFF */
#else  /* CHECKJAC */
      lmder1_(fcn,&M,&N,X,FVEC,FJAC,&LDFJAC,&TOL,&INFO,IPVT,WA,&LWA);
      MODE=1;
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      fcn(&M,&N,XP,FVECP,FJAC,&LDFJAC,&MODE);
      MODE=2;
      fcn(&M,&N,X,FVEC,FJAC,&LDFJAC,&MODE);
      chkder_(&M,&N,X,FVEC,FJAC,&LDFJAC,XP,FVECP,&MODE,ERR);
      for(n=0;n<M;n++){
            printf("%d %f\n",n,ERR[n]);
//          for(a=0;a<N;a++)printf("%.2f ",FJAC[n+a*LDFJAC]);printf("\n");
      }
      exit(0);
#endif /* CHECKJAC */
      fprintf(stderr,"\nlmder1 exit code: %d\n",INFO);
#ifdef VERBOSE
      for(n=0;n<nump;n++)
      {
            fprintf(stderr,"\n % 3f % 3f",
                   X[ANGLES+2*n+THETA]*180/pi,X[ANGLES+2*n+PHI]*180/pi);
            if(known)fprintf(stderr," -  % 3f % 3f",
                        real_ang[2*n+THETA],real_ang[2*n+PHI]);
      }
#endif /* VERBOSE */
#if 1
      for(n=0;n<4;n++){
            fprintf(stderr,"d%d: % 3f\ts%d: % 3f\n",
                  n,X[D1+n],n,(X[S1+n]-1)*100);
            if(known)fprintf(stderr,"    % 3f\t    % 3f\n",
                         real_x[D1+n],real_x[S1+n]);
      }
      if(known)
      {
            for(sum=sumsq=max=n=0;n<nump;n++) {
                  s=fabs(X[ANGLES+2*n+THETA]/pi*180);
                  if(s>180)s=360-s;
                  s-=real_ang[2*n+THETA];
                  sum+=s;
                  sumsq+=s*s;
                  if(fabs(s)>max)max=fabs(s);
            }
            fprintf(stderr,"Max error: %f\nStd. error:%f\nFn.
evaluations:"
                  "%d\nJacobian evaluations %d\n\n",max,
                  sqrt(sumsq/nump),fev,jev);
      }
#endif
      printf("% .5f\t% .5f\t% .5f\n\t\t% .5f\t% .5f\n",
              ax,ay,az,by,bz);
      for(n=0;n<4;n++)printf("% .5f\t% .5f\n",X[D1+n],X[S1+n]);
      return 0;
```

```
}
int fcn2(m,n,x,fvec,iflag)
int*m,*n,*iflag;
double *x,*fvec;
{
      int if2=1;
      fcn(m,n,x,fvec,0,0,&if2);return 0;
}
int fcn(m,n,x,fvec,fjac,ldfjac,iflag)
int*m,*n;
double*x,*fvec,*fjac;
int*ldfjac,*iflag;
{
      int i,j,k;
      double g[3],tmp,q=0;
      switch(*iflag)
      {
      case 1:
            fev++;
            for(i=0;i<nump;i++)
            {
                  g[0]=cos(x[ANGLES+2*i+THETA]);
                  g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=x[D1+j];
                        for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                        tmp-=data[4*i+j];
                        fvec[4*i+j]=tmp;
                        q+=tmp*tmp;
                  }
            }
//          for(i=0;i<*m;i++)printf("%d %f\n",i,fvec[i]);
//          printf("%f\n",q);
            return 0;

      case 2:
            jev++;

            memset(fjac,0,*ldfjac**n*sizeof(double));
            for(i=0;i<nump;i++)
            {
                  g[0]=-sin(x[ANGLES+2*i+THETA]);
                  g[1]=cos(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=cos(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=0;
                        for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                        fjac[4*i+j+*ldfjac*(ANGLES+2*i+THETA)]=tmp;
                  }
                  g[0]=0;
                  g[1]=sin(x[ANGLES+2*i+THETA])*-sin(x[ANGLES+2*i+PHI]);
                  g[2]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=0;
                        for(k=0;k<3;k++)tmp+=x[S1+j]*mat[j][k]*g[k];
                        fjac[4*i+j+*ldfjac*(ANGLES+2*i+PHI)]=tmp;
                  }
                  g[0]=cos(x[ANGLES+2*i+THETA]);
                  g[1]=sin(x[ANGLES+2*i+THETA])*cos(x[ANGLES+2*i+PHI]);
                  g[2]=sin(x[ANGLES+2*i+THETA])*sin(x[ANGLES+2*i+PHI]);
                  for(j=0;j<4;j++)
                  {
                        tmp=0;
                        for(k=0;k<3;k++)tmp+=mat[j][k]*g[k];
                        fjac[4*i+j+*ldfjac*(S1+j)]=tmp;
                        fjac[4*i+j+*ldfjac*(D1+j)]=1;
                  }
            }
            return 0;

      default:
            fprintf(stderr,"Iflag=%d\n",*iflag);
            exit(14);
      }
}

void mtrx(da,db,ax,ay,az,by,bz,a)
double da,db,ax,ay,az,by,bz,(*a)[3];
{
      double T[4][3]=
#include "matrix.h"
      ;
      memcpy(a,T,sizeof(T));
}
```

A.1.12 pts.c

