IMM
INFORMATICS AND MATHEMATICAL MODELLING

Technical University of Denmark
DK-2800 Kongens Lyngby – Denmark

ROBUST NON-GRADIENT C SUBROUTINES FOR NON-LINEAR OPTIMIZATION

Pernille Brock
Kaj Madsen
Hans Bruun Nielsen

Revised version of report NI-91-04

IMM-Technical Report-2004-22

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, where gradient information is not required. 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 requiring gradient information. The parameter lists for the subroutines in both collections are similar so it is easy to switch between the non-gradient and the gradient methods. All of the subroutine names in this report start with `MIO`. The corresponding names of the gradient subroutines are obtained by changing `0` to `1`.

The present report is a new and updated version of a previous report NI-91-04 with the title *Non-gradient c Subroutines for Non-Linear Optimization*, [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been translated from Fortran to C. The reason for writing the present report is that some of the C subroutines have been replaced by more effective and robust versions translated from the original Fortran subroutines to C by the Bandler Group, see [1].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 the function value $F(\mathbf{x})$. In case of a vector function the user's subroutine must return the vector $\mathbf{f}(\mathbf{x}) = [f_1 \ldots f_m]^\top$. For an efficient performance of the optimization algorithm the function must be implemented without errors. It is however not possible to check the correctness of the implementation of F (or $\mathbf{f}$).

1.2. Non-gradient versus gradient methods

Most efficient optimization software requires user-implemented gradients of the functions. This gives faster and more reliable methods described in e.g. [3]. A recurrent reason, however, for erroneous runs with programs including this kind of methods is that the user makes errors when implementing the gradients. Mostly such an error will prevent the program from providing convergence. Occasionally we see convergence in spite of erroneous gradients, but the accuracy and speed of the method are ruined. Often the functions to be minimized are difficult or impossible to differentiate although they are differentiable. Thus there are many reasons for providing subroutines which do not rely upon user-implemented gradients but either compute a difference approximation or use special non-gradient algorithms.

For most of the unconstrained problem types, namely the subroutines `MIOF`, `MIOL2`, and `MIOINF`, we provide two subroutines:

(i) An implementation of a non-gradient method where gradient approximations are calculated automatically using the function values during the iteration.

(ii) An implementation of a gradient method (from [3]) where the gradients are approximated by differences at each iterate. Thus for an n-dimensional problem $(n + 1)$ function evaluations are needed for each iteration of the method.

The two methods have similar argument lists so, in principle, the user only needs to change one parameter in order to change method. The choice beteen the two methods is communicated through the variable `icontr` of the parameter list.

The first method is a non-gradient method, the second is a gradient method with difference approximations of the gradients. The rationale is that non-gradient methods are faster but not always robust, i.e. they may fail to provide convergence. Since the two methods are different, e.g. in regard to the size of the necessary work area, they are described seperately but illustrated with a common example. Note that the extra n function calls needed to approximate the gradient at each iterate are not counted by the examples. Thus the number of function evaluations is about $(n + 1)$ times the number of iterations given.

For the unconstrained problem, `MIOL1`, as well as for the constrained problem types, `MIOCF`, `MIOCL1`, and `MIOCIN`, we only provide subroutines of type (ii).

1.3. Approximation of gradients

In the subroutine class (ii), see the previous section, the gradients are approximated using a computed difference approximation at a single point. This is not an easy task, because the result depends on the step length used in the difference. If this step length is too large then the result may be useless because of truncation errors, and if the step length is too small then the result may be destroyed by rounding errors.

As an illustration we describe the difference approximation via a real function in one variable. The approximations are made using function values from the center point x and one other point.

The approximation to $\frac{\mathrm{d}f}{\mathrm{d}x}(x)$ is the forward difference approximation given by:

$$\frac{f(x+h)-f(x)}{h} \tag{1.1}$$

If the function is twice differentiable then the truncation error associated with this approximation is:

$$\begin{aligned} D_F(h) &= \frac{\mathrm{d}f}{\mathrm{d}x}(x) - \frac{f(x+h)-f(x)}{h} \\ &\simeq k_1 h \end{aligned} \tag{1.2}$$

where k_1 is a constant.

Now let g be the computed derivative of f at the point x:

$$g = fl\left(\frac{\mathrm{d}f}{\mathrm{d}x}(x)\right) \tag{1.3}$$

where fl denotes the floating point number representation.
The deviation between g and the difference approximation has three contributions:

the *truncation* error,

the *rounding* errors in the computation of the difference approximation and of g,

the errors (if any) in the implementation of the derivative $\frac{\mathrm{d}f}{\mathrm{d}x}(x)$.

These three deviation contributions are characterized by the way in which they depend on h:

(1) The truncation error decreases with decreasing h as $D_F(h) = O(h)$.

(2) The rounding errors usually varies like $O(h^{-1})$, i.e. it decreases when h decreases.

(3) The implementation error is presumably independent of h.

As h decreases, the truncation contribution to the deviation decreases whereas the rounding error contribution increases. If h gets small enough, then the rounding error contribution starts to dominate, usually this happens when h passes a threshold value of about $|x|\sqrt{\epsilon_M}$, ϵ_M being the machine accuracy. For even smaller values of h the deviations will increase with decreasing step length h.

