TECHNISCHE UNIVERSITÄT
TUHH
HAMBURG-HARBURG

Berichte des
Forschungsschwerpunktes
Informations- und
Kommunikationstechnik

Olaf Knüppel, Thomas Simenec

PROFIL/BIAS Extensions

November 1993 Bericht 93.5

Olaf Knüppel, Thomas Simenec:

November 1993

PROFIL/BIAS Extensions

Bericht 93.5

IMPORTANT NOTE:

Because the program is licensed free of charge, there is no warranty for the program, to the extent permitted by applicable law. The entire risk as to the quality and performance of the program is with you. Should the program prove defective, you assume the cost of all necessary servicing, repair or correction.

In no event unless required by applicable law or agreed to in writing will any copyright holder, or any other party who may modify and/or redistribute the program as permitted above, be liable to you for damages, including any general, special, incidental or consequential damages arising out of the use or inability to use the program (including but not limited to loss of data or data being rendered inaccurate or losses sustained by you or third parties or a failure of the program to operate with any other programs), even if such holder or other party has been advised of the possibility of such damages.

Prof. Dr. S. M. Rump
Technische Universität Hamburg-Harburg
Technische Informatik III
D-21071 Hamburg
Germany

This report is available
via anonymous ftp from:
`ti3sun.ti3.tu-harburg.de`
in the file:
`/pub/reports/report93.5.ps.Z`

PROFIL/BIAS Extensions

Olaf Knüppel Thomas Simenec

Technische Informatik III
Technische Universität Hamburg-Harburg
D–21071 Hamburg

email: knueppel@tu-harburg.d400.de
simenec@tu-harburg.d400.de

November 19, 1993

Abstract

After the first official version of PROFIL has been released, several extensions including a set of test matrices, general linear lists, automatic differentiation, and more have been added as a separate package to the PROFIL distribution. Some examples and test programs are also contained in the package.

This is a brief description of these extensions and their usage. It is not intended to explain the specific implementation details of the extensions.

Additionally, some hints of how to adapt the IEEE version of BIAS [3] to other than the currently supported architectures are given.

Contents

1 Introduction **3**

2 Installation **3**

2.1 Installing the Libraries and Include Files 4

3 Some Miscellaneous Functions **4**

4 Test Matrices **5**

5 Subroutines for Local Minimization **13**

6 General Linear Singly Linked Lists **16**

6.1 Example 16

6.2 Sorted Lists 17

6.3 Supported Functions 18

7 Automatic Differentiation **19**

8 Sample Programs **21**

9 Adaptation of BIAS to non-supported Architectures **21**

Bibliography **25**

1 Introduction

This documentation describes in a compact way the usage of the PROFIL extensions which are contained in the extension package.

In some cases it is assumed that the reader is familiar with basic C programming techniques and terms as header files, modules, libraries, and so forth.

Unexperienced readers may take the sample programs and test and modify them.

In an extra section we will give some hints for the adaptation of the BIAS IEEE version to non-supported architectures. For this section some experience with C and good knowledge about the architecture (that is how to switch the rounding mode of the floating point unit) is required.

2 Installation

The installation of the PROFIL extensions is similar to the installation of PROFIL. As a prerequisite it is assumed that PROFIL and BIAS are properly installed [4]. To install the extensions, change to the PROFIL directory and copy the package file into this directory. The package file is called `profext.tar.Z` under Unix systems and `profext.tgz` under DOS or OS/2. To unpack the compressed package archive, type

```
zcat profext.tar.Z | tar xf -
```

under Unix or

```
gzip -dc profext.tgz | tar xf -
```

under DOS or OS/2. After that the compressed package (`profext.tar.Z` or `profext.tgz`) may be deleted.