```
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include <stdio.h>
#include <unistd.h>

int main(int argc,char**argv)
{
      int n,m;
      srand(time(0)+getpid());
      if(argc!=2)
      {
            fprintf(stderr,"Usage: pts <pts>\n\n");
            exit(1);
      }
      sscanf(argv[1],"%d",&n);
      printf("pts=%d\n",n);
      for(m=0;m<n;m++)printf("%.3f %.3f\n",
            acos((double)rand()/RAND_MAX*2-1)/4/atan(1)*180,
            (double)rand()/RAND_MAX*360);
      return 0;
}
```

A.2 C Header Files

A.2.1 drv_ax.h

```
{
 {
  cos(ax)*sin(az)+sin(ax)*sin(ay)*cos(az),
  sin(ax)*sin(ay)*sin(az)-cos(ax)*cos(az),
  -sin(ax)*cos(ay)
 },
 {
  cos(ax)*sin(az)*(sin(da)-cos(da))+sin(ax)*sin(ay)*cos(az)*(sin(da)-
cos(da)),
  cos(ax)*cos(az)*(cos(da)-sin(da))+sin(ax)*sin(ay)*sin(az)*(sin(da)-
cos(da)),
  sin(ax)*cos(ay)*(cos(da)-sin(da))
 },
 {0,0,0},
 {0,0,0}
}
```

A.2.2 drv_ay.h

```
{
 {
  -cos(ax)*cos(ay)*cos(az)-sin(ay)*cos(az),
  -cos(ax)*cos(ay)*sin(az)-sin(ay)*sin(az),
  cos(ay)-cos(ax)*sin(ay),
 },
 {
  cos(ax)*cos(ay)*cos(az)*(cos(da)-sin(da))-
sin(ay)*cos(az)*(cos(da)+sin(da)),
  cos(ax)*cos(ay)*sin(az)*(cos(da)-sin(da))-
sin(ay)*sin(az)*(cos(da)+sin(da)),
  cos(ax)*sin(ay)*(cos(da)-sin(da))+cos(ay)*(cos(da)+sin(da)),
 },
 {0,0,0,},
 {0,0,0,}
}
```

A.2.3 drv_az.h

```
{
 {
  cos(ax)*sin(ay)*sin(az)+sin(ax)*cos(az)-cos(ay)*sin(az),
  -cos(ax)*sin(ay)*cos(az)+sin(ax)*sin(az)+cos(ay)*cos(az),
  0,
 },
 {
 cos(ax)*sin(ay)*sin(az)*(sin(da)-cos(da))+sin(ax)*cos(az)*(sin(da)-
cos(da))-cos(ay)*sin(az)*(cos(da)+sin(da)),
 cos(ax)*sin(ay)*cos(az)*(cos(da)-sin(da))+sin(ax)*sin(az)*(sin(da)-
cos(da))+cos(ay)*cos(az)*(cos(da)+sin(da)),
 0
 },
 {0,0,0},
 {0,0,0}
}
```

A.2.4 drv_by.h

```
{
 {0,0,0},
 {0,0,0},
 {
  sin(by)*cos(bz),
  sin(by)*sin(bz),
  -cos(by)
 },
 {
  sin(by)*cos(bz)*(cos(db)-sin(db)),
  sin(by)*sin(bz)*(cos(db)-sin(db)),
  cos(by)*(sin(db)-cos(db))
 }
}
```

A.2.5 drv_bz.h

```
{
 {0,0,0},
 {0,0,0},
 {
  cos(by)*sin(bz)-cos(bz),
  -cos(by)*cos(bz)-sin(bz),
  0
 },
 {
  cos(by)*sin(bz)*(cos(db)-sin(db))+cos(bz)*(cos(db)+sin(db)),
  cos(by)*cos(bz)*(sin(db)-cos(db))+sin(bz)*(cos(db)+sin(db)),
  0
 }
}
```

A.2.6 drv_da.h

