IMM
INFORMATICS AND MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Kongens Lyngby – Denmark

ROBUST C SUBROUTINES FOR NON-LINEAR OPTIMIZATION

Pernille Brock
Kaj Madsen
Hans Bruun Nielsen

Revised version of report NI-91-03

IMM-Technical Report-2004-21

IMM

Contents

1. Introduction

This report presents a package of robust and easy-to-use C subroutines for solving unconstrained and constrained non-linear optimization problems. The intention is that the routines should use the currently best algorithms available. All routines have standardized calls, and the user does not have to worry about special parameters controlling the iterations. For convenience we include an option for numerical checking of the user's implementation of the gradient.

Note that another report [3] presents a collection of robust subroutines for both unconstrained and constrained optimization but not requiring gradient information. The parameter lists for the subroutines in both collections are similar so it is easy to switch between the gradient and the non-gradient methods. All of the subroutine names in this report start with `MI1`. The coresponding names of the non-gradient subroutines are obtained by changing `1` to `0`.

The present report is a new and updated version of a previous report NI-91-03 with the same title, [16]. Both the previous and the present report describe a collection of subroutines, which have been translated from Fortran to C. The reason for writing the present report is that some of the C subroutines have been replaced by more effective and robust versions translated from the original Fortran subroutines to C by the Bandler Group, see [1]. Also the test examples have been modified to some extent. For a description of the original Fortran subroutines see the report [17]. The software changes are listed in Section 1.5.

1.1. Problem Formulation

We consider minimization of functions of vector arguments, $F : \mathbb{R}^n \mapsto \mathbb{R}$. The function may be a norm of vector valued function $\mathbf{f} : \mathbb{R}^n \mapsto \mathbb{R}^m$. For the scalar case the user must provide a subroutine, which – for a given $\mathbf{x}$ – returns both the function value $F(\mathbf{x})$ and the *gradient* $\mathbf{g}(\mathbf{x}) \in \mathbb{R}^n$, defined by

$$\mathbf{g} = \begin{bmatrix} \frac{\partial F}{\partial x_1} \\ \vdots \\ \frac{\partial F}{\partial x_n} \end{bmatrix} . \tag{1.1}$$

In case of a vector function the user's subroutine must return the vector $\mathbf{f}(\mathbf{x})$ and the *Jacobian matrix* $\mathbf{J}(\mathbf{x}) \in \mathbb{R}^{m \times n}$, defined by

$$\mathbf{J} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} , \tag{1.2}$$

i.e., the ith row in $\mathbf{J}$ is the gradient of f_i, the ith component of $\mathbf{f}$.

For an efficient performance of the optimization algorithm the function and the gradients must be implemented without errors. It is not possible to check the correctness of the implementation of F (or $\mathbf{f}$), but we provide the possibility of checking the corresponding gradient (or Jacobian).

1.2. Checking the Gradients

This is done by *difference approximations.* First, consider a scalar function $F(\mathbf{x})$: For given $\mathbf{x}$ and step length h we compute

$$\left.\begin{aligned} D_j^F &= \big(F(\mathbf{x}+h\mathbf{e}_j) - F(\mathbf{x})\big)/h \\ D_j^B &= \big(F(\mathbf{x}) - F(\mathbf{x}-\tfrac{1}{2}h\mathbf{e}_j)\big)/(\tfrac{1}{2}h) \\ D_j^E &= \big(D_j^F + 2D_j^B\big)/3 \end{aligned}\right\} , \quad j = 1, \ldots, n , \tag{1.3}$$

where $\mathbf{e}_j$ is the jth unit vector (the jth column of $\mathbf{I}$), and the superscripts stand for Forward, Backward and Extrapolated difference approximation, respectively.

We assume that F is three times continuously differentiable with respect to each of its arguments. Then a Taylor expansion from $\mathbf{x}$

shows that

$$\begin{aligned} F(\mathbf{x}) + \eta\mathbf{e}_j) &= F(\mathbf{x}) + \eta\frac{\partial F}{\partial x_j}(\mathbf{x}) + \tfrac{1}{2}\eta^2\frac{\partial^2 F}{\partial x_j^2}(\mathbf{x}) + \mathcal{O}(\eta^3) \\ &= F(\mathbf{x}) + \eta g_j(\mathbf{x}) + \eta^2 S_j(\mathbf{x}) + \mathcal{O}(\eta^3) \quad . \end{aligned} \tag{1.4}$$

Inserting this in (1.3) we see that

$$\begin{aligned} D_j^F &= g_j + hS_j + \mathcal{O}(h^2) \\ D_j^B &= g_j - \tfrac{1}{2}hS_j + \mathcal{O}(h^2) \qquad \text{with} \quad S_j = \tfrac{1}{2}\,\frac{\partial^2 F}{\partial x_j^2}(\mathbf{x}) \ . \\ D_j^E &= g_j + \mathcal{O}(h^2) \end{aligned} \tag{1.5}$$

Now, let G_j denote the jth component of the gradient as returned from the user's subroutine, and let

$$G_j = g_j - \psi_j \ , \tag{1.6}$$

where $\psi_j = \psi_j(\mathbf{x})$ is zero if the implementation is correct. Inserting this in (1.5) we get

$$\begin{aligned} \delta_j^F &\equiv D_j^F - G_j = \psi_j + hS_j + \mathcal{O}(h^2) \ , \\ \delta_j^B &\equiv D_j^B - G_j = \psi_j - \tfrac{1}{2}hS_j + \mathcal{O}(h^2) \ , \\ \delta_j^E &\equiv D_j^E - G_j = \psi_j + \mathcal{O}(h^2) \quad . \end{aligned} \tag{1.7}$$

If $\psi_j = 0$, $S_j \neq 0$ and h is so small that the last term in each right-hand side of (1.7) can be neglected, then we can expect $\delta_j^B \simeq -\frac{1}{2}\delta_j^F$ and δ_j^E to be of the order of magnitude $(\delta_j^F)^2$. Also, if the approximation is recomputed with h replaced by θh, where $0 < \theta < 1$, then both δ_j^F and δ_j^B are reduced by a factor θ, while δ_j^E is reduced by a factor θ^2.

If $\psi_j \neq 0$ and h is sufficiently small, then the error will be recognized by $\delta_j^F \simeq \delta_j^B \simeq \delta_j^E \simeq \psi_j$.

The computed values are affected by rounding errors. Especially, instead of $F(\mathbf{z})$ we get $\mathsf{fl}(F(\mathbf{z})) = F(\mathbf{z}) + \varepsilon$. The best that we can hope for is that $|\varepsilon| \leq u \cdot |F(\mathbf{x})|$, where u is the "unit round-off". (The subroutines use `double` corresponding to $u = 2^{-53} \simeq 10^{-16}$ on most computers). This has the consequence that for the computed difference approximations (1.7) should be replaced by

$$\begin{aligned} |\delta_j^F| &\le |\psi_j + hS_j| + A_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2)\ , \\ |\delta_j^B| &\le |\psi_j - \tfrac{1}{2}hS_j| + A_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2)\ , \\ |\delta_j^E| &\le |\psi_j| + B_j h^{-1} + \mathcal{O}(u) + \mathcal{O}(h^2))\ \ , \end{aligned} \tag{1.8}$$

where A_j and B_j are positive values, that depend on F and $\mathbf{x}$, but not on h. In the case of correct implementation of the gradient, (1.8) shows that for large h the errors are dominated by truncation error, while effects of rounding errors dominate if h is too small. Assuming that $|S_j|$ and A_j are of the same order of magnitude, the smallest error with the forward and backward difference approximations is obtained with $h \simeq \sqrt{u}\|\mathbf{x}\|$. Similarly, we can expect that $|\delta_j^E|$ is minimal for $h \simeq \sqrt[3]{u}\|\mathbf{x}\|$.

In order to enhance accuracy the one-sided difference approximations in (1.3) should be computed by the formulae

$$D_j^F = \frac{F(\mathbf{x}{+}h\mathbf{e}_j) - F(\mathbf{x})}{\widehat{h}_j}, \quad D_j^B = \frac{F(\mathbf{x}) - F(\mathbf{x}{-}\frac{1}{2}h\mathbf{e}_j)}{\widetilde{h}_j}\ , \tag{1.9a}$$

where $\widehat{h}_j$ and $\widetilde{h}_j$ are the **actual** steps,

$$\widehat{h}_j = \mathsf{fl}\big(\mathsf{fl}(x_j{+}h) - x_j\big), \quad \widetilde{h}_j = \mathsf{fl}\big(x_j - \mathsf{fl}(x_j{-}\tfrac{1}{2}h)\big)\ . \tag{1.9b}$$

Note that if $|h|$ is too small, then we get $\widehat{h}_j = 0$ and/or $\widetilde{h}_j = 0$. In that case the gradient checker gives an error return.

The subroutines `MI1F` and `MI1CF` deal with scalar functions of vector variables. If they are called with the option of checking the gradient, then they return $\{\delta_j^A, j^A\}$ defined by

$$j^A = \underset{j=1,\ldots,n}{\operatorname{argmax}}\ \{|\delta_j^A|\}, \qquad \delta^A = \delta_{j^A}^A \tag{1.10}$$

for $A = F, B, E$, i.e. δ^A is the extreme value and j^A is its position.

The other subroutines deal with problems where $F(\mathbf{x})$ is some norm of a vector function $\mathbf{f}(\mathbf{x})$. In this case it is relevant to check the implementation of the Jacobian $\mathbf{J}(\mathbf{x})$, (1.2). The ith row in $\mathbf{J}$ is the gradient of f_i, the ith component of $\mathbf{f}$, and a straightforward generalization of (1.3) is

$$\left.\begin{aligned} D^F_{ij} &= \big(f_i(\mathbf{x}{+}h\mathbf{e}_j) - f_i(\mathbf{x})\big)/h \\ D^B_{ij} &= \big(f_i(\mathbf{x}) - f_i(\mathbf{x}{-}\tfrac{1}{2}h\mathbf{e}_j)\big)/(\tfrac{1}{2}h) \\ D^E_{ij} &= \big(D^F_{ij} + 2D^B_{ij}\big)/3 \end{aligned}\right\}, \quad \begin{cases} i=1,\ldots,m \\ j=1,\ldots,n \end{cases}, \tag{1.11}$$

leading to

$$\begin{aligned} \delta^F_{ij} &\equiv D^F_{ij} - \mathtt{J}_{ij} = \psi_{ij} + hS_{ij} + \mathcal{O}(h^2) + \mathcal{O}(uh^{-1}) \ , \\ \delta^B_{ij} &\equiv D^B_{ij} - \mathtt{J}_{ij} = \psi_{ij} - \tfrac{1}{2}hS_{ij} + \mathcal{O}(h^2) + \mathcal{O}(uh^{-1}) \ , \\ \delta^E_{ij} &\equiv D^E_{ij} - \mathtt{J}_{ij} = \psi_{ij} + \mathcal{O}(h^2) + \mathcal{O}(uh^{-1}) \ \ , \end{aligned} \tag{1.12}$$

where $\mathtt{J}_{ij}$ is the (i,j)th element in the implemented Jacobian, ψ_{ij} is its error and $S_{ij} = \frac{1}{2}\partial^2 f_i/\partial x_j^2$. If the subroutines are called with the option of checking the Jacobian, they return δ^A, i^A, j^A for $A = F, B, E$ defined as in (1.10).