Type `cd Tools` to change to the directory containing the extensions. Depending on your compiler, type the following to build all libraries:

Command	Architecture	Compiler used
`dmake xlC`	IBM RS/6000	IBM xlc
`dmake ibmgcc`	IBM RS/6000	gcc
`dmake hppaCC`	HP 9000/700	HP CC
`dmake hppagcc`	HP 9000/700	gcc
`dmake sun4CC`	Sparc	SUN CC
`dmake sun4gcc`	Sparc	gcc
`dmake pcgcc`	80x86	gcc

After the `dmake` finishes, the following libraries have been built:

Library	Description
`ProfilLIN`	linear subroutines
`ProfilNLI`	non-linear subroutines
`ProfilMISC`	miscellaneous subroutines

If `_test` is appended to the above build commands, additionally a set of test routines is also built. These test routines perform a simple check of some of the new functions. Under Unix, they are executed automatically, but under DOS or OS/2 you have to start them explicitly. If the test routines terminate silently, the test succeeded. Otherwise, some information is given.

2.1 Installing the Libraries and Include Files

Until now, the libraries and the include files with the PROFIL extensions are contained in the `Tools` directory. To install them in a different place, please refer to [4], where the appropriate steps are described for the PROFIL distribution. Note that files ending with "`.hgen`" and "`.Cgen`" ("`.hge`" and "`.cge`" under DOS) must be treated like include files.

Due to limitations in file name length on some systems, some of the file names mentioned below may be different on your system. If any differences occur, those can be identified in the file `names` where the first name in each line corresponds to the name mentioned in this documentation and the second name is the file name on your system. On such systems for example the library `ProfilMISC` is called `prmis` instead (i.e. the file name is `libprmis.a`).

3 Some Miscellaneous Functions

The file `MiscFunctions.h` contains the definitions of some functions that do not fit into any other header file but are useful enough to provide them for end users. In the following we

denote expressions of type `INT` by `k` or `p`.

Operation	Return Type	Description
`Rand01()`	`REAL`	Returns a pseudo-random number in the range $[0, 1]$.
`Random()`	`REAL`	Returns a pseudo-random number in the range $[-1, 1]$.
`Rand()`	`UNSIGNED`	Returns an integer pseudo-random number between 0 and (`PROFIL_RAND_MAX` $- 1$).
`Randomize()`	—	Initializes the pseudo-random number generator with a new time dependent random sequence.
`Binom(k,p)`	`INT`	Returns the binomial coefficient $$\binom{k}{p} := \frac{k!}{(k-p)! \cdot p!}$$
`Legendre(k,p)`	`INT`	Returns the Legendre coefficient $$\left(\frac{k}{p}\right) = \begin{cases} 0 & \text{if } p \text{ divides } k \\ 1 & \text{if } k \text{ is congruent to a square mod } p \\ -1 & \text{otherwise} \end{cases}$$ where `p` must be a prime number.

4 Test Matrices

The extension package provides a set of test matrices, which mainly will be used for testing and comparing linear algorithms. All test matrices are defined in the file `TestMatrices.h`. Many of them have been taken from [2].

To get a quick overview concerning the matrix properties, each of the test matrices has a four character type field in the first line of the header comments. The values used for each of the four characters are:

Position	Character	Description
1.	`S`	symmetric matrix
2.	`PD`	positive definite matrix
3.	`M`	M-matrix

The symbol "$*$" in any position denotes the absence of a certain property.

Assuming that `n` and `m` are of type `INT`, the return type of all test matrix generating routines is always `MATRIX`, where the dimension of the matrix is `n` $\times$ `m`, if both `n` and `m` are parameters.

Otherwise, the dimension is `n` × `n`. Furthermore assume `lb`,`ub` to be of type `REAL`. We denote the generated matrix by A and the element in the i-th row and j-th column by a_{ij}. Except symmetry, any mentioned properties of the matrices below (e.g. being positive definite or a given range for the eigenvalues) are not guaranteed due to the finite precision of the floating point arithmetic used to calculate the matrices.