```
{
 {0,0,0},
 {
  -cos(ax)*sin(ay)*cos(az)*(cos(da)+sin(da))+
sin(ax)*sin(az)*(cos(da)+sin(da))+cos(ay)*cos(az)*(cos(da)-sin(da)),
  -cos(ax)*sin(ay)*sin(az)*(cos(da)+sin(da))-
sin(ax)*cos(az)*(cos(da)+sin(da))+cos(ay)*sin(az)*(cos(da)-sin(da)),
  cos(ax)*cos(ay)*(cos(da)+sin(da))+sin(ay)*(cos(da)-sin(da)),
 },
 {0,0,0},
 {0,0,0}
}
```

A.2.7 drv_db.h

```
{
 {0,0,0},
 {0,0,0},
 {0,0,0},
 {
  cos(by)*cos(bz)*(cos(db)+sin(db))+sin(bz)*(cos(db)-sin(db)),
  cos(by)*sin(bz)*(cos(db)+sin(db))+cos(bz)*(sin(db)-cos(db)),
  sin(by)*(cos(db)+sin(db))
 }
}
```

A.2.8 matrix.h

```
{
 {
  -cos(ax)*sin(ay)*cos(az)+sin(ax)*sin(az)+cos(ay)*cos(az),
  -cos(ax)*sin(ay)*sin(az)-sin(ax)*cos(az)+cos(ay)*sin(az),
  cos(ax)*cos(ay)+sin(ay)
 },
 {
  cos(ax)*sin(ay)*cos(az)*(cos(da)-sin(da))+sin(ax)*sin(az)*(sin(da)-
cos(da))+cos(ay)*cos(az)*(cos(da)+sin(da)),
  cos(ax)*sin(ay)*sin(az)*(cos(da)-sin(da))+sin(ax)*cos(az)*(cos(da)-
sin(da))+cos(ay)*sin(az)*(cos(da)+sin(da)),
  cos(ax)*cos(ay)*(sin(da)-cos(da))+sin(ay)*(cos(da)+sin(da))
 },
 {
  -cos(by)*cos(bz)-sin(bz),
  cos(bz)-cos(by)*sin(bz),
  -sin(by)
 },
 {
  cos(by)*cos(bz)*(sin(db)-cos(db))+sin(bz)*(cos(db)+sin(db)),
  cos(by)*sin(bz)*(sin(db)-cos(db))-cos(bz)*(cos(db)+sin(db)),
  sin(by)*(sin(db)-cos(db))
 }
}
```

A.3 Script Files

A.3.1 doit

```
#!/bin/tcsh
./aln 5 0.001 > $1.aln
./off 0.2 25 > $1.off
./pts 8 > $1.pts
./lpts 12 0.5 > $1.lpt
./dat $1.aln $1.off $1.pts 1 > $1.dat
./dat $1.aln $1.off $1.lpt 1 > $1.ldt
./cal $1.dat $1.ldt $1.aln $1.off $1.pts $1.lpt 0.001 > $1.cal
./off `echo $2 \* 0.002 | bc –lq` `echo $2 \* 0.015 | bc –lq` > $1.cof
./add $1.off $1.cof > $1.pof
#./pts 8 > $1.ppt
```

```
#./dat $1.aln $1.pof $1.ppt 4.5 > $1.pdt
#./pic $1.cal $1.pdt $1.pof $1.ppt > $1.pcl
## ./fpic $1.cal $1.pdt $1.pof $1.ppt > $1.pcl
#./pts 100 > $1.bpt
#./dat $1.aln $1.pof $1.bpt 4.5 > $1.bdt
#./fnl $1.bdt $1.pcl $1.bpt >& $1.fin

./pts 100 > $1.tpt
./dat $1.aln $1.pof $1.tpt 1 > $1.tdt
./fnl $1.tdt $1.cal $1.tpt >& $1.fin
```

A.3.2 dolots

```
#!/bin/tcsh
rm $1.fin
foreach aa (0 1 2 3 4 5 6 7 8 9)
foreach a (0 1 2 3 4 5 6 7 8 9)
./doit $1_$aa$a $2 >& /dev/null
echo Run $aa$a | cat - $1_$aa$a.fin >> $1.fin
end
echo -n $aa > /dev/tty
end
./analyse < $1.fin | tee $1.anl
```

A.3.3 domany

```
#!/bin/tcsh

rm $1.final
foreach q (0 0.5 1 2 3 4 5 6 8 10 12 14 16)
echo temp=$q
echo temp=$q >> $1.final
./dolots $1_$q $q >>$1.final
end
```

A.3.4 convert.awk

```
#!/usr/bin/gawk -f
BEGIN {
      RS = "";
      FS = "\n"
}
 /=/ { split($1,x,"="); }
!/=/ { split($2,max,"\:");
       split($3,maxerr,"\:");
       split($4,err,"\:");
       print x[2],err[2],maxerr[2],max[2]; }
```

A.3.5 rawdata.awk

