Open access • Book Chapter • DOI:10.1016/B978-044482107-2/50036-4

Fitting of circles and ellipses least squares solution — Source link

Walter Gander, Rolf Strebel, Gene H. Golub

Institutions: Stanford University

Published on: 01 Jan 1995

Topics: Non-linear least squares, Total least squares, Least squares, Iteratively reweighted least squares and Residual sum of squares

Fitting of circles and ellipses
Least squares solution

Eidgenössische
Technische Hochschule
Zürich

Departement Informatik
Institut für
Wissenschaftliches Rechnen

Walter Gander
Gene H. Golub
Rolf Strebel

Fitting of Circles and Ellipses Least Squares Solution

June 1994

217

ETH Zürich
Departement Informatik
Institut für Wissenschaftliches Rechnen
Prof. Dr. W. Gander

Walter Gander
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: gander@inf.ethz.ch

Gene H. Golub
Computer Science Department
Stanford University
Stanford, California 94305
e-mail: golub@sccm.stanford.edu

Rolf Strebel
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
e-mail: strebel@inf.ethz.ch

This report is also available via anonymous ftp from ftp.inf.ethz.ch
as doc/tech-reports/1994/217.ps.

Fitting of Circles and Ellipses Least Squares Solution

Walter Gander
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
gander@inf.ethz.ch

Gene H. Golub
Computer Science Department
Stanford University
Stanford, California 94305
golub@sccm.stanford.edu

Rolf Strebel
Institut für Wissenschaftliches Rechnen
Eidgenössische Technische Hochschule
CH-8092 Zürich
strebel@inf.ethz.ch

November 11, 1994

Abstract

Fitting circles and ellipses to given points in the plane is a problem that arises in many application areas, e.g. computer graphics [1], coordinate metrology [2], petroleum engineering [11], statistics [7]. In the past, algorithms have been given which fit circles and ellipses in *some* least squares sense without minimizing the geometric distance to the given points [1], [6].

In this paper[1] we present several algorithms which compute the ellipse for which the *sum of the squares of the distances to the given points is minimal.* These algorithms are compared with classical simple and iterative methods.

Circles and ellipses may be represented algebraically i.e. by an equation of the form $F(\mathbf{x}) = 0$. If a point is on the curve then its coordinates $\mathbf{x}$ are a zero of the function F. Alternatively, curves may be represented in parametric form, which is well suited for minimizing the sum of the squares of the distances.

Keywords. least squares, curve fitting, singular value decomposition.

[1]This report and the MATLAB sources are available via anonymous ftp from ftp.inf.ethz.ch as doc/tech-reports/1994/217.ps and doc/tech-reports/1994/217.matlab.tar respectively. A shortened version of this report is to appear in BIT 34(1994), pp. 556–577.

Contents

1 Preliminaries

Ellipses, for which the *sum of the squares of the distances to the given points is minimal* will be referred to as "best fit" or "geometric fit", and the algorithms will be called "geometric".

Determining the parameters of the algebraic equation $F(\mathbf{x}) = 0$ in the least squares sense will be denoted by "algebraic fit" and the algorithms will be called "algebraic".

We will use the well known Gauss-Newton method to solve the nonlinear least squares problem [15]. Let $\mathbf{u} = (u_1, \ldots, u_n)^{\mathrm{T}}$ be a vector of unknowns and consider the nonlinear system of m equations $\mathbf{f}(\mathbf{u}) = \mathbf{0}$.

If $m > n$, then we want to minimize

$$\sum_{i=1}^{m} f_i(\mathbf{u})^2 = \min .$$

This is a *nonlinear least squares problem*, which we will solve iteratively by a sequence of linear least squares problems.

We approximate the solution $\hat{\mathbf{u}}$ by $\tilde{\mathbf{u}} + \mathbf{h}$. Developing

$$\mathbf{f}(\mathbf{u}) = (f_1(\mathbf{u}), f_2(\mathbf{u}), \ldots, f_m(\mathbf{u}))^{\mathrm{T}}$$

around $\tilde{\mathbf{u}}$ in a Taylor series, we obtain

$$\mathbf{f}(\tilde{\mathbf{u}} + \mathbf{h}) \simeq \mathbf{f}(\tilde{\mathbf{u}}) + J(\tilde{\mathbf{u}})\mathbf{h} \approx \mathbf{0}, \tag{1}$$

where J is the Jacobian. We solve equation (1) as a linear least squares problem for the correction vector $\mathbf{h}$:

$$J(\tilde{\mathbf{u}})\mathbf{h} \approx -\mathbf{f}(\tilde{\mathbf{u}}). \tag{2}$$

An iteration then with the Gauss-Newton method consists of the two steps:

1. Solving equation (2) for $\mathbf{h}$.
2. Update the approximation $\tilde{\mathbf{u}} := \tilde{\mathbf{u}} + \mathbf{h}$.

We define the following notation: a given point P_i will have the coordinate vector $\mathbf{x}_i = (x_{i1}, x_{i2})^{\mathrm{T}}$. The $m \times 2$ matrix $X = [\mathbf{x}_1, \ldots, \mathbf{x}_m]^{\mathrm{T}}$ will therefore contain the coordinates of a set of m points. The 2-norm $\|\cdot\|_2$ of vectors and matrices will simply be denoted by $\|\cdot\|$.

To improve the readability of the paper, we have moved the MATLAB implementation of the algorithms and some input data to the appendix. The MATLAB sources for the examples are available via anonymous ftp from `ftp.inf.ethz.ch`.

2 Circle: Minimizing the algebraic distance

Let us first consider an algebraic representation of the circle in the plane:

$$F(\mathbf{x}) = a\mathbf{x}^{\mathrm{T}}\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c = 0, \tag{3}$$

where $a \neq 0$ and $\mathbf{x}, \mathbf{b} \in \mathbb{R}^2$. To fit a circle, we need to compute the coefficients a, $\mathbf{b}$ and c from the given data points.

If we insert the coordinates of the points into equation (3), we obtain a linear system of equations $B\mathbf{u} = \mathbf{0}$ for the coefficients $\mathbf{u} = (a, b_1, b_2, c)^{\mathrm{T}}$, where

$$B = \begin{pmatrix} x_{11}^2 + x_{12}^2 & x_{11} & x_{12} & 1 \\ \vdots & \vdots & \vdots & \vdots \\ x_{m1}^2 + x_{m2}^2 & x_{m1} & x_{m2} & 1 \end{pmatrix}.$$

To obtain a non-trivial solution, we impose some constraint on $\mathbf{u}$, e.g. $u_1 = 1$ (commonly used) or $\|\mathbf{u}\| = 1$.

For $m > 3$, in general, we cannot expect the system to have a solution, unless all the points happen to be on a circle. Therefore, we solve the overdetermined system $B\mathbf{u} = \mathbf{r}$ where $\mathbf{u}$ is chosen to minimize $\|\mathbf{r}\|$. We obtain a standard problem (c.f. [3]):

$$\|B\mathbf{u}\| = \min \quad \text{subject to} \quad \|\mathbf{u}\| = 1.$$

This problem is equivalent to finding the right singular vector associated with the smallest singular value of B. If $a \neq 0$, we can transform equation (3) to

$$\left(x_1 + \frac{b_1}{2a}\right)^2 + \left(x_2 + \frac{b_2}{2a}\right)^2 = \frac{\|\mathbf{b}\|^2}{4a^2} - \frac{c}{a}\,, \tag{4}$$

from which we obtain the center and the radius, if the right hand side of (4) is positive:

$$\mathbf{z} = (z_1, z_2) = \left(-\frac{b_1}{2a}, -\frac{b_2}{2a}\right) \qquad r = \sqrt{\frac{\|\mathbf{b}\|^2}{4a^2} - \frac{c}{a}}\,.$$

The MATLAB procedure `algcircle` computes the center $\mathbf{z} = (z_1, z_2)$ and the radius r of the circle by minimizing the algebraic distance. This approach has the advantage of being simple. The disadvantage is that we are uncertain what we are minimizing in a geometrical sense. For applications in coordinate metrology this kind of fit is often unsatisfactory. In such applications, one wishes to minimize the sum of the squares of the distances. Figure 1 shows two circles fitted to the set of points

x	1	2	5	7	9	3
y	7	6	8	7	5	7

. (5)

Minimizing the algebraic distance, we obtain the dashed circle with radius $r = 3.0370$ and center $\mathbf{z} = (5.3794, 7.2532)$.

The algebraic solution is often useful as a starting vector for methods minimizing the geometric distance.

3 Circle: Minimizing the geometric distance

To minimize the sum of the squares of the distances $d_i^2 = (\|\mathbf{z} - \mathbf{x}_i\| - r)^2$ we need to solve a nonlinear least squares problem. Let $\mathbf{u} = (z_1, z_2, r)^{\mathrm{T}}$, we want to determine $\tilde{\mathbf{u}}$ so that

$$\sum_{i=1}^{m} d_i(\mathbf{u})^2 = \min\,.$$

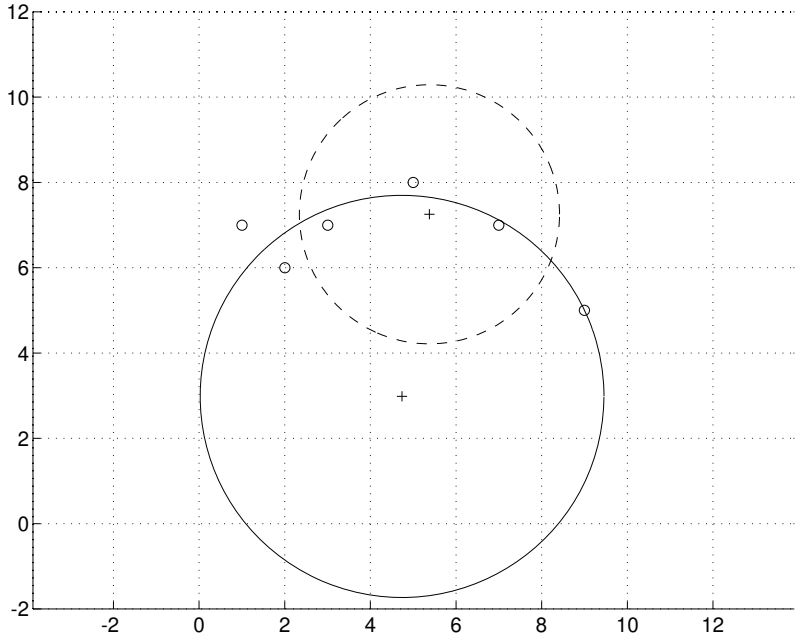

Figure 1: algebraic vs. best fit

———— Best fit
– – – – Algebraic fit

The Jacobian defined by the partial derivatives $\partial d_i(\mathbf{u})/\partial u_j$ is given by:

$$J(\mathbf{u}) = \begin{pmatrix} \dfrac{u_1 - x_{11}}{\sqrt{(u_1 - x_{11})^2 + (u_2 - x_{12})^2}} & \dfrac{u_2 - x_{12}}{\sqrt{(u_1 - x_{11})^2 + (u_2 - x_{12})^2}} & -1 \\ \vdots & \vdots & \vdots \\ \dfrac{u_1 - x_{m1}}{\sqrt{(u_1 - x_{m1})^2 + (u_2 - x_{m2})^2}} & \dfrac{u_2 - x_{2m}}{\sqrt{(u_1 - x_{m1})^2 + (u_2 - x_{m2})^2}} & -1 \end{pmatrix}.$$

A good starting vector for the Gauss-Newton method may often be obtained by solving the linear problem as given in the previous paragraph. The MATLAB procedure `circle` then iteratively computes the "best" circle.

If we use the set of points (5) and start the iteration with the values obtained from the linear model (minimizing the algebraic distance), then after 11 Gauss-Newton steps the norm of the correction vector is $2.05E{-}6$. We obtain the best fit circle with center $\mathbf{z} = (4.7398, 2.9835)$ and radius $r = 4.7142$ (the solid circle in figure 1).

4 Circle: Geometric fit in parametric form

The parametric form commonly used for the circle is given by

$$x = z_1 + r\cos\varphi \tag{6}$$
$$y = z_2 + r\sin\varphi\ . \tag{7}$$

The distance d_i of a point $P_i = (x_{i1}, x_{i2})$ may be expressed by

$$d_i^2 = \min_{\varphi_i} \left[(x_{i1} - x(\varphi_i))^2 + (x_{i2} - y(\varphi_i))^2 \right].$$

Now since we want to determine z_1, z_2 and r by minimizing

$$\sum_{i=1}^{m} d_i^2 = \min,$$

we can simultaneously minimize for z_1, z_2, r and $\{\varphi_i\}_{i=1\ldots m}$; i.e. find the minimum of the quadratic function

$$Q(\varphi_1, \varphi_2, \ldots, \varphi_m, z_1, z_2, r) = \sum_{i=1}^{m} \left[(x_{i1} - x(\varphi_i))^2 + (x_{i2} - y(\varphi_i))^2 \right].$$

This is equivalent to solving the nonlinear least squares problem

$$\begin{array}{rcll} z_1 + r\cos\varphi_i - x_{i1} & \approx & 0 & \\ z_2 + r\sin\varphi_i - x_{i2} & \approx & 0 & \quad \text{for } i = 1, 2, \ldots, m\,. \end{array}$$

Let $\mathbf{u} = (\varphi_1, \ldots, \varphi_m, z_1, z_2, r)$. The Jacobian associated with Q is

$$J = \begin{pmatrix} rS & A \\ -rC & B \end{pmatrix},$$

where $S = \mathrm{diag}(\sin\varphi_i)$ and $C = \mathrm{diag}(\cos\varphi_i)$ are $m \times m$ diagonal matrices. A and B are $m \times 3$ matrices defined by:

$$\begin{array}{rclrclrcl} a_{i1} & = & -1 & a_{i2} & = & 0 & a_{i3} & = & -\cos\varphi_i \\ b_{i1} & = & 0 & b_{i2} & = & -1 & b_{i3} & = & -\sin\varphi_i\,. \end{array}$$

For large m, J is very sparse. We note that the first part $\binom{rS}{-rC}$ is orthogonal. To compute the QR decomposition of J we use the orthonormal matrix

$$Q = \begin{pmatrix} S & C \\ -C & S \end{pmatrix}.$$

Multiplying from the left we get

$$Q^{\mathrm{T}} J = \begin{pmatrix} rI & SA - CB \\ O & CA + SB \end{pmatrix}.$$

So to obtain the QR decomposition of the Jacobian, we only have to compute a QR decomposition of the $m \times 3$ sub-matrix $CA + SB = UP$. Then

$$\begin{pmatrix} I & 0 \\ O & U^{\mathrm{T}} \end{pmatrix} Q^{\mathrm{T}} J = \begin{pmatrix} rI & SA - CB \\ O & P \end{pmatrix},$$

and the solution is obtained by backsubstitution. In general we may obtain good starting values for z_1, z_2 and r for the Gauss-Newton iteration, if we first solve the linear problem

by minimizing the algebraic distance. If the center is known, initial approximations for $\{\varphi_k\}_{k=1\ldots m}$ can be computed by

$$\varphi_k = \arg\left((x_{k1} - z_1) + i\,(x_{k2} - z_2)\right).$$

The MATLAB procedure `parcircle` computes the best fit circle using the parametric form. We use again the points (5) and start the iteration with the values obtained from the linear model (minimizing the algebraic distance). After 21 Gauss-Newton steps the norm of the correction is $3.43E{-}06$ and we obtain the same results as before: center $\mathbf{z} = (4.7398, 2.9835)$ and radius $r = 4.7142$ (the solid circle in figure 1).

5 Ellipse: Minimizing the algebraic distance

Given the quadratic equation

$$\mathbf{x}^{\mathrm{T}} A\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c = 0 \tag{8}$$

with A symmetric and positive definite, we can compute the geometric quantities of the conic as follows.