As a compromise between the two first deviation contributions h is chosen as $10^{-3}|x|$, a fair choice for the accuracy used by today's computers.

It is straigthforward to generalize the case of a scalar function to an m-dimensional vector function of an n-dimensional variable. In this case the exact gradients are given by the *Jacobian matrix* $\mathbf{J}(\mathbf{x}) \in \mathbb{R}^{m \times n}$, defined by

$$\mathbf{J} = \begin{bmatrix} \dfrac{\partial f_1}{\partial x_1} & \cdots & \dfrac{\partial f_1}{\partial x_n} \\ \vdots & & \vdots \\ \dfrac{\partial f_m}{\partial x_1} & \cdots & \dfrac{\partial f_m}{\partial x_n} \end{bmatrix}, \tag{1.4}$$

i.e., the ith row in $\mathbf{J}$ is the gradient of f_i, the ith component of $\mathbf{f}$. Now each of the Jacobian elements should be approximated by a forward difference approximation calculated by taking a step $\mathbf{h}$ in the proper direction. Jacobian element (i, j) is then approximated by the difference approximation,

$$\frac{\partial f_i}{\partial x_j}(\mathbf{x}) \simeq \frac{f_i(\mathbf{x}+h\mathbf{e}_j) - f_i(\mathbf{x})}{h} \tag{1.5}$$

where h is the step length and $\mathbf{e}_j$ is a unit vector in the jth direction.

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],

$$\begin{aligned} f_1(\mathbf{x}) &= \;\; 1.5 \;\; - x_1(1-x_2) \\ f_2(\mathbf{x}) &= \;\; 2.25 \; - x_1(1-x_2^2) \\ f_3(\mathbf{x}) &= 2.625 - x_1(1-x_2^3) \end{aligned} \tag{1.6}$$

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 directory. 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 `MIOCIN` 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].

1.6. Package overview

The package contains the following subroutines:

`MI0F`: Unconstrained minimization of a scalar function.
Origin of non-gradient method: `VA04A`, [14].
Origin of gradient method: `VA13CD`, [14].

`MI0L2`: Unconstrained minimization of the ℓ_2-norm of a vector function (least squares).
Origin of non-gradient method: `VA05AD`, [14].
Origin of gradient method: `NL2SOL`, [5] and [6].

`MI0L1`: Unconstrained minimization of the ℓ_1-norm of a vector function.
Origin: `L1NLS`, [11]

`MI0INF`: Unconstrained minimization of the ℓ_∞-norm of a vector function.
Origin of non-gradient method: `VG02AD`, [14].
Origin of gradient method: `SUB1W`, [15].

`MI0CF`: Minimization of a non-linear function subject to non-linear constraints (mathematical programming).
Origin: `VF13AD`, [20].

`MI0CL1`: Linearly constrained minimization of the ℓ_1-norm of a vector function.
Origin: `L1NLS`, [11].

`MI0CIN`: 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. MI0F. Minimization of a Scalar Function

Non-gradient version

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 $F(\mathbf{x})$.

Method. A variation of the simple method of changing one variable at a time is used. The method has the property that when applied to a quadratic form, it causes conjugate directions of search to be chosen, thus finishing in at most n steps. Thus when applied to a general function the ultimate rate of convergence is fast, see [19]. The minimum will almost never be found in less than n iterations, each iteration using at least $2n$ different values of the function. Each iteration causes the function to decrease, except when the ultimate convergence criterion is met.

Origin. `MI0F` uses a modified version of the subroutine `VA04AD` from [14], modified such that it is consistent with the other routines in this package.

Ultimate convergence criterion. First, convergence will be assumed when an iteration changes each variable by less than 10% of the required accuracy as defined by `eps`. Such a point is found and is often displaced by 10 times the required accuracy in each variable. Minimization is then continued from the new point until a change of less than 10% is again made by an iteration. The two estimates of the minimum are finally compared and the best is returned.

Recommendations. The following points are recommmended:

(i) Set `dx` as large as reasonable, remembering that it should prevent the maximum step from jumping from one valley to another.

(ii) Set the required accuracy so that `dx` > 100.

(iii) If the results appear unreasonable, try with different initial values of the variables set in the point `x`.