```
#!/usr/bin/gawk -f
BEGIN {
      RS = "\n\n+(#[^\n]*)*\n*"
      FS = "\n"
      for(i=1;i<=4;i++) {
            min[i]=20
            max[i]=-20
      }
      ARGV[2]=ARGV[1]
      ARGC=3
      OFMT="% .5f"
}
{
      for(i=1;i<=4;i++) {
            split($i,a," ")
            k[i]=a[1]/(a[1]+a[2])
            if(ARGIND==1)
            {
                  if(min[i]>k[i]) min[i]=k[i]
                  if(max[i]<k[i]) max[i]=k[i]
            }
            else k[i]=2*(k[i]-min[i])/(max[i]-min[i])-1
      }
      if(ARGIND!=1)print k[1],k[2],k[3],k[4]
}
```

A.3.6 makefile

```
LDFLAGS=-lm
CFLAGS= -Wall -O3

.PHONY:all clean;

all: lpts pts aln off dat cal pic add fpic fnl analyse

lpts off dat aln: error.o

cal: cal.o
      $(CC) $^ -lminpack $(LDFLAGS) -o $@

pic: pic.o
      $(CC) $^ -lminpack $(LDFLAGS) -o $@

fpic: fpic.o
      $(CC) $^ -lsminpack $(LDFLAGS) -o $@

fnl: fnl.o
      $(CC) $^ -lminpack $(LDFLAGS) -o $@

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

B Sample Output

B.1 Run 22 of Noisy Self Recalibration with 8 Points

B.1.1 nsr_8_22.aln

```
aln=5 aln_xy=0.001
0.0001 0.0002
7.9643 9.2783 -3.6924
      4.5592 3.7236
```

B.1.2 nsr_8_22.off

```
off=0.2 scl=25
-0.08850 2.94181
0.03741 -10.61356
-0.10575 -36.47886
0.05472 20.84703
```

B.1.3 nsr_8_22.pts

```
pts=8
90.654 219.953
119.904 178.064
112.828 168.426
137.507 154.100
67.703 317.466
99.905 221.306
137.566 324.762
26.668 258.268
```

B.1.4 nsr_8_22.lpt

```
lpts=12 lerr=0.5
90.080 347.294
89.182 23.502
90.143 243.965
90.538 203.121
89.721 250.185
89.937 297.702
89.649 280.845
90.149 80.797
90.262 328.984
89.327 70.663
89.908 346.570
89.774 272.352
```

B.1.5 nsr_8_22.dat

```
( aln=5 aln_xy=0.001 : off=0.2 scl=25 : pts=8 ) noise 1.000
-0.70074  0.44918 -0.52074  1.11340
-0.30167 -0.54842 -0.28635  1.72546
-0.02038 -0.54907 -0.38918  1.63203
-0.24322 -0.97108  0.01550  1.63462
-0.63551  0.92426  0.07432 -1.18679
-0.84809  0.29192 -0.39653  1.26170
-1.27328 -0.40773  0.73663  0.21579
 0.16588  1.27228 -0.73872 -0.79100
```

B.1.6 nsr_8_22.ldt

```
( aln=5 aln_xy=0.001 : off=0.2 scl=25 : lpts=12 lerr=0.5 ) noise 1.000
-0.53910  0.25216  0.48283 -1.17549
 0.20907 -0.18647  0.40707 -1.17784
-1.05856  0.66562 -0.31898  0.70881
-0.37616  0.26224 -0.62481  1.28295
-1.12006  0.70864 -0.26434  0.57454
-1.21685  0.71167  0.21402 -0.45781
-1.27108  0.77210  0.05223 -0.09932
 1.03536 -0.67574 -0.05675 -0.24253
-0.86654  0.45743  0.43058 -0.99069
 0.96216 -0.62108  0.03466 -0.47485
-0.55126  0.26435  0.48094 -1.17469
-1.26539  0.77290 -0.03388  0.09449
```

B.1.7 nsr_8_22.cal

```
ratio 0.001000 : ( aln=5 aln_xy=0.001 : off=0.2 scl=25 : pts=8 ) noise
1.000
( aln=5 aln_xy=0.001 : off=0.2 scl=25 : lpts=12 lerr=0.5 ) noise 1.000
    XXX

 0.13860     0.16145    -0.06030
             0.08062     0.06809
-0.08774     1.03096
 0.03805     0.89217
-0.10653     0.63452
 0.05390     1.20935
```

B.1.8 nsr_8_22.cof

```
off=0.05 scl=0.5
0.07715 -0.61083
0.06708 0.31216
0.06365 0.50803
-0.05075 -0.42088
```