1.3. Examples

First, consider the scalar problem $(n=2)$

$$F(\mathbf{x}) = \cos x_1 + e^{2x_2}, \qquad \mathbf{g}(\mathbf{x}) = \begin{bmatrix} -\sin x_1 \\ 2e^{2x_2} \end{bmatrix} \ , \tag{1.13}$$

implemented by the subroutine (note the sign error in g_1)

```
     void fdf ( const int *n, const double x[], double df[], double *f )
/*   Scalar function with gradient error   */
        {
            double cexp;

            cexp = exp(2.0 * x[1]);
            *f = cos(x[0]) + cexp;
            df[0] = sin(x[0]);
            df[1] = 2.0 * cexp;
        }
```

If we call e.g. `MI1F` with the checking option with the point $\mathbf{x} = [1,1]^\top$ and $h = 10^{-3}$, we get the results

h	$\lvert\delta^F\rvert$	j^F	$\lvert\delta^B\rvert$	j^B	$\lvert\delta^E\rvert$	j^E
1.0e+00	2.40e+02	2	-4.02e+01	2	5.31e+01	2
1.0e-01	1.17e+01	2	-5.28e+00	2	3.74e-01	2
1.0e-02	1.10e+00	2	-5.44e-01	2	3.65e-03	2
1.0e-03	1.09e-01	2	-5.46e-02	2	3.64e-05	2
1.0e-04	1.09e-02	2	-5.46e-03	2	3.64e-07	2
1.0e-05	1.09e-03	2	-5.46e-04	2	2.67e-09	2
1.0e-06	1.09e-04	2	-5.46e-05	2	6.93e-09	2
1.0e-07	1.07e-05	2	-5.75e-06	2	-3.48e-07	2
1.0e-08	6.46e-07	2	-1.49e-06	2	-1.49e-06	2
1.0e-09	1.41e-05	2	7.04e-06	2	7.04e-06	2
1.0e-10	4.97e-05	2	-9.58e-05	1	-9.58e-05	1
1.0e-11	6.18e-04	2	1.88e-04	1	1.33e-03	2
1.0e-12	1.41e-02	2	3.03e-03	1	1.41e-02	2

Table 1.1. *Gradient check with varying h*

$$\begin{aligned} \texttt{max|df| = 1.0920e+02},\quad \delta^F &= \texttt{-1.6832E+00},\quad \jmath^F = 1\ ,\\ \delta^B &= \texttt{-1.6828E+00},\quad \jmath^B = 1\ ,\\ \delta^E &= \texttt{-1.6829E+00},\quad \jmath^E = 1\ , \end{aligned}$$

indicating an error in the first element of the computed gradient. After correcting the error we get

$$\delta^F = \texttt{1.0927e-01},\quad \delta^B = \texttt{-5.4580e-02},\quad \delta^E = \texttt{3.6408e-05}$$

and $j^F = j^B = j^E = 2$. This agrees with expectation: $\delta^B \simeq -\frac{1}{2}\delta^F$ and δ^E is orders of magnitude smaller.

To illustrate the behaviour for varying step length we give results in Table 1.1 for the extreme values of of the differences for $h = 1, 10^{-1}, \cdots, 10^{-12}$.

For large values of h (the first two rows) the results are dominated by truncation error. Then follows a series of results where the δ^A behave as described above **and** $\delta^A(0.1h) \simeq 0.1\delta^A(h)$ for the forward and backward approximation, while $\delta^E(0.1h) \simeq 0.01\delta^E(h)$. Finally, for the smallest h-values rounding errors dominate. For the one-sided approximations this happens for $h \simeq 10^{-8} \simeq \sqrt{u}$ and for the extrapolated approximation the turning point is $h \simeq 10^{-5} \simeq 2\sqrt[3]{u}$, where $u = 2^{-53} \simeq 10^{-16}$ is the unit round-off used for the computations.

This agrees with the discussion after (1.8).

1.4. Test Functions

Many of the examples in this report use the following set of functions, $\mathbf{f} : \mathbb{R}^2 \mapsto \mathbb{R}^3$, originally given by Beale [2],

$$\begin{aligned} f_1(\mathbf{x}) &= \ \ 1.5 \ \ - x_1(1 - x_2) \\ f_2(\mathbf{x}) &= \ 2.25 \ - x_1(1 - x_2^2) \\ f_3(\mathbf{x}) &= 2.625 - x_1(1 - x_2^3) \end{aligned} \tag{1.14}$$

1.5. Modifications

The following modifications were made compared with the previous version of the package described in [16],

1° The descriptions of the algorithms have been modified to a certain extent.

2° The test examples and results have been updated.

3° In the package is included makefiles for a UNIX platform, linking the files in the respective directories. These makefiles must be modified by the user, if the subroutines should work on a different platform. A succesful linkage and compilation relies on the existence of the header file `f2c.h` in the parent directory.

4° The subroutine `MI1CIN` and the auxiliary functions called from this subroutine have been replaced by more effective and robust versions, compared to the previous package. The new versions have been translated from Fortran to C by the Bandler Group, see [1].

5° The functions used for checking the user-implemented gradients have been modified such that the three maximum errors are returned as signed values.

1.6. Package overview

The package contains the following subroutines:

`MI1F`: Unconstrained minimization of a scalar function.
Origin: `VA13CD`, [14].

`MI1L2`: Unconstrained minimization of the ℓ_2-norm of a vector function (least squares).
Origin: `NL2SOL`, [5] and [6].

`MI1L1`: Unconstrained minimization of the ℓ_1-norm of a vector function.
Origin: `L1NLS`, [11].

`MI1INF`: Unconstrained minimization of the ℓ_∞-norm of a vector function.
Origin: `SUB1W`, [15].

`MI1CF`: Minimization of a non-linear function subject to non-linear constraints (mathematical programming).
Origin: `VF13AD`, [19].

`MI1CL1`: Linearly constrained minimization of the ℓ_1-norm of a vector function.
Origin: `L1NLS`, [11].

`MI1CIN`: Linearly constrained minimax optimization of a vector function.
Origin: `MLA1QS`, [10] translated from Fortran to C by the Bandler Group, [1].

2. Unconstrained Optimization

2.1. MI1F. Minimization of a Scalar Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ is the vector of unknown parameters and the scalar objective function F is twice continuously differentiable. The user must supply a subroutine that evaluates $F(\mathbf{x})$ and the gradient $\mathbf{g}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{g}$.

Method. The algorithm is a quasi-Newton method with BFGS updating of the inverse Hessian[1)] and soft line search,, see e.g. [7, Chapters 9 (and 6)] or [18, Chapters 3, 4 and 8].

Origin. `MI1F` uses a modified version of the subroutine `VA13CD` from [14], modified such that it is consistent with the other routines in this package. In the Harwell Library `VA13CD` is called from the driver routine `VA13AD`.

Use. The subroutine call is

```
mi1f(fdf,n,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf(  const int *N, const double x[],
           double df[], double *f )
```

It must calculate the value of the objective function and its gradient at the point $\mathbf{x} = [$`x[0]`$, \ldots,$ `x[n-1]`$]^\top$, $n =$ `*N`, and store these numbers as follows,

$$\texttt{*f} = F(\mathbf{x}),$$

$$\texttt{df(j-1)} = \frac{\partial F}{\partial x_{\texttt{j}}}(\mathbf{x}), \quad \texttt{j} = 1, \ldots \texttt{n}.$$

The name of this subroutine can be chosen freely by the user.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

---

1) The Hessian $\mathbf{H}(\mathbf{x})$ is the matrix of second derivatives, $H_{ij} = \dfrac{\partial^2 F}{\partial x_i \partial x_j}$.

`x` — `double array` with `n` elements. The use depends on the entry value of `icontr`.
`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.
`icontr` ≤ 0 : Point at which the Jacobian should be checked. Is not changed.

`dx` — `double`. The use depends on the entry value of `icontr`.
`icontr` > 0 : `dx` does not enter into the computations.
`icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3). Must be positive. Is not changed.

`eps` — `double`. Desired accuracy.
Used only if the entry value of `icontr` is positive. The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k + \mathbf{h}_k$ with $\|\mathbf{h}_k\| <$ `eps`$\cdot\|\mathbf{x}_k\|$. Must be positive. Is not changed.

`maxfun` — `integer`. Used only if the entry value of `icontr` is positive.
On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` — `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.

On exit with $\texttt{icontr}_{\text{entry}} > 0$:
`w[0]` $= F($`X`$)$, the computed minimum.

On exit with $\texttt{icontr}_{\text{entry}} \le 0$: Results of the gradient check are returned in the first 7 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in |`df`|.
`w[1]`, `w[4]`	δ^F and j^F.
`w[2]`, `w[5]`	δ^B and j^B.
`w[3]`, `w[6]`	δ^E and j^E.

In case of an error the indices `w[4..6]` point out the erroneous gradient component.

`iw` — `integer`. Length of work space `w`.
Must be at least $\frac{1}{2}n(n+15) + 1$. Is not changed.

`icontr` — `integer`.
On entry: Controls the computation,

`icontr` > 0 : Start minimization.
`icontr` ≤ 0 : Check gradient. No iteration.

On exit: Information about performance,
`icontr` $= 0$: Successful call.
`icontr` $= 2$: Iteration stopped because the maximum number of calls to `fdf` was exceeded, see parameter `maxfun`.
`icontr` < 0 : Computation did not start for the following reason,
`icontr` $= -2$: `n` ≤ 0
`icontr` $= -4$: `dx` ≤ 0.0
`icontr` $= -5$: `eps` ≤ 0.0
`icontr` $= -6$: `maxfun` ≤ 0
`icontr` $= -8$: `iw` $< \frac{1}{2}n(n+15) + 1$

Example. Minimize

$$F(\mathbf{x}) = \sin(x_1 x_2) + 2e^{x_1+x_2} + e^{-x_1-x_2} \ .$$

```
#include <stdio.h>
#include "math.h"
#include "f2c.h"


/*      TEST OF MI1F      23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *N, const double x[],
       double df[], double *f)
{
    double ccos, cexp;


    /* Function Body */
    ccos = cos(x1 * x2);
    cexp = exp(x1 + x2);
    *f = sin(x1 * x2) + cexp * 2 + 1 / cexp;

    df[0] = x2 * ccos + cexp * 2 - 1 / cexp; /* df/dx1(x) */
```

```
    df[1] = x1 * ccos + cexp * 2 - 1 / cexp; /* df/dx2(x) */

} /* fdf */

static int opti(int icontr)
{

#define N 2
#define IW N*(N+15)/2 + 1

    extern void mi1f(
      void (*fdf)(const int *n, const double x[],
                  double df[], double *f),
      int n,
      double x[],
      const double *dx,
      const double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double w[IW], x[2];
    int index[4], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 2.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = .001;

    mi1f(fdf, N, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 1; k < 4; ++k) {
    index[k] = (int) w[k + 3];
```

```
    }
    printf("TEST OF GRADIENTS\n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e at Variable No %d\n",
    w[1],index[1]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e at Variable No %d\n",
    w[2],index[2]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e at Variable No %d\n",
    w[3],index[3]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);

    }
    else {
/*     RESULTS FROM OPTIMIZATION */
    printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 2:
    printf("NB: MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("NUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUE AT THE SOLUTION: %18.10e\n",w[0]);
    }
    return 0;
}

int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 1.97e-02 at Variable No 2
MAXIMUM BACKWARD DIFFERENCE: -9.83e-03 at Variable No 2
MAXIMUM CENTRAL  DIFFERENCE: 3.62e-06 at Variable No 1

MAXIMUM ELEMENT IN DF: 3.97e+01

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL

                       SOLUTION:  -1.4385237849e+00
                                   1.0919501946e+00
NUMBER OF CALLS OF FDF: 19

FUNCTION VALUE AT THE SOLUTION:   1.8284271247e+00
```

The results indicate that there is no error in the gradient.