The following test matrices are provided:

`Lietzke(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = n - |i - j|, \qquad i, j = 1, 2, \ldots, n.$$

Example for $n = 4$:

$$A = \begin{pmatrix} 4 & 3 & 2 & 1 \\ 3 & 4 & 3 & 2 \\ 2 & 3 & 4 & 3 \\ 1 & 2 & 3 & 4 \end{pmatrix}.$$

`Pascal(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$\begin{aligned} a_{1j} &= a_{j1} = 1, & j &= 1, 2, \ldots, n, \\ a_{ij} &= a_{i-1,j} + a_{i,j-1}, & i, j &= 2, 3, \ldots, n. \end{aligned}$$

`Pascal(n,k)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$\begin{aligned} a_{1j} &= a_{j1} = k \neq 0, & j &= 1, 2, \ldots, n, \\ a_{ij} &= a_{i-1,j} + a_{i,j-1}, & i, j &= 2, 3, \ldots, n. \end{aligned}$$

If k is the reciprocal of an integer, the elements of A^{-1} are integers.
Example for $n = 4$, $k = \frac{1}{7}$:

$$A = \begin{pmatrix} \frac{1}{7} & \frac{1}{7} & \frac{1}{7} & \frac{1}{7} \\ \frac{1}{7} & \frac{2}{7} & \frac{3}{7} & \frac{4}{7} \\ \frac{1}{7} & \frac{3}{7} & \frac{6}{7} & \frac{10}{7} \\ \frac{1}{7} & \frac{4}{7} & \frac{10}{7} & \frac{20}{7} \end{pmatrix}.$$

`Hilbert(n)` returns the finite segments of the (infinite) Hilbert Matrix.
$A = (a_{ij})$ is the $n \times n$ matrix defined by

$$a_{ij} = \frac{1}{i + j - 1}, \qquad i, j = 1, 2, \ldots, n.$$

The general structure of the matrix is:

$$A = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} & \cdots & \frac{1}{n} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n-1} \end{pmatrix} .$$

`ExactHilbert(n)` returns a scalar Hilbert Matrix.

$A = (a_{ij})$ is the $n \times n$ matrix defined by

$$a_{ij} = \frac{\mathrm{lcm}(1, 2, \ldots, n)}{i + j - 1}, \qquad i, j = 1, 2, \ldots, n.$$

Only matrices which are *exactly* representable on the computer are returned. So beyond a machine-dependent matrix dimension the program aborts with an error message.

`InverseHilbert(n)` returns the finite segments of the (infinite) inverse Hilbert Matrix.

$A = (a_{ij})$ is the $n \times n$ matrix defined by

$$a_{ij} = \frac{(-1)^{i+j}(n + i - 1)!(n + j - 1)!}{(i + j - 1)[(i - 1)!(j - 1)!]^2(n - i)!(n - j)!} , \qquad i, j = 1, 2, \ldots, n.$$

Example for $n = 4$:

$$A = \begin{pmatrix} 16 & -120 & 240 & -140 \\ -120 & 1200 & -2700 & 1680 \\ 240 & -2700 & 6480 & -4200 \\ -140 & 1680 & -4200 & 2800 \end{pmatrix} .$$

`Lotkin(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$\begin{aligned} a_{1j} &= 1, && j = 1, 2, \ldots, n, \\ a_{ij} &= (i + j - 1)^{-1}, && i = 2, 3, \ldots, n, \quad j = 1, 2, \ldots, n. \end{aligned}$$

The general structure of the matrix is:

$$A_n = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \cdots & \frac{1}{n+1} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \cdots & \frac{1}{n+2} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \frac{1}{n} & \frac{1}{n+1} & \frac{1}{n+2} & \cdots & \frac{1}{2n+1} \end{pmatrix} .$$

`Westlake(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \begin{cases} i/j, & \text{if } i \le j, \\ j/i, & \text{if } i > j. \end{cases}$$

Example for $n = 3$:

$$A = \begin{pmatrix} 1 & \frac{1}{2} & \frac{1}{3} \\ \frac{1}{2} & 1 & \frac{2}{3} \\ \frac{1}{3} & \frac{2}{3} & 1 \end{pmatrix}.$$

`Newman(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \left(\frac{2}{n+1}\right)^{\frac{1}{2}} \cdot \sin\left(\frac{ij\pi}{n+1}\right) \qquad i, j = 1, 2, \ldots, n.$$

A is orthogonal, and $A^{-1} = A$.