B.1.9 nsr_8_22.pof

```
( off=0.2 scl=25 ) : ( off=0.05 scl=0.5 )
-0.01135     2.33098
 0.10449    -10.30140
-0.04210    -35.97083
 0.00397     20.42615
```

B.1.10 nsr_8_22.ppt

```
pts=8
113.770 352.398
51.306 329.443
148.937 158.294
91.003 294.243
105.269 116.866
65.449 68.539
84.797 221.025
138.527 53.470
```

B.1.11 nsr_8_22.pdt

```
( aln=5 aln_xy=0.001 : ( off=0.2 scl=25 ) : ( off=0.05 scl=0.5 ) : pts=8 )
noise 4.500
-0.66467 -0.16507  0.78114 -0.68952
-0.08123  1.08152 -0.04804 -1.51916
-0.40986 -0.94611  0.24374  1.55277
-1.16873  0.78037  0.26904 -0.42150
 0.86334 -0.82438 -0.16483  0.77529
 1.25982 -0.06243 -0.16949 -0.96157
-0.55085  0.62991 -0.51628  0.94040
-0.09681 -1.04046  0.67394  0.28796
```

B.1.12 nsr_8_22.pcl

```
{ ratio 0.001000 : ( aln=5 aln_xy=0.001 : off=0.2 scl=25 : pts=8 ) noise
1.000
  ( aln=5 aln_xy=0.001 : off=0.2 scl=25 : lpts=12 lerr=0.5 ) noise 1.000 }
( aln=5 aln_xy=0.001 : ( off=0.2 scl=25 ) : ( off=0.05 scl=0.5 ) : pts=8 )
noise 4.500
 0.13860     0.16145    -0.06030
             0.08062     0.06809
-0.00995     1.02602
 0.10890     0.89934
-0.04273     0.64079
 0.00535     1.20221
```

B.1.13 nsr_8_22.bpt

```
pts=100
158.831 197.948
57.609 294.406
```

```
62.876 85.361
104.979 141.935
140.371 84.165
      ...
167.835 84.526
112.131 129.915
17.215 38.669
```

B.1.14 nsr_8_22.bdt

```
( aln=5 aln_xy=0.001 : ( off=0.2 scl=25 ) : ( off=0.05 scl=0.5 ) : pts=100
) noise 4.500
-0.84898 -0.79776  0.38555  1.49319
-0.52447  1.24223 -0.16197 -0.97174
 1.38980 -0.06403 -0.34972 -0.68622
 0.62049 -0.63861 -0.35329  1.20401
 0.07656 -1.14565  0.49258  0.72280
      ...
-0.59162 -1.05955  0.62366  1.05262
 0.61501 -0.83396 -0.17892  1.11152
 0.95083  0.96771 -0.56236 -1.37202
```

B.1.15 nsr_8_22.fin

```
Max error: 0.746783
Std. error:0.241566
```

B.1.16 Screen output of the cal program

```
lmder1 exit code: 7

  90.308209  347.306471 -   90.080000  347.294000
  89.459425  23.448270 -   89.182000  23.502000
  90.062601  243.962033 -   90.143000  243.965000
  90.325519  203.166446 -   90.538000  203.121000
  89.600252  250.152149 -   89.721000  250.185000
  90.032576  297.696044 -   89.937000  297.702000
  89.666116  280.864683 -   89.649000  280.845000
  90.317827  80.744872 -   90.149000  80.797000
  90.485911  328.948387 -   90.262000  328.984000
  89.519603  70.640965 -   89.327000  70.663000
  90.138477  346.560985 -   89.908000  346.570000
  89.785373  272.339147 -   89.774000  272.352000
ax:  7.941243     ay:  9.250379     az: -3.455135
     7.964300          9.278300         -3.692400
            by:  4.619087     bz:  3.901356
                 4.559200          3.723600
d0: -0.087742     s0:  3.096430
    -0.088500          2.941810
d1:  0.038049     s1: -10.783184
     0.037410         -10.613560
d2: -0.106527     s2: -36.548263
    -0.105750         -36.478860
```

```
d3:  0.053903     s3:  20.934948
     0.054720          20.847030
Max error: 0.266288
Std. error:0.160579
```

B.1.17 Screen output of the pic program

```
lmder1 exit code: 7
d0: -0.087115     s0:  3.060242
    -0.088500          2.941810
d1:  0.037868     s1: -10.779551
     0.037410         -10.613560
d2: -0.106623     s2: -36.481327
    -0.105750         -36.478860
d3:  0.053897     s3:  20.919641
     0.054720          20.847030
Max error: 0.253651
Std. error:0.150217
Fn. evaluations:9
Jacobian evaluations 7
```

B.2 Vertical legs

B.2.1 vertical.bpt

```
pts=4
0.140 250.612
0.001 82.580
179.812 266.318
180.000 203.276
```