2.2. MI1L2. Minimization of the ℓ_2-Norm of a Vector Function (Least Squares)

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where

$$F(\mathbf{x}) = \tfrac{1}{2}\sum_{i=1}^{m} (f_i(\mathbf{x}))^2 \quad . \tag{2.1}$$

Here $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ is the vector of unknown parameters and $f_i,\ i=1,\ldots,m$ is a set of functions that are twice continuously differentiable. The user must supply a subroutine that evaluates $\mathbf{f}(\mathbf{x})$ and the Jacobian $\mathbf{J}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{J}$.

Method. The algorithm amounts to a variation on Newton's method in which part of the Hessian matrix is computed exactly and part is approximated by the secant (quasi-Newton) updating method. Once the iterates come sufficiently close to a local solution, they usually converge quite rapidly. To promote convergence from poor starting guess, the algorithm uses a model/trust region technique along with an adaptive choice of the model Hessian. Consequently, the algorithm sometimes reduces to a Gauss-Newton or Levenberg-Marquardt method (see e.g. [8, Section 5.2]). On large residual problems (in which $F(\mathbf{x}^*)$ is large), the present method often works much better than these methods. The algorithm is described in [5] and [6].

Origin. Subroutine `NL2SOL` from [5] and [6]. Available from Netlib.

Use. The subroutine call is

```
mi1l2(fdf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf(  const int *N, const int *M,
           const double x[],
           double df[], double f[] )
```

It must calculate the values of the functions and their gradients at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = $ `*N`, $m = $ `*M` and store these numbers as follows,

$$\texttt{f[i-1]} = f_{\texttt{i}}(\mathbf{x}),\ \ \texttt{i} = 1, \ldots, \texttt{m},$$

$$\texttt{df[(j-1)m+(i-1)]} = \frac{\partial f_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}),\ \begin{cases} \texttt{i} = 1, \ldots, \texttt{m} \\ \texttt{j} = 1, \ldots, \texttt{n} \end{cases}$$

The name of this subroutine can be chosen freely by the user.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`m` `integer`. Number of functions, m.
Must be positive. Is not changed.

`x` `double array` with `n` elements. The use depends on the entry value of `icontr`.
`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.
`icontr` ≤ 0 : Point at which the Jacobian should be checked.
Is not changed.

`dx` `double`. The use depends on the entry value of `icontr`.
`icontr` > 0 : `dx` does not enter into the computations.
`icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3).
Must be positive. Is not changed.

`eps` `double`. Used only if the entry value of `icontr` is positive.
On entry: Desired accuracy.
The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k + \mathbf{h}_k$ with $\|\mathbf{h}_k\| <$ `eps`$\cdot\|\mathbf{x}_k\|$. Must be positive.
On exit: If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate, and `eps` is set to 0.0. Otherwise not changed.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.

On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.
$\texttt{icontr}_{\text{entry}} > 0$: The function values at the computed solution, i.e.

$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}), \ \texttt{i} = 1, \ldots, \texttt{m}.$

$\texttt{icontr}_{\text{entry}} \leq 0$: Results of the gradient check are returned in the first 10 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in $\lvert\texttt{df}\rvert$.
`w[1]`, `w[4]`, `w[5]`	$\delta^F, \ i^F, \ j^F.$
`w[2]`, `w[6]`, `w[7]`	$\delta^B, \ i^B, \ j^B.$
`w[3]`, `w[8]`, `w[9]`	$\delta^E, \ i^E, \ j^E.$

In case of an error the indices point out the erroneous element of the Jacobian matrix.

`iw` `integer`. Length of work space `w`.
Must be at least $m(2n+4)+\frac{1}{2}n(3n+33)+93$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
$\texttt{icontr} > 0$: Start minimization.
$\texttt{icontr} \leq 0$: Check gradient. No iteration.

On exit: Information about performance,
$\texttt{icontr} = 0$: Successful call.
$\texttt{icontr} = 1$: Successful call.
$\texttt{icontr} = 2$: Iteration stopped because the maximum number of calls of `fdf` was exceeded, see parameter `maxfun`.

`icontr` < 0 : Computation did not start for the following reason,

`icontr` $= -2$: `n` ≤ 0
`icontr` $= -3$: `m` ≤ 0
`icontr` $= -5$: `dx` ≤ 0.0
`icontr` $= -6$: `eps` ≤ 0.0
`icontr` $= -7$: `maxfun` ≤ 0
`icontr` $= -9$: `iw` $< m(2n+4) + \frac{1}{2}n(3n+33) + 93$

Example. Minimize

$$F(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{3} f_i^2(\mathbf{x}) \quad ,$$

where the f_i are given by (1.14), page 11.

```
#include <stdio.h>
#include <math.h>
#include "f2c.h"


/*      TEST OF MI1L2     23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *n, const int *m, const double x[],
          double df[], double f[])
{
    int df_dim1 = *m;
    double x2_2 = x2*x2, x2_3 = x2_2*x2;

    /* Function Body */
    f[0] = 1.5 - x1 * (1. - x2); /* f1(x) */
    f[1] = 2.25 - x1 * (1. - x2_2); /* f2(x) */
    f[2] = 2.625 - x1 * (1. - x2_3); /* f3(x) */

    df[0] = x2 - 1.; /* df1/dx1(x) */
    df[0 + df_dim1] = x1; /* df1/dx2(x) */
    df[1] = x2_2 - 1.; /* df2/dx1(x) */
    df[1 + df_dim1] = x1 * 2. * x2; /* df2/dx2(x) */
    df[2] = x2_3 - 1.; /* df3/dx1(x) */
```

```
    df[2 + df_dim1] = x1 * 3. * x2_2; /* df3/dx2(x) */
} /* fdf */

static int opti(int icontr)
{

#define N 2
#define M 3
#define IW (2*N+4)*M+N*(3*N+33)/2+93

    extern void mi1l2(
      void (*fdf)(const int *n, const int *m, const double x[],
                  double df[], double f[]),
      int n,
      int m,
      double x[],
      const double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double w[IW], x[2];
    int index[8], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun =  25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = .001;

    mi1l2(fdf, N, M, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 2; k <= 4; ++k) {
```

```
index[k - 1] = (int) w[k * 2];
index[k + 3] = (int) w[(k << 1) + 1];
    }
    printf("TEST OF GRADIENTS\n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[1],index[1],index[5]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[2],index[2],index[6]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[3],index[3],index[7]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);
    }
    else {
/*     RESULTS FROM OPTIMIZATION */
printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 1:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 2:
    printf("NB: MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("\nNUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUES AT THE SOLUTION: %18.10e\n",w[0]);
    for (j = 1; j < M; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",w[j]);
    }
    }
    return 0;
}

int main()
{
    opti(0); /* check gradients */
```

```
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 3.00e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM BACKWARD DIFFERENCE: -1.50e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM CENTRAL  DIFFERENCE: 5.00e-07 AT FUNCTION NO 3 AND VARIABLE NO 2

MAXIMUM ELEMENT IN DF: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL

                       SOLUTION:   3.0000000000e+00
                                   5.0000000000e-01

NUMBER OF CALLS OF FDF: 9

FUNCTION VALUES AT THE SOLUTION:   7.8039796847e-12
                                   1.0468514944e-11
                                   1.1182166304e-11
```

The results indicate that there is no error in the Jacobian.

2.3. MI1L1. Minimization of the ℓ_1-Norm of a Vector Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where

$$F(\mathbf{x}) = \sum_{i=1}^{m} | f_i(\mathbf{x}) | \quad . \tag{2.2}$$

Here $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ is the vector of unknown parameters and $f_i,\ i=1,\ldots,m$ is a set of functions that are twice continuously differentiable. The user must supply a subroutine that evaluates $\mathbf{f}(\mathbf{x})$ and the Jacobian $\mathbf{J}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{J}$.

Method. The algorithm is iterative. It is based on successive linearizations of the non-linear functions f_i, combining a first order trust region method with a local method which uses approximate second order information. The method is described in [13].

Origin. Subroutine `L1NLS` from [11].

Remark. The trust region around the the current $\mathbf{x}$ is the ball centered at $\mathbf{x}$ with radius Δ defined so that the linearizations of the non-linear functions f_i are reasonably accurate for all points inside the ball. During iteration this bound is adjusted according to how well the linear approximations centered at the previous iterate predict the gain in F.

The user has to give an initial value for Δ. If the functions are almost linear, then we recommend to use an estimate of the distance between $\mathbf{x}_0$ and the solution $\mathbf{x}^*$. Otherwise, we recommend $\Delta_0 = 0.1\|\mathbf{x}_0\|$. However the initial choice of Δ is not critical because it is adjusted by the subroutine during the iteration.

Use. The subroutine call is

```
mi1l1(fdf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf(   const int *N, const int *M,
            const double x[],
            double df[], double f[] )
```

It must calculate the values of the functions and their gradients at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$ and store these numbers as follows,

$$\texttt{f[i-1]} = f_{\texttt{i}}(\mathbf{x}), \;\; \texttt{i} = 1, \ldots, \texttt{m},$$

$$\texttt{df[(j-1)m+(i-1)]} = \frac{\partial f_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}), \;\; \begin{cases} \texttt{i} = 1, \ldots, \texttt{m} \\ \texttt{j} = 1, \ldots, \texttt{n} \end{cases}$$

The name of this subroutine can be chosen freely by the user.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`m` `integer`. Number of functions, m.
Must be positive. Is not changed.

`x` `double array` with `n` elements. The use depends on the entry value of `icontr`.

`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.

`icontr` ≤ 0 : Point at which the Jacobian should be checked. Not changed.

`dx` `double`. The use depends on the entry value of `icontr`.