Use. The subroutine call is

```
mi0f(calf,n,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

It must calculate the value of the objective function at the point $\mathbf{x} = [$`x[0]`$, \ldots,$ `x[n-1]`$]^\top$, $n =$ `*N`, and store the function value as `*f` $= F(\mathbf{x})$.

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

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

`x` `double array` with `n` elements.
On entry: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution. `x[j-1]` will contain the value of $\mathbf{x}_j$ at the minimum position to the required accuracy, see **Ultimate Convergence criterion** above.

`dx` `double`.
`dx` must be set by the user to control the step length of the iterations. The variable $\mathbf{x}_j$ will be changed by not more than `dx` $\cdot$ `eps`. See **Recommendations** above. Must be positive. Is not changed.

`eps` `double`. Desired accuracy.
`eps` must be set by the user to the desired absolute accuracy of the solution. It is assumed that the variables are scaled to be roughly proportional. Must be positive. Is not changed.

`maxfun` `integer`.
On entry: Upper bound on the number of iterations. Must be positive.
On exit: Number of calls of `calf`.

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: `w[0]` $= F($`X`$)$, the computed minimum.

`iw` `integer`. Length of work space `w`.
Must be at least $n(n+4)$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
`icontr` $= 0$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration successful.
`icontr` $= 1$: Iteration successful.
`icontr` $= 2$: Iteration stopped because the maximum number of iterations was exceeded, see parameter `maxfun`.
`icontr` $= 3$: Maximum change does not alter function.
`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` $< n(n+4)$
`icontr` $= -9$: $\texttt{icontr}_{\text{entry}} < 0$

Numerical gradient version

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 $F(\mathbf{x})$.

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], except for the fact that first order derivatives are approximated by finite differences.

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

```
miOf(calf,n,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

It must calculate the value of the objective function at the point $\mathbf{x} = [$`x[0]`$, \ldots,$ `x[n-1]`$]^\top$, $n =$ `*N`, and store the function value as `*f` $= F(\mathbf{x})$.

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

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

`x` `double array` with `n` elements.
On entry: Initial approximation to $\mathbf{x}^*$.
On exit: Computed solution.

`dx` `double`.

---

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

`dx` does not enter into the computations, but is present to be consistent with the other methods described in this report.

`eps` — `double`. Desired accuracy.
`eps` must be set by the user to the desired absolute accuracy of the solution. 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. Is not changed.

`maxfun` — `integer`.
On entry: Upper bound on the number of calls of `calf`. Must be positive.
On exit: Number of calls of `calf`.

`w` — `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: `w[0]` $= F(\texttt{X})$, the computed minimum.

`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` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration successful.
`icontr` $= 2$: Iteration stopped because the maximum number of calls to `calf` was exceeded, see parameter `maxfun`.
`icontr` < 0 : Computation did not start for the following reason,

- `icontr` $= -2$: `n` ≤ 0
- `icontr` $= -5$: `eps` ≤ 0.0
- `icontr` $= -6$: `maxfun` ≤ 0
- `icontr` $= -8$: `iw` $< n(n+4)$
- `icontr` $= -9$: $\texttt{icontr}_{\text{entry}} < 0$

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 MIOF      23.11.2004 */

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

void calf( const int *n, const double x[],
           double *f)
{
    double cexp;

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

} /* calf */

static int opti(int icontr)
{

#define N 2
#define IW max(N*(N+4),N*(N+15)/2+1)

    extern void mi0f(
      void (*calf)(),
      int n,
      double x[],
      const double *dx,
      const double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i,k;
    double w[IW], x[2];
    int maxfun, method = icontr;
```

```
    double eps, dx;

/*     SET PARAMETERS */
    eps = (method==0) ? 0.005 : 1e-10;
    maxfun = (method==0) ? 4 :  19;
    dx = 101;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 2.;

    switch (method) {
    case 0:
      printf("Optimize : non-gradient method\n\n");
      break;
    case 1:
      printf("Optimize : gradient method with approx. gradient\n\n");
      break;
    default:
      printf("Not Implemented\n\n");
      break;
    }

    mi0f(calf, 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;
    }

    switch (icontr) {
    case 3:
    printf("NB: maximum does not alter function\n\n");
    break;
    case 1:
    /* NOT an error : message below would only cause confusion
    printf("NB: accuracy limited by errors in f\n\n");
    */
    break;
    case 2:
    printf("NB: maximum number of %s exceeded\n\n",
        (method == 0) ? "iterations" : "function calls" );
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("Solution: %18.10e\n",x[0]);
    for (i = 1; i < N; ++i) {
        for (k = 1; k <52-18 ; ++k) putchar(' ');
        printf("%18.10e\n\n",x[i]);
    }
```

```
    printf("Number of %s: %d\n\n",
        (method == 1) ? "iterations" : "function calls" ,
maxfun);
    printf("Function value at the solution: %18.10e\n\n\n",w[0]);

    return 0;
}

int main()
{
    opti(0); /* non-gradient method */
    opti(1); /* gradient method with numerical gradient */