B.2.2 vertical.fin

```
Max error: 0.366636
Std. error:0.273127

0.498582    250.723435  -  0.140000       250.612000
0.156427    332.941142  -  0.001000       82.580000
179.706016  152.126701  -  179.812000     266.318000
179.633364  130.508828  -  180.000000     203.276000
```

B.3 Scaling

B.3.1 scale.bdt

```
( aln=5 aln_xy=0.001 : ( off=0.2 scl=25 ) : ( off=0.05 scl=0.5 ) : pts=100
) noise 4.500 !!  2nd member of pair is 1.05 times 1st  !!
-0.09949 -1.26727 -0.06347 -0.20664
-0.10446 -1.33063 -0.06664 -0.21697

-1.44551  1.33249 -0.14301 -0.26415
-1.51779  1.39911 -0.15061 -0.27736
```

```
 0.08395  0.83921 -0.29924  0.26580
 0.08815  0.88117 -0.31420  0.27909
```

B.3.2 scale.fin

```
q=0.000123048
q=0.00264439
q=4.87357e-05
q=0.00724418
q=0.000118392
q=0.00224575
```

C LMDER1 Documentation

```
0                                                                 Page
0              Documentation for MINPACK subroutine LMDER1
0                        Single precision version
0                      Argonne National Laboratory
0         Burton S. Garbow, Kenneth E. Hillstrom, Jorge J. More
0                               March 1980
0
  1. Purpose.
0       The purpose of LMDER1 is to minimize the sum of the squares of
        nonlinear functions in N variables by a modification of the
        Levenberg-Marquardt algorithm.  This is done by using the more
        general least-squares solver LMDER.  The user must provide a
        subroutine which calculates the functions and the Jacobian.
0
  2. Subroutine and type statements.
0       SUBROUTINE LMDER1(FCN,M,N,X,FVEC,FJAC,LDFJAC,TOL,
       *                  INFO,IPVT,WA,LWA)
        INTEGER M,N,LDFJAC,INFO,LWA
        INTEGER IPVT(N)
        REAL TOL
        REAL X(N),FVEC(M),FJAC(LDFJAC,N),WA(LWA)
        EXTERNAL FCN
0
  3. Parameters.
0       Parameters designated as input parameters must be specified on
        entry to LMDER1 and are not changed on exit, while parameters
        designated as output parameters need not be specified on entry
        and are set to appropriate values on exit from LMDER1.
0       FCN is the name of the user-supplied subroutine which calculate
          the functions and the Jacobian.  FCN must be declared in an
          EXTERNAL statement in the user calling program, and should be
          written as follows.
0         SUBROUTINE FCN(M,N,X,FVEC,FJAC,LDFJAC,IFLAG)
          INTEGER M,N,LDFJAC,IFLAG
          REAL X(N),FVEC(M),FJAC(LDFJAC,N)
          ----------
          IF IFLAG = 1 CALCULATE THE FUNCTIONS AT X AND
          RETURN THIS VECTOR IN FVEC.  DO NOT ALTER FJAC.
          IF IFLAG = 2 CALCULATE THE JACOBIAN AT X AND
          RETURN THIS MATRIX IN FJAC.  DO NOT ALTER FVEC.
          ----------
          RETURN
          END
1
```

```
0                                                                Page
0         The value of IFLAG should not be changed by FCN unless the
          user wants to terminate execution of LMDER1.  In this case se
          IFLAG to a negative integer.
0       M is a positive integer input variable set to the number of
          functions.
0       N is a positive integer input variable set to the number of
          variables.  N must not exceed M.
0       X is an array of length N.  On input X must contain an initial
          estimate of the solution vector.  On output X contains the
          final estimate of the solution vector.
0       FVEC is an output array of length M which contains the function
          evaluated at the output X.
0       FJAC is an output M by N array.  The upper N by N submatrix of
          FJAC contains an upper triangular matrix R with diagonal ele-
          ments of nonincreasing magnitude such that
0                T     T           T
                P *(JAC *JAC)*P = R *R,
0         where P is a permutation matrix and JAC is the final calcu-
          lated Jacobian.  Column j of P is column IPVT(j) (see below)
          of the identity matrix.  The lower trapezoidal part of FJAC
          contains information generated during the computation of R.
0       LDFJAC is a positive integer input variable not less than M
          which specifies the leading dimension of the array FJAC.
0       TOL is a nonnegative input variable.  Termination occurs when
          the algorithm estimates either that the relative error in the
          sum of squares is at most TOL or that the relative error
          between X and the solution is at most TOL.  Section 4 contain
          more details about TOL.
0       INFO is an integer output variable.  If the user has terminated
          execution, INFO is set to the (negative) value of IFLAG.  See
          description of FCN.  Otherwise, INFO is set as follows.
0         INFO = 0  Improper input parameters.
0         INFO = 1  Algorithm estimates that the relative error in the
                    sum of squares is at most TOL.
0         INFO = 2  Algorithm estimates that the relative error between
                    X and the solution is at most TOL.
0         INFO = 3  Conditions for INFO = 1 and INFO = 2 both hold.
0         INFO = 4  FVEC is orthogonal to the columns of the Jacobian t
                    machine precision.
1
```

```
0                                                                  Page
0         INFO = 5  Number of calls to FCN with IFLAG = 1 has reached
                    100*(N+1).
0         INFO = 6  TOL is too small.  No further reduction in the sum
                    of squares is possible.
0         INFO = 7  TOL is too small.  No further improvement in the
                    approximate solution X is possible.
0         Sections 4 and 5 contain more details about INFO.
0       IPVT is an integer output array of length N.  IPVT defines a
          permutation matrix P such that JAC*P = Q*R, where JAC is the
          final calculated Jacobian, Q is orthogonal (not stored), and
          is upper triangular with diagonal elements of nonincreasing
          magnitude.  Column j of P is column IPVT(j) of the identity
          matrix.
0       WA is a work array of length LWA.
0       LWA is a positive integer input variable not less than 5*N+M.
0
  4. Successful completion.
0       The accuracy of LMDER1 is controlled by the convergence parame-