`icontr` > 0 : *On entry*: `dx` must be set by the user to an initial value of the trust region radius, which controls the step length of the iterations. See **Remark** above. Must be positive.
On exit: Final trust region radius.

`icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3). Must be positive. Is not changed.

`eps` `double`. Used only if the entry value of `icontr` is positive.
On entry: Desired accuracy.
The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k+\mathbf{h}_k$ with $\|\mathbf{h}_k\| < \texttt{eps}\cdot\|\mathbf{x}_k\|$. Must be positive.
On exit: If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate, and `eps` is set to 0.0 and `icontr` is set to 2. Otherwise not changed.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.
On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.
$\texttt{icontr}_{\text{entry}} > 0$: The function values at the computed solution, i.e.

$$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1,\ldots,\texttt{m}.$$

$\texttt{icontr}_{\text{entry}} \le 0$: Results of the gradient check are returned in the first 10 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in $\lvert\texttt{df}\rvert$.
`w[1]`, `w[4]`, `w[5]`	$\delta^F,\ i^F,\ j^F$.
`w[2]`, `w[6]`, `w[7]`	$\delta^B,\ i^B,\ j^B$.
`w[3]`, `w[8]`, `w[9]`	$\delta^E,\ i^E,\ j^E$.

In case of an error the indices point out the erroneous element of the Jacobian matrix.

`iw` `integer`. Length of work space `w`.
Must be at least $2nm+5n^2+11n+5m+5$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
$\texttt{icontr} > 0$: Start minimization.
$\texttt{icontr} \le 0$: Check gradient. No iteration.

On exit: Information about performance,
$\texttt{icontr} = 0$: Successful call.
$\texttt{icontr} = 1$: Successful call.

`icontr` $= 2$:	Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `fdf` was exceeded, see parameter `maxfun`. The best solution approximation is returned in `x`.
`icontr` < 0 :	Computation did not start for the following reason, `icontr` $= -2$: `n` ≤ 0 `icontr` $= -3$: `m` ≤ 0 `icontr` $= -5$: `dx` ≤ 0.0 `icontr` $= -6$: `eps` ≤ 0.0 `icontr` $= -7$: `maxfun` ≤ 0 `icontr` $= -9$: `iw` $< 2nm+5n^2+11n+5m+5$

Example. Minimize

$$F(\mathbf{x}) = \sum_{i=1}^{3} |f_i(\mathbf{x})| \quad ,$$

where the f_i are given by (1.14), page 11.

```
#include <stdio.h>
#include <math.h>
#include "f2c.h"


/*      TEST OF MI1L1     23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *n, const int *m, const double x[],
          double df[], double f[])
{
    int df_dim1 = *m;
    double x2_2 = x2*x2, x2_3 = x2_2*x2;


    /* Function Body */
    f[0] = 1.5 - x1 * (1. - x2); /* f1(x) */
    f[1] = 2.25 - x1 * (1. - x2_2); /* f2(x) */
```

```
    f[2] = 2.625 - x1 * (1. - x2_3); /* f3(x) */

    df[0] = x2 - 1.; /* df1/dx1(x) */
    df[0 + df_dim1] = x1; /* df1/dx2(x) */
    df[1] = x2_2 - 1.; /* df2/dx1(x) */
    df[1 + df_dim1] = x1 * 2. * x2; /* df2/dx2(x) */
    df[2] = x2_3 - 1.; /* df3/dx1(x) */
    df[2 + df_dim1] = x1 * 3. * x2_2; /* df3/dx2(x) */
} /* fdf */

static int opti(int icontr)
{

#define N 2
#define M 3
#define IW 2*N*M+5*N*N+11*N+5*M+5

    extern void mi1l1(
      void (*fdf)(const int *n, const int *m, const double x[],
                  double df[], double f[]),
      int n,
      int m,
      double x[],
      const double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double w[IW], x[2];
    int index[8], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun =  25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = (optim) ? 0.1 : .001;

    mi1l1(fdf, N, M, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
```

```
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 2; k <= 4; ++k) {
index[k - 1] = (int) w[k * 2];
index[k + 3] = (int) w[(k << 1) + 1];
    }
    printf("TEST OF GRADIENTS \n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[1],index[1],index[5]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[2],index[2],index[6]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[3],index[3],index[7]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);
    }
    else {
/*     RESULTS FROM OPTIMIZATION */
printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 1:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 2:
    printf("NB: EPS IS TOO SMALL OR\n"
           "NB: MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("\nNUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUES AT THE SOLUTION: %18.10e\n",w[0]);
    for (j = 1; j < M; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",w[j]);
```

```
    }
    }
    return 0;
}

int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 3.00e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM BACKWARD DIFFERENCE: -1.50e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM CENTRAL  DIFFERENCE: 5.00e-07 AT FUNCTION NO 3 AND VARIABLE NO 2

MAXIMUM ELEMENT IN DF: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL

                       SOLUTION:   3.0000000000e+00
                                   5.0000000000e-01

NUMBER OF CALLS OF FDF: 10

FUNCTION VALUES AT THE SOLUTION:   2.2204460493e-16
                                   4.4408920985e-16
                                   4.4408920985e-16
```

The results indicate that there is no error in the Jacobian.

2.4. MI1INF. Minimization of the ℓ_∞-Norm of a Vector Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where

$$F(\mathbf{x}) = \max_i |f_i(\mathbf{x})| \quad . \tag{2.3}$$

Here $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ is the vector of unknown parameters and $f_i,\ i=1,\ldots,m$ is a set of functions that are twice continuously differentiable. The user must supply a subroutine that evaluates $\mathbf{f}(\mathbf{x})$ and the Jacobian $\mathbf{J}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{J}$.

Method. The algorithm is iterative. It is based on successive linearizations of the non-linear functions f_i and uses constraints on the step vector. The linearized problems are solved by a linear programming technique. The method is described in [15].

Origin. The main part of the subroutine was written by K. Madsen and was published as `VE01AD` in the the Harwell Subroutine Library [14]. We use K. Madsen's original subroutine `SUB1W` which is consistent with the other subroutines in the present package

Remark. The user has to give an initial value for Δ, which appears in the constraint $\|\mathbf{h}\| \le \Delta$, where $\mathbf{h}$ is the step between two consecutive iterates. During iteration this bound (trust region radius) is adjusted according to how well the current linear approximations predict the actual gain in F.

If the functions f_i are almost linear, then we recommend to use a value for Δ_0, which is an estimate of the distance between $\mathbf{x}_0$ and the solution $\mathbf{x}^*$. Otherwise, we recommend $\Delta_0 = 0.1\|\mathbf{x}_0\|$. However the initial choice of Δ is not critical because it is adjusted by the subroutine during the iteration.

Use. The subroutine call is

```
mi1inf(fdf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf( const int *N, const int *M,
          const double x[],
          double df[], double f[] )
```

It must calculate the values of the functions and their gradients at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$ and store these numbers as follows,

$$\texttt{f[i-1]} = f_{\texttt{i}}(\mathbf{x}),\ \texttt{i} = 1, \ldots, \texttt{m},$$
$$\texttt{df[(j-1)m+(i-1)]} = \frac{\partial f_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}),\ \begin{cases} \texttt{i} = 1, \ldots, \texttt{m} \\ \texttt{j} = 1, \ldots, \texttt{n} \end{cases}$$

The name of this subroutine can be chosen freely by the user.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`m` `integer`. Number of functions, m.
Must be positive. Is not changed.

`x` `double array` with `n` elements. The use depends on the entry value of `icontr`.
`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.
`icontr` ≤ 0 : Point at which the Jacobian should be checked. Not changed.

`dx` `double`. The use depends on the entry value of `icontr`.
`icontr` > 0 : *On entry*: `dx` must be set by the user to an initial value of the trust region radius, which controls the step length of the iterations. See **Remark** above. Must be positive.
On exit: Final trust region radius.
`icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3). Must be positive. Is not changed.

`eps` `double`. Used only if the entry value of `icontr` is positive.
On entry: Desired accuracy.
The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k+\mathbf{h}_k$ with $\|\mathbf{h}_k\| <$ `eps`$\cdot\|\mathbf{x}_k\|$. Must be positive.
On exit: If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate, and `eps` is set to 0.0 and `icontr`is set to 2. Otherwise not changed.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.
On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.
$\texttt{icontr}_{\text{entry}} > 0$: The function values at the computed solution, i.e.

$$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1, \ldots, \texttt{m}.$$

$\texttt{icontr}_{\text{entry}} \le 0$: Results of the gradient check are returned in the first 10 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in |`df`|.
`w[1]`, `w[4]`, `w[5]`	$\delta^F,\ i^F,\ j^F$.
`w[2]`, `w[6]`, `w[7]`	$\delta^B,\ i^B,\ j^B$.
`w[3]`, `w[8]`, `w[9]`	$\delta^E,\ i^E,\ j^E$.

In case of an error the indices point out the erroneous element of the Jacobian matrix.

`iw` `integer`. Length of work space `w`.
Must be at least $2nm + n^2 + 14n + 4m + 11$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
`icontr` > 0 : Start minimization.
`icontr` ≤ 0 : Check gradient. No iteration.

On exit: Information about performance,
`icontr` $= 0$: Successful call.
`icontr` $= 1$: Successful call.

`icontr` $= 2$: Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `fdf` was exceeded, see parameter `maxfun`. The best solution approximation is returned in `x`.