`Frank(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = a_{ji} = n + 1 - i, \qquad \text{if } i \ge j.$$

The general structure of the matrix is:

$$A = \begin{pmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n-1 & n-2 & \cdots & 2 & 1 \\ n-2 & n-2 & n-2 & \cdots & 2 & 1 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ 2 & 2 & 2 & \cdots & 2 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix}.$$

`BoothroydMax(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \max(i, j), \qquad i, j = 1, 2, \ldots, n.$$

The general structure of the matrix is:

$$A = \begin{pmatrix} 1 & 2 & 3 & \ldots & n-1 & n \\ 2 & 2 & 3 & \ldots & n-1 & n \\ 3 & 3 & 3 & \ldots & n-1 & n \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \\ n-1 & n-1 & n-1 & \ldots & n-1 & n \\ n & n & n & \ldots & n & n \end{pmatrix}.$$

`Hadamard(n)` returns an $n \times n$ Hadamard Matrix $A = (a_{ij})$.

$p = (n+1)$ must be prime, $n \geq 2$. For an arbitrary integer k let

$$\left(\frac{k}{p}\right) = \begin{cases} 0, & \text{if } p \text{ divides } k, \\ 1, & \text{if } k \text{ is congruent to a square mod } p, \\ -1, & \text{otherwise.} \end{cases}$$

Then $A = (a_{ij})$ is the $n \times n$ matrix defined by

$$a_{ij} = \left(\frac{i+j}{n+1}\right), \qquad i, j = 1, 2, \ldots, n.$$

Example for $n = 4$:

$$A = \begin{pmatrix} -1 & -1 & 1 & 0 \\ -1 & 1 & 0 & 1 \\ 1 & 0 & 1 & -1 \\ 0 & 1 & -1 & -1 \end{pmatrix}.$$

`Binomial(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = (-1)^{i-1} \binom{i-1}{j-1}, \qquad i, j = 1, 2, \ldots, n.$$

Example for $n \geq 5$:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots \\ 1 & -1 & 0 & 0 & 0 & \cdots \\ 1 & -2 & 1 & 0 & 0 & \cdots \\ 1 & -4 & 6 & -4 & 1 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots \end{pmatrix}.$$

`Aergerter(n)` returns the $n \times n$ matrix $A_n = (a_{ij})$ defined by

$$A_n = \begin{pmatrix} & & & & 1 \\ & & & & 2 \\ & I_{n-1} & & & \vdots \\ & & & & n-1 \\ 1 & 2 & \cdots & n-1 & n \end{pmatrix}.$$

I_{n-1} denotes the $(n-1) \times (n-1)$ identity matrix.

`Todd(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = |i - j|, \qquad i, j = 1, 2, \ldots, n.$$

`Milnes(n)` returns the $n \times n$ matrix $A = A(a_1, a_2, \ldots, a_{n-1}) = (a_{ij})$ defined by

$$a_{ij} = \begin{cases} 1, & j \geq i, \\ a_j, & j < i. \end{cases}$$

Example for $n = 4$:

$$A = \begin{pmatrix} 1 & 1 & 1 & 1 \\ a_1 & 1 & 1 & 1 \\ a_1 & a_2 & 1 & 1 \\ a_1 & a_2 & a_3 & 1 \end{pmatrix}.$$

The constants $a_1, a_2, \ldots, a_{n-1}$ are random numbers.
A is singular iff $a_i = 1$ for some i.