    return 0;
}
```

We get the results

```
Optimize : non-gradient method

                       Solution:  -1.4423577074e+00
                                   1.0931956724e+00

Number of function calls: 26

Function value at the solution:   1.8284544974e+00


Optimize : gradient method with approx. gradient

                       Solution:  -1.4375562056e+00
                                   1.0911558152e+00

Number of iterations: 14

Function value at the solution:   1.8284295839e+00
```

2.2. MI0L2. Minimization of the ℓ_2-Norm of a Vector Function (Least Squares)

Non-gradient version

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 $\mathbf{f}(\mathbf{x})$.

Method. The method is a compromise between three different algorithms for minimizing a sum of squares, namely Newton-Raphson, Steepest Descent, and Marquardt. Moreover it automatically obtains and improves an approximation to the first derivative matrix following the ideas of Broyden.

Origin. `MI0L2` uses a modified version of the subroutine `VA05AD` from [14], modified such that it is consistent with the other routines in this package.

Choice of scaling. The method requires some step length control, for which the "distance" between two estimates $\mathbf{a} = [a_1, \ldots, a_n]^\top$ and $\mathbf{b} = [b_1, \ldots, b_n]^\top$ of the vector $\mathbf{x} = [x_1, \ldots, x_n]^\top$ is

$$\left(\sum_{j=1}^{n}(a_j - b_j)^2\right)^{\frac{1}{2}} \tag{2.2}$$

Therefore it is important that the user shall scale the variables, perhaps by multiplying them by appropriate constants, so that their magnitudes are similar.

Accuracy. Note that a normal return from the subroutine occurs when it is predicted that the required accuracy has been obtained. Sometimes this prediction may be wrong, and we quote from [14]:

> *which has been shown by an example with $M = 60$, $N = 30$, having the property that the correct final value of the sum of squares, $F(\mathbf{x})$ is large. Here the subroutine finishes too soon, but the final answer was very close to the required one. Experience with the subroutine on about thirty other examples has shown that the convergence criterion is usually adequate.*

The next section describes a method that often works much better than this method on large residual problems.

Use. The subroutine call is

```
mi0l2(calf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

It must calculate the value of the vector function at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$ and store the function values as,

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

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

`dx` `double`.
`dx` is used as the absolute step length in the difference approximations which serve as estimates for the partial derivatives of the vector function. See section 1.3 for further details. See also **Choice of scaling** above. `dx` must be non-zero. `dx` is not altered.

`eps` `double`. Desired accuracy.
`eps` must be set by the user to the desired absolute accuracy of the solution. Must be positive. The required accuracy is considered obtained when it is predicted that the best calculated value of $F(\mathbf{x})$ is not more than `eps` greater than the minimum value. See **Accuracy** above.

`maxfun` `integer`.
On entry: Upper bound on the number of calls of `calf`. As an exception to this upper bound, at least $(n+1)$ calls of `calf` are made due to the nature of the non-gradient method. Must be positive.
On exit: Number of calls of `calf`.

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The function values at the computed solution are stored in the first m elements of `w` as,

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

The approximations to the partial derivatives at the computed solution are stored in the next $n \cdot m$ elements of `w` as,

$$\mathtt{w[n(i-1)+(j-1)+m]} = \frac{\partial f_{\mathtt{i}}}{\partial x_{\mathtt{j}}}(\mathbf{x}),\ \mathtt{i} = 1, \ldots, \mathtt{m}, \mathtt{j} = 1, \ldots, \mathtt{n}.$$

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

`icontr` `integer`.
On entry: Controls the computation,
`icontr` $= 0$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration successful.
`icontr` $= 1$: Iteration successful.
`icontr` $= 2$: Iteration stopped because the maximum number of calls to `calf` 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` $< 2nm + 2n^2 + 3m + 5n$
`icontr` $= -10$: $\texttt{icontr}_{\text{entry}} < 0$

Numerical gradient version

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

$$F(\mathbf{x}) = \tfrac{1}{2}\sum_{i=1}^{m}(f_i(\mathbf{x}))^2 \quad . \tag{2.3}$$

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 $\mathbf{f}(\mathbf{x})$.

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 guesses, 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 identical to that described in [5] and [6] except for the fact that first order derivatives are approximated by finite differences.

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

Use. The subroutine call is

```
mi0l2(calf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

It must calculate the value of the vector function at the point $\mathbf{x} = [$`x[0]`$, \ldots,$ `x[n-1]`$]^\top$, $n =$ `*N`, $m =$ `*M` and store the function value as,

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

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

`dx` `double`.
`dx` does not enter into the computations, but is present to be consistent with the other methods described in this report.

`eps` `double`.
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 $\|\mathbf{h}_k\| < \mathtt{eps}\cdot\|\mathbf{x}_k\|$.
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`.
On entry: Upper bound on the number of calls of `calf`. Must be positive.
On exit: Number of calls of `calf`.

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.
$\mathtt{w[i-1]} = f_{\mathtt{i}}(\mathtt{x}),\ \mathtt{i} = 1, \ldots, \mathtt{m}.$

`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,
`icontr` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful.
`icontr` $= 1$: Iteration succesful.

`icontr` $= 2$: Iteration stopped, because the maximum number of calls of `calf` 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` $= -6$: `eps` ≤ 0.0
- `icontr` $= -7$: `maxfun` ≤ 0
- `icontr` $= -9$: `iw` $< m(2n+4) + \frac{1}{2}n(3n+33) + 93$
- `icontr` $= -10$: $\texttt{icontr}_{\text{entry}} < 0$

Example. Minimize

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

where the f_i are given by (1.6), page 9.

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


/*      TEST OF MI0L2     23.11.2004 */

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

void calf( const int *n, const int *m,
           const double x[], double f[])
{
    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) */
```

```
} /* calf */

static int opti(int icontr)
{

#define N 2
#define M 3
#define IW max(2*M*N+2*N*N+3*M+5*N,(2*N+4)*M+N*(3*N+33)/2+93)

extern void mi0l2(
      void (*calf)(),
      int n,
      int m,
      double x[],
      double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j;
    double w[IW], x[N];
    int maxfun, method = icontr;
    double eps, dx;

/*     SET PARAMETERS */
    dx = 0.07;
    eps = 1e-10;
    maxfun = 40;

/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;

    switch (method) {
    case 0:
    printf("\nOptimize with non-gradient method\n\n");
    break;
    case 1:
    printf("\nOptimize with approx. gradient method\n\n");
    break;
    }

    mi0l2(calf, 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;
    }

/*     RESULTS FROM OPTIMIZATION */
    switch (icontr) {
    case 1:
    printf("Sum of squares fails to decrease\n\n");
    break;
    case 2:
    printf("Upper Limit for Function Evaluations Exceeded.\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("Solution: %18.10e\n",x[0]);
    for (i = 1; i < N; ++i) {
        for (j = 1; j <52-18 ; ++j) putchar(' ');
        printf("%18.10e\n",x[i]);
    }

    printf("Number of %s: %d\n\n",
        (method == 1) ? "iterations" : "function calls" ,
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]);
    }
    if (method == 0) {
      printf("Approx. of partial derivatives\n");
      for (i = 0; i < M; i++)
      for (j = 0; j < N; j++)
        printf("%18.10e\n",w[M + N*i + j]);
    }

    return 0;
}

int main()
{
    opti(0); /* non-gradient method */
    opti(1); /* gradient method with numerical gradient */

    return 0;
}
```