We introduce new coordinates $\bar{\mathbf{x}}$ with $\mathbf{x} = Q\bar{\mathbf{x}} + \mathbf{t}$, thus rotating and shifting the conic. Then equation (8) becomes

$$\bar{\mathbf{x}}^{\mathrm{T}}(Q^{\mathrm{T}}AQ)\bar{\mathbf{x}} + (2\mathbf{t}^{\mathrm{T}}A + \mathbf{b}^{\mathrm{T}})Q\bar{\mathbf{x}} + \mathbf{t}^{\mathrm{T}}A\mathbf{t} + \mathbf{b}^{\mathrm{T}}\mathbf{t} + c = 0.$$

Defining $\bar{A} = Q^{\mathrm{T}}AQ$, and similarly $\bar{\mathbf{b}}$ and $\bar{c}$, this equation may be written

$$\bar{\mathbf{x}}^{\mathrm{T}}\bar{A}\bar{\mathbf{x}} + \bar{\mathbf{b}}^{\mathrm{T}}\bar{\mathbf{x}} + \bar{c} = 0.$$

We may choose Q so that $\bar{A} = \mathrm{diag}(\lambda_1, \lambda_2)$; if the conic is an ellipse or a hyperbola, we may further choose $\mathbf{t}$ so that $\bar{\mathbf{b}} = \mathbf{0}$. Hence, the equation may be written

$$\lambda_1 \bar{x}_1^2 + \lambda_2 \bar{x}_2^2 + \bar{c} = 0, \tag{9}$$

and this defines an ellipse if $\lambda_1 > 0$, $\lambda_2 > 0$ and $\bar{c} < 0$. The center and the axes of the ellipse in the non-transformed system are given by

$$\begin{aligned} \mathbf{z} &= \mathbf{t} \\ a &= \sqrt{-\bar{c}/\lambda_1} \\ b &= \sqrt{-\bar{c}/\lambda_2}\,. \end{aligned}$$

Since $Q^{\mathrm{T}}Q = I$, the matrices A and $\bar{A}$ have the same (real) eigenvalues λ_1, λ_2. It follows that each function of λ_1 and λ_2 is invariant under rotation and shifts. Note

$$\begin{aligned} \det A &= a_{11}a_{22} - a_{21}a_{12} = \lambda_1\lambda_2 \\ \mathrm{trace}\, A &= a_{11} + a_{22} = \lambda_1 + \lambda_2\,, \end{aligned}$$

which serve as a basis for all polynomials symmetric in λ_1, λ_2. As a possible application of above observations, let us express the quotient $\kappa = a/b$ for the ellipse's axes a and b. With $a^2 = -\bar{c}/\lambda_1$ and $b^2 = -\bar{c}/\lambda_2$ we get

$$\kappa^2 + 1/\kappa^2 = \frac{\lambda_2}{\lambda_1} + \frac{\lambda_1}{\lambda_2} = \frac{\lambda_1^2 + \lambda_2^2}{\lambda_1\lambda_2} = \frac{(\mathrm{trace}\, A)^2 - 2\det A}{\det A} = \frac{a_{11}^2 + a_{22}^2 + 2a_{12}^2}{a_{11}a_{22} - a_{12}^2}$$

and therefore

$$\kappa^2 \;=\; \mu \pm \sqrt{\mu^2 - 1}$$

where

$$\mu \;=\; \frac{(\operatorname{trace} A)^2}{2 \det A} - 1 \; .$$

To compute the coefficients $\mathbf{u}$ from given points, we insert the coordinates into equation (8) and obtain a linear system of equations $B\mathbf{u} = 0$, which we may solve again as constrained least squares problem: $\|B\mathbf{u}\| = \min$ subject to $\|\mathbf{u}\| = 1$. The MATLAB procedure `algellipse` fits an ellipse by minimizing this algebraic distance.

The disadvantage of the constraint $\|\mathbf{u}\| = 1$ is its non-invariance for Euclidean coordinate transformations

$$\mathbf{x} = Q\bar{\mathbf{x}} + \mathbf{t}, \quad \text{where } Q^{\mathrm{T}}Q = I .$$

For this reason Bookstein [9] recommended solving the constrained least squares problem

$$\mathbf{x}^{\mathrm{T}}A\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c \;\approx\; 0 \tag{10}$$

$$\lambda_1^2 + \lambda_2^2 = a_{11}^2 + 2a_{12}^2 + a_{22}^2 \;=\; 1 \; . \tag{11}$$

While [9] describes a solution based on eigenvalue decomposition, we may solve the same problem more efficiently and accurately with a singular value decomposition as described in [12]. In the simple algebraic solution by SVD, we solve the system for the parameter vector

$$\mathbf{u} \;=\; (a_{11}, 2a_{12}, a_{22}, b_1, b_2, c)^{\mathrm{T}} \tag{12}$$

with the constraint $\|\mathbf{u}\| = 1$, which is not invariant under Euclidean transformations. If we define vectors

$$\begin{aligned} \mathbf{v} &= (b_1, b_2, c)^{\mathrm{T}} \\ \mathbf{w} &= (a_{11}, \sqrt{2}\, a_{12}, a_{22})^{\mathrm{T}} \end{aligned}$$

and the coefficient matrix

$$S = \begin{pmatrix} x_{11} & x_{12} & 1 & x_{11}^2 & \sqrt{2}\, x_{11}x_{12} & x_{12}^2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_{m1} & x_{m2} & 1 & x_{m1}^2 & \sqrt{2}\, x_{m1}x_{m2} & x_{m2}^2 \end{pmatrix},$$

then the Bookstein constraint (11) may be written $\|\mathbf{w}\| = 1$, and we have the reordered system

$$S \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \approx \mathbf{0}.$$

The QR decomposition of S leads to the equivalent system

$$\begin{pmatrix} R_{11} & R_{12} \\ 0 & R_{22} \end{pmatrix} \begin{pmatrix} \mathbf{v} \\ \mathbf{w} \end{pmatrix} \approx \mathbf{0},$$

which may be solved in following steps:

$$\begin{aligned} R_{22}\mathbf{w} &\approx \mathbf{0} \\ \|\mathbf{w}\| &= 1 \; . \end{aligned}$$

Using the singular value decomposition of $R_{22} = U\Sigma V^{\mathrm{T}}$, finding $\mathbf{w} = \mathbf{v}_3$, and then

$$\mathbf{v} \;=\; -{R_{11}}^{-1}R_{12}\mathbf{w}.$$

The MATLAB procedure `bookstein` and the MATLAB procedure `bookstein_svd` implement the two algorithms. Note that the problem

$$(S_1\ S_2)\begin{pmatrix}\mathbf{v}\\ \mathbf{w}\end{pmatrix} \approx \mathbf{0} \quad \text{where } \|\mathbf{w}\| = 1$$

is equivalent to the generalized total least squares problem finding a matrix $\hat{S_2}$ such that

$$\begin{aligned}\operatorname{rank}(S_1\ \hat{S_2}) &\leq 5\\ \|(S_1\ \hat{S_2}) - (S_1\ S_2)\| &= \inf_{\operatorname{rank}(S_1\ \bar{S_2})\leq 5} \|(S_1\ \bar{S_2}) - (S_1\ S_2)\|\,.\end{aligned}$$

In other words, find a best rank 5 approximation to S that leaves S_1 fixed. A description of this problem may be found in [16].

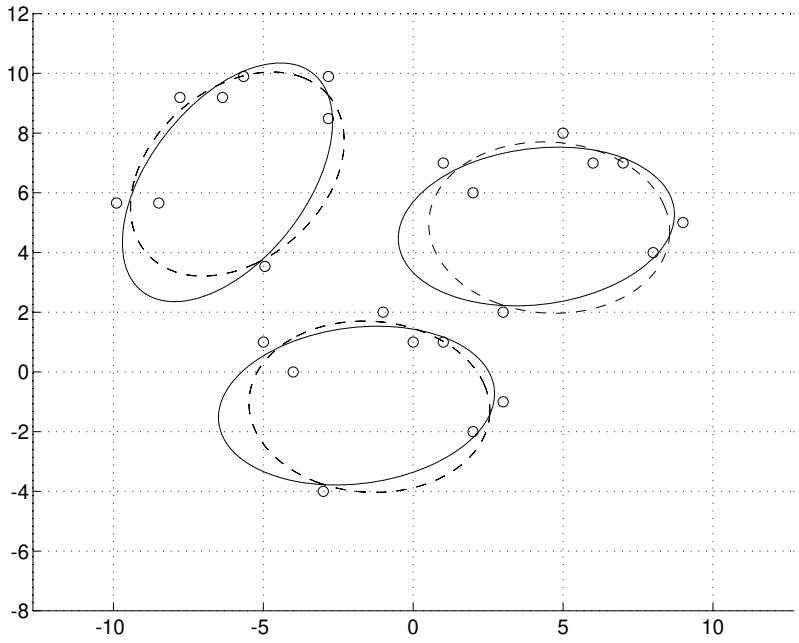

Figure 2: Euclidean-invariant algorithms

———— Constraint $\lambda_1^2 + \lambda_2^2 = 1$
– – – – Constraint $\lambda_1 + \lambda_2 = 1$

To demonstrate the influence of different coordinate systems, we have computed the ellipse fit for this set of points:

x	1	2	5	7	9	6	3	8
y	7	6	8	7	5	7	2	4

, (13)

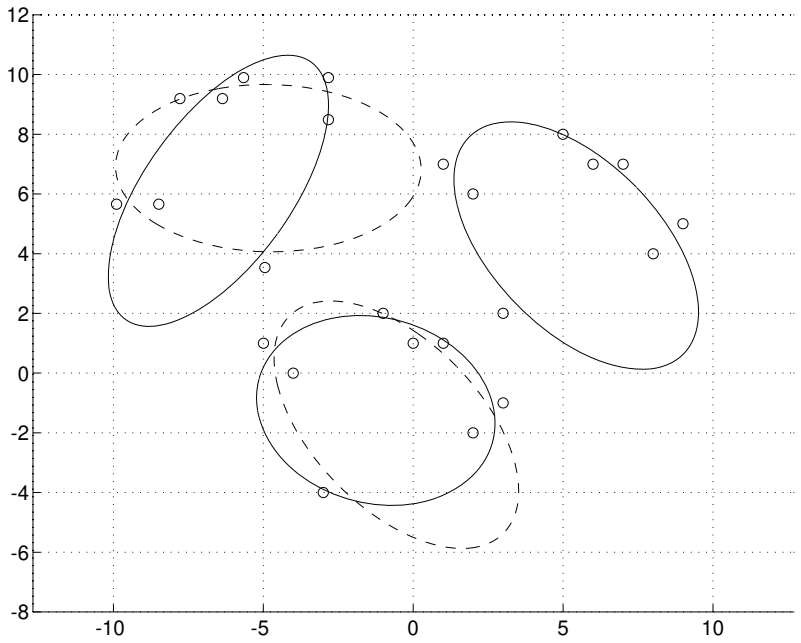

Figure 3: Non-invariant algebraic algorithm

———— Fitted ellipses
– – – – Originally fitted ellipse after transformation

which are first shifted by $(-6,-6)$, and then by $(-4,4)$ and rotated by $\pi/4$. See figures 2–3 for the fitted ellipses.

Since $\lambda_1^2 + \lambda_2^2 \neq 0$ for ellipses, hyperbolas and parabolas, the Bookstein constraint is appropriate to fit any of these. But all we need is an invariant $I \neq 0$ for ellipses—and one of them is $\lambda_1 + \lambda_2$. Thus we may invariantly fit an ellipse with the constraint

$$\lambda_1 + \lambda_2 = a_{11} + a_{22} = 1,$$

which results in the linear least squares problem

$$\begin{pmatrix} 2x_{11}x_{12} & x_{22}^2 - x_{11}^2 & x_{11} & x_{12} & 1 \\ \vdots & \vdots & & & \vdots \\ 2x_{m1}x_{m2} & x_{m2}^2 - x_{m1}^2 & x_{m1} & x_{m2} & 1 \end{pmatrix} \mathbf{v} \approx \begin{pmatrix} -x_{11}^2 \\ \vdots \\ -x_{m1}^2 \end{pmatrix}.$$

See the dashed ellipses in figure 2. The MATLAB procedure `alge_simple` implements this algorithm.

6 Ellipse: Geometric fit in parametric form

In order to fit an ellipse in parametric form, we consider the equations

$$\mathbf{x} = \mathbf{z} + Q(\alpha)\mathbf{x}', \quad \mathbf{x}' = \begin{pmatrix} a\cos\varphi \\ b\sin\varphi \end{pmatrix}, \quad Q(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix}.$$

Minimizing the sum of squares of the distances of the given points to the "best" ellipse is equivalent to solving the nonlinear least squares problem:

$$\mathbf{g}_i = \begin{pmatrix} x_{i1} \\ x_{i2} \end{pmatrix} - \begin{pmatrix} z_1 \\ z_2 \end{pmatrix} - Q(\alpha) \begin{pmatrix} a\cos\varphi_i \\ b\sin\varphi_i \end{pmatrix} \approx 0, \quad i = 1, \ldots, m.$$

Thus we have $2m$ nonlinear equations for $m+5$ unknowns: $\varphi_1, \ldots, \varphi_m$, α, a, b, z_1, z_2. To compute the Jacobian we need the partial derivatives:

$$\begin{aligned}
\frac{\partial \mathbf{g}_i}{\partial \varphi_j} &= -\delta_{ij} Q(\alpha) \begin{pmatrix} -a\sin\varphi_i \\ b\cos\varphi_i \end{pmatrix} \\
\frac{\partial \mathbf{g}_i}{\partial \alpha} &= -\dot{Q}(\alpha) \begin{pmatrix} a\cos\varphi_i \\ b\sin\varphi_i \end{pmatrix} \\
\frac{\partial \mathbf{g}_i}{\partial a} &= -Q(\alpha) \begin{pmatrix} \cos\varphi_i \\ 0 \end{pmatrix} \\
\frac{\partial \mathbf{g}_i}{\partial b} &= -Q(\alpha) \begin{pmatrix} 0 \\ \sin\varphi_i \end{pmatrix} \\
\frac{\partial \mathbf{g}_i}{\partial z_1} &= \begin{pmatrix} -1 \\ 0 \end{pmatrix} \\
\frac{\partial \mathbf{g}_i}{\partial z_2} &= \begin{pmatrix} 0 \\ -1 \end{pmatrix}
\end{aligned}$$

where we have used the notation

$$\delta_{ij} = \begin{cases} 1, & i = j \\ 0, & i \neq j \end{cases} .$$

Thus the Jacobian becomes:

$$J = \begin{pmatrix} -Q\binom{-as_1}{bc_1} & & & -\dot{Q}\binom{ac_1}{bs_1} & -Q\binom{c_1}{0} & -Q\binom{0}{s_1} & \binom{-1}{0} & \binom{0}{-1} \\ & \ddots & & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & -Q\binom{-as_m}{bc_m} & -\dot{Q}\binom{ac_m}{bs_m} & -Q\binom{c_m}{0} & -Q\binom{0}{s_m} & \binom{-1}{0} & \binom{0}{-1} \end{pmatrix},$$

where we have used as abbreviation $s_i = \sin\varphi_i$ and $c_i = \cos\varphi_i$. Note that

$$\dot{Q}(\alpha) = \begin{pmatrix} -\sin\alpha & -\cos\alpha \\ \cos\alpha & -\sin\alpha \end{pmatrix} \quad \text{and therefore} \quad Q^{\mathrm{T}}\dot{Q} = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}.$$

Since Q is orthogonal, the $2m \times 2m$ block diagonal matrix $U = -\operatorname{diag}(Q, \ldots, Q)$ is orthogonal, too, and

$$U^{\mathrm{T}}J = \begin{pmatrix} \binom{-as_1}{bc_1} & & & \binom{-bs_1}{ac_1} & \binom{c_1}{0} & \binom{0}{s_1} & \binom{c}{-s} & \binom{s}{c} \\ & \ddots & & \vdots & \vdots & \vdots & \vdots & \vdots \\ & & \binom{-as_m}{bc_m} & \binom{-bs_m}{ac_m} & \binom{c_m}{0} & \binom{0}{s_m} & \binom{c}{-s} & \binom{s}{c} \end{pmatrix},$$

where $s = \sin\alpha$ and $c = \cos\alpha$. If we permute the equations, we obtain a similar structure for the Jacobian as in the circle fit:

$$\bar{J} = \begin{pmatrix} -aS & A \\ bC & B \end{pmatrix}.$$

That is, $S = \mathrm{diag}(\sin\varphi_i)$ and $C = \mathrm{diag}(\cos\varphi_i)$ are two $m \times m$ diagonal matrices and A and B are $m \times 5$ and are defined by:

$$\begin{array}{rcl} A(i,1:5) & = & [\ -b\sin\varphi_i \quad \cos\varphi_i \quad 0 \quad \cos\alpha \quad \sin\alpha\ ] \\ B(i,1:5) & = & [\ a\cos\varphi_i \quad 0 \quad \sin\varphi_i \quad -\sin\alpha \quad \cos\alpha\ ]\ . \end{array}$$

We cannot give an explicit expression for an orthogonal matrix to triangularize the first m columns of J in a similar way as we did in fitting a circle. However, we can use Givens rotations to do this in m steps. The MATLAB procedure `rot_cossin`—which computes the angles for the Givens rotation matrices—is used by the MATLAB procedure `pare` to compute the best ellipse fit using the parametric form.