`Milnes(n,U)` returns the matrix `Milnes(n)` with user-specified values a_1, a_2, ..., a_{n-1} in vector U.

`Combinatorial(n)` returns the $n \times n$ matrix $A = (a_{ij})$:

$$A = \begin{pmatrix} x+y & y & y & \cdots & y \\ y & x+y & y & \cdots & y \\ y & y & x+y & \cdots & y \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ y & y & y & \cdots & x+y \end{pmatrix}.$$

The constants x and y are random numbers between 0 and 1.

`Combinatorial(n,x,y)` returns the matrix `Combinatorial(n)` with user-specified values x and y.

`Cauchy(n)` returns the $n \times n$ matrix $A = (a_{ij})$:

$$A = \begin{pmatrix} (x_1+y_1)^{-1} & (x_1+y_2)^{-1} & \cdots & (x_1+y_n)^{-1} \\ (x_2+y_1)^{-1} & (x_2+y_2)^{-1} & \cdots & (x_2+y_n)^{-1} \\ \cdots & \cdots & \cdots & \cdots \\ (x_n+y_1)^{-1} & (x_n+y_2)^{-1} & \cdots & (x_n+y_n)^{-1} \end{pmatrix}.$$

It holds $0 < x_i \leq 1$, $0 < y_i \leq 1$ for $i = 1, 2, \ldots, n$.
The constants x_i and y_i are random numbers.

`Cauchy(n,x,y)` returns the matrix `Cauchy(n)` with user-specified values x_1, x_2, ..., x_n in vector x, and y_1, y_2, ..., y_n in vector y.

`Vandermonde(n,x)` returns the $n \times n$ Vandermonde matrix $A = (a_{ij})$:

$$A(x_1, x_2, \ldots, x_n) = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ x_1 & x_2 & \cdots & x_n \\ x_1^2 & x_2^2 & \cdots & x_n^2 \\ \cdots & \cdots & \cdots & \cdots \\ x_1^{n-1} & x_2^{n-1} & \cdots & x_n^{n-1} \end{pmatrix}.$$

The x_i are distinct and non-zero chosen by the user in a vector x.

`Boothroyd(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = \binom{n+i-1}{i-1} \cdot \binom{n-1}{n-j} \cdot n \cdot \frac{1}{i+j-1}, \qquad i,j = 1,2,\ldots,n.$$

`InverseBoothroyd(n)` returns the $n \times n$ matrix $A = (a_{ij})$ defined by

$$a_{ij} = (-1)^{i+j} \cdot \binom{n+i-1}{i-1} \cdot \binom{n-1}{n-j} \cdot n \cdot \frac{1}{i+j-1}, \qquad i,j = 1,2,\ldots,n.$$

`RandomM_Matrix(n)` returns an $n \times n$ M-matrix with random coefficients.

A matrix A is an *M-matrix* iff $A \in \mathbb{R}^{n \times n}$, $a_{ij} \leq 0$ for all $i \neq j$, and $Au > 0$ for some positive vector $u \in \mathbb{R}^n$.

`RandomPD(n)` returns a positive definite $n \times n$ matrix $A = (a_{ij})$ constructed by similarity transformation:

$$A = (I - uv^T) \cdot D \cdot (I - uv^T)^{-1}, \qquad \sum_{i=1}^{n} u_i v_i = 0.$$

The vectors u, v and the diagonal matrix D are choosen at random.

`RandomPD(n,lb,ub)` returns for `ub` $\geq$ `lb` > 0 a positive definite $n \times n$ matrix $A = (a_{ij})$ constructed by similarity transformation:

$$A = (I - uv^T) \cdot D \cdot (I - uv^T)^{-1}, \qquad \sum_{i=1}^{n} u_i v_i = 0.$$

The vectors u and v are choosen at random. The eigenvalues of A are contained in the interval [`lb`, `ub`] where the smallest eigenvalue of A is equal to `lb` and the largest eigenvalue is equal to `ub`.