We get the results

```
Optimize with non-gradient method

                       Solution:   3.0000056975e+00
                                   5.0000129484e-01
Number of function calls: 25

Function Values at the Solution:   1.0357872369e-06
                                  -3.8858482787e-07
                                  -2.0719062168e-06
Approx. of partial derivatives
 -4.8186487403e-01
  2.9893557811e+00
 -7.6599384516e-01
  3.1860482934e+00
 -9.1547852714e-01
  2.5503763257e+00

Optimize with approx. gradient method

                       Solution:   3.0000000000e+00
                                   5.0000000000e-01
Number of iterations: 10

Function Values at the Solution:   1.7208456882e-13
                                   1.5329959524e-12
                                   2.8510527272e-12
```

2.3. MI0L1. Minimization of the ℓ_1-Norm of a Vector Function

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

$$F(\mathbf{x}) = \sum_{i=1}^{m} |\, f_i(\mathbf{x})\,| \quad . \tag{2.4}$$

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 $\mathbf{f}(\mathbf{x})$.

Method. The method 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 identical to that described in [13], except from the fact that first order derivatives are approximated by finite differences.

Origin. Subroutine `L1NLS` from [11].

Remark. The trust region around 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

```
mi0l1(calf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

It must calculate the value of the vector function at the point $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$, $n = \texttt{*N}$, $m = \texttt{*M}$, and store the function value as,

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

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

`dx` `double`.
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.

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

`w` `double array` with `iw` elements. Work space.

On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.
`w[i-1]` $= f_{\mathtt{i}}(\mathtt{x}),\ \mathtt{i} = 1,\ldots,\mathtt{m}$.

`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,
`icontr` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful.
`icontr` $= 1$: Iteration succesful.
`icontr` $= 2$: Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `calf` 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` $< 2nm + 5n^2 + 11n + 5m + 5$
`icontr` $= -10$: $\mathtt{icontr}_{\mathrm{entry}} \le 0$

Example. Minimize

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

where the f_i are given by (1.6), page 9.

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

```
/*      TEST OF MIOL1     23.11.2004 */

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

void calf( const int *n, const int *m,
           const double x[], double f[])
{
    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) */
} /* calf */

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 mi0l1(
      void (*calf)(),
      int n,
      int m,
      double x[],
      double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

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

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun =  25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
    dx = 0.1;
```

```
    mi0l1(calf, 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;
    }
    printf("\nRESULTS FROM OPTIMIZATION\n\n");
    if ((icontr == 0) || (icontr == 1)) {
    printf("Iteration succesful.\n\n");
    }
    if (icontr == 2) {
    printf("Maximum number of function evaluations exceeded.\n\n");
    }
    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 iterations: %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(1);
    return 0;
}
```

We get the results

```
RESULTS FROM OPTIMIZATION

Iteration succesful.

                       Solution:   3.0000000000e+00
                                   5.0000000000e-01
Number of iterations: 11

Function values at the solution:   2.2204460493e-16
                                   4.4408920985e-16
                                   4.4408920985e-16
```

2.4. MI0INF. Minimization of the ℓ_∞-Norm of a Vector Function

Non-gradient version

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

$$F(\mathbf{x}) = \max_i |f_i(\mathbf{x})| \quad . \tag{2.5}$$

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 $\mathbf{f}(\mathbf{x})$. An important application of the subroutine is non-linear minimax data fitting, and it is also an efficient method for finding a zero of a set of non-linear equations.

Origin. This is a modified version of `VG02AD` from [14], modified such that it is consistent with the other routines in this package.

Remark. The trust region around 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

```
mi0inf(calf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

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

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

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

`dx` `double`.
`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.