Figure 4 shows two ellipses fitted to the points given by

x	1	2	5	7	9	3	6	8
y	7	6	8	7	5	7	2	4

. (14)

By minimizing the algebraic distance with $\|\mathbf{u}\| = 1$ we obtain the large cigar shaped dashed ellipse with $\mathbf{z} = (13.8251, -2.1099)$, $a = 29.6437$, $b = 1.8806$ and residual norm $r = 1.80$. If we minimize the sum of squares of the distances then we obtain the solid ellipse with $\mathbf{z} = (2.6996, 3.8160)$, $a = 6.5187$, $b = 3.0319$ and $r = 1.17$. In order to obtain

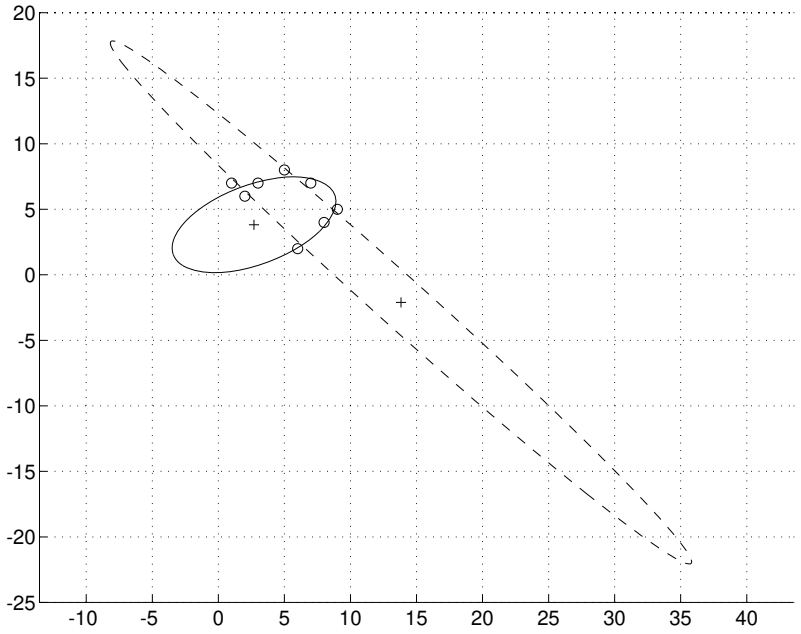

Figure 4: algebraic versus best fit

———— Best fit
– – – – Algebraic fit ($\|\mathbf{u}\| = 1$)

starting values for the nonlinear least squares problem we used the center, obtained by fitting the best circle. We cannot use the approximation $b_0 = a_0 = r$, since the Jacobian becomes singular for $b = a$! Therefore, we used $b_0 = r/2$ as a starting value. With $\alpha_0 = 0$, we needed 71 iteration steps to compute the "best ellipse" shown in Figure 4.

7 Ellipse: Iterative algebraic solutions

In this section, we will present modifications to the algebraic fit of ellipses. The algebraic equations may be weighted depending on a given estimation—thus leading to a simple iterative mechanism. Most algorithms try to weight the points such that the algebraic solution comes closer to the geometric solution. Another idea is to favor non-eccentric ellipses.

7.1 Curvature weights

The solution of

$$\mathbf{x}^{\mathrm{T}} A\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c \approx 0$$

in the least squares sense leads to an equation for each point. If the equation for point (x_{i1}, x_{i2}) is multiplied by $\omega_i > 1$, the solution will approximate this point more accurately. In [6], ω_i is set to $1/R_i$, where R_i is the curvature radius of the ellipse at a point $\mathbf{p}_i$ associated with (x_{i1}, x_{i2}). The point $\mathbf{p}_i$ is determined by intersecting the ray from the ellipse's center to (x_{i1}, x_{i2}) and the ellipse. The MATLAB procedure `lyle` fits ellipses using these curvature weights.

Tests on few data sets show, that this weighting scheme leads to better shaped ellipses in some cases, especially for eccentric ellipses; but it does not systematically restrict the solutions to ellipses. Lets look at the curvature weight solution for two problems. Figure 5 shows the result for the data set (14) presented earlier: unluckily, the algorithm finds a hyperbola for the weighted equations in the first step. On the other side, the algorithm is successful indeed for the data set in appendix C.1. Figure 6 shows the large solid ellipse (residual norm 2.20) found by the curvature weights algorithm. The small dotted ellipse is the solution of the unweighted algebraic solution (6.77); the dashed ellipse is the best fit solution using Gauss-Newton (1.66), and the dash-dotted ellipse (1.69) is found by the geometric-weight algorithm described later.

7.2 Geometric distance weighting

We are interested in weighting schemes which result in a least square solution for the geometric distance. If we define

$$Q(\mathbf{x}) = \mathbf{x}^{\mathrm{T}} A\mathbf{x} + \mathbf{b}^{\mathrm{T}}\mathbf{x} + c \ ,$$

then the simple algebraic method minimizes Q for the given points in the least squares sense. Q has the following geometric meaning: Let $h(\mathbf{x})$ be the geometric distance from the center to point $\mathbf{x}$

$$h(\mathbf{x}) = \sqrt{(x_1 - z_1)^2 + (x_2 - z_2)^2}$$

and determine $\mathbf{p}_i$ by intersecting the ray from the ellipse's center to $\mathbf{x}_i$ and the ellipse. Then, as pointed out in [9]

$$\begin{aligned} Q(\mathbf{x}_i) &= \kappa((h(\mathbf{x}_i)/h(\mathbf{p}_i))^2 - 1) && (15) \\ &\simeq 2\kappa \frac{h(\mathbf{x}_i) - h(\mathbf{p}_i)}{h(\mathbf{p}_i)} \ , \quad \text{if } \mathbf{x}_i \simeq \mathbf{p}_i && (16) \end{aligned}$$

for some constant κ. This explains why the simple algebraic solution tends to neglect points far from the center.

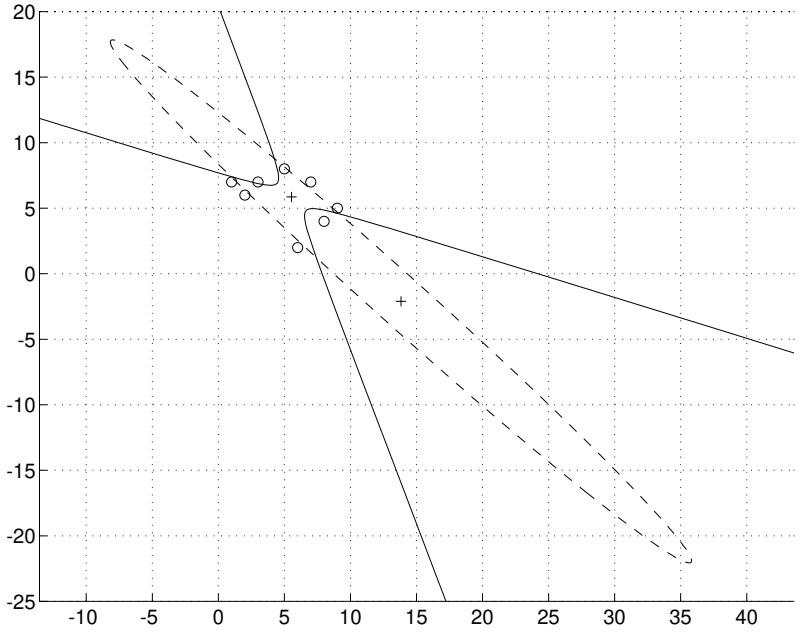

Figure 5: algebraic fit with curvature weights

——————— Conic after first curvature-weight step
– – – – Unweighted algebraic fit

Thus, we may say that the algebraic solution fits the ellipse with respect to the relative distances, i.e. a distant point has not the same importance as a near point. If we prefer to minimize the absolute distances, we may solve a weighted problem with weights

$$\omega_i = h(\mathbf{p}_i)$$

for a given estimated ellipse. The resulting estimated ellipse may then be used to determine new weights ω_i, thus iteratively solving weighted least squares problems.

Consequently, we may go a step further and set weights so that the equations are solved in the least squares sense for the geometric distances. If $d(\mathbf{x})$ is the geometric distance of $\mathbf{x}$ from the currently estimated ellipse, then weights are set

$$\omega_i = d(\mathbf{x}_i)/Q(\mathbf{x}_i).$$

The MATLAB procedure `wate2` implements this geometric weight algorithm. See the dash-dotted ellipse in figure 6 for an example.

The advantage of this method compared to the non-linear method to compute the geometric fit is, that no derivatives for the Jacobian or Hessian matrices are needed. The disadvantage of this method is, that it does not generally minimize the geometric distance. To show this, let us restate the problem:

$$\|G(\mathbf{x})\mathbf{x}\|^2 = \min \quad \text{where } \|\mathbf{x}\| = 1. \tag{17}$$

An iterative algorithm determines a sequence $(\mathbf{y}_i)$, where $\mathbf{y}_{k+1}$ is the solution of

$$\|G(\mathbf{y}_k)\,\mathbf{y}\|^2 = \min \quad \text{where } \|\mathbf{y}\| = 1. \tag{18}$$

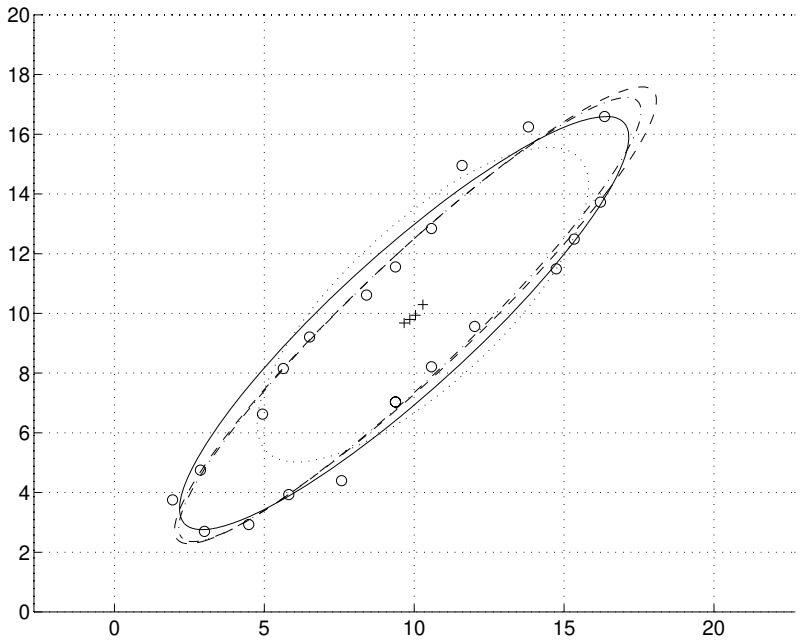

Figure 6: comparison of different fits

———— Curvature weights solution
– – – – Best fit
· · · · · · · Unweighted algebraic fit
– · – · – Geometric weights solution

The sequence $(\mathbf{y}_i)$ may have a fixed point $\tilde{\mathbf{y}} = \mathbf{y}_\infty$ without solving (17), since the conditions for critical points $\tilde{\mathbf{x}}$ and $\tilde{\mathbf{y}}$ are different for the two equations. To show this, we shall use the notation $d\mathbf{z}$ for an infinitesimal change of $\mathbf{z}$. For all $d\mathbf{x}$ with $\tilde{\mathbf{x}}^{\mathrm{T}} d\mathbf{x} = 0$ the following holds

$$2(\tilde{\mathbf{x}}^{\mathrm{T}} G^{\mathrm{T}} dG\tilde{\mathbf{x}} + \tilde{\mathbf{x}}^{\mathrm{T}} G^{\mathrm{T}} G d\mathbf{x}) = d\|G\tilde{\mathbf{x}}\|^2 = 0$$

for equation (17). Whereas for equation (18) and $\tilde{\mathbf{y}}^{\mathrm{T}} d\mathbf{y} = 0$ the condition is

$$2(\tilde{\mathbf{y}}^{\mathrm{T}} G^{\mathrm{T}} G d\mathbf{y}) = d\|G\tilde{\mathbf{y}}\|^2 = 0.$$

This problem is common to all iterative algebraic solutions of this kind, so no matter how good the weights approximate the real geometric distances, we may not generally expect that the sequence of estimated ellipses converges to the optimal solution.

We give a simple example for a fixed point of the iteration scheme (18), which does not solve (17). For $\mathbf{z} = (x, y)^{\mathrm{T}}$ consider

$$G = \begin{pmatrix} 2 & 0 \\ -4y & 4 \end{pmatrix};$$

then $\mathbf{z}_0 = (1, 0)^{\mathrm{T}}$ is a fixed point of (18), but $\mathbf{z} = (0.7278, 0.6858)^{\mathrm{T}}$ is the solution of (17).

Another severe problem with iterative algebraic methods is the lack of convergence in the general case—especially if the problem is ill-conditioned. We will shortly examine the

solution of (18) for small changes to G. Let

$$\begin{aligned} G &= U\Sigma V^{\mathrm{T}} \\ \bar{G} &= G + dG = \bar{U}\bar{\Sigma}\bar{V}^{\mathrm{T}} \end{aligned}$$

and denote with $\sigma_1, \ldots, \sigma_n$ the singular values in descending order, with $\mathbf{v}_i$ the associated right singular vectors—where $\mathbf{v}_n$ is the solution of equation (18) for G. Then we may bound $\|d\mathbf{v}_n\| = \|\bar{\mathbf{v}}_n - \mathbf{v}_n\|$ as follows. First, we define

$$\begin{aligned} \lambda &= \bar{V}^{\mathrm{T}}\mathbf{v}_n \qquad \text{thus } \|\lambda\| = 1 \\ \mu &= 1 - {\lambda_n}^2 \\ \varepsilon &= \|dG\| \end{aligned}$$

and note that

$$\sigma_i - \varepsilon \le \bar{\sigma}_i \le \sigma_i + \varepsilon \quad \text{for all } i.$$

We may conclude

$$\|\bar{\Sigma}\lambda\| = \|\bar{U}\bar{\Sigma}\lambda\| = \|\bar{G}\mathbf{v}_n\|$$

and

$$\|\bar{G}\mathbf{v}_n\| \le \|G\mathbf{v}_n\| + \|dG\mathbf{v}_n\| \le \sigma_n + \varepsilon,$$

thus

$$\sum_{i=1}^{n} \lambda_i^2 \bar{\sigma}_i^2 \le \sigma_n^2 + 2\varepsilon\sigma_n + \varepsilon^2.$$

Using that $\bar{\sigma}_i \ge \bar{\sigma}_{n-1}$ for $i \le n-1$, and that $\|\lambda\| = 1$, we simplify the above expression to

$$(1 - \lambda_n^2)\bar{\sigma}_{n-1}^2 + \lambda_n^2\bar{\sigma}_n^2 \le \sigma_n^2 + 2\varepsilon\sigma_n + \varepsilon^2. \tag{19}$$

Assuming that $\sigma_n \neq 0$ (otherwise the solution is exact) and $\varepsilon \le \sigma_n$, we have

$$\sigma_i^2 - 2\varepsilon\sigma_i + \varepsilon^2 \le \bar{\sigma}_i^2.$$

Applying this to inequality (19), we get

$$\mu \le \frac{4\varepsilon\sigma_n}{\bar{\sigma}_{n-1}^2 - \bar{\sigma}_n^2}\ .$$

Note that

$$\|\mathbf{v}_n - \bar{\mathbf{v}}_n\|^2 = \|\lambda - (0, \ldots, 1)^{\mathrm{T}}\|^2 = (\lambda_n - 1)^2 + (1 - \lambda_n^2) = 2(1 - \lambda_n).$$

Assuming that $\|\mathbf{v}_n - \bar{\mathbf{v}}_n\|$ is small (and thus $\lambda_n \approx 1$, $\mu \approx 0$), then we may write σ for $\bar{\sigma}$ and $\mu = (1 + \lambda_n)(1 - \lambda_n) \approx 2(1 - \lambda_n)$; thus we finally get

$$\|\mathbf{v}_n - \bar{\mathbf{v}}_n\| \le \sqrt{\frac{4\varepsilon\sigma_n}{\sigma_{n-1}^2 - \sigma_n^2}} \le \sqrt{\frac{2\varepsilon}{\sigma_{n-1} - \sigma_n}}\ .$$

This shows that the convergence behavior of iterative algebraic methods depends on how well—in a figurative interpretation—the solution vector $\tilde{\mathbf{v}}_n$ is separated from its hyperplane with respect to the residual norm $\|G\mathbf{v}\|$. If $\tilde{\sigma}_{n-1} \approx \tilde{\sigma}_n$, the solution is poorly determined, and the algorithm may not converge.