`icontr` < 0 : Computation did not start for the following reason,
- `icontr` $= -2$: `n` ≤ 0
- `icontr` $= -3$: `m` ≤ 0
- `icontr` $= -5$: `dx` ≤ 0.0
- `icontr` $= -6$: `eps` ≤ 0.0
- `icontr` $= -7$: `maxfun` ≤ 0
- `icontr` $= -9$: `iw` $< 2nm + n^2 + 14n + 4m + 11$

Example. Minimize

$$F(\mathbf{x}) = \max_i |\, f_i(\mathbf{x}) \,| \quad ,$$

where the f_i are given by (1.14), page 11.

```
#include <stdio.h>
#include <math.h>
#include "f2c.h"


/*      TEST OF MI1INF    23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *n, const int *m, const double x[],
          double df[], double f[])
{
    int df_dim1 = *m;
    double x2_2 = x2*x2, x2_3 = x2_2*x2;


    /* Function Body */
    f[0] = 1.5 - x1 * (1. - x2); /* f1(x) */
    f[1] = 2.25 - x1 * (1. - x2_2); /* f2(x) */
    f[2] = 2.625 - x1 * (1. - x2_3); /* f3(x) */
```

```
    df[0] = x2 - 1.; /* df1/dx1(x) */
    df[0 + df_dim1] = x1; /* df1/dx2(x) */
    df[1] = x2_2 - 1.; /* df2/dx1(x) */
    df[1 + df_dim1] = x1 * 2. * x2; /* df2/dx2(x) */
    df[2] = x2_3 - 1.; /* df3/dx1(x) */
    df[2 + df_dim1] = x1 * 3. * x2_2; /* df3/dx2(x) */
} /* fdf_ */

static int opti(int icontr)
{

#define N 2
#define M 3
#define IW 2*N*M+N*N+14*N+4*M+11

    extern void mi1inf(
      void (*fdf)(const int *n, const int *m, const double x[],
                  double df[], double f[]),
      int n,
      int m,
      double x[],
      const double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double w[IW], x[2];
    int index[8], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun =  25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = (optim) ? 0.1 : .001;

    mi1inf(fdf, N, M, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
```

```
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 2; k <= 4; ++k) {
index[k - 1] = (int) w[k * 2];
index[k + 3] = (int) w[(k << 1) + 1];
    }
    printf("TEST OF GRADIENTS \n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[1],index[1],index[5]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[2],index[2],index[6]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[3],index[3],index[7]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);
    }
    else {
/*     RESULTS FROM OPTIMIZATION */
printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 1:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 2:
    printf("NB: EPS IS TOO SMALL OR\n"
           "MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("\nNUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUES AT THE SOLUTION: %18.10e\n",w[0]);
    for (j = 1; j < M; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",w[j]);
    }
    }
```

```
    return 0;
}

int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 3.00e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM BACKWARD DIFFERENCE: -1.50e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM CENTRAL  DIFFERENCE: 5.00e-07 AT FUNCTION NO 3 AND VARIABLE NO 2

MAXIMUM ELEMENT IN DF: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL

                       SOLUTION:   3.0000000000e+00
                                   5.0000000000e-01

NUMBER OF CALLS OF FDF: 11

FUNCTION VALUES AT THE SOLUTION:  -4.4408920985e-16
                                  -4.4408920985e-16
                                   0.0000000000e+00
```

The results indicate that there is no error in the Jacobian.

3. Constrained Optimization

3.1. MI1CF. Constrained Minimization of a Scalar Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where the vector of unknown parameters $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ must satisfy the following non-linear equality and inequality constraints,

$$\begin{aligned} c_i(\mathbf{x}) &= 0 \ , \quad i = 1, 2, \ldots, l_{\text{eq}} \ , \\ c_i(\mathbf{x}) &\geq 0 \ , \quad i = l_{\text{eq}}+1, \ldots, l \ . \end{aligned}$$

The objective function F and the constraint functions $\{c_i\}$ must be twice continuously differentiable. The user must supply a subroutine that evaluates $F(\mathbf{x})$, $\{c_i(\mathbf{x})\}$ and the gradients of F and $\{c_i\}$. There is an option for checking the implementation of these gradients.

Method. The algorithm is iterative. It is based on successively approximating the non-linear problem with quadratic problems, i.e. at the current iterate the objective function is approximated by a quadratic function and the constraints are approximated by linear functions. The algorithm uses the so-called *"Watch-dog technique"* as described in [4] and [19]. The quadratic programming algorithm is described in [20]

Origin. Harwell subroutine `VF13AD` from [14].

Use. The subroutine call is

```
mi1cf(fdfcdc,n,l,leq,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdfcdc` Subroutine written by the user with the following declaration

```
void fdfcdc(   const int *N, const int *L,
               const double x[],
               double *f, double df[] )
               double c[], double dc[] )
```

where n = `*N`, l = `*L`. It must calculate the value of the objective function and its gradient at the point $\mathbf{x}$ = $[$`x[0]`$, \ldots,$ `x[n-1]`$]^\top$ and store these numbers as follows,

$$\texttt{*f} = F(\mathbf{x}),$$

$$\texttt{df[j-1]} = \frac{\partial F}{\partial x_{\texttt{j}}}(\mathbf{x}) \ , \qquad \texttt{j} = 1, \ldots \texttt{n}$$

$$\texttt{c[i-1]} = c_{\texttt{i}} \ , \qquad \texttt{i} = 1, \ldots \texttt{l}$$

$$\texttt{dc[(j-1)l+(i-1)} = \frac{\partial c_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}) \ , \quad \texttt{i} = 1, \ldots \texttt{l} \text{ and} \texttt{j} = 1, \ldots \texttt{n}$$

The name of the subroutine can be chosen freely by the user.

It is essential that the equality constraints (if any) are numbered first.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`l` `integer`. Number of constraints, m.
Must be positive. Is not changed.

`leq` `integer`. Number of equality constraints, l_{eq}.
Must satisfy $0 \le$ `leq` $\le \min\{$`l`, `n`$\}$. Is not changed.

`x` `double array` with `n` elements. The use depends on the entry value of `icontr`.
`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
It needs not satisfy the constraints.
On exit: Computed solution.
`icontr` ≤ 0 : Point at which the Jacobian should be checked.
Is not changed.

`dx` `double`. The use depends on the entry value of `icontr`.
`icontr` > 0 : `dx` does not enter into the computations.

`icontr` ≤ 0 : Gradient check for the objective function and for the constraints in `fdfcdc` with `dx` used for h in (1.3). Must be nonzero. Is not changed.

`eps` `double`. The use depends on the entry value of `icontr`.

`icontr` > 0 : Must be set by the user to indicate the desired accuracy of the results. Must be positive. The iteration stops when the Kuhn-Tucker conditions are approximately satisfied within a tolerance of `eps`. Is not changed.

`icontr` ≤ 0 : `eps` does not enter into the computations.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.

On entry: Upper bound on the number of calls of `fdfcdc`. Must be positive.

On exit: Number of calls of `fdfcdc`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.

$\texttt{icontr}_{\text{entry}} > 0$:

`w[0]` $= F(\texttt{x})$, the computed minimum.

`w[i]` $= c_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1, \ldots, \texttt{l}$

$\texttt{icontr}_{\text{entry}} \le 0$: Results of the gradient check are returned in the first 17 elements of `w` as follows, cf. (1.10)

Objective function:	
`w[0]`	Maximum element in $\vert\texttt{df}\vert$.
`w[1]`,`w[8]`	δ^F and j^F.
`w[2]`,`w[9]`	δ^B and j^B.
`w[3]`,`w[10]`	δ^E and j^E.
Constraints:	
`w[4]`	Maximum element in $\vert\texttt{dc}\vert$.
`w[5]`,`w[11]`,`w[12]`	δ^F, i^F and j^F.
`w[6]`,`w[13]`,`w[14]`	δ^B, i^B and j^B.
`w[7]`,`w[15]`,`w[16]`	δ^E, i^E and j^E.

In case of an error the indices `w[9..16]` point out the erroneous gradient component.

`iw` `integer`. Length of work space `w`.
Must be at least $\frac{5}{2}n(n+9) + (n+8)l + 15$. Is not changed.

`icontr` `integer`.

On entry: Controls the computation,

`icontr` > 0 : Start minimization.

`icontr` ≤ 0 : Check gradient. No iteration

On exit: Information about performance,

`icontr` $= 1$: Successful call.

`icontr` $= 2$: Iteration stopped because the maximum number of calls of `fdfcdc` was exceeded, see `maxfun`.

`icontr` $= 3$: Iteration stopped because more than 5 calls of `fdfcdc` was needed in one line search. Check your gradients.

`icontr` $= 4$: Iteration stopped because an uphill search direction was suggested. Check your gradients.

`icontr` $= 5$: Iteration failed because it was not possible to find a starting point satisfying all constraints.

`icontr` < 0 : Computation did not start for the following reason,

- `icontr` $= -2$: `n` ≤ 0
- `icontr` $= -3$: `l` ≤ 0
- `icontr` $= -4$: `leq` < 0 or `leq` $> \min\{$`l`, `n`$\}$
- `icontr` $= -6$: `dx` $= 0.0$ in case of gradient check
- `icontr` $= -7$: `eps` ≤ 0
- `icontr` $= -8$: `maxfun` ≤ 0
- `icontr` $= -9$: `iw` $< \frac{5}{2}n(n+9) + (n+8)l + 15$

Example. Minimize

$$F(\mathbf{x}) = \sin(x_1 x_2) + 2e^{x_1+x_2} + e^{-x_1-x_2}$$

subject to the constraints

$$\begin{aligned} c_1(\mathbf{x}) &\equiv 1 - x_1^2 - x_2^2 \geq 0 \\ c_2(\mathbf{x}) &\equiv x_2 - x_1^3 \qquad \geq 0 \\ c_3(\mathbf{x}) &\equiv x_1 + 2x_2 \qquad \geq 0 \end{aligned}$$

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

```
#include "f2c.h"


/*      TEST OF MI1CF      23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdfcdc(
    const int *n, const int *l,
    const double x[],
    double *f, double df[], double c[], double dc[])
{
    int dc_dim1 = *l;
    double x1_2 = x1*x1, x1_3 = x1_2*x1;
    double ccos, cexp;

    /* Function Body */
    ccos = cos(x1 * x2);
    cexp = exp(x1 + x2);
    *f = sin(x1 * x2) + cexp * 2 + 1 / cexp;
    df[0] = x2 * ccos + cexp * 2 - 1 / cexp; /* df/dx1(x) */
    df[1] = x1 * ccos + cexp * 2 - 1 / cexp; /* df/dx2(x) */

/*     CONSTRAINTS */

    c[0] = -x1_2 - x2 * x2 + 1.;
    c[1] = -x1_3 + x2;
    c[2] = x1 + x2 * 2.;

    dc[0] = x1 * -2; /* dc1/dx1(x) */
    dc[0 + dc_dim1] = x2 * -2; /* dc1/dx2(x) */
    dc[1] = x1_2 * -3; /* dc2/dx1(x) */
    dc[1 + dc_dim1] = 1.; /* dc2/dx2(x) */
    dc[2] = 1.; /* dc3/dx1(x) */
    dc[2 + dc_dim1] = 2.; /* dc3/dx2(x) */

} /* fdfcdc */

static int opti(int icontr)
{

#define N 2
#define L 3
#define LEQ 0
#define IW 5*N*N/2+45*N/2+L*(N+8)+15

    extern void mi1cf(
      void (*fdf)(const int *n, const int *l,
```

```
                  const double x[],
                  double *f, double df[],
  double c[], double dc[]),
      int n,
      int l,
      int leq,
      double x[],
      const double *dx,
      const double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double w[IW], x[N];
    int index[8], indexc[16], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = (optim) ? 0.1 : .001;

    mi1cf(fdfcdc, N, L, LEQ, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 1; k < 4; ++k) {
index[k] = (int) w[k + 7];
    }
    printf("TEST OF GRADIENTS \n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e AT VARIABLE NO %d\n",
    w[1],index[1]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e AT VARIABLE NO %d\n",
    w[2],index[2]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e AT VARIABLE NO %d\n",
    w[3],index[3]);
```