        ter TOL.  This parameter is used in tests which make three type
        of comparisons between the approximation X and a solution XSOL.
        LMDER1 terminates when any of the tests is satisfied.  If TOL i
        less than the machine precision (as defined by the MINPACK func
        tion SPMPAR(1)), then LMDER1 only attempts to satisfy the test
        defined by the machine precision.  Further progress is not usu-
        ally possible.  Unless high precision solutions are required,
        the recommended value for TOL is the square root of the machine
        precision.
0       The tests assume that the functions and the Jacobian are coded
        consistently, and that the functions are reasonably well
        behaved.  If these conditions are not satisfied, then LMDER1 ma
        incorrectly indicate convergence.  The coding of the Jacobian
        can be checked by the MINPACK subroutine CHKDER.  If the Jaco-
        bian is coded correctly, then the validity of the answer can be
        checked, for example, by rerunning LMDER1 with a tighter toler-
        ance.
0       First convergence test.  If ENORM(Z) denotes the Euclidean norm
          of a vector Z, then this test attempts to guarantee that
0               ENORM(FVEC) .LE. (1+TOL)*ENORM(FVECS),
0         where FVECS denotes the functions evaluated at XSOL.  If this
          condition is satisfied with TOL = 10**(-K), then the final
          residual norm ENORM(FVEC) has K significant decimal digits an
          INFO is set to 1 (or to 3 if the second test is also
1
0                                                                   Page
0         satisfied).
0       Second convergence test.  If D is a diagonal matrix (implicitly
          generated by LMDER1) whose entries contain scale factors for
          the variables, then this test attempts to guarantee that
0               ENORM(D*(X-XSOL)) .LE. TOL*ENORM(D*XSOL).
0         If this condition is satisfied with TOL = 10**(-K), then the
          larger components of D*X have K significant decimal digits an
          INFO is set to 2 (or to 3 if the first test is also satis-
          fied).  There is a danger that the smaller components of D*X
          may have large relative errors, but the choice of D is such
          that the accuracy of the components of X is usually related t
          their sensitivity.
0       Third convergence test.  This test is satisfied when FVEC is
          orthogonal to the columns of the Jacobian to machine preci-
          sion.  There is no clear relationship between this test and
          the accuracy of LMDER1, and furthermore, the test is equally
          well satisfied at other critical points, namely maximizers an
          saddle points.  Therefore, termination caused by this test
          (INFO = 4) should be examined carefully.
0
  5. Unsuccessful completion.
0       Unsuccessful termination of LMDER1 can be due to improper input
        parameters, arithmetic interrupts, or an excessive number of
        function evaluations.
0       Improper input parameters.  INFO is set to 0 if N .LE. 0, or
          M .LT. N, or LDFJAC .LT. M, or TOL .LT. 0.E0, or
          LWA .LT. 5*N+M.
0       Arithmetic interrupts.  If these interrupts occur in the FCN
          subroutine during an early stage of the computation, they may
          be caused by an unacceptable choice of X by LMDER1.  In this
          case, it may be possible to remedy the situation by not evalu
          ating the functions here, but instead setting the components
          of FVEC to numbers that exceed those in the initial FVEC,
          thereby indirectly reducing the step length.  The step length
          can be more directly controlled by using instead LMDER, which
          includes in its calling sequence the step-length- governing
          parameter FACTOR.
0       Excessive number of function evaluations.  If the number of
          calls to FCN with IFLAG = 1 reaches 100*(N+1), then this indi
          cates that the routine is converging very slowly as measured
          by the progress of FVEC, and INFO is set to 5.  In this case,
          it may be helpful to restart LMDER1, thereby forcing it to
          disregard old (and possibly harmful) information.
1
```

```
0                                                                  Page
0 6. Characteristics of the algorithm.
0       LMDER1 is a modification of the Levenberg-Marquardt algorithm.
        Two of its main characteristics involve the proper use of
        implicitly scaled variables and an optimal choice for the cor-
        rection.  The use of implicitly scaled variables achieves scale
        invariance of LMDER1 and limits the size of the correction in
        any direction where the functions are changing rapidly.  The
        optimal choice of the correction guarantees (under reasonable
        conditions) global convergence from starting points far from th
        solution and a fast rate of convergence for problems with small
        residuals.
0       Timing.  The time required by LMDER1 to solve a given problem
          depends on M and N, the behavior of the functions, the accu-
          racy requested, and the starting point.  The number of arith-
          metic operations needed by LMDER1 is about N**3 to process
          each evaluation of the functions (call to FCN with IFLAG = 1)
          and M*(N**2) to process each evaluation of the Jacobian (call
          to FCN with IFLAG = 2).  Unless FCN can be evaluated quickly,
          the timing of LMDER1 will be strongly influenced by the time
          spent in FCN.
0       Storage.  LMDER1 requires M*N + 2*M + 6*N single precision sto-
          rage locations and N integer storage locations, in addition t
          the storage required by the program.  There are no internally
          declared storage arrays.
0
  7. Subprograms required.
0       USER-supplied ...... FCN
0       MINPACK-supplied ... SPMPAR,ENORM,LMDER,LMPAR,QRFAC,QRSOLV
0       FORTRAN-supplied ... ABS,AMAX1,AMIN1,SQRT,MOD
0
  8. References.
0       Jorge J. More, The Levenberg-Marquardt Algorithm, Implementatio
        and Theory. Numerical Analysis, G. A. Watson, editor.
        Lecture Notes in Mathematics 630, Springer-Verlag, 1977.
0
```