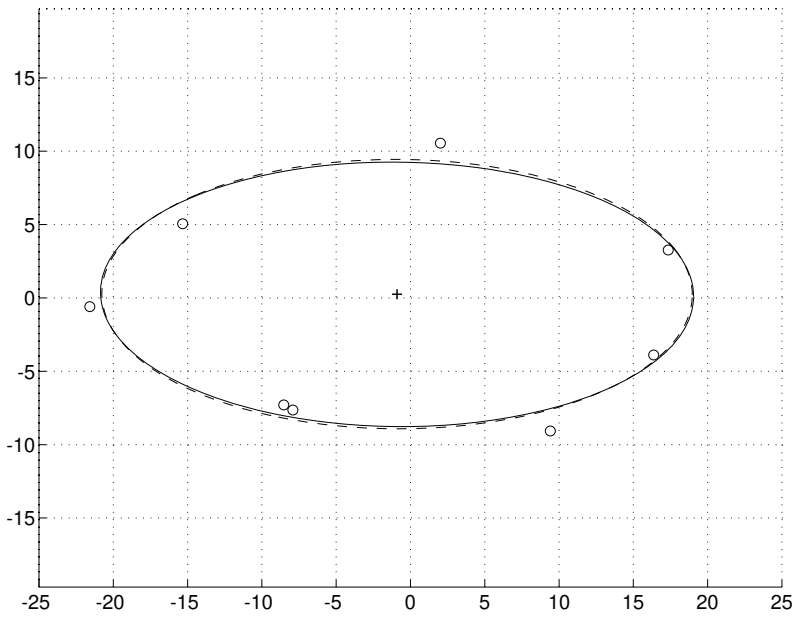

Figure 7: geometric weight fit vs. best fit

———— Geometric weight solution
– – – – Best fit

While the analytical results are not encouraging, we obtained solutions close to the optimum using the geometric-weight algorithm for several examples. Figure 7 shows the geometric-weight (solid) and the best (dashed) solution for such an example. Note that the calculation of geometric distances is relatively expensive, so a pragmatic way to limit the cost is to perform a fixed number of iterations, since convergence is not guaranteed anyway.

Appendix C.1 lists the points to be approximated, resulting in a residual norm of 2.784, compared to 2.766 for the best geometric fit. Since the algebraic method minimizes $w_i Q_i$ in the least squares sense, it remains to check that $w_i Q_i$ is proportional to the geometric distance d_i for the estimated conic. We compute

$$M_i = |(w_i Q_i)/d_i| \tag{20}$$

$$h_i = 1 - M_i/\|M\|_\infty \tag{21}$$

and find—as expected—that $\|\mathbf{h}\| = 1.1E{-}5$.

7.3 Circle weight algorithm

The main difficulty with above algebraic methods is that the solution may be any conic—not necessarily an ellipse. To cope with this case, we extend the system by weighted equations, which favour circles and non-eccentric ellipses:

$$\omega\,(a_{11} - a_{22}) \approx 0 \tag{22}$$

$$\omega\, 2a_{12} \approx 0\ . \tag{23}$$

Note that these equations are Euclidean-invariant only if ω is the same in both equations, and the problem is solved in the least squares sense. What we are really minimizing in this case is

$$\omega^2((a_{11}-a_{22})^2+4a_{12}^2)=\omega^2(\lambda_1-\lambda_2)^2;$$

hence, the constraints (22) and (23) are equivalent to the equation

$$\omega\,(\lambda_1-\lambda_2) \quad \approx \quad 0.$$

The weight ω is fixed by

$$\omega=\epsilon\, f((\lambda_1-\lambda_2)^2), \tag{24}$$

where ϵ is a parameter representing the badness of eccentric ellipses, and

$$f\colon [0,\infty[\,\to[0,\infty[\quad \text{continuous, strictly increasing.}$$

The larger ϵ is chosen, the larger will ω be, and thus the more important are equations (22–23), which make the solution be more circle-shaped. The parameter ω is determined iteratively (starting with $\omega_0=0$), where following conditions hold

$$0=\omega_0\le\omega_2\le\ldots\le\omega\le\ldots\le\omega_3\le\omega_1. \tag{25}$$

Thus, the larger the weight ω, the less eccentric the ellipse; we prove this in the following. Given weights $\chi<\omega$ and

$$F=\begin{pmatrix}1&0&-1&0&0&0\\0&1&0&0&0&0\end{pmatrix},$$

then we find solutions $\mathbf{x}$ and $\mathbf{z}$ for the equations

$$\begin{aligned}\left\|\begin{pmatrix}B\\ \chi F\end{pmatrix}\mathbf{x}\right\| &= \min\\ \left\|\begin{pmatrix}B\\ \omega F\end{pmatrix}\mathbf{z}\right\| &= \min\end{aligned}$$

respectively. It follows that

$$\begin{aligned}\|B\mathbf{x}\|^2+\chi^2\|F\mathbf{x}\|^2 &\le \|B\mathbf{z}\|^2+\chi^2\|F\mathbf{z}\|^2\\ \|B\mathbf{z}\|^2+\omega^2\|F\mathbf{z}\|^2 &\le \|B\mathbf{x}\|^2+\omega^2\|F\mathbf{x}\|^2\end{aligned}$$

and by adding

$$\chi^2\|F\mathbf{x}\|^2+\omega^2\|F\mathbf{z}\|^2 \le \chi^2\|F\mathbf{z}\|^2+\omega^2\|F\mathbf{x}\|^2$$

and since $\omega^2-\chi^2>0$

$$\|F\mathbf{z}\|^2 \le \|F\mathbf{x}\|^2\ ,$$

so that

$$\bar\omega=\varepsilon\, f(\|F\mathbf{z}\|^2) \le \varepsilon\, f(\|F\mathbf{x}\|^2)=\bar\chi\ .$$

This completes the proof, since f was chosen strictly increasing. One obvious choice for f is the identity function, which was used in our test programs.

8 Comparison of geometric algorithms

The discussion of algorithms minimizing the geometric distance is somewhat different from the algebraic distance problems. The problems in the latter are primarily stability and "good-looking" solutions; the former must be viewed by their efficiency, too. Generally, the simple algebraic solution is orders of magnitude cheaper than the geometric counterparts (for accurate results about a factor 10–100); thus iterative algebraic methods are a valuable alternative. But a comparison between algebraic and geometric algorithms would not be very enlightening—because there is no objective criterion to decide which estimate is better. However, we may compare different nonlinear least square algorithms to compute the geometric fit with respect to stability and efficiency.

8.1 Algorithms

Several known nonlinear least square algorithms have been implemented:

1. Gauss-Newton (`gauss`)
2. Newton (`newton`)
3. Gauss-Newton with Marquardt modification (`marq`)
4. Variable projection (`varpro`)
5. Orthogonal distance regression (`odr`)

The `odr` algorithm (see [13], [14]) solves the implicit minimization problem

$$\begin{aligned} f(\mathbf{x}_i + \delta_i, \beta) &= 0 \\ \sum_i \|\delta_i\|^2 &= \min \end{aligned}$$

where

$$f(\mathbf{x}, \beta) = \beta_3(x_1 - \beta_1)^2 + 2\beta_4(x_1 - \beta_1)(x_2 - \beta_2) + \beta_5(x_2 - \beta_2)^2 - 1 \; .$$

Whereas the `gauss`, `newton`, `marq` and `varpro` algorithms solve the problem

$$Q(\mathbf{x}, \varphi_1, \varphi_2, \ldots, \varphi_m, z_1, z_2, r) = \sum_{i=1}^{m} \Big[(x_{i1} - x(\varphi_i))^2 + (x_{i2} - y(\varphi_i))^2 \Big] = \min \; .$$

For $a \approx b$, the Jacobian matrix is nearly singular, so the `gauss` and `newton` algorithms are modified to apply Marquardt steps in this case. Not surprising, this modification makes all algorithms behave similar with respect to stability if the initial parameters are accurate and the problem is well posed.

The MATLAB procedure `pare` implements algorithms 1–3, the MATLAB procedure `varpro` the variable projection algorithm, and the MATLAB procedure `odr` the orthogonal distance regression. The algorithms are described in the appendix.

8.2 Results

To appreciate the results given in table 1, it must be said that the `varpro` algorithm is written for any separable functional and cannot take profit from the sparsity of the Jacobian. The algorithms were tested with example data—each consisting of 8 points—for following problems (the data sets are listed in appendix C.2)

1. Special set of points (14)
2. Uniformly distributed data in a square
3. Points on a circle
4. Points on an ellipse with $a/b = 2$
5. Points on hyperbola branch

	`gauss`	`newton`	`marq`	`varpro`	`odr`
Special	146	<u>85</u>	468	1146	◇
Random	◇	◇	◇	<u>2427</u>	◇
Circle	22	22	22	36	<u>7</u>
Circle+	86	<u>67</u>	189	717	69
Ellipse	<u>30</u>	37	67	143	41
Ellipse+	186	◇	633	1977	<u>103</u>
Hyperbola	22	22	22	36	<u>10</u>
Hyperbola+	◇	◇	◇	◇	◇

Table 1: Geometric fit with initial parameters of algebraic circle

#flops/1000, minimum is underlined
'◇' if non-convergence

The tests with points on a conic were done both with and without perturbations. For table 1, the initial parameters were derived from the algebraically best fitting circle (radius r_a, center z_a); initial center $z_0 = z_a$, axes $a_0 = r_a$, $b_0 = r_a$ and $\alpha_0 = 0$. Note that these initial values are somewhat rudimentary, so they serve to check the algorithms' stability, too. Table 1 shows the number of flops (in 1000) for the respective algorithm and problem; the smallest number is underlined. If the algorithm didn't terminate after 100 steps, it was assumed non-convergent and a '◇' is shown instead. Table 2 contains the results if the initial parameters were obtained from the Bookstein algorithm.

Table 2 shows that all algorithms converge quickly with the more accurate initial data for exact conics. For the perturbed ellipse data, it's primarily the `newton` algorithm which profits from the starting values close to the solution. Note further that the `newton` algorithm does not find the correct solution for the special data, since the algebraic estimation—which serves as initial approximation—is completely different from the geometric solution.

General conclusions from this (admittedly) small test series are

- All algorithms are prohibitively expensive compared to the simple algebraic solution (factor 10–100).
- If the problem is well posed, and the accuracy of the result should be high, the `newton` method applied to the parameterized algorithm is the most efficient.
- The `odr` algorithm—although a simple general-purpose optimizing scheme—is competitive with algorithms specifically written for the ellipse fitting problem. If one

	gauss	newton	marq	varpro	odr
Special	165	896	566	1506	◇
Random	◇	◇	◇	2819	◇
Circle	32	32	32	102	7
Circle+	76	63	145	574	66
Ellipse	22	22	22	112	7
Ellipse+	161	40	435	1870	74
Hyperbola	◇	◇	◇	1747	◇
Hyperbola+	◇	◇	◇	2986	◇

Table 2: Geometric estimation with initial data of algebraic ellipse

#flops/1000, minimum is underlined
'◇' if non-convergence

takes into consideration further, that we didn't use a highly optimized `odr` procedure, the method of solution is surprisingly simple and efficient.

- The `varpro` algorithm seems to be the most expensive. Reasons for its inefficiency are that most parameters are non-linear and that the algorithm does not make use of the special matrix structure for this problem.

References

[1] Vaughan Pratt, *Direct Least Squares Fitting of Algebraic Surfaces*, ACM J. Computer Graphics, Volume 21, Number 4, July 1987

[2] M. G. Cox, A. B. Forbes, *Strategies for Testing Assessment Software*, NPL Report DITC 211/92

[3] G. Golub, Ch. Van Loan, *Matrix Computations (second ed.)*, The Johns Hopkins University Press, Baltimore, 1989

[4] D. Sourlier, A. Bucher, *Normgerechter Best-fit-Algorithmus für Freiform-Flächen oder andere nicht-reguläre Ausgleichs-Flächen in Parameter-Form*, Technisches Messen 59, (1992), pp. 293–302

[5] H. Späth, *Orthogonal Least Squares Fitting with Linear Manifolds*, Numer. Math., 48:441–445, 1986.

[6] Lyle B. Smith, *The use of man-machine interaction in data fitting problems*, TR No. CS 131, Stanford University, March 1969 (thesis)

[7] Y. Vladimirovich Linnik, *Method of least squares and principles of the theory of observations*, New York, Pergamon Press, 1961

[8] Charles L. Lawson, *Contributions to the Theory of Linear Least Maximum Approximation*, Dissertation University of California, Los Angeles, 1961

[9] Fred L. Bookstein, *Fitting Conic Sections to Scattered Data*, Computer Graphics and Image Processing 9, (1979), pp. 56–71

[10] G. H. Golub, V. Pereyra, *The Differentiation of Pseudo-Inverses and Nonlinear Least Squares Problems whose Variables Separate*, SIAM J. Numer. Anal. 10, No. 2, (1973); pp. 413–432

[11] M. Heidari, P. C. Heigold, *Determination of Hydraulic Conductivity Tensor Using a Nonlinear Least Squares Estimator*, Water Resources Bulletin, June 1993, pp. 415–424

[12] W. Gander, U. von Matt, *Some Least Squares Problems*, Solving Problems in Scientific Computing Using Maple and Matlab, W. Gander and J. Hřebíček ed., 251-266, Springer-Verlag, 1993.

[13] Paul T. Boggs, Richard H. Byrd and Robert B. Schnabel, *A stable and efficient algorithm for nonlinear orthogonal distance regression*, SIAM Journal Sci. and Stat. Computing 8(6):1052–1078, November 1987.

[14] Paul T. Boggs, Richard H. Byrd, Janet E. Rogers and Robert B. Schnabel, *User's Reference Guide for ODRPACK Version 2.01—Software for Weighted Orthogonal Distance Regression*, National Institute of Standards and Technology, Gaithersburg, June 1992.

[15] P. E. Gill, W. Murray and M. H. Wright, *Practical Optimization*, Academic Press, New York, 1981.

[16] Gene H. Golub, Alan Hoffmann, G. W. Stewart, *A Generalization of the Eckart-Young-Mirsky Matrix Approximation Theorem*, Linear Algebra and its Appl. 88/89:317–327(1987)

A Algorithms

A.1 Gauss-Newton with Marquardt correction

Given parameters $\mathbf{x} = (\varphi_1, \ldots, \varphi_m, \alpha, a, b, z_1, z_2)^{\mathrm{T}}$, we consider the simplified equation of the Gauss-Newton method

$$J\mathbf{h} = \begin{pmatrix} -aS & A \\ bC & B \end{pmatrix} \mathbf{h} \approx \mathbf{x}. \tag{26}$$

Then, the equation for the same problem with Marquardt correction is

$$\begin{pmatrix} J \\ \Lambda \end{pmatrix} \mathbf{h} \approx \begin{pmatrix} \mathbf{x} \\ \mathbf{0} \end{pmatrix}. \tag{27}$$

If Λ is chosen diagonal (e.g. $\Lambda = \lambda I$), we can triangularize the first m colums of the extended system with $2m$ Givens rotations. Thus, as in the case of the Gauss-Newton algorithm, the full QR algorithm has to be applied only to the last 5 columns.

The λ_i parameter for the i^{th} step is determined as follows: $\lambda_i = \nu^k \lambda_{i-1}$ ($\nu > 1, k \geq -1$) where k is chosen minimal so that $\|\mathbf{r}_i\| \leq \|\mathbf{r}_{i-1}\|$ holds for the residual $\mathbf{r}$.

A.2 The Newton algorithm

Using the terminology of the previous section, the equations for one Newton step may be written as

$$(J^{\mathrm{T}}J + H)\mathbf{h} = -J^{\mathrm{T}}\mathbf{x}, \tag{28}$$

With parameter vector $\mathbf{x} = (\varphi_1, \ldots, \varphi_m, \alpha, a, b, z_1, z_2)^{\mathrm{T}}$, we may define H by

$$H_{jk} = \sum_i \mathbf{g}_i{}^{\mathrm{T}} \frac{\partial^2 \mathbf{g}_i}{\partial x_j \partial x_k} .$$

For completeness, we'll list the second derivatives of the residual functions $\mathbf{g}_i$:

$$\begin{aligned}
\frac{\partial^2 \mathbf{g}_i}{\partial \varphi_j \partial \varphi_k} &= \delta_{ij}\delta_{ik} Q(\alpha) \begin{pmatrix} a\cos\varphi_i \\ b\sin\varphi_i \end{pmatrix} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \varphi_j \partial \alpha} &= -\delta_{ij} \dot{Q}(\alpha) \begin{pmatrix} -a\sin\varphi_i \\ b\cos\varphi_i \end{pmatrix} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \varphi_j \partial a} &= -\delta_{ij} Q(\alpha) \begin{pmatrix} \sin\varphi_i \\ 0 \end{pmatrix} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \varphi_j \partial b} &= -\delta_{ij} Q(\alpha) \begin{pmatrix} 0 \\ \cos\varphi_i \end{pmatrix} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \varphi_j \partial z_k} &= \mathbf{0} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \alpha^2} &= Q(\alpha) \begin{pmatrix} a\cos\varphi_i \\ b\sin\varphi_i \end{pmatrix} \\
\frac{\partial^2 \mathbf{g}_i}{\partial \alpha \partial a} &= -\dot{Q}(\alpha) \begin{pmatrix} \cos\varphi_i \\ 0 \end{pmatrix}
\end{aligned}$$

$$
\begin{aligned}
\frac{\partial^2 \mathbf{g}_i}{\partial\alpha\partial b} &= -\dot{Q}(\alpha)\begin{pmatrix}0\\ \sin\varphi_i\end{pmatrix}\\
\frac{\partial^2 \mathbf{g}_i}{\partial\alpha\partial z_k} &= \mathbf{0}\\
\frac{\partial^2 \mathbf{g}_i}{\partial a^2} &= \frac{\partial^2 \mathbf{g}_i}{\partial a\partial b} = \frac{\partial^2 \mathbf{g}_i}{\partial a\partial z_k} = \frac{\partial^2 \mathbf{g}_i}{\partial b^2} = \frac{\partial^2 \mathbf{g}_i}{\partial b\partial z_k} = \frac{\partial^2 \mathbf{g}_i}{\partial z_j\partial z_k} = \mathbf{0}\ .
\end{aligned}
$$

The main disadvantage of the Newton method is that we have to solve a linear system with a non-sparse matrix, which is awfully conditioned for ill-posed problems.