`RandomSPD(n)` returns a symmetric positive definite $n \times n$ matrix $A = (a_{ij})$ constructed by Householder similarity transformation:

$$A = (I - 2vv^T) \cdot D \cdot (I - 2vv^T)^{-1}, \qquad \sum_{i=1}^{n} v_i^2 = 1.$$

The vector v and the diagonal matrix D are choosen at random.

`RandomSPD(n,lb,ub)` returns for `ub` $\geq$ `lb` > 0 a symmetric positive definite $n \times n$ matrix $A = (a_{ij})$ constructed by Householder similarity transformation:

$$A = (I - 2vv^T) \cdot D \cdot (I - 2vv^T)^{-1}, \qquad \sum_{i=1}^{n} v_i^2 = 1.$$

The vector v is choosen at random. The eigenvalues of A are contained in the interval [`lb`, `ub`] where the smallest eigenvalue of A is equal to `lb` and the largest eigenvalue is equal to `ub`.

`RandomToeplitz(n)` returns an $n \times n$ Toeplitz matrix $A = (a_{ij})$ defined by

$$a_{ij} = r_{j-i}, \qquad i, j = 1, 2, \ldots, n.$$

$r_{-n+1}, \ldots, r_0, \ldots, r_{n-1}$ are random scalars between -1 and 1. The general structure of the matrix is ($n = 4$):

$$A = \begin{pmatrix} r_0 & r_1 & r_2 & r_3 \\ r_{-1} & r_0 & r_1 & r_2 \\ r_{-2} & r_{-1} & r_0 & r_1 \\ r_{-3} & r_{-2} & r_{-1} & r_0 \end{pmatrix}.$$

`RandomMatrix(n)` returns an $n \times n$ matrix with random coefficients in the range $[-1, 1]$.

`RandomMatrix(n, m)` returns a rectangular $n \times m$ matrix with random coefficients in the range $[-1, 1]$.

`RandomSymmetric(n)` returns a symmetric $n \times n$ matrix with random coefficients in the range $[-1, 1]$.

`RandomPersymmetric(n)` returns a persymmetric $n \times n$ matrix with random coefficients in the range $[0, 1]$.

5 Subroutines for Local Minimization

Two methods for finding a local minimum of $f : \mathbb{R}^n \to \mathbb{R}$ without using derivatives are contained in the package. The first method is the downhill simplex method due to Nelder and Mead [5]. It is a robust but rather slow method in terms of required function evaluations. The implementation has been taken from [6] with some slight modifications. The file `NelderMead.h` contains the definitions needed to use this method.

The second method is due to Brent [1] and defined in `Brent.h`. Both methods are available as easy-to-use and as expert version. The latter having more parameters allowing more freedom in customization.

```
REAL NelderMead (VECTOR & x, VECTOR & diam, REAL tol, REAL abstol,
                 INT & ok, INT niter, REAL (*pfunc)(VECTOR &));
```

Return value: An approximation of the minimum value found.

Parameters:

`x` On input, contains the start point. On output, contains the approximation of the minimum point.

`diam` On input, contains the approx. diameter of the box with center `x`, which contains a minimum point. This parameter is used to construct the start simplex. Unchanged on output.

`tol` On input, relative termination tolerance. Unchanged on output.

`abstol` On input, absolute termination tolerance. Unchanged on output.

`ok` On output, success flag. A value of 1 indicates that the algorithm terminates normally. Otherwise, a value of 0 is used.

`niter` On input, denotes the maximal number of iterations to be performed. Unchanged on output.

`pfunc` On input, pointer to the function to be minimized. Unchanged on output.

```
VOID NelderMead_Expert (MATRIX & p, VECTOR & y, REAL tol, REAL abstol,
                        INT & ok, INT niter, PREAL parms,
                        REAL (*pfunc)(VECTOR &));
```

Return value: none.