`eps` `double`.
On entry: Desired accuracy.
The algorithm stops when $\mathbf{x} = [\texttt{x[0]}, \ldots, \texttt{x[n-1]}]^\top$ is a nearby stationary point, more precisely when
$\max_i |f_i(\mathbf{x})| - \min_{\|\mathbf{h}\| \le \|\mathbf{x}\|} (\max_i |f_i(\mathbf{x}) + (\Delta f_i(\mathbf{x}), \mathbf{h})|) < \texttt{eps}$
Must be positive.
On exit: If `eps` was chosen too small, then the iteration stops when there is indication that rounding errors dominate. If the inequality above is satisfied, `eps` will not be changed, otherwise `eps` will be set to 0.0.

`maxfun` `integer`.
On entry: Upper bound on the number of calls of `calf`. Must be positive.
On exit: Number of calls of `calf`. If `maxfun` is exceeded, `eps` will be set to 0.0.

`w` `double array` with `iw` elements. Work space.

On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.
$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1, \ldots, \texttt{m}$.

`iw` `integer`. Length of work space `w`.
Must be at least $nm + 2n^2 + 3m + 13m + 16$. Is not changed.

`icontr` `integer`.
On entry: Controls the computation,
`icontr` $= 0$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful.
`icontr` $= 1$: Iteration succesful.
`icontr` $= 2$: Iteration stopped, because the maximum number of calls of `calf` 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` $< nm + 2n^2 + 3m + 13m + 16$
`icontr` $= -10$: $\texttt{icontr}_{\text{entry}} < 0$

Numerical gradient version

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

$$F(\mathbf{x}) = \max_i |f_i(\mathbf{x})| \quad . \tag{2.6}$$

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 $\mathbf{f}(\mathbf{x})$.

Method. The method 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], apart from the fact that first order derivatives are approximated by finite differences.

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

```
mi0inf(calf,n,m,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

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

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

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

`dx` `double`.
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.

`eps` `double`.
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 1.

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

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.

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

`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` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful.
`icontr` $= 1$: Iteration stopped because rounding errors dominate.
`icontr` $= 2$: Iteration stopped, because the maximum number of calls of `calf` 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` $< 2nm + n^2 + 14n + 4m + 11$
`icontr` $= -10$: $\texttt{icontr}_{\text{entry}} < 0$

Example. Minimize

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

where the f_i are given by (1.6), page 9.

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

/*      TEST OF MI0INF     23.11.2004 */

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

void calf( const int *n, const int *m,
```

```
   const double x[], double f[])
{
    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) */

}


static int opti(int icontr)
{

#define N 2
#define M 3
#define IW max(M*N+2*N*N+3*M+13*N+16,2*N*M+N*N+14*N+4*M+11)

extern void mi0inf(
      void (*calf)(),
      int n,
      int m,
      double x[],
      double *dx,
      double *eps,
      int *maxfun,
      double w[],
      int iw,
      int *icontr);

    /* Local variables */
    int i, j;
    double w[IW], x[N];
    double dx, eps;
    int maxfun, method = icontr;

/*     SET PARAMETERS */
    maxfun = 34;
    eps = 1e-10;
    dx = (method==0) ? 0.07 : 0.1;

/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;

    switch (method) {
    case 0:
      printf("\nOptimize : non-gradient method\n\n");
```

```
      break;
    case 1:
      printf("\nOptimize : approx. gradient method\n\n");
      break;
    }

    mi0inf(calf, 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;
    }

/*     RESULTS FROM OPTIMIZATION */
    switch (icontr) {
    case 1:
    printf("Solution surrounded by errors.\n\n");
    break;
    case 2:
    printf("Upper Limit for Function Evaluations Exceeded.\n\n");
    break;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("Solution: %18.10e\n",x[0]);
    for (i = 1; i < N; ++i) {
        for (j = 1; j <52-18 ; ++j) putchar(' ');
        printf("%18.10e\n",x[i]);
    }

    printf("Number of %s: %d\n\n",
(method==1) ? "Iterations" : "Function Calls",
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); /* non-gradient method */
    opti(1); /* gradient method with numerical gradient */

    return 0;
}
```

We get the results

```
Optimize : non-gradient method

                       Solution:   3.0000000000e+00
                                   5.0000000000e-01
Number of Function Calls: 22

Function values at the solution:   0.0000000000e+00
                                   0.0000000000e+00
                                   0.0000000000e+00

Optimize : approx. gradient method

                       Solution:   3.0000000000e+00
                                   5.0000000000e-01
Number of Iterations: 12

Function values at the solution:  -2.2204460493e-16
                                  -4.4408920985e-16
                                  -8.8817841970e-16
```

3. Constrained Optimization

3.1. MI0CF. 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,

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

The objective function F and the constraint functions $\{c_i\}$ must be twice continuously differentiable. The user must supply a subroutine that evaluates $F(\mathbf{x})$, $\{c_i(\mathbf{x})\}$.

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 [20]. The quadratic programming algorithm is described in [21], except from the fact that first order derivatives are approximated by finite differences.