```
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);

    for (k = 6; k <= 8; ++k) {
indexc[k - 1] = (int) w[(k << 1) - 1];
indexc[k + 7] = (int) w[k * 2];
    }
    printf("TEST OF CONSTRAINT GRADIENTS\n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[5],indexc[5],indexc[13]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[6],indexc[6],indexc[14]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[7],indexc[7],indexc[15]);
    printf("\nMAXIMUM ELEMENT IN DC: %8.2e\n",w[4]);
    }
    else {
    printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    case 1:
    printf("ITERATION SUCCESSFUL\n\n");
    break;
    case 2:
    printf("NB: MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    case 3:
    case 4:
    printf("ITERATION HAS FAILED. THE TYPE OF THE FAILURE\n");
    printf("INDICATES THAT YOUR GRADIENTS MAY BE ERRONEOUS.\n");
    printf("USE THE GRADIENT CHECKING FACILITY.\n\n");
    return icontr;
    case 5:
    printf("MI1CF HAS FAILED TO FIND A STARTING POINT\n");
    printf("SATISFYING ALL OF THE CONSTRAINTS.\n\n");
    return icontr;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (i = 1; i <52-18 ; ++i) putchar(' ');
    for (i = 1; i < N; ++i) printf("%18.10e\n\n",x[i]);

    printf("NUMBER OF CALLS OF FDFCDC: %d\n\n", maxfun);
    printf("FUNCTION VALUE AT THE SOLUTION: %21.10e\n\n\n",w[0]);
    printf("CONSTRAINT VALUES AT THE SOLUTION: %18.10e\n",w[1]);
    for (j = 2; j <= L; ++j) {
        for (i = 1; i <54-18 ; ++i) putchar(' ');
```

```
        printf("%18.10e\n",w[j]);
    }
    }
    return 0;
}
int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 7.04e-03 AT VARIABLE NO 1
MAXIMUM BACKWARD DIFFERENCE: -3.52e-03 AT VARIABLE NO 1
MAXIMUM CENTRAL  DIFFERENCE: 1.18e-06 AT VARIABLE NO 1

MAXIMUM ELEMENT IN DF: 1.52e+01
TEST OF CONSTRAINT GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: -3.00e-03 AT FUNCTION NO 2 AND VARIABLE NO 1
MAXIMUM BACKWARD DIFFERENCE: 1.50e-03 AT FUNCTION NO 2 AND VARIABLE NO 1
MAXIMUM CENTRAL  DIFFERENCE: -5.00e-07 AT FUNCTION NO 2 AND VARIABLE NO 1

MAXIMUM ELEMENT IN DC: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL

                       SOLUTION:  -8.2644738993e-01
                                   5.6301395336e-01

NUMBER OF CALLS OF FDFCDC: 10

FUNCTION VALUE AT THE SOLUTION:      2.3895160145e+00


CONSTRAINT VALUES AT THE SOLUTION:  -4.4408920985e-16
                                     1.1274901557e+00
                                     2.9958051679e-01
```

The results indicate that the gradients of both the objective function and the constraints are implemented correctly.

3.2. MI1CL1. Linearly Constrained Minimization of the ℓ_1-Norm of a Vector Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where

$$F(\mathbf{x}) = \sum_{i=1}^{m} |\, f_i(\mathbf{x})\,| \quad , \tag{3.1a}$$

and where the vector of unknown parameters $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ must satisfy the following linear equality and inequality constraints,

$$\begin{aligned} c_i(\mathbf{x}) &\equiv \mathbf{d}_i^\top \mathbf{x} + c_i = 0 \ , \quad i = 1, 2, \ldots, l_{\text{eq}} \ , \\ c_i(\mathbf{x}) &\equiv \mathbf{d}_i^\top \mathbf{x} + c_i \geq 0 \ , \quad i = l_{\text{eq}}{+}1, \ldots, l \end{aligned} \tag{3.1b}$$

for given vectors $\{\mathbf{d}_i\}$ and scalars $\{c_i\}$. The $f_i, \ i=1, \ldots, m$ is a set of functions that are twice continuously differentiable. The user must supply a subroutine that evaluates $\mathbf{f}(\mathbf{x})$ and the Jacobian $\mathbf{J}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{J}$.

Method. The algorithm is iterative. It is based on successive linearizations of the non-linear functions f_i, combining a first order trust region method with a local method that uses approximate second order information. The method is described in [13].

Origin. Subroutine `L1NLS` by Jørgen Hald [11].

Remarks. The trust region around the the current $\mathbf{x}$ is the ball centered at $\mathbf{x}$ with radius Δ defined so that the linearizations of the non-linear functions f_i are reasonably accurate for all points inside the ball. During iteration this bound is adjusted according to how well the linear approximations centered at the previous iterate predict the gain in F.

The user has to give an initial value for Δ. If the functions are almost linear, then we recommend to use an estimate of the distance between $\mathbf{x}_0$ and the solution $\mathbf{x}^*$. Otherwise, we recommend $\Delta_0 = 0.1\|\mathbf{x}_0\|$. However the initial choice of Δ is not critical because it is adjusted by the subroutine during the iteration.

A solution is said to be *"regular"* when it is a strict local minimum, i.e. there exists a positive number K such that

$$F(\mathbf{x}) - F(\mathbf{x}^*) \geq K\|\mathbf{x} - \mathbf{x}^*\|$$

for any feasible $\mathbf{x}$ near $\mathbf{x}^*$. Otherwise, the solution is said to be *"singular"*.

Use. The subroutine call is

```
mi1cl1(fdf,n,m,l,leq,c,dc,x,&dx,&eps,&maxfun,w,iw,&icontr
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf(  const int *N, const int *M,
           const double x[],
           double df[], double f[] )
```

It must calculate the values of the functions and their gradients at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$ and store these numbers as follows,

$$\texttt{f[i-1]} = f_{\texttt{i}}(\mathbf{x}),\ \texttt{i} = 1, \ldots, \texttt{m},$$

$$\texttt{df[(j-1)m+(i-1)]} = \frac{\partial f_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}),\ \begin{cases} \texttt{i} = 1, \ldots, \texttt{m} \\ \texttt{j} = 1, \ldots, \texttt{n} \end{cases}$$

The name of this subroutine can be chosen freely by the user.

`n` `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`m` `integer`. Number of functions, m.
Must be positive. Is not changed.

`l` `integer`. Number of constraints, l.
Must be positive. Is not changed.

`leq` `integer`. Number of equality constraints, l_{eq}.
Must satisfy $0 \le \texttt{leq} \le \min\{\texttt{l}, \texttt{n}\}$. Is not changed.

`c` `double array` with `l` elements. The constant terms in the constraints (3.1b) are stored in the following way

$$\texttt{c[i-1]} = c_{\texttt{i}},\ \ \texttt{i} = 1, \ldots, \texttt{l}\ .$$

Is not changed.

`dc` `double array` with `l·n` elements. The coefficients of the constraints (3.1b) stored in the following way,

$$\texttt{dc[(i-1)n+(j-1)]} = d_{\texttt{i}}^{(\texttt{j})}, \quad \texttt{i} = 1, \ldots, \texttt{l}, \quad \texttt{j} = 1, \ldots, \texttt{n} .$$

Is not changed.

`x` `double array` with `n` elements. The use depends on the entry value of `icontr`.

`icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.

`icontr` ≤ 0 : Point at which the Jacobian should be checked. Is not changed.

`dx` `double`. The use depends on the entry value of `icontr`.

`icontr` > 0 : *On entry*: `dx` must be set by the user to an initial value of the trust region radius, which controls the step length of the iterations. See **Remarks** above. Must be positive.
On exit: Final trust region radius.

`icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3). Must be positive. Is not changed.

`eps` `double`. Used only if the entry value of `icontr` is positive.
On entry: Desired accuracy.
The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k+\mathbf{h}_k$ with $\|\mathbf{h}_k\| < \texttt{eps}\cdot\|\mathbf{x}_k\|$. Must be positive.
On exit: `eps` contains the length of the last step of the iteration. If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate, and `icontr` is set to 2.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.
On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.
$\texttt{icontr}_{\text{entry}} > 0$: The function values at the computed solution, i.e.

$$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}), \quad \texttt{i} = 1, \ldots, \texttt{m}.$$

`icontr`$_{\text{entry}} \leq 0$: Results of the gradient check are returned in the first 10 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in $\lvert$`df`$\rvert$.
`w[1]`, `w[4]`, `w[5]`	δ^F, i^F, j^F.
`w[2]`, `w[6]`, `w[7]`	δ^B, i^B, j^B.
`w[3]`, `w[8]`, `w[9]`	δ^E, i^E, j^E.

In case of an error the indices point out the erroneous element of the Jacobian matrix.

`iw` `integer`. Length of work space `w`.
Must be at least $2nm + 5n^2 + 5m + 10n + 4l$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
`icontr` > 0 : Start minimization.
`icontr` ≤ 0 : Check gradient. No iteration.

On exit: Information about performance,
`icontr` $= 0$: Successful call. Regular solution.
`icontr` $= 1$: Successful call. Singular solution.
`icontr` $= 2$: Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `fdf` was exceeded, see parameter `maxfun`. The best solution approximation is returned in `x`.
`icontr` $= 3$: The subroutine failed to find a point `x` satisfying all the constraints. The feasible region is presumably empty.
`icontr` < 0 : Computation did not start for the following reason,
`icontr` $= -2$: `n` ≤ 0
`icontr` $= -3$: `m` ≤ 0
`icontr` $= -4$: `l` ≤ 0
`icontr` $= -5$: `leq` < 0 or `leq` $> \min\{$`l`, `n`$\}$
`icontr` $= -9$: `dx` $= 0.0$
`icontr` $= -10$: `eps` ≤ 0.0
`icontr` $= -11$: `maxfun` ≤ 0
`icontr` $= -13$: `iw` $< 2nm+5n^2+5m+10n+4l$

Example. Minimize

$$F(\mathbf{x}) = \sum_{i=1}^{3} |f_i(\mathbf{x})| \quad ,$$

subject to the constraint

$$c(\mathbf{x}) \equiv -x_1 + x_2 + 2 \geq 0 \ .$$

The f_i are given by (1.14), page 11.