A.3 `varpro`—The variable projection algorithm

This chapter only gives a survey on the algorithm; for a detailed description see [10]. The following problem is considered: Find optimal parameters $\hat{\mathbf{a}} = (\hat{a}_1, \ldots, \hat{a}_n)^{\mathrm{T}}$, $\hat{\alpha} = (\hat{\alpha}_1, \ldots, \hat{\alpha}_k)^{\mathrm{T}}$ that minimize the nonlinear functional

$$
r(\mathbf{a}, \alpha) = \sum_{i=1}^{m} \Big[y_i - \sum_{j=1}^{n} a_j \phi_j(\alpha; t_i) \Big]^2 . \tag{29}
$$

Let

$$
\{\Phi\}_{i,j} = \phi_j(\alpha; t_i), \qquad i = 1, \ldots, m; j = 1, \ldots, n.
$$

Then (29) can be written as

$$
r(\mathbf{a}, \alpha) = \|\mathbf{y} - \mathbf{\Phi}(\alpha)\mathbf{a}\|^2 . \tag{30}
$$

The idea is to minimize first a modified functional which depends only on the nonlinear parameters α, and then proceed to obtain the linear parameters $\mathbf{a}$. In order to obtain the separation of variables, the modified functional

$$
r_2(\alpha) = \|\mathbf{y} - \mathbf{\Phi}(\alpha)\mathbf{\Phi}^{+}(\alpha)\mathbf{y}\|^2 \tag{31}
$$

is considered, which is called the *variable projection functional.* Once optimal parameters $\hat{\alpha}$ have been obtained by minimizing (31), then the parameters $\hat{\mathbf{a}}$ are obtained as a solution of $\mathbf{\Phi}(\hat{\alpha})\mathbf{a} \approx \mathbf{y}$. This approach to finding a critical point requires an important hypothesis: not only must the involved functions be continuously differentiable, but the matrix $\mathbf{\Phi}(\alpha)$ must have constant rank for an open set containing the desired solution. If this condition does not hold, the pseudo-inverse of $\mathbf{\Phi}$ is a discontinuous function in the critical point.

For any given α we have the minimal least squares solution

$$
\hat{\mathbf{a}}(\alpha) = \mathbf{\Phi}^{+}(\alpha)\mathbf{y}. \tag{32}
$$

Thus,

$$
\min_{\mathbf{a}} r(\mathbf{a}, \alpha) = r(\hat{\mathbf{a}}, \alpha) = \|\mathbf{y} - \mathbf{\Phi}(\alpha)\mathbf{\Phi}^{+}(\alpha)\mathbf{y}\|^2 = r_2(\alpha). \tag{33}
$$

The modified functional is then the variable projection functional that we mentioned earlier. See [10] for a proof that—under the assumption mentioned above—($\hat{\mathbf{a}}$, $\hat{\alpha}$) is a minimizer for r if and only if $\hat{\alpha}$ is a minimizer for r_2. To apply the Gauss-Newton algorithm (possibly with Marquardt correction), it is necessary to differentiate the pseudo-inverse; this is the primary contribution of [10]. Further, the developed formulas allow to study the stability of the solution of perturbed linear least square problems.

A.4 `odr`—The Orthogonal Distance Regression algorithm

The `odr` algorithm provides an elegant method to the general solution of the problem

$$\begin{aligned} y_i + \epsilon_i &= f(\mathbf{x}_i + \delta_i, \beta) \\ \sum_i \|\delta_i\|^2 + \epsilon_i^2 &= \min \ , \end{aligned}$$

which may further be extended by a general weighting scheme. See [13] for details, and [14] for an implementation in FORTRAN. The form finally chosen results in the general ODR problem

$$\min_{\beta,\delta} \sum_i w_i^2 \left[(f(\mathbf{x}_i + \delta_i, \beta) - y_i)^2 + {\delta_i}^{\mathrm{T}} D_i^2 \delta_i \right] \tag{34}$$

where $w_i > 0$, and $D_i = \mathrm{diag}(d_{ij})$ where $d_{ij} > 0$.

The `odr` algorithm solves the problem (34) using Levenberg-Marquardt. The number of unknowns involved is the number of model parameters plus the number of data points. By exploiting sparsity, however, the algorithm has a per step computational effort similar to the Levenberg-Marquardt method for ordinary least squares.

The implicit problem

$$\begin{aligned} 0 &= f(\mathbf{x}_i + \delta_i, \beta) \\ \sum_i \|\delta_i\|^2 &= \min \ , \end{aligned}$$

may be solved by a penalty function method (see e.g. [15]). The penalty function is

$$P(\beta, \delta; r_k) = \sum_i r_k f(\mathbf{x}_i + \delta_i, \beta)^2 + {\delta_i}^{\mathrm{T}} D_i^2 \delta_i$$

with penalty parameter r_k. A sequence of unconstrained minimization problems

$$\min_{\beta,\delta} P(\beta, \delta; r_k)$$

for a sequence of values $\{r_k\}$ results in weighted orthogonal distance regression problems (which are explicit)

$$\epsilon_i = f(\mathbf{x}_i + \delta_i, \beta)$$

and their solutions approach for $r_k \to \infty$ the solution of

$$0 = f(\mathbf{x}_i + \delta_i, \beta).$$

To apply this algorithm to the ellipse fitting problem, we may use the algebraic equation

$$f(\beta) = \beta_3(x - \beta_1)^2 + 2\beta_4(x - \beta_1)(y - \beta_2) + \beta_5(y - \beta_2)^2 - 1,$$

which is simple to differentiate and does not require trigonometric functions.

B MATLAB Implementation

Various small MATLAB functions have been written for this paper, documented herein to provide an easy way to reproduce the results.

B.1 Utilities

Linear least squares problem with special constraint

```
function [c, n] = clsq (A, dim);
%CLSQ   Special constrained least squares
%
%       [c, n] = clsq (A, dim) solves the constrained
%       least squares Problem
%       A (c n)' == 0 subject to norm(n,2)=1
%       dim=length(n)

  [m,p] = size(A);
  if p < dim+1, error ('not enough unknowns'); end;
  if m < dim, error ('not enough equations'); end;
  m = min (m, p);
  R = triu (qr (A));
  [U,S,V] = svd(R(p-dim+1:m,p-dim+1:p));
  n = V(:,dim);
  c = -R(1:p-dim,1:p-dim)\R(1:p-dim,p-dim+1:p)*n;

end % clsq
```

Draw circle

```
function drawcircle (z, r, pat, OPTIONS);
%DRAWCIRCLE Draw circle
%
% drawcircle (z, r, pat{'-'}, OPTIONS{[]})
% draws a circle into the current figure.
%
% z, r: center and radius of circle
% pat: pattern to be used (e.g. '--' for dashed)

  if (nargin < 4), OPTIONS = [2]; end;
  if (nargin < 3), pat = '-'; end;

  theta = [0:0.02:2*pi];
  u = z(1) + r*cos(theta);
  v = z(2) + r*sin(theta);
  plot(u, v, pat);
  if (find(OPTIONS==2) == []),
    plot(z(1),z(2),'+');
  end

end % drawcircle
```

Draw ellipse

```
function drawellipse (z, a, b, alpha, pat, OPTIONS)
%DRAWELLIPSE Draw ellipse
%
% drawellipse (z, a, b, alpha, pat{'-'}, OPTIONS{[]})
% draws ellipse into current figure.
%
% z, a, b, alpha: parameters of ellipse
% pat: pattern to be used

  if (nargin < 6), OPTIONS = [2]; end;
  if (nargin < 5), pat = '-'; end;
```

```
  s = sin(alpha); c = cos(alpha);
  Q =[c -s; s c];
  theta = [0:0.02:2*pi];
  u = diag(z)*ones(2,length(theta)) + ...
      Q*[a*cos(theta); b*sin(theta)];
  plot(u(1,:),u(2,:), pat);
  if (find(OPTIONS==2) == []),
    plot(z(1),z(2),'+');
  end
  if (find(OPTIONS==1) ~= []),
    if (a < b),
      alpha = alpha + pi/2;
      tmp = a; a = b; b = tmp;
    end
    alpha = alpha - pi*floor(alpha/pi);
    at = text(z(1),z(2),'  a');
    at2 = text(z(1)+1,z(2),'  a');
    set(at2,'Visible','off');
    ext = get(at, 'Extent');
    xa2  = ext(1);
    set(at, 'FontName', 'Symbol');
    ext = get(at, 'Extent');
    wa  = ext(3);
    xa  = ext(1);
    bt  = text(z(1)+(xa2-xa)/wa,z(2),sprintf(' = %1.4f',alpha));
  end

end % drawellipse
```

Compute angles for Givens-Rotation matrix

```
function [c, s] = rot_cossin (x, y);
%ROT_COSSIN     Givens rotation angles
%
% [c, s] = rot_cossin (x, y);
% returns cos and sin vectors for Givens-rotation matrix
% which rotates y to zero.
%
% x, y: vectors
% c(i), s(i): [c(i) -s(i); s(i) c(i)]*[x(i); y(i)] == [..; 0]

  m = size(x,1);
  c = zeros(m,1); s = zeros(m,1);
  for i=1:m,
    if (abs(y(i)) > abs(x(i))),
      cot = -x(i)/y(i); si = 1/sqrt(1+cot^2); co = si*cot;
    else
      tan = -y(i)/x(i); co = 1/sqrt(1+tan^2); si = co*tan;
    end
    s(i) = si; c(i) = co;
  end

end % rot_cossin
```

Compute nearest points on ellipse to given points

```
function phi = ellipse_phi (X, a, b, phi, myeps);
%ELLIPSE_PHI    Compute nearest points to ellipse
%
% phi = ellipse_phi (X, a, b, phi{}, myeps{sqrt(myeps)});
```

```
% compute angles for nearest points on ellipse to given points X
%
% X: given points <X(i,1),X(i,2)>
% a, b: ellipse parameters with a-axis in x-coordinate
% phi: starting values for iteration
% myeps: convergence limit
%
% phi: phi(i) angle of point on ellipse (in parametric form)
%      nearest to point <X(i,1), X(i,2)>

  if (nargin < 5), myeps = sqrt(eps); end;
  if (nargin < 4), phi = []; end;
  if (phi == []),  phi = angle (X(:,1)/a + sqrt(-1)*X(:,2)/b); end;

  m = size(X,1);
  for i = 1:m,
    par = [b^2 - a^2; a*X(i,1); b*X(i,2)];
    ni  = phi(i);
    step = 0;
    while (1),
      c = cos(ni);
      s = sin(ni);
      dni = - (par(1)*s*c + par(2)*s - par(3)*c) / ...
              (par(1)*(c^2 - s^2) + par(2)*c + par(3)*s);
      while ((X(i,1) - a*c)^2 + (X(i,2) - b*s)^2) < ...
            ((X(i,1) - a*cos(ni+dni))^2 + (X(i,2) - b*sin(ni+dni))^2),
        dni = -0.2*dni;
      end
      ni = ni + dni;
      if (abs(dni) <= sqrt(eps)) break; end;
      step = step + 1;
      if (step > 40),
        disp ('warning: no convergence');
        str = sprintf ('angle, diff-angle = %g, %g', ni, dni);
        disp (str);
        break;
      end
    end
    phi(i) = ni;
  end % for all points

end % ellipse_phi
```

Compute distance vectors from given points to ellipse

```
function [Y] = ellipse_residual (X, z, a, b, alpha);
%ELLIPSE_RESIDUAL       Computes the residual vector "X - ellipse"
%
% [Y] = ellipse_residual (X, z, a, b, alpha);
% computes the residual vector "X - ellipse" in the
% transformed system (rotated by -alpha).
%
% X: given points <X(i,1), X(i,2)>
% z, a, b, alpha: ellipse parameters
%
% Y: residual vector "X - ellipse" for the transformed system.
%    <Y(i), Y(i+m)> difference vector for i-th point (m == nofpoints)

  s = sin(alpha);
  c = cos(alpha);
  Q = [c -s; s c];

  x = [X(:,1) - z(1), X(:,2) - z(2)]*Q;
  phi = ellipse_phi (x, a, b);
```

```
  Y = x - [a*cos(phi), b*sin(phi)];
  Y = [Y(:,1); Y(:,2)];

end % ellipse_residual
```

Find ellipse parameters from algebraic equation

```
function [z, a, b, alpha, err] = ellipse_params (u, show);
%ELLIPSE_PARAMS Get ellipse params from algebraic equation
%
%       [z, a, b, alpha, err] = ellipse_params (u, show{0});
%       get the ellipse parameters
%       from algebraic equation
%         u(1)x^2 + u(2)xy + u(3)y^2 + ...
%         u(4)x + u(5)y + u(6) = 0.
%
%       u: coefficients of algebraic equation
%       show: == 1, then plot figure if error.
%
%       z, a, b, alpha: ellipse parameters
%       err: != 0, if not an ellipse

  if (nargin < 2) show = 0; end;
  err = 0;

  A   = [u(1) u(2)/2; u(2)/2 u(3)];
  bb  = [u(4); u(5)];
  c   = u(6);

  [Q D] = eig(A);
  det   = D(1,1)*D(2,2);
  if (det <= 0),
    err = 1;
    if (show == 1), drawconic (u); end;
    z = [0;0];
    a = 1; b = 1; alpha = 0;
  else
    bs    = Q'*bb;
    alpha = atan2(Q(2,1), Q(1,1));
    zs    = -(2*D)\bs;
    z     = Q*zs;
    h     = -bs'*zs/2-c;
    a     = sqrt(h/D(1,1));
    b     = sqrt(h/D(2,2));
  end

end % ellipse_params
```

B.2 Circle estimation

Algebraic circle solution

```
function [z, r] = algcircle (X);
%ALGCIRCLE      Algebraic circle fit
%
% [z, r] = algcircle (X);
% fits a circle by minimizing the "algebraic distance"
% in the least squares sense  a x'x + b'x + c = 0
%
% X : given points <X(i,1, X(i,2)>
```

```
%
% z, r: center and radius of the found circle

  B  = [ X(:,1).^2+X(:,2).^2  X(:,1)  X(:,2) ones(size(X(:,1)))];
  [U S V]= svd(B);
  u = V(:,4); a = u(1); b =[u(2); u(3)]; c = u(4);
  z = -b/2/a; r = sqrt(norm(z)^2 - c/a);

end % algcircle
```

Geometric circle solution in explicit form

```
function [z, r] = circle (X, z, r);
%CIRCLE         Geometric circle fit
%
% [z,r] = circle (X, z, r)
% fits the best circle by nonlinear least squares
% for true geometric distance.
%
% X: given points <X(i,1), X(i,2)>
% z, r: starting values for ellipse solution
%
% z, r: parameters for ellipse found

  u = [z(1), z(2) , r]'    % Starting values
  h = u;
  while norm(h)>norm(u)*1e-6,
    a = u(1)-X(:,1); b = u(2)-X(:,2);
    fak = sqrt(a.*a + b.*b);
    J = [a./fak b./fak -ones(size(a))];
    f = fak -u(3);
    h = -J\f;
    u = u + h;
  end;
  z = u(1:2); r = u(3);

end % circle
```