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

Use. The subroutine call is

```
mi0cf(calfc,n,l,leq,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

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

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

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

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 $0 \le \texttt{leq} \le \min\{\texttt{l}, \texttt{n}\}$. Is not changed.

`x` `double array` with `n` elements.
On entry: Initial approximation to $\mathbf{x}^*$. It needs not satisfy the constraints.
On exit: Computed solution.

`dx` `double`.
`dx` does not enter into the computations, but is present to be consistent with the other methods described in this report.

`eps` `double`. Desired accuracy.
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.

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

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The optimal objective function value and the constraint function values are stored in the following way,

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

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

`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` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful.
`icontr` $= 1$: Iteration succesful.
`icontr` $= 2$: Iteration stopped because the maximum number of calls of `calfc` was exceeded, see `maxfun`.
`icontr` $= 3$: Iteration stopped due to problems with the method. Try a new starting point.
`icontr` $= 4$: Iteration stopped due to problems with the method. Try a new starting point.
`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` $= -7$: `eps` ≤ 0.0
`icontr` $= -8$: `maxfun` ≤ 0
`icontr` $= -10$: `iw` $< \frac{5}{2}n(n+9) + (n+8)l + 15$
`icontr` $= -11$: $\texttt{icontr}_{\text{entry}} \leq 0$

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 MIOCF      23.11.2004 */

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

void calfc( const int *n, const int *l,
            const double x[],
            double *f, double c[])
{
    double x1_2 = x1*x1, x1_3 = x1_2*x1;
    double cexp;

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

/*     CONSTRAINTS */

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

} /* calfc */

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 mi0cf(
      void (*calfc)(),
      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;
    double w[IW], x[2];
    double dx, eps;
    int maxfun;

/*     SET PARAMETERS */
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
/*     GRADIENT CHECK OR MINIMIZATION */
    dx = 0.1;

    mi0cf(calfc, 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;
    }
    printf("RESULTS FROM OPTIMIZATION\n\n");
    switch (icontr) {
    case 0:
    case 1:
    break;
    case 2:
    printf("NB: maximum number of function evaluations exceeded");
    break;
    case 3:
    case 4:
    printf("Method has failed. Try a new starting point\n");
    return icontr;
    case 5:
    printf("MI0CF 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 < N; ++i) {
        for (j = 1; j <52-18 ; ++j) putchar(' ');
        printf("%18.10e\n\n",x[i]);
    }

    printf("Number of iterations: %d\n\n", maxfun);
    printf("Function value at the solution: %21.10e\n\n\n",w[0]);
    for (j = 1; j <= L; ++j) {
```

```
        if (j==1)
            printf("Constraint values at the solution: %18.10e\n",w[1]);
        else {
            for (i = 1; i <54-18 ; ++i) putchar(' ');
            printf("%18.10e\n",w[j]);
        }
    }
    return 0;
}
int main()
{
    opti(1); /* optimize        */

    return 0;
}
```

We get the results

```
RESULTS FROM OPTIMIZATION

                       Solution:  -8.2645035400e-01
                                   5.6300960240e-01

Number of iterations: 10

Function value at the solution:      2.3895160146e+00


Constraint values at the solution:  -2.3047785902e-11
                                     1.1274918783e+00
                                     2.9956885079e-01
```

3.2. MI0CL1. 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,

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

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 $\mathbf{f}(\mathbf{x})$.

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 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

```
mi0cl1(calf,n,m,l,leq,c,dc,x,&dx,&eps,&maxfun,w,iw,&icontr
```

The parameters are

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

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

It must calculate the value of the objective function at the point $\mathbf{x} = [\mathtt{x[0]}, \ldots, \mathtt{x[n-1]}]^\top$, $n = \mathtt{*N}$, $m = \mathtt{*M}$ and store the function values as,

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

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 $0 \le \mathtt{leq} \le \min\{\mathtt{l}, \mathtt{n}\}$. Is not changed.

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

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

Is not changed.

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

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

Is not changed.

`x` `double array` with `n` elements.
On entry: Initial approximation to $\mathbf{x}^*$. It needs not satisfy the constraints.
On exit: Computed solution.

`dx` `double`.

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.

`eps` `double`.
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` will contain the step length used in the last 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`.
On entry: Upper bound on the number of calls of `calf`. Must be positive.
On exit: Number of calls of `calf`.

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.
$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}),\ \texttt{i} = 1, \ldots, \texttt{m}$.

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

`icontr` `integer`.
On entry: Controls the computation,
`icontr` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful. Regular solution.
`icontr` $= 1$: Iteration succesful. Singular solution.
`icontr` $= 2$: Iteration stopped, either because `eps` is too small, or because the maximum number of calls of `calf` was exceeded, see parameter `maxfun`. The best solution approximation is returned in `x`.

`icontr` $= 3$: The subroutine failed to find a point $\mathbf{x}$ 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` $> \min\{\mathtt{l}, \mathtt{n}\}$
`icontr` $= -9$: `dx` $= 0.0$
`icontr` $= -10$: `eps` ≤ 0.0
`icontr` $= -11$: `maxfun` ≤ 0
`icontr` $= -13$: `iw` $< 2nm+5n^2+5m+10n+4l$
`icontr` $= -14$: $\mathtt{icontr}_{\text{entry}} \le 0$

Example. Minimize

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

subject to the constraint

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

The f_i are given by (1.6), page 9.