```
#include <stdio.h>
#include <math.h>
#include "f2c.h"


/*      TEST OF MI1CL1     23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *n, const int *m, const double x[],
          double df[], double f[])
{
    int df_dim1 = *m;
    double x2_2 = x2*x2, x2_3 = x2_2*x2;


    /* Function Body */
    f[0] = 1.5 - x1 * (1. - x2); /* f1(x) */
    f[1] = 2.25 - x1 * (1. - x2_2); /* f2(x) */
    f[2] = 2.625 - x1 * (1. - x2_3); /* f3(x) */

    df[0] = x2 - 1.; /* df1/dx1(x) */
    df[0 + df_dim1] = x1; /* df1/dx2(x) */
    df[1] = x2_2 - 1.; /* df2/dx1(x) */
    df[1 + df_dim1] = x1 * 2. * x2; /* df2/dx2(x) */
    df[2] = x2_3 - 1.; /* df3/dx1(x) */
    df[2 + df_dim1] = x1 * 3. * x2_2; /* df3/dx2(x) */
}

static int opti(int icontr)
{

#define N 2
#define M 3
#define L 1
```

```
#define LEQ 0
#define IW 2*M*N+5*N*N+5*M+10*N+4*L

    extern void mi1cl1(
      void (*fdf)(const int *n, const int *m, const double x[],
                  double df[], double f[]),
      int n,
      int m,
      int l,
      int leq,
      const double c[],
      const double dc[],
      double x[],
      double *dx,
      double *eps,
      int *maxfun,
      double *w,
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double c[1];
    double dc[2], w[IW], x[2];
    int index[8],optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    c[0] = 2.;
    dc[0] = -1.;
    dc[1] = 1.;
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = (optim) ? 0.1 : .001;

    mi1cl1(fdf, N, M, L, LEQ, c, dc, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
```

```
    for (k = 2; k <= 4; ++k) {
index[k - 1] = (int) w[k * 2];
index[k + 3] = (int) w[(k << 1) + 1];
    }
    printf("TEST OF GRADIENTS \n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[1],index[1],index[5]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[2],index[2],index[6]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
           "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[3],index[3],index[7]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);
    }
    else {
/*     RESULTS FROM OPTIMIZATION */
    printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL, REGULAR SOLUTION\n\n");
    break;
    case 1:
    printf("ITERATION SUCCESSFUL, SINGULAR SOLUTION\n\n");
    break;
    case 2:
    printf("NB: EPS IS TOO SMALL OR\n"
           "MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n\n");
    break;
    case 3:
    printf("NO FEASIBLE POINT FOUND - CHECK YOUR CONSTRAINTS\n\n");
    return 3;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("NUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUES AT THE SOLUTION: %18.10e\n",w[0]);
    for (j = 1; j < M; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",w[j]);
    }
    }
    return 0;
```

```
}

int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 3.00e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM BACKWARD DIFFERENCE: -1.50e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM CENTRAL  DIFFERENCE: 5.00e-07 AT FUNCTION NO 3 AND VARIABLE NO 2

MAXIMUM ELEMENT IN DF: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL, REGULAR SOLUTION

                       SOLUTION:   2.3660254038e+00
                                   3.6602540378e-01
NUMBER OF CALLS OF FDF: 9

FUNCTION VALUES AT THE SOLUTION:   0.0000000000e+00
                                   2.0096189432e-01
                                   3.7500000000e-01
```

The results indicate that the gradients of the $\{f_i\}$ are implemented correctly.

3.3. MI1CIN. Linearly Constrained Minimax Optimization of a Vector Function

Purpose. Find $\mathbf{x}^*$ that minimizes $F(\mathbf{x})$, where

$$F(\mathbf{x}) = \max_i \{ f_i(\mathbf{x}) \} \ , \tag{3.2a}$$

and where the vector of unknown parameters $\mathbf{x} = [x_1, \ldots, x_n]^\top \in \mathbb{R}^n$ must satisfy the following linear equality and inequality constraints,

$$\begin{aligned} c_i(\mathbf{x}) &\equiv \mathbf{d}_i^\top \mathbf{x} + c_i = 0 \ , \quad i = 1, 2, \ldots, l_{\text{eq}} \ , \\ c_i(\mathbf{x}) &\equiv \mathbf{d}_i^\top \mathbf{x} + c_i \geq 0 \ , \quad i = l_{\text{eq}}+1, \ldots, l \end{aligned} \tag{3.2b}$$

for given vectors $\{\mathbf{d}_i\}$ and scalars $\{c_i\}$. The f_i, $i = 1, \ldots, m$ is a set of functions that are twice continuously differentiable. The user must supply a subroutine that evaluates $\mathbf{f}(\mathbf{x})$ and the Jacobian $\mathbf{J}(\mathbf{x})$. There is an option for checking the implementation of $\mathbf{J}$.

Method. The algorithm is iterative. It is based on successive linearizations of the non-linear functions f_i, combining a first order trust region method with a local method that uses approximate second order information. The method is described in [12].

Origin. Subroutine `MLA1QS` by Jørgen Hald [11], translated from Fortran to C by the Bandler Group, [1].

Remarks. The trust region around the the current $\mathbf{x}$ is the ball centered at $\mathbf{x}$ with radius Δ defined so that the linearizations of the non-linear functions f_i are reasonably accurate for all points inside the ball. During iteration this bound is adjusted according to how well the linear approximations centered at the previous iterate predict the gain in F.

The user has to give an initial value for Δ. If the functions are almost linear, then we recommend to use an estimate of the distance between $\mathbf{x}_0$ and the solution $\mathbf{x}^*$. Otherwise, we recommend $\Delta_0 = 0.1\|\mathbf{x}_0\|$. However the initial choice of Δ is not critical because it is adjusted by the subroutine during the iteration.

The user must also supply an initial approximation for the solution $\mathbf{x}^*$. The algorithm requires that the initial point is feasible. For determination of a feasible starting point, see eg. [12] or [13].

A solution is said to be *"regular"* when it is a strict local minimum, i.e. there exists a positive number K such that

$$F(\mathbf{x}) - F(\mathbf{x}^*) \geq K\|\mathbf{x} - \mathbf{x}^*\|$$

for any feasible $\mathbf{x}$ near $\mathbf{x}^*$. Otherwise, the solution is said to be *"singular"*.

`MI1CIN` can also be used to compute a linearly constrained minimizer of the ℓ_∞-norm of $\mathbf{f}$,

$$F(\mathbf{x}) = \max_i |\, f_i(\mathbf{x}) \,| \quad . \tag{3.3a}$$

For that purpose we introduce the extended vector function $\widehat{\mathbf{f}} : \mathbb{R}^n \mapsto \mathbb{R}^{2m}$ defined by

$$\widehat{f}_i(\mathbf{x}) = \begin{cases} f_i(\mathbf{x}) & \text{for } i = 1, 2, \ldots, m \\ -f_{i-m}(\mathbf{x}) & \text{for } i = m{+}1, \ldots, 2m \end{cases} \quad . \tag{3.3b}$$

It is easily seen that $\displaystyle\max_{i=1,\ldots,2m} \{\widehat{f}_i(\mathbf{x})\} = \max_{i=1,\ldots,m} \{|f_i(\mathbf{x})|\}$.

Use. The subroutine call is

```
mi1cin(fdf,n,m,l,leq,c,dc,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

`fdf` Subroutine written by the user with the following declaration

```
void fdf(  const int *N, const int *M,
           const double x[],
           double df[], double f[] )
```

It must calculate the values of the functions and their gradients at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$ and store these numbers as follows,

$\texttt{f[i-1]} = f_{\texttt{i}}(\mathbf{x}),\ \texttt{i} = 1, \ldots, \texttt{m},$

$\texttt{df[(j-1)m+(i-1)]} = \dfrac{\partial f_{\texttt{i}}}{\partial x_{\texttt{j}}}(\mathbf{x}),\ \begin{cases} \texttt{i} = 1, \ldots, \texttt{m} \\ \texttt{j} = 1, \ldots, \texttt{n} \end{cases}$

The name of this subroutine can be chosen freely by the user.

`n` — `integer`. Number of unknowns, n.
Must be positive. Is not changed.

`m` — `integer`. Number of functions, m.
Must be positive. Is not changed.

`l` — `integer`. Number of constraints, l.
Must be positive. Is not changed.

`leq` — `integer`. Number of equality constraints, l_{eq}.
Must satisfy $0 \le \texttt{leq} \le \min\{\texttt{l}, \texttt{n}\}$. Is not changed.

`c` — `double array` with `l` elements. The constant terms in the constraints (3.2b) are stored in the following way

$$\texttt{c[i-1]} = c_{\texttt{i}}, \quad \texttt{i} = 1, \ldots, \texttt{l}\ .$$

Is not changed.

`dc` — `double array` with `l`·`n` elements. The coefficients of the constraints (3.2b) stored in the following way,

$$\texttt{dc[(i-1)n+(j-1)]} = d_{\texttt{i}}^{(\texttt{j})}, \quad \texttt{i} = 1, \ldots, \texttt{l}, \quad \texttt{j} = 1, \ldots, \texttt{n}\ .$$

Is not changed.

`x` — `double array` with `n` elements. The use depends on the entry value of `icontr`.

- `icontr` > 0 : *On entry*: Initial approximation to $\mathbf{x}^*$. Must be feasible.
  On exit: Computed solution.
- `icontr` ≤ 0 : Point at which the Jacobian should be checked. Is not changed.

`dx` — `double`. The use depends on the entry value of `icontr`.

- `icontr` > 0 : `dx` must be set by the user to an initial value of the trust region radius, which controls the step length of the iterations. See **Remarks** above. Must be positive. Is not changed.
- `icontr` ≤ 0 : Gradient check with `dx` used for h in (1.3). Must be positive. Is not changed.

`eps` `double`. Used only if the entry value of `icontr` is positive.
On entry: Desired accuracy.
The algorithm stops when it suggests to change the iterate from $\mathbf{x}_k$ to $\mathbf{x}_k+\mathbf{h}_k$ with $\|\mathbf{h}_k\| < \texttt{eps}\cdot\|\mathbf{x}_k\|$. Must be positive.
On exit: `eps` contains the length of the last step of the iteration. If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate, and `icontr` is set to 2.

`maxfun` `integer`. Used only if the entry value of `icontr` is positive.
On entry: Upper bound on the number of calls of `fdf`. Must be positive.
On exit: Number of calls of `fdf`.

`w` `double array` with `iw` elements. Work space.
Entry values are not used.
Exit values depend on the entry value of `icontr`.
$\texttt{icontr}_{\text{entry}} > 0$: The function values at the computed solution, i.e.

$$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1,\dots,\texttt{m}.$$

$\texttt{icontr}_{\text{entry}} \le 0$: Results of the gradient check are returned in the first 10 elements of `w` as follows, cf. (1.10)

`w[0]`	Maximum element in $\lvert\texttt{df}\rvert$.
`w[1]`, `w[4]`, `w[5]`	$\delta^F,\ i^F,\ j^F$.
`w[2]`, `w[6]`, `w[7]`	$\delta^B,\ i^B,\ j^B$.
`w[3]`, `w[8]`, `w[9]`	$\delta^E,\ i^E,\ j^E$.

In case of an error the indices point out the erroneous element of the Jacobian matrix.

`iw` `integer`. Length of work space `w`.
Must be at least $2nm+5n^2+4m+8n+4l+3$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
$\texttt{icontr} > 0$: Start minimization.
$\texttt{icontr} \le 0$: Check gradient. No iteration.

On exit: Information about performance,
$\texttt{icontr} = 0$: Successful call. Regular solution.
$\texttt{icontr} = 1$: Successful call. Singular solution.

`icontr` $= 2$: Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `fdf` was exceeded, see parameter `maxfun`. The best solution approximation is returned in `x`.

`icontr` $= 2$: The subroutine failed to find a point **x** satisfying all the constraints. The feasible region is presumably empty.