Geometric circle solution in parametric form

```
function [z, r, phi, step] = parcircle (X, z, r, show);
%PARCIRCLE      Geometric circle fit
%
% [z, r, phi, step] = parcircle (X, z, r, show{0});
% computes the best fit circle in parameterform
% x = z(1) + r  cos(phi), y = z(2) + r sin(phi)
%
% X: given points <X(i,1), X(i,2)>
% z, r: starting values for circle
% show: if (show == 1), test output
%
% z, r: circle found
% phi: phi(i) angle to nearest point on circle
% step: nof iterations

  m = size(X,1);

  % compute initial approximations for phi_i
  phi  = angle(X(:,1)-z(1)+ i*(X(:,2)-z(2)));
  step = 0;
  h    = 1;
  while (norm(h) > 1e-5),
```

```
    step  = step+1;
    if (step > 100),
      disp ('warning: number of iterations exceeded limit');
      break;
    end
  % form Jacobian
    S =  diag(sin(phi));
    C =  diag(cos(phi));
    A = [ -ones(size(phi)) zeros(size(phi)) -cos(phi)];
    B = [ zeros(size(phi)) -ones(size(phi)) -sin(phi)];
    J = [r*S A; -r*C B];

  % QR decomposition of Jacobian
    Q = [S C; -C S];
    [U R] = qr(C*A+S*B);
    RR =  R(1:3, 1:3);
    RRR = [r*eye(m) S*A-C*B; zeros(size(A')) RR];
  % transform right hand  side
    Y = [X(:,1)-z(1)-r*cos(phi); X(:,2)-z(2)-r*sin(phi)];
    Y = [eye(m) zeros(m); zeros(m) U]'*Q'*Y;
  % solve triangular system
    h = -RRR\Y(1:m+3);

  % update solution
    phi = phi + h(1:m);
    z = z +  h(m+1:m+2);
    r = r+h(m+3);

    if (show == 1),
      drawcircle (z,r);
      [z' r phi']
    end
  end

end % parcircle
```

B.3 Algebraic ellipse estimation

SVD algebraic ellipse estimation

```
function [z, a, b, alpha, err] = algellipse (X, W, show);
%ALGELLIPSE     Algebraic least square ellipse fit
%
%       [z, a, b, alpha, err] = ...
%         algellipse (X, W{default ones}, show{default 0})
%       fits an ellipse by minimizing the "algebraic distance"
%       in the least squares sense x'A x + b'x + c = 0
%       weighting the i-th data by W(i)
%
%       X: given points Pi = [X(i,1), X(i,2)]
%       W: weight W(i) for the i-th equation
%       show: if (show == 1) make test output
%
%       z, a, b, alpha: parameters for found ellipse
%       err: error indication
%         if (err == 1), not an ellipse
%         if (err == 0), ok

  if (nargin < 2), W = ones(size(X,1), 1); end;
  if (nargin < 3), show = 0; end;

  [U S V] = svd(diag(W) * [X(:,1).^2  X(:,1).*X(:,2) X(:,2).^2 ...
```

```
                          X(:,1) X(:,2) ones(size(X(:,1)))];
  u   = V(:,6);

  [z, a, b, alpha, err] = ellipse_params (u, show);

end % algellipse
```

Linear algebraic ellipse estimation

```
function [z, a, b, alpha, err] = alge_simple (X, show);
%ALGE_SIMPLE    Algebraic least squares ellipse fit.
%
%       [z, a, b, alpha, err] = alge_simple (X, show{default 0})
%       fits an ellipse by minimizing the "algebraic distance"
%       in the least squares sense x'A x + b'x + c = 0
%       weighting the i-th data by W(i).
%       Constraint: A11 + A22 == 1.
%
%       X: given points Pi = [X(i,1), X(i,2)]
%       show: if (show == 1) make test output
%
%       z, a, b, alpha: parameters for found ellipse
%       err: error indication
%       if (err == 1), not an ellipse
%       if (err == 0), ok

  if (nargin < 2), show = 0; end;

  b = -X(:,1).^2;
  A = [X(:,1).*X(:,2), X(:,2).^2-X(:,1).^2 ...
       X(:,1) X(:,2) ones(size(X(:,1)))];
  x = A\b;
  u = [1-x(2);x];

  [z, a, b, alpha, err] = ellipse_params (u, show);

end % alge_simple
```

Bookstein algorithm

```
function [z, a, b, alpha] = bookstein (X, show);
%BOOKSTEIN       Algebraic ellipse fit
%
% [z, a, b, alpha] = bookstein (X, show{0});
%
% Approximate ellipse to points <X(i,1),X(i,2)>.
% Invariant under euclidian transformation, see
% BOOKSTEIN, "Fitting conic secions to scattered data", in
% Computer graphics & image processing 9, 56-71 (1979).
%
% X: given points <X(i,1),X(i,2)>
% show: if (show == 1), test output
%
% z, a, b, alpha: parameters for ellipse found

  if (nargin < 2), show = 0; end;

  m      = size(X,1);
  A      = [X(:,1).^2  X(:,1).*X(:,2) X(:,2).^2 ...
            X(:,1) X(:,2) ones(size(X(:,1)))];

  S = A'*A;
```

```
  T = S(1:3,1:3) - S(1:3,4:6)*(S(4:6,4:6)'\S(4:6,1:3));
  T = diag([1,2,1])*T;
  [V, D] = eig(T);

  emin = 0;
  kmin = 0;
  for k = 1:3,
    A = V(1,k); B = V(2,k); C = V(3,k);
    I0 = (A + C);
    I1 = (A*C - B^2/4);
    if (I1 <= 0),
      % this is not an ellipse !
    else
      val   = (I0^2 - 4*I1)/(I0^2 - 2*I1);
      if (emin == 0 | val < emin),
        emin = val;
        kmin = k;
      end
    end
  end
  if (kmin == 0), kmin = 1; end; % not an ellipse
  y1 = V(:,kmin);
  y2 = -(S(4:6,4:6)')\(S(1:3,4:6)'*y1);
  u  = [y1; y2];

  [z, a, b, alpha, err] = ellipse_params (u, show);

end % bookstein
```

SVD solution for Bookstein constraint

```
function [z, a, b, alpha] = bookstein_svd (X, show);
%BOOKSTEIN_SVD  Algebraic ellipse fit
%
% [z, a, b, alpha] = bookstein_svd (X, show{0});
%
% Approximate ellipse to points <X(i,1),X(i,2)>.
% Invariant under euclidian transformation, see
% BOOKSTEIN, "Fitting conic secions to scattered data", in
% Computer graphics & image processing 9, 56-71 (1979).
% unlike BOOKSTEIN, SVD is used for solution.
%
% X: given points <X(i,1),X(i,2)>
% show: if (show == 1), test output
%
% z, a, b, alpha: parameters for ellipse found

  if (nargin < 2), show = 0; end;

  m      = size(X,1);
  AA     = [ X(:,1) X(:,2) ones(size(X(:,1))) ...
             X(:,1).^2  sqrt(2)*X(:,1).*X(:,2) X(:,2).^2];

  [d, a] = clsq (AA, 3);
  u = [a(1); sqrt(2)*a(2); a(3); d];

  [z, a, b, alpha, err] = ellipse_params (u, show);

end % bookstein_svd
```

Curvature weights

```
function [z, a, b, alpha, step] = lyle (X, show);
%LYLE   Iterative algebraic ellipse fit
%
% [z, a, b, alpha, step] = lyle (X, show{0});
% fit ellipse with algebraic method using curvature weights.
%
% X: points given <X(i,1), X(i,2)>
% show: if (show == 1), test output
%
% z, a, b, alpha: ellipse found
% step: nof iterations

  delta = 1;
  omega = 2.0;
  myeps = 1e-3;

  new = [0;0;0;0;0];
  old = new;
  W = ones(size(X,1), 1);
  step = 0;
  m = size(X, 1);

  while (delta > myeps),
    step  = step + 1;
    if (step > 20),
      disp ('warning: number of steps exceeded limit');
      break;
    end
    myeps = omega*myeps;
    [z, a, b, alpha, err] = algellipse (X, W, show);
    if (err),
      disp ('warning: found non-ellipse');
      break;
    end
    new = [z; a; b; alpha];
    if (step == 1),
      delta = 1;
    else
      delta = norm (new - old);
    end
    old = new;

    c = cos(alpha); s = sin(alpha);
    Q = [c -s; s c];
    % compute initial approximations for phi_i
    du  = Q'*[ X(:,1)-z(1) X(:,2)-z(2)]';
    phi = angle(du(1,:)/a  + sqrt(-1)*du(2,:)/b)';

    %% weights is inverse of ellipse radius
    C = cos(phi); S = sin(phi);
    W = (a*b) ./ (a^2*S.^2 + b^2*C.^2).^(3/2);

  end % while

  z  = new(1:2);
  a  = new(3);
  b  = new(4);
  alpha = new(5);

end % lyle
```

Geometric weights

```
function [z, a, b, alpha, step] = wate2 (X, show);
```

```
% [z, a, b, alpha, step] = wate2 (X, show{0});
% fit ellipse with algebraic method using geometric distance weights.
%
% X: given points <X(i,1), X(i,2)>
% show: if (show), test output
%
% z, a, b, alpha: ellipse found
% step: nof iterations

if (nargin < 2), show = 0; end;

delta = 1;
omega = 0;
myeps = 1e-6;

m     = size(X, 1);
sumX  = sum(X)/m;
X     = X - ones(m,1)*sumX;
normX = norm(X, inf);
X     = X/normX;

oldu = zeros(6,1);
W1   = [ones(size(X,1), 1)];
A1   = [X(:,1).^2  X(:,1).*X(:,2) X(:,2).^2 ...
        X(:,1) X(:,2) ones(size(X(:,1)))];
A2   = [1 0 -1 0 0 0; ...
        0 1  0 0 0 0];
step = 0;

while (1), %% breaks

  [U S V] = svd([diag(W1) * A1; omega * norm(W1,inf) * A2]);
  u = V(:,6);
  if (norm (oldu - u) <= myeps), break; end;

  [z, a, b, alpha, err] = ellipse_params (u);
  if (err),
    %% the result is not an ellipse, adjust weight !
    disp ('warning: weighted');
    if (omega == 0), omega = 1;
    else             omega = 4*omega;
    end
  else
    oldu = u;

    c = cos(alpha); s = sin(alpha);
    Q = [c -s; s c];
    % compute initial approximations for phi_i
    du  = Q'*[ X(:,1)-z(1) X(:,2)-z(2)]';
    phi = angle(du(1,:)/a  + sqrt(-1)*du(2,:)/b)';
    du  = du';

    phi_geom = ellipse_phi (du, a, b, phi);

    %% weights is "real distance / algebraic weight"
    C      = cos(phi);      S      = sin(phi);
    C_geom = cos(phi_geom); S_geom = sin(phi_geom);

    D_geom = sqrt ((du(:,1) - a*C_geom).^2 + (du(:,2) - b*S_geom).^2);
    D      = (du(:,1).^2 + du(:,2).^2) ./ (a^2*C.^2 + b^2*S.^2) - ...
             ones (size(du,1), 1);
    W1     = D_geom./D;

  end % if (is ellipse ?)

  step = step + 1;
```

```
    if (step > 20),
      disp ('warning: number of steps exceeeded limit');
      break;
    end
  end % while

  z = z*normX;
  a = a*normX;
  b = b*normX;
  z = z + sumX';

  if (show == 3),
    M = abs((diag(W1)*(A1*u))./D_geom);
    M = M/norm(M,-inf);
    disp ('output of M = wQ/g, M = M/norm(M), norm(M - 1)');
    norm (M - 1)
  end

end % wate2
```

B.4 Geometric ellipse estimation

B.4.1 Utilities

Parameter vector to parameter translation

```
function [phi, alpha, a, b, z] = pare_get (x);
%PARE_GET       Vector to param conversion
%
% [phi, alpha, a, b, z] = pare_get (x);
% gets single parameters from param vector
%
% x: x == [phi; alpha; a; b; z]
%
% phi, alpha, a, b, z: splitted parameters

  m = size(x, 1) - 5;
  phi   = x(1:m);
  alpha = x(m+1);
  a     = x(m+2);
  b     = x(m+3);
  z     = x(m+4:m+5);

end % pare_get
```

Parameter to parameter vector translation

```
function x = pare_set (x, phi, alpha, a, b, z);
%PARE_SET       Param to vector conversion
%
% x = pare_set (x, phi, alpha, a, b, z);
% stores single parameters into param vector
%
% x: x == previous values of [phi; alpha; a; b; z]
% phi, alpha, a, b, z:
%   if (<val> != []), then set this param
%
% x: updated param vector

  m = size(x, 1) - 5;
```

```
  if (~isempty(phi)),   x(1:m) = phi; end;
  if (~isempty(alpha)), x(m+1) = alpha; end;
  if (~isempty(a)),     x(m+2) = a; end;
  if (~isempty(b)),     x(m+3) = b; end;
  if (~isempty(z)),     x(m+4:m+5) = z; end;

end % pare_set
```

Set initial angle estimation

```
function x = pare_initphi (X, x);
%PARE_INITPHI
%
%       x = pare_initphi (X, x);
%       assigns approximate values for angles
%       relative to the transformed system.
%
%       X: given points <X(i,1), X(i,2)>
%       x: parameters
%
%       x: parameters, with 'phi' values approximate
%       nearest point angles

  [phi, alpha, a, b, z] = pare_get (x);

  c = cos(alpha); s = sin(alpha);
  Q = [c -s; s c];
  % compute initial approximations for phi_i
  du  = Q'*[ X(:,1)-z(1) X(:,2)-z(2)]';
  phi = angle(du(1,:)/a  + sqrt(-1)*du(2,:)/b)';

  x = pare_set (x, phi, [], [], [], []);

end % pare_initphi
```

Computing the residual vector

```
function res = pare_residual (X, x);
%PARE_RESIDUAL  Residual vector for ellipse
%
% res = pare_residual (X, x);
% compute residual vector in the transformed system
% for parameter ellipse.
%
% X: given points <X(i,1), X(i,2)>
% x: ellipse parameters
%
% res: residual "X - ellipse" for the transformed system.
%   <res(i), res(m+i)> is residual for i-th point (m == nofpoints).

  [phi, alpha, a, b, z] = pare_get (x);

  s = sin(alpha);
  c = cos(alpha);
  Q = [c -s; s c];

  Xs = X*Q;
  zs = Q'*z;
  res = [Xs(:,1)-zs(1)-a*cos(phi); Xs(:,2)-zs(2)-b*sin(phi)];

end % pare_residual
```

QR algorithm with rank determination

```
function  [Q, R, P, r] = varpro_qr (X, defeps);
%VARPRO_QR      Special QR decomposition
%
% [Q, R, P, r] = varpro_qr (X, defeps);
% makes Q*X*P == R ( == [R1(r,r), R2(r,n-r); 0(m-r,r), 0(m-r,n-r)]).
%
% defeps: limit for rank-deficiency
%
% r: rank of R (with respect to defeps)

  [m, n] = size(X);
  [Q, R, P] = qr(X);

  normr = norm(R);
  r = 1;
  while (r <= min(m,n)),
    if (norm(R(r,r:n)) < defeps*normr), break; end;
    r = r + 1;
  end
  r = r - 1;

  Q = Q';

end % varpro_qr
```

B.4.2 Gauss-Newton, Newton and Marquardt algorithms

Gauss-Newton iteration step

```
function [x, lambda] = pare_gauss_step(X, x, lambda);
%PARE_GAUSS_STEP        Gauss-Newton iteration step
%
% [x, lambda] = pare_gauss_step(X, x, lambda);
% makes basic step for this x. Adds marquart correction if
% abs(a - b)/(a + b) < lambda(1).
%
% X: given points <X(i,1), X(i,2)>
% x: given parameters
% lambda: lambda(1) marquardt correction factor
%
% x: updated parameters
% lambda: (possibly new) marquardt factor

  [phi, alpha, a, b, z] = pare_get (x);

  if (abs(a - b)/(a + b) < lambda(1)),
    [x, lambda] = pare_marq_step(X, x, lambda);
  else
    m = size(X,1);
    Y = pare_residual (X, x);

  %% form Jacobian
    S = sin(phi);
    C = cos(phi);
    s = sin(alpha);
    c = cos(alpha);
    A = [-b*S C zeros(size(phi))  c*ones(size(phi)) s*ones(size(phi))];
    B = [ a*C zeros(size(phi)) S -s*ones(size(phi)) c*ones(size(phi))];
```

```
    [cg, sg] = rot_cossin (-a*S, b*C);
    G = sparse ([diag(cg), -diag(sg); diag(sg), diag(cg)]);
    Y = G*Y;

    D = diag (- a*S.*cg - b*C.*sg);
    J = [[D; zeros(m,m)], G*[A; B]];

    [qq, J(m+1:2*m, m+1:m+5)] = qr(J(m+1:2*m, m+1:m+5));
    Y(m+1:m+5, :) = qq(:,1:5)'*Y(m+1:2*m,:);

    h = J(1:m+5, 1:m+5)\Y(1:m+5);

    x = x + h;

  end % if (marq term necessary)

end % pare_gauss_step
```