D Data Sheets

74AC00: QUAD 2-INPUT NAND GATE 83
AD22100: VOLTAGE OUTPUT TEMPERATURE SENSOR WITH SIGNAL CONDITIONING 87
HEAT SINK BONDER 93
LT1013: DUAL PRECISION OP AMP 94
ADXL202: LOW COST DUAL AXIS ACCELEROMETER 113

References

[1] B. Ellis, “Introduction to Cave Surveying,” Cave Studies Series No. 2, publ. by British Cave Research Association Sales (London, 1988).

[2] B. Thrun et al., “BCRA Grade Definitions,” *Compass Points* **14** (1996), 4 – 7.

[3] M. Stephens, “Instrument Error Experiment at SWCC,” *Compass Points* **19** (1998), 7 – 12.

[4] L. Brod, “Errors in the Suunto Compass Used for Cave Surveying,” *Compass Points* **21** (1998), 7 – 13

[5] D. Gibson, “3-D Vector Processing of Magnetometer and Inclinometer Data,” *BCRA Cave Radio and Electronics Group J.* **25** (1996), 18 – 22.

[6] P. T. Boggs, R. H. Byrd, J. E. Rogers and R. B. Schnabel, “User’s Reference Guide for ODRPACK Version 2.01: Software for Weighted Orthogonal Distance Regression,” Publ. Ref. NISTIR 92-4834, NIST (Guthersberg, Maryland, USA, 1992).

[7] W. H. Press, S. A. Teukolsky, W. T. Vetterling and B. P. Flannery, “Numerical Recipes in C: The Art of Scientific Computing,” Cambridge Univ. Press (Cambridge, 1992).

[8] J. J. Moré in: “Numerical Analysis,” G. A. Watson (ed.), Lecture Notes in Mathematics Vol. **630** Springer-Verlag (Berlin, 1977), pp.105 – 116.

[9] J. E. Dennis, Jr. and R. B. Schnabel, “Numerical Methods for Unconstrained Optimization and Nonlinear Equations,” Prentice-Hall (Englewood Cliffs, New Jersey, USA, 1983).

[10] Å. Björck, “Numerical Methods for Least Squares Problems,” SIAM (Philadelphia, USA, 1996).

[11] G. E. P. Box, M. E. Muller and G. Marsaglia, *Annals Math. Stat.* **28** (1958), 610.

[12] D. E. Knuth, “The Art of Computer Programming,” Vol. **2**: Seminumerical Algorithms, Addison Wesley (Reading, Mass., USA, 1981), p.117