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


/*      TEST OF MIOCL1      23.11.2004 */

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

void calf( const int *n, const int *m,
   const double x[], double f[])
{
    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) */

} /* calf */

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 mi0cl1(
      void (*calf)(),
      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;
    double c[1];
    double dc[2], w[IW], x[2];
    double dx, eps;
    int maxfun;

/*     SET PARAMETERS */
    c[0] = 2.;
    dc[0] = -1.;
    dc[1] = 1.;
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
```

```
    dx =  0.1;

    mi0cl1(calf, 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;
    }
    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: Required accuracy not achieved\n\n");
    break;
    case 3:
    printf("No feasible point found - check your constraints\n");
    return 3;
    }
    for (i = 1; i <= 23; ++i) putchar(' ');
    printf("Solution: %18.10e\n",x[0]);
    for (i = 1; i < N; ++i) {
        for (j = 1; j <52-18 ; ++j) putchar(' ');
        printf("%18.10e\n",x[i]);
    }

    printf("Number of calls of calf: %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(1);

    return 0;
}
```

We get the results

```
RESULTS FROM OPTIMIZATION

Iteration successful, regular solution

                       Solution:   2.3660254038e+00
                                   3.6602540378e-01
Number of calls of calf: 9

Function values at the solution:  -2.2204460493e-16
                                   2.0096189432e-01
                                   3.7500000000e-01
```

3.3. MI0CIN. 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,

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

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 $\mathbf{f}(\mathbf{x})$.

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], except from the fact that first order derivatives are approximated by finite differences.

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

Remarks. The trust region around 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"*.

MIOCIN 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 $\max\limits_{i=1,\ldots,2m} \{\widehat{f}_i(\mathbf{x})\} = \max\limits_{i=1,\ldots,m} \{|f_i(\mathbf{x})|\}$.

Use. The subroutine call is

```
mi0cin(calf,n,m,l,leq,c,dc,x,&dx,&eps,&maxfun,w,iw,&icontr)
```

The parameters are

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

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

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

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

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 $0 \le$ `leq` $\le \min\{$`l`, `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.
On entry: Initial approximation to $\mathbf{x}^*$. It must satisfy the constraints.
On exit: Computed solution.

`dx` `double`.
`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.

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

`w` `double array` with `iw` elements. Work space.
On entry: The values of `w` are not used.
On exit: The function values at the computed solution, i.e.
$\texttt{w[i-1]} = f_{\texttt{i}}(\texttt{x}), \ \texttt{i} = 1, \ldots, \texttt{m}$.

`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,
`icontr` $= 1$: Start minimization.

On exit: Information about performance,
`icontr` $= 0$: Iteration succesful. Regular solution.
`icontr` $= 1$: Iteration succesful. Singular solution.
`icontr` $= 2$: Iteration stopped, because `eps` is too small. The best solution approximation is returned in `x`.
`icontr` $= 3$: Iteration stopped, because the maximum number of calls of `calf` 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` $= -4$: `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` $< 2nm+5n^2+5m+10n+4l$
`icontr` $= -14$: $\texttt{icontr}_{\text{entry}} \le 0$

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 (1.6), page 9. 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 MI0CIN     23.11.2004 */

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

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

    /* find second half of function values */
    for (mhalf = j = 3; j < df_dim1; j++) {
f[j] = -f[j-mhalf];
    }
} /* calf */

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 mi0cin(
      void (*calf)(),
      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;
    double c[1];
    double dc[2], w[IW], x[2];
    double dx, eps;
    int maxfun;

/*     SET PARAMETERS */
    c[0] = 2.;
    dc[0] = -1.;
    dc[1] = 1.;
    eps = 1e-10;
    maxfun = 25;
/*     SET INITIAL GUESS */
    x1 = 1.;
    x2 = 1.;
    dx = .1;

    mi0cin(calf, 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;
    }
    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: Required accuracy not achieved\n\n");
    break;
    case 3:
    printf("Maximum number of function evaluations exceeded\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 calf: %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(1);

    return 0;
}
```

We get the results

```
RESULTS FROM OPTIMIZATION

Maximum number of function evaluations exceeded

                       Solution:   2.3682683689e+00
                                   3.6826836888e-01
Number of calls of calf: 25

Function values at the solution:   3.8899604048e-03
                                   2.0291995645e-01
                                   3.7501513179e-01
                                  -3.8899604048e-03
                                  -2.0291995645e-01
                                  -3.7501513179e-01
```

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 C Subroutines for Non-Linear Optimization.* IMM-Technical report-2004-21, 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): *Non-gradient c Subroutines for Non-Linear Optimization.* Report NI-91-04, 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 (1964): *An efficient method for finding the minimum of a function of several variables without calculating derivatives*, Computer Journal, Vol. 7, No. 2.

[20] 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.

[21] M.J.D. Powell (1985): *On the Quadratic Programming Algorithm of Goldfarb and Idnani.* MATHEMATICAL PROGRAMMING STUDY **25**, 46 – 61.

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