Newton iteration step

```
function [x, lambda] = pare_newton_step (X, x, lambda);
%PARE_NEWTON_STEP       Newton iteration step
%
% [x, lambda] = pare_newton_step (X, x, lambda);
% makes basic step for this x. Adds marquart correction if
% abs(a - b)/(a + b) < lambda(1).
%
% X: given points <X(i,1), X(i,2)>
% x: given parameters
% lambda: lambda(1) marquardt correction factor
%
% x: updated parameters
% lambda: (possibly new) marquardt factor

  [phi, alpha, a, b, z] = pare_get (x);

  if (abs(a - b)/(a + b) < lambda(1)),
    [x, lambda] = pare_marq_step(X, x, lambda);
  else
    m = size(X,1);
    ALPHA = m + 1;
    A     = m + 2;
    B     = m + 3;

    Y = pare_residual (X, x);
    YY = [Y(1:m, 1), Y(m+1:2*m, 1)];

  %% form Jacobian
    S = sin(phi);
    C = cos(phi);
    s = sin(alpha);
    c = cos(alpha);
    JA = [-b*S C zeros(size(phi))  c*ones(size(phi)) s*ones(size(phi))];
    JB = [ a*C zeros(size(phi)) S -s*ones(size(phi)) c*ones(size(phi))];

    DA = -a*sparse(diag(S));
    DB = b*sparse(diag(C));

    H  = zeros(m+5, m+5);
    for i = 1:m, H(i,i)     = [a*C(i), b*S(i)]*YY(i,:)'; end;
    for i = 1:m, H(i,ALPHA) = [b*C(i), a*S(i)]*YY(i,:)'; end;
    H(1:m,A)                = S.*YY(:,1);
    H(1:m,B)                = -C.*YY(:,2);
    H(ALPHA, ALPHA)         = [a*C', b*S']*Y;
```

```
    H(ALPHA, A)             = C'*YY(:,2);
    H(ALPHA, B)             = -S'*YY(:,1);
    H = H + triu(H, 1)';

    DD = a^2*S.^2 + b^2*C.^2;
    J1 = DA*JA + DB*JB;
    J2 = [diag(DD), J1; J1', JA'*JA + JB'*JB];
    Y2 = [DA*Y(1:m) + DB*Y(m+1:2*m); JA'*Y(1:m) + JB'*Y(m+1:2*m)];
    s = (J2 + H)\Y2;

    x = x + s;

  end % if (marq term necessary)

end % pare_newton_step
```

Gauss-Newton iteration step with Marquardt modification

```
function [x, lambda] = pare_marq_step(X, x, lambda);
%PARE_MARQ_STEP        Gauss-Newton step with Marquardt
%
% [x, lambda] = pare_marq_step(X, x, lambda);
% makes basic step for this x with marquardt correction.
%
% X: given points <X(i,1), X(i,2)>
% x: given parameters
% lambda: lambda(i) i-th previous marquardt correction factor
%
% x: updated parameters
% lambda: updated marquardt factors

  [phi, alpha, a, b, z] = pare_get (x);

  mu      = 1e-3;
  omega   = 0.5;

  W = ones(size(x,1), 1);
  m = size(X,1);
  Y = pare_residual (X, x);

%% form Jacobian
  S = sin(phi);
  C = cos(phi);
  s = sin(alpha);
  c = cos(alpha);
  A = [-b*S C zeros(size(phi))  c*ones(size(phi)) s*ones(size(phi))];
  B = [ a*C zeros(size(phi)) S -s*ones(size(phi)) c*ones(size(phi))];

  [cg, sg] = rot_cossin (-a*S, b*C);
  G = sparse([diag(cg), -diag(sg); diag(sg), diag(cg)]);
  Y = G*Y;

  D  = - a*S.*cg - b*C.*sg;
  AB = G*[A; B];

%% do marquart step
  istep = 0;
  while (1),
    [cg, sg] = rot_cossin (D, lambda(1)*ones(m,1));
    DD = D.*cg - lambda(1)*sg;
    AA = sparse(diag(cg))*AB(1:m,:);
    YY = [sparse(diag(cg))*Y(1:m,:); Y(m+1:2*m,:); ...
          sparse(diag(sg))*Y(1:m,:)];
    BB = [AB(m+1:2*m,:); sparse(diag(sg))*AB(1:m,:); ...
```

```
          lambda(1)*eye(5,5)];
    [qq, RR] = qr(BB);
    YY = [YY(1:m,:); qq(1:2*m,1:5)'*YY(m+1:3*m,:)];
    JJ = [diag(DD), AA; zeros(5,m), RR(1:5,:)];
    s  = JJ\YY;

    h = norm(Y) - norm(pare_residual (X, x + s));
    if (h >= 0), break; end;
    lambda(1) = lambda(1)/omega;
    istep = istep + 1;
  end % while
  if (istep == 0),
    lambda(1) = lambda(1)*omega;
  end

  x = x + s;

end % pare_marq_step
```

Geometric estimation loop

```
function [z, a, b, alpha, phi, step] = ...
         pare (X, z, a, b, alpha, meth, show);
%PARE   Geometric ellipse fit loop
%
% [z, a, b, alpha, phi, step] = ...
%      pare (X, z, a, b, alpha, meth, show{0});
% computes the best fit ellipse in parameterform
% x = z(1) + a  cos(phi-alpha), y = z(2) + b sin(phi-alpha)
%
% X: given points <X(i,1), X(i,2)>
% z, a, b, alpha: starting values for ellipse
% meth:
%   0 --> gauss-newton with marquardt for a near b
%   1 --> newton with marquardt for a near b
%   2 --> marquardt
%   3 --> gauss-newton
% show: if (show == 1), test output
%
% z, a, b, alpha: ellipse found
% phi: values for the nearest points (in parametric form)
% step: nof iterations

  if (nargin < 7), show = 0; end;
  if (nargin < 6), meth = 1; end;

  epsr  = 1e-5;

  m = size (X, 1);
  x = zeros (m+5, 1);
  x = pare_set (x, phi, alpha, a, b, z);
  x = pare_initphi (X, x);

  step   = 0;
  normr  = 1;
  norma  = 1;
  lambda = [1;1;1];

  while (normr > epsr*norma),

    if     (meth == 0), [x, lambda] = pare_gauss_step (X, x, lambda);
    elseif (meth == 1), [x, lambda] = pare_newton_step (X, x, lambda);
    elseif (meth == 2), [x, lambda] = pare_marq_step (X, x, lambda);
    elseif (meth == 3), [x, lambda] = pare_gauss_step (X, x, [0;0;0]);
```

```
    else                error ('unknown meth');
    end

    if (step > 0),
      normr = norm (x - prevx);
      norma = norm (x);
    end

    prevx = x;
    step  = step+1;

    if (show == 1),
      [phi, alpha, a, b, z] = pare_get (x);
      drawellipse(z, a, b, alpha)
    end

    if (step > 100),
      disp ('warning: number of steps exceeded limit');
      break;
    end

  end % while

  [phi, alpha, a, b, z] = pare_get (x);

end % pare
```

B.4.3 The varpro algorithm

The general `varpro` procedure

```
function [vn, vl, err, step] = varpro (HOOK, Y, vn, vl, OPTIONS, ...
                 P0, P1, P2, P3, P4, P5, P6, P7, P8, P9);
%VARPRO         Variable Projection Algorithm
%
% [vn, vl, err, step] = ...
%      varpro (HOOK, Y, vn, vl, OPTIONS, P0, P1, ... );
%
%   Computes values for the non-linear 'vn' and the linear 'vl'
%   to approximate 'Y' in the least-squares sense.
%
%   On input, 'vn' contains approximative values of the solution
%   (as good as possible); 'vl' is used for its size only.
%
%   err == 0: everything OK.
%   err == 1: too many iterations.
%
%   Algorithm by GOLUB/PEREYRA:
%   "The differentiation of pseudo-inverses and nonlinear least
%    squares problems whose variables separate".
%   SIAM J. Num. Anal. 10(2), april 1973.
%
% HOOK is a user-supplied function receiving
%   (what, vn, OPTIONS, P0, ...) as arguments.
%   It evaluates
%     - function values (what == 'Phi') with linear factor
%     - function values without linear factor ('Psi')
%     - derivatives ('DPhi' and 'DPhi_Inc' for packed version)
%       ('DPsi' for the independent term).
%     - incidence matrix ('Inc'), to tell that a function
%       does not depend on certain variables.
%     - (possibly) Cholesky factor of positive definite
```

```
%       symmetric matrix to be used in the marquart step.
%
% if     (what == 'Phi'),
%   Ret := Phi = [Phi(1) ... Phi(4)];
% elseif (what == 'DPhi' | 'DPhi_Inc'),
%   Ret := Derivative of Phi
%     [dPhi/dvn(1) ... dPhi/dvn(nof-nonlinear)],
%     that is, jacobians are stored sequentially into Ret.
%   For 'DPhi_Inc', some columns are left out,
%     (all dPhi(j)/dvn(k) where INC(i,j) == 0),
%     INC is passed as the last arg to function.
% elseif (what == 'Inc'),
%   Ret := incidence matrix, to tell the 'varpro' routine that
%     some Phi(i) does not depend from vn(j). See 'DPhi_Inc'
% elseif (what == 'Psi'),
%   Ret := Psi, function without linear factor
% elseif (what == 'DPsi'),
%   Ret := Derivative (jacobian) of 'Psi'
% elseif (what == 'LM'),
%   Ret := Cholesky factor for Levenberg-Marquardt
%          positive-definite matrix (default eye)
% end

  fun = [HOOK];
  arg = [];
  if ~any(fun<48)
    fun = [fun, '('];
    arg = [arg, ', vn, OPTIONS'];
    for i = 1:nargin - 4
      arg = [arg,',P',int2str(i-1)];
    end
    arg_open = arg;
    arg = [arg, ')'];
  end
  if (nargin < 5), OPTIONS=[]; end

  OPTI_EPS     = 1;
  OPTI_LMSPEC  = 2;
  OPTI_NOFSTEP = 3;
  OPTI_ERRPR   = 4;
  OPTI_PACKM   = 5; % should memory be packed ?
  OPTI_MARQ    = 6;
  OPTI_PSI     = 7;

  ERR_OK       = 0;
  ERR_NOFSTEP  = 1;

  epss  = 1.0e-8;
  epsr  = 1.0e-5;

  k = size(vn,1);
  m = size(Y,1);
  n = size(vl,1);

  F      = eye(k,k);
  lambda = 1.0;
  omega  = sqrt(0.5);
  nofstep= 100;
  err    = ERR_OK;
  errpr  = 1;
  packm  = 1;
  marq   = 1;
  psi    = 0;

  for i = 1:size(OPTIONS,1),
    kind = OPTIONS(i,1);
```

```
  if     (kind == OPTI_EPS)     epsr = OPTIONS(i,2);
  elseif (kind == OPTI_LMSPEC)  F = eval([fun, '''LM''', arg]);
  elseif (kind == OPTI_NOFSTEP) nofstep = OPTIONS(i,2);
  elseif (kind == OPTI_ERRPR)   errpr = OPTIONS(i,2);
  elseif (kind == OPTI_PACKM)   packm = OPTIONS(i,2);
  elseif (kind == OPTI_MARQ)    marq = OPTIONS(i,2);
  elseif (kind == OPTI_PSI)     psi = OPTIONS(i,2);
  else                          error ('unknown option');
  end
end

if (~psi), YY = Y; end;
step = 0;
if (k > 0), % number of non-linear variables
  normr = 1;
  norma = 1;
  while (normr > epsr*norma),

    Phi  = eval ([fun, '''Phi''', arg]);
    %
    % Phi (i,k) is k-th component of phi (i)
    %

    if (packm),
      Inc  = eval ([fun, '''Inc''', arg]);
      DPhi = eval ([fun, '''DPhi_Inc''', arg_open, ',Inc', ')']);
    else
      DPhi = eval ([fun, '''DPhi''', arg]);
    end
    %
    % DPhi contains columns dphi(i)/dalpha(k), for
    %   - unpacked at DPhi(:, 1+(k-1)*n + i)
    %   - packed in the same order, but only for Inc(i,k) != 0.
    %

    [Q, T, S, r] = varpro_qr(Phi, epss);

    if (psi),
      Psi = eval ([fun, '''Psi''', arg]);
      YY  = Y - Psi;
    end
    v = Q*YY;
    C = Q*DPhi;
    x = S(:,1:r)*(T(1:r,1:r)\v(1:r));

    if (packm),
      U = zeros(n,k); Dx = zeros(m-r,k);
      p = 0; % column into DPhi
      for i = 1:k,
        for j = 1:n,
          if (Inc(j,i)),
            p = p+1;
            U(j,i)  = C(r+1:m,p)'*v(r+1:m);
            Dx(:,i) = Dx(:,i) + C(r+1:m,p)*x(j);
          end
        end
      end
    else
      U = []; Dx = [];
      for i = 1:k,
        U = [U, C(r+1:m,1+(i-1)*n:i*n)'*v(r+1:m)];
        Dx= [Dx, C(r+1:m,1+(i-1)*n:i*n)*x];
      end
    end

    H = S'*U;
```

```
    W = T(1:r,1:r)'\H(1:r,:);

    B   = -Q'*[W; Dx];
    res = Q(r+1:m,:)'*v(r+1:m);

    if (psi),
      DPsi= eval ([fun, '''DPsi''', arg]);
      B   = B - Q(r+1:m,:)'*Q(r+1:m,:)*DPsi;
    end

    if (marq),
      vvn = vn;
      istep = 0;
      while (1),
        h   = - [B; lambda*F]\[res; zeros(k,1)];
        vn  = vvn + h;
        Phi = eval ([fun, '''Phi''', arg]);
        if (psi),
          Psi = eval ([fun, '''Psi''', arg]);
          YY  = Y - Psi;
        end
        [Q, T, S, r] = varpro_qr (Phi, epss);
        tmp = Q(r+1:m,:)*YY;
        if (norm(tmp) <= norm(v(r+1:m))),
          break;
        end
        lambda = lambda/omega;
        istep = istep + 1;
      end
      if (istep == 0),
        lambda = lambda*omega;
      end
      vn = vvn;
    else
      h = -B\res;
    end

    norma = norm(vn);
    normr = norm(h);

    vn = vn + h;

    step  = step + 1;
    if (step > nofstep),
      err = ERR_NOFSTEP;
      break;
    end

  end % while
end % if vn == []

if (n > 0),
  Phi = eval ([fun, '''Phi''', arg]);
  if (psi),
    Psi = eval ([fun, '''Psi''', arg]);
    YY  = Y - Psi;
  end
  vl = Phi\YY;
end

if (~(err == ERR_OK) & errpr),
  if     (err == ERR_OK),
    disp ('no error');
  elseif (err == ERR_NOFSTEP),
    disp ('warning: number of steps exceeded limit');
  else
```

```
      error ('fatal: illegal error number');
    end
  end

end % varpro
```

The varpro interface procedure

```
function [z, a, b, alpha, phi, step] = ...
         pare_varpro (X, z, a, b, alpha, show);
%PARE_VARPRO    Geometric ellipse fit using varpro
%
% [z, a, b, alpha, phi, step] = ...
%      pare_varpro (X, z, a, b, alpha, show{0});
% computes the best fit ellipse in parameterform
% x = z(1) + a  cos(phi-alpha), y = z(2) + b sin(phi-alpha)
% using the varpro algorithm.
%
% X: given points <X(i,1), X(i,2)>
% z, a, b, alpha: starting values
% show: if (show), test output
%
% z, a, b, alpha: ellipse found
% phi: values for the nearest points (in parametric form)
% step: nof iterations

  m = size(X,1);

  s = sin(alpha);
  c = cos(alpha);
  Q = [c -s; s c];

  % compute initial approximations for phi_i
  du  = Q'*[X(:,1)-z(1) X(:,2)-z(2)]';
  phi = angle(du(1,:)/a + sqrt(-1)*du(2,:)/b)';

  [vn, vl, err, step] = varpro ('pare_varpro_hook', ...
                          [X(:,1);X(:,2)], [phi;alpha], [a;b;z]);

  phi   = vn(1:m);
  alpha = vn(1+m);
  a     = vl(1);
  b     = vl(2);
  z     = vl(3:4);

end % pare_varpro
```