`icontr` < 0 : Computation did not start for the following reason,

`icontr` $= -2$: `n` ≤ 0
`icontr` $= -3$: `m` ≤ 0
`icontr` $= -4$: `l` ≤ 0
`icontr` $= -5$: `leq` < 0 or `leq` $> \min\{$`l`$,$`n`$\}$
`icontr` $= -8$: The initial point `x` is not feasible
`icontr` $= -9$: `dx` $= 0.0$
`icontr` $= -10$: `eps` ≤ 0.0
`icontr` $= -11$: `maxfun` ≤ 0
`icontr` $= -13$: `iw` $< 2nm + 5n^2 + 4m + 8n + 4l + 3$

Example. Minimize

$$F(\mathbf{x}) = \max_i |\, f_i(\mathbf{x})\,| \quad ,$$

subject to the constraint

$$c(\mathbf{x}) \equiv -x_1 + x_2 + 2 \ge 0 \ .$$

The f_i are given by (1.14), page 11. This is a problem of computing a linearly constrained minimizer of the ℓ_∞-norm of $\mathbf{f}$, and we extend the vector $\mathbf{f}$ to $\widehat{\mathbf{f}}$ as defined in (3.3b).

```
#include <stdio.h>
#include <math.h>
#include "f2c.h"


/*      TEST OF MI1CIN     23.11.2004 */

#define x1 x[0]
#define x2 x[1]

void fdf( const int *n, const int *m, const double x[],
          double df[], double f[])
{
    int df_dim1 = *m;
    double x2_2 = x2*x2, x2_3 = x2_2*x2;
    int i,j,mhalf;

    /* Function Body */
    f[0] = 1.5 - x1 * (1. - x2); /* f1(x) */
    f[1] = 2.25 - x1 * (1. - x2_2); /* f2(x) */
    f[2] = 2.625 - x1 * (1. - x2_3); /* f3(x) */

    df[0] = x2 - 1.; /* df1/dx1(x) */
    df[0 + df_dim1] = x1; /* df1/dx2(x) */
    df[1] = x2_2 - 1.; /* df2/dx1(x) */
    df[1 + df_dim1] = x1 * 2. * x2; /* df2/dx2(x) */
    df[2] = x2_3 - 1.; /* df3/dx1(x) */
    df[2 + df_dim1] = x1 * 3. * x2_2; /* df3/dx2(x) */

    /* find second half of function and gradient values */
    for (mhalf = j = 3; j < df_dim1; j++) {
f[j] = -f[j-mhalf];
for (i = 0; i < *n; i++)
    df[j + i * df_dim1] = -df[j - mhalf + i * df_dim1];
    }
} /* fdf */

static int opti(int icontr)
{

#define N 2
#define M 6
#define L 1
#define LEQ 0
#define IW 2*N*M+5*N*N+4*M+8*N+4*L+3

    extern void mi1cin(
      void (*fdf)(const int *n, const int *m, const double x[],
                  double df[], double f[]),
```

```
      int n,
      int m,
      int l,
      int leq,
      const double c[],
      const double dc[],
      double x[],
      double *dx,
      double *eps,
      int *maxfun,
      double *w,
      int iw,
      int *icontr);

    /* Local variables */
    int i, j, k;
    double c[1];
    double dc[2], w[IW], x[2];
    int index[8], optim, maxfun;
    double dx, eps;

/*     SET PARAMETERS */
    c[0] = 2.;
    dc[0] = -1.;
    dc[1] = 1.;
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    optim = icontr > 0;
    dx = (optim) ? 0.1 : .001;

    mi1cin(fdf, N, M, L, LEQ, c, dc, x, &dx, &eps, &maxfun, w, IW, &icontr);
    if (icontr < 0) {
/*    PARAMETER OUTSIDE RANGE */
    printf( "INPUT ERROR. PARAMETER NUMBER %d "
            "IS OUTSIDE ITS RANGE.\n",-icontr);
    return -icontr;
    }
    if (! optim) {
/*     RESULTS FROM GRADIENT TEST */
    for (k = 2; k <= 4; ++k) {
index[k - 1] = (int) w[k * 2];
index[k + 3] = (int) w[(k << 1) + 1];
    }
    printf("TEST OF GRADIENTS \n\n");
    printf("MAXIMUM FORWARD  DIFFERENCE: %8.2e "
```

```
            "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[1],index[1],index[5]);
    printf("MAXIMUM BACKWARD DIFFERENCE: %8.2e "
            "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[2],index[2],index[6]);
    printf("MAXIMUM CENTRAL  DIFFERENCE: %8.2e "
            "AT FUNCTION NO %d AND VARIABLE NO %d\n",
    w[3],index[3],index[7]);
    printf("\nMAXIMUM ELEMENT IN DF: %8.2e\n",w[0]);
    }
    else {
    printf("\nRESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    printf("ITERATION SUCCESSFUL, REGULAR SOLUTION\n\n");
    break;
    case 1:
    printf("ITERATION SUCCESSFUL, SINGULAR SOLUTION\n\n");
    break;
    case 2:
    printf("EPS IS TOO SMALL\n\n");
    break;
    case 3:
    printf("MAXIMUM NUMBER OF FUNCTION EVALUATIONS EXCEEDED\n");
    return 3;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("SOLUTION: %18.10e\n",x[0]);
    for (j = 1; j < N; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",x[j]);
    }

    printf("NUMBER OF CALLS OF FDF: %d\n\n", maxfun);
    printf("FUNCTION VALUES AT THE SOLUTION: %18.10e\n",w[0]);
    for (j = 1; j < M; ++j) {
        for (i = 1; i <52-18 ; ++i) putchar(' ');
        printf("%18.10e\n",w[j]);
    }
    }
    return 0;
}

int main()
{
    opti(0); /* check gradients */
    opti(1); /* optimize        */
    return 0;
}
```

We get the results

```
TEST OF GRADIENTS

MAXIMUM FORWARD  DIFFERENCE: 3.00e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM BACKWARD DIFFERENCE: -1.50e-03 AT FUNCTION NO 3 AND VARIABLE NO 2
MAXIMUM CENTRAL  DIFFERENCE: 5.00e-07 AT FUNCTION NO 3 AND VARIABLE NO 2

MAXIMUM ELEMENT IN DF: 3.00e+00

RESULTS FROM OPTIMIZATION

ITERATION SUCCESSFUL, SINGULAR SOLUTION

                       SOLUTION:   2.3660254038e+00
                                   3.6602540378e-01
NUMBER OF CALLS OF FDF: 15

FUNCTION VALUES AT THE SOLUTION:  -2.2204460493e-16
                                   2.0096189432e-01
                                   3.7500000000e-01
                                   2.2204460493e-16
                                  -2.0096189432e-01
                                  -3.7500000000e-01
```

The results indicate that the gradients of the $\{f_i\}$ are implemented correctly.

References

[1] J.W. Bandler with *Simulation Optimization Systems Research Laboratory, Department of Electrical and Computer Engineering, McMaster University, Hamilton, ON, Canada L8S 4K1* and with *Bandler Corporation, Dundas, ON, Canada L9H 5E7* (WWW: http://www.sos.mcmaster.ca and email: bandlermcmaster.ca.)

[2] E.M.L. Beale (1958): *On an Iterative Method of Finding a Local Minimum of a Function of More than one Variable.* Princeton Univ. Stat. Techn. Res. Group, Techn. Rep. 25.

[3] P. Brock, K. Madsen, and H.B. Nielsen (2004): *Robust Nongradient C Subroutines for Non-Linear Optimization.* IMM-Technical report-2004-22, Informatics and Mathematical Modelling (IMM), Technical University of Denmark.

[4] R.M. Chamberlain, C. Lemarechal, H.C. Pedersen and M.J.D. Powell (1982): *The Watchdog Technique for Forcing Convergence in Algorithms for Constrained Optimization.* MATHEMATICAL PROGRAMMING STUDY **16**, 1 – 17.

[5] J.E. Dennis, D.M. Gay and R.E. Welsch (1981a): *An adaptive nonlinear least-squares algorithm.* ACM Trans. Math. Software, Vol. 7, pp. 348-368.

[6] J.E. Dennis, D.M. Gay and R.E. Welsch (1981b): *ALGORITHM 573. NL2SOL - An adaptive nonlinear least-squares algorithm.* ACM Trans. Math. Software, Vol. 7, pp. 364-383.

[7] J.E. Dennis and R.B. Schnabel (1983): *Numerical Methods for Unconstrained Optimization and Nonlinear Equations.* Prentice Hall Series in Computational Mathematics.

[8] R. Fletcher (1987): *Practical Methods of Optimization,* 2nd edition. Wiley.

[9] P.E. Gill, W. Murray, M.A. Saunders and M.H. Wright (1983): *Computing forward difference intervals for numerical optimization.* SIAM J. SCI. STAT. COMPUT. Vol. 4, pp. 310-321.

[10] J. Hald (1981a): *MMLA1Q, a Fortran Subroutine for Linearly Constrained Minimax Optimization.* Report NI-81-01, Institute for Numerical Analysis (now part of IMM), Technical University of Denmark.

[11] J. Hald (1981b): *A 2-Stage Algorithm for Nonlinear ℓ_1 Optimization.* Report NI-81-03, Institute for Numerical Analysis (now part of IMM), Technical University of Denmark.

[12] J. Hald and K. Madsen (1981): *Combined LP and Quasi-Newton Methods for Minimax Optimization.* Mathematical Programming **20**, 49 – 62.

[13] J. Hald and K. Madsen (1985): *Combined LP and Quasi-Newton Methods for Nonlinear ℓ_1 Optimization.* SIAM J. Numer. Anal. **20**, 68 – 80.

[14] *Harwell Subroutine Library.* (1984). Report R9185, Computer Science and Systems Division, Harwell Laboratory, Oxfordshire, OX11 ORA, England.

[15] K. Madsen (1975): *An Algorithm for Minimax Solution of Overdetermined Systems of Nonlinear Equations.* J. IMA **16**, 321 – 328.

[16] K. Madsen, P. Hegelund and P.C. Hansen (1991): *Robust c Subroutines for Non-Linear Optimization.* Report NI-91-03, Institute for Numerical Analysis (now part of IMM), Technical University of Denmark.

[17] K. Madsen, H.B. Nielsen and J.Søndergaard (2002): *Robust Subroutines for Non-Linear Optimization.* Technical Report IMM-REP-2002-02, Informatics and Mathematical Modelling (IMM), Technical University of Denmark.

[18] J. Nocedal and S.J. Wright (1999): *Numerical Optimization.* Springer, New York.

[19] M.J.D. Powell (1982): *Extension to Subroutine VF02AD.* In R.F. Drenik and F. Kozin (eds.), "System Modeling and Optimization", Lecture Notes in Control and Informations Sciences **38**, Springer-Verlag, 529 – 538.

[20] M.J.D. Powell (1985): *On the Quadratic Programming Algorithm of Goldfarb and Idnani.* Mathematical Programming Study **25**, 46 – 61.

[21] P. Wolfe (1982): *Checking the Calculation of Gradients.* ACM TOMS., Vol. 8. pp. 337-343.