The varpro hook function

```
function Ret = pare_varpro_hook (what, vn, OPTIONS, INC);
%PARE_VARPRO_HOOK       Hook for varpro ellipse fit
%
% Ret = pare_varpro_hook (what, vn, OPTIONS, INC);
% Hook for the 'varpro' routine, returns data depending on 'what'.
% given data is Y = [x-coord(1:m,1); y-coord(1:m,1)]
% estimation function is
% a*Phi(1) + b*Phi(2) + z(1)*Phi(3) + z(2)*Phi(4)
%
% what: kind of data needed (see 'varpro')
% vn: non-linear parameters
% OPTIONS: same as options passed to 'varpro' routine
% INC: incidence matrix, for what == 'DPhi_Inc' only
```

```
%
% Ret: see 'varpro'

  m = size(vn,1) - 1;
  if (strcmp(what,'Phi') | strcmp(what,'DPhi') | ...
      strcmp(what,'DPhi_Inc')),
    C = cos (vn(1:m));
    S = sin (vn(1:m));
    c = cos (vn(m+1));
    s = sin (vn(m+1));
  end

  if    (strcmp(what, 'Phi')),
    Ret = [ c*C, -s*S,  ones(m,1), zeros(m,1);
            s*C,  c*S, zeros(m,1),  ones(m,1)];
  elseif (strcmp(what, 'DPhi')),
    Ret = [];
    for i = 1:m,
      J = zeros(2*m,4);
      J(i,1)   = -c*S(i);
      J(i+m,1) = -s*S(i);
      J(i,2)   = -s*C(i);
      J(i+m,2) =  c*C(i);
      Ret = [Ret, J];
    end
    Ret = [Ret, [-s*C, -c*S, zeros(m,2); c*C, -s*S, zeros(m,2)]];
  elseif (strcmp(what, 'DPhi_Inc')),
    %% INC(i,j) if column phi(i)/dalpha(j) should be added.
    p   = ones(1,4)*INC*ones(m+1,1);
    Ret = zeros(2*m,p);
    p   = 0;
    for i = 1:m,
      if (INC(1,i)), p = p+1; Ret(i,p) = -c*S(i);
                              Ret(i+m,p) = -s*S(i);
      end;
      if (INC(2,i)), p = p+1; Ret(i,p) = -s*C(i);
                              Ret(i+m,p) =  c*C(i);
      end;
      if (INC(3,i)), p = p+1; end;
      if (INC(4,i)), p = p+1; end;
    end
    i = m+1;
    if (INC(1,i)), p = p+1; Ret(:,p) = [-s*C;  c*C]; end;
    if (INC(2,i)), p = p+1; Ret(:,p) = [-c*S; -s*S]; end;
    if (INC(3,i)), p = p+1; end;
    if (INC(4,i)), p = p+1; end;
  elseif (strcmp(what, 'Inc')),
    Ret = [ones(2,m+1); zeros(2,m+1)];
  else
    str = sprintf ('unknowm: command %s.', what);
    error (str);
  end

end % pare_varpro_hook
```

B.4.4 The odr algorithm

The general `odr` procedure

```
function [b, delta, err, step] = ...
         odr (HOOK, x, y, b, OPTIONS, W, D, delta, ...
              P0, P1, P2, P3, P4, P5, P6, P7, P8, P9);
```

```
%ODR    Orthogonal Distance Regression
%
%       [b, delta, err, step] = odr (HOOK, x, y, b, ...
%                                OPTIONS, W, D, delta, P0, ...);
%
%       determines eps[i], delta[i] so that
%         - HOOK ('f', x+delta, b) = y+eps
%         - norm(eps,2)^2 + norm(delta,2)^2 = minimal
%
%       You may think of it as a flexible total least
%       squares algorithm, or just an algorithm to fit
%       a curve with minimal geometric distances.
%
%       HOOK is user-supplied function evaluating
%         - ('f') --> f(x,b)
%         - ('df') --> [df/db, df/dx](x,b)

%       Algorithm by Boggs/Byrd/Schnabel
%       "A stable and efficient algorithm for nonlinear
%       orthogonal distance regression"
%       SIAM J. Sci. Stat. Comput. 8(6):1052--1078, nov. 87.

  fun = [HOOK];
  arg = [];
  if ~any(fun<48)
    fun = [fun, '('];
    arg = [arg, ', x+delta, b'];
    for i = 1:nargin - 8,
      arg = [arg,',P',int2str(i-1)];
    end
    arg_open = arg;
    arg = [arg, ')'];
  end
  if (nargin < 8), delta = []; end
  if (nargin < 7), D = []; end
  if (nargin < 6), W = []; end
  if (nargin < 5), OPTIONS=[]; end

  OPTI_EPS = 1;
  OPTI_NOFSTEP = 3;
  OPTI_ERRPR = 4;
  OPTI_ONEW = 9;
  OPTI_ONED = 10;
  OPTI_DIFFCHK = 11;

  ERR_OK = 0;
  ERR_NOFSTEP = 1;

  epss = 1.0e-8;
  epsr = 1.0e-5;
  oned = 0;
  onew = 0;
  err  = ERR_OK;
  nofstep = 100;
  diffchk = 0;
  alpha = 0.01;

  [n, m] = size(x);
  p = size(b, 1);
  if (delta == []), delta = zeros(n,m); end
  if (D == []), oned = 1; end
  if (W == []), onew = 1; end

  for i = 1:size(OPTIONS,1),
    kind = OPTIONS(i,1);
    if     (kind == OPTI_EPS)     epsr = OPTIONS(i,2);
```

```
    elseif (kind == OPTI_NOFSTEP) nofstep = OPTIONS(i,2);
    elseif (kind == OPTI_ERRPR)   errpr = OPTIONS(i,2);
    elseif (kind == OPTI_ONEW)    onew = OPTIONS(i,2);
    elseif (kind == OPTI_ONED)    oned = OPTIONS(i,2);
    elseif (kind == OPTI_DIFFCHK) diffchk = OPTIONS(i,2);
    else                          error ('unknown option');
    end
  end

  if (onew), W = onew*ones(n,1); end
  if (oned), D = oned*ones(n,m); end
  if (size(y,2) ~= 1) error ('y must be column vector'); end
  if (size(y,1) ~= n) error ('x and y incompatible'); end
  if (size(b,2) ~= 1) error ('b must be column vector'); end
  if (size(W,2) ~= 1) error ('W must be column vector'); end
  if (size(W,1) ~= n) error ('W incompatible'); end
  if (size(D,2) ~= m) error ('D incompatible'); end
  if (size(D,1) ~= n) error ('D incompatible'); end

% flag variables (scalar coeff?)
  wscalar = (onew ~= 0);
  dscalar = (oned ~= 0);
  scalar  = (wscalar & dscalar);

% size variables
  omega = zeros(n,1);
  M = zeros(n,1);
  yb = zeros(n,1);
  JB = zeros(n,p);
  t = zeros(n,m);

% loop until change small
% (or nof steps too large)
  step = 0;
  res  = -1;
  normr = 1;
  norma = 1;
  while (normr > epsr*norma),
    step = step + 1;
    if (step > nofstep),
      err = ERR_NOFSTEP;
      disp ('warning: number of steps exceeded limit');
      break;
    end
    f    = eval ([fun, '''f''', arg]);
    df   = eval ([fun, '''df''', arg]);

if ((df == []) | (diffchk ~= 0)),
    save_x = x;
    save_b = b;
    h = 1e-5;
    DF = [];
    for i=1:size(b,1),
      b(i) = b(i) - h;
      f1 = eval ([fun, '''f''', arg]);
      b(i) = b(i) + 2*h;
      f2 = eval ([fun, '''f''', arg]);
      b(i) = b(i) - h;
      DF = [DF, (f2 - f1)/(2*h)];
    end
    hv = h*ones(size(x,1),1);
    for i=1:size(x,2),
      x(:,i) = x(:,i) - hv;
      f1 = eval ([fun, '''f''', arg]);
      x(:,i) = x(:,i) + 2*hv;
      f2 = eval ([fun, '''f''', arg]);
```

```
      x(:,i) = x(:,i) - hv;
      DF = [DF, (f2 - f1)/(2*h)];
    end
    if (df == []),
      df = DF;
    else
      if (norm(DF-df) > epsr*norm(DF)),
        disp ('warning: differentiate may be inexact');
        if (diffchk == 2), disp('DF - df ='); disp(DF-df); end;
      else
        disp ('status: differentiate OK');
      end
    end
    x = save_x;
    b = save_b;
end

if (onew == 1),
    G1 = (f - y);
elseif (onew)
    G1 = onew*(f - y);
else
    G1 = W.*(f - y);
end
if (onew*oned == 1),
    G2 = delta;
elseif (scalar),
    G2 = (oned*onew)*delta;
else
    G2 = D.*delta;
    for i=1:n,
      G2(i,:) = W(i)*G2(i,:);
    end
end
    V    = df(:,p+1:p+m);
    J    = df(1:n,1:p);

    alpha = alpha/2;
    while (1),
if (oned),
      E    = oned^2 + alpha;
      Ei   = 1/E;
      for i=1:n,
        omega(i) = (V(i,:)*Ei)*V(i,:)';
      end
      M = sqrt(1./(1+omega));
      Tmp = (Ei*oned)*G2;
else
      E    = D.^2 + alpha*ones(n,m);
      Ei   = 1./E;
      for i=1:n,
        omega(i) = (V(i,:).*Ei(i,:))*V(i,:)';
      end
      M = sqrt(1./(1+omega));
      Tmp = Ei.*D.*G2;
end

if (onew == 1),
      for i=1:n,
        JB(i,:) = M(i)*J(i,:);
      end
elseif (onew),
      for i=1:n,
        JB(i,:) = M(i)*onew*J(i,:);
      end
else
```

```
      for i=1:n,
        JB(i,:) = M(i)*W(i)*J(i,:);
      end
end
      for i=1:n,
        yb(i) = -M(i)*(G1(i) - V(i,:)*Tmp(i,:)');
      end
      s = [JB; sqrt(alpha)*eye(p)]\[yb; zeros(p,1)];
      tmp = -JB*s + yb;
      tmp = tmp.*M;
if (oned),
      for i=1:n,
        t(i,:) = tmp(i)*V(i,:)*Ei - Tmp(i,:);
      end
else
      for i=1:n,
        t(i,:) = tmp(i)*V(i,:).*Ei(i,:) - Tmp(i,:);
      end
end
      b = b + s;
      delta = delta + t;

      newres = 0;
      newf = eval ([fun, '''f''', arg]);
      epsilon = newf - y;
if (oned),
      for i=1:n,
        newres = newres + ...
          W(i)^2*(epsilon(i)^2 + (oned*norm(delta(i,:),2))^2);
      end
else
      for i=1:n,
        newres = newres + ...
          W(i)^2*(epsilon(i)^2 + norm(delta(i,:).*D(i,:),2)^2);
      end
end
      if ((res < 0) | (newres < res*(1+epsr))),
        norma = norm(delta) + norm(b);
        normr = norm(t) + norm(s);
        res   = newres;
        break;
      end

      b = b - s;
      delta = delta - t;
      alpha = alpha*3;
    end

  end

end % odr
```

The `odr` interface procedure

```
function [z, a, b, alpha, step] = ...
        alge_odr (X, z, a, b, alpha, show);
%ALGE_ODR
%
%       [z, a, b, alpha, step] = ...
%       alge_odr (X, z, a, b, alpha, show);
%
%       computes the best fit ellipse in parameterform
%       x = z(1) + a  cos(phi-alpha), y = z(2) + b sin(phi-alpha)
%       using the odr algorithm.  %
```

```
%       X: given points <X(i,1), X(i,2)>
%       z, a, b, alpha: starting values
%       show: if (show), test output
%
%       z, a, b, alpha: ellipse found
%       step: nof iterations
%
%       See also ODR, ALGE_ODR_HOOK

%       This implicit problem is solved by successive
%       solution of weighted explicit problems.
%       For further discussion see
%       Gill/Murray/Wright
%       "Practical Optimization"
%       Academic Press, New York, 1981.

  OPTI_ONEW = 9;
  OPTI_ONED = 10;

  wate = sqrt(eps);
  awate = sqrt(sqrt(wate))+1e-3;

  c = cos(alpha); s = sin(alpha);
  Q = [c -s; s c];
  A = Q*[1/(a^2) 0; 0 1/(b^2)]*Q';
  beta = [z(1); z(2); A(1,1); A(1,2); A(2,2)];
  delta = zeros(size(X));

  step = 0;
  while (awate >= wate),
    OPTIONS = [[OPTI_ONED, awate]];
    [beta, delta, err, astep] = odr ('alge_odr_hook', ...
      X, zeros(size(X,1),1), beta, OPTIONS, [], [], delta);
    awate = awate^2;
    step = step + astep;
  end

  [zno, a, b, alpha, err] = ellipse_params (...
    [beta(3); 2*beta(4); beta(5); 0; 0; -1]);
  z = [beta(1); beta(2)];

end % alge_odr
```

The `odr` hook function

```
function [res] = alge_odr_hook (what, x, b);
%ALGE_ODR_HOOK
%
%       Function called by odr,
%       wants either 'f' or 'df'
%
%       See also ODR.

  xc = x(:,1) - b(1)*ones(size(x,1),1);
  yc = x(:,2) - b(2)*ones(size(x,1),1);

  if     (strcmp(what, 'f')),
    res = b(3)*xc.^2 + 2*b(4)*xc.*yc + b(5)*yc.^2 - ones(size(x,1),1);
  elseif (strcmp(what, 'df')),
    res = zeros(size(x,1),7);
    res(:,6) = 2*b(3)*xc + 2*b(4)*yc;
    res(:,7) = 2*b(5)*yc + 2*b(4)*xc;
    res(:,1) = -res(:,6);
    res(:,2) = -res(:,7);
```

```
    res(:,3) = xc.^2;
    res(:,4) = 2*xc.*yc;
    res(:,5) = yc.^2;
  else
    str = sprintf ('unknown command: %s', what);
    error (str);
  end

end % alge_odr_hook
```

C Data sets

This section lists the data sets used in the paper.

C.1 Various data sets

Data for eccentric ellipse

```
X = [
    1.9400    3.7540
    2.8640    4.7500
    4.9420    6.6280
    5.6360    8.1520
    6.5120    9.2080
    8.4060   10.6160
    9.3760   11.5540
   10.5780   12.8440
   11.5940   14.9560
   13.8100   16.2460
   16.3520   16.5980
   16.2120   13.7240
   15.3340   12.4920
   14.7340   11.4960
   12.0100    9.5600
   10.5780    8.2120
    9.3760    7.0380
    9.3760    7.0380
    7.5760    4.3980
    5.8200    3.9300
    4.4800    2.9320
    3.0020    2.6980
];
```

Points near ellipse

```
X = [
    2.0143   10.5575
   17.3465    3.2690
   -8.5257   -7.2959
   -7.9109   -7.6447
   16.3705   -3.8815
  -15.3434    5.0513
  -21.5840   -0.6013
    9.4111   -9.0697
];
```

C.2 Comparison of the geometric estimation algorithms

Special data

```
X = [
     1     7
     2     6
     5     8
     7     7
     9     5
     3     7
     6     2
     8     4
];
```

Random data

```
X = [
    4.3792    0.6914
    0.9409    1.0692
   13.5773   10.5940
   13.5859   13.4230
   18.6939    0.1540
    7.6700    7.6683
   10.3883    1.3368
   16.6193    8.3497
];
```

Points on a circle

```
X = [
    3.8760   19.6208
   19.1326    5.8261
   -8.6445  -18.0353
   -8.5955  -18.0587
   18.3397   -7.9784
  -14.8771   13.3668
  -19.8514   -2.4339
    9.7412  -17.4674
];
```

Points on a circle, with perturbations

```
X = [
   -0.7783   21.4885
   14.6672    6.7159
   -8.3475  -13.7310
   -6.8840  -14.5970
   13.4167   -7.7091
  -16.0429    9.2865
  -24.1829   -0.8947
    8.9161  -18.3074
];
```

Points on ellipse

```
X = [
    3.8760    9.8104
   19.1326    2.9130
   -8.6445   -9.0177
   -8.5955   -9.0294
   18.3397   -3.9892
  -14.8771    6.6834
  -19.8514   -1.2169
    9.7412   -8.7337
];
```

Points on ellipse, with perturbations

```
X = [
   -0.7783   11.6781
   14.6672    3.8028
   -8.3475   -4.7133
   -6.8840   -5.5677
   13.4167   -3.7199
  -16.0429    2.6031
  -24.1829    0.3222
    8.9161   -9.5737
];
```

Points on hyperbola branch

```
X = [
   20.0658   -0.8118
   20.1711   -1.3107
   20.0266    0.5163
   20.0268    0.5176
   20.1575    1.2575
   20.0113   -0.3362
   20.0003    0.0560
   20.0913    0.9564
];
```

Points on hyperbola branch, with perturbations

```
X = [
   15.4115    1.0559
   15.7057   -0.4209
   20.3236    4.8207
   21.7383    3.9792
   15.2345    1.5268
   18.8455   -4.4166
   15.6687    1.5952
   19.2661    0.1164
];
```