Parameters:

`p` On input, the rows of the matrix contain the start simplex. The number of rows must be $(1 + n)$ where n is the dimension of the problem. On output, contains the last calculated simplex.

`y` On input/output, contains the function values of each simplex point given by `p`.

`tol` On input, relative termination tolerance. Unchanged on output.

`abstol` On input, absolute termination tolerance. Unchanged on output.

`ok` On output, success flag. A value of 1 indicates that the algorithm terminates normally. Otherwise, a value of 0 is used.

`niter` On input, denotes the maximal number of iterations to be performed. Unchanged on output.

`parms` On input, points to an array of dimension 3 containing the parameters of the minimization method:

`parms[0]` expansion factor for new trial point
`parms[1]` shrink factor
`parms[2]` expansion factor

Unchanged on output.

`pfunc` On input, pointer to the function to be minimized. Unchanged on output.

```
REAL BrentMinimize (VECTOR & p, VECTOR & diam, REAL rtol,
                    REAL atol, REAL (*pfunc)(VECTOR &));
```

Return value: An approximation of the minimum value found.

Parameters:

`p` On input, contains the start point. On output, contains the approximation of the minimum point.

`diam` On input, contains the approx. diameter of the box with center `x`, which contains a minimum point. This parameter is used to set up some initial parameters. Unchanged on output.

`rtol` On input, relative termination tolerance. Unchanged on output.

`atol` On input, absolute termination tolerance. Unchanged on output.

`pfunc` On input, pointer to the function to be minimized. Unchanged on output.

```
REAL BrentMinimize_Expert (VECTOR & p, REAL h, REAL rtol, REAL atol,
                           REAL scbd, INT illc, INT ktm,
                           REAL (*pfunc)(VECTOR &));
```

Return value: An approximation of the minimum value found.

Parameters:

`p` On input, contains the start point. On output, contains the approximation of the minimum point.

`h` On input, contains an estimation of the distance from `p` to the minimum point. Unchanged on output.

`rtol` On input, relative termination tolerance. Unchanged on output.

`atol` On input, absolute termination tolerance. Unchanged on output.

`scbd` On input, contains the scale factor. Unchanged on output.

`illc` On input, contains the ill-conditioned flag. The value 1 denotes an ill-conditioned problem. Otherwise the value must be 0. Unchanged on output.

`ktm` On input, the number of iterations minus one which are allowed to be taken until an improvement must take place. In most cases 1 should do. Unchanged on output.

`pfunc` On input, pointer to the function to be minimized. Unchanged on output.

6 General Linear Singly Linked Lists

The extension package of PROFIL supports the creation and usage of linear singly linked lists of arbitrary data types. The general list definitions and implementations are contained in the two generic files `LinearList.hgen` and `LinearList.Cgen`.

The creation of the new list data types is done by defining some macros with the names of the data types to be used and including the generic files mentioned above.

The following macros control the generation of the linear lists:

Macro	Description
`LIST_ELEMENT`	defines the type name of the list elements (e.g. a C++ class name)
`LIST`	defines the type name of the list to be generated
`LISTOBJECT`	defines the type name of one list node (only used internally)

After defining these macros, the file `LinearList.hgen` can be included to generate all definitions of the new list data type. By including the file `LinearList.Cgen` (the file `LinearList.hgen` must have been included before), all necessary routines for the list implementation are generated.

6.1 Example

As an example, we consider a list of real vectors (data type `VECTOR`). The new list data type should be named `VECTORLIST`. With `VECTORLISTOBJECT` we want to denote the list node data type.

To create the definitions (headers) and implementations (subroutines) of the new list type, use the following code fragment:

```
#define LIST_ELEMENT    VECTOR
#define LIST            VECTORLIST
#define LISTOBJECT      VECTORLISTOBJECT

#include "LinearList.hgen"

#include "LinearList.Cgen"

#undef LISTOBJECT
#undef LIST
#undef LIST_ELEMENT
```

The `#undef` commands may be necessary, if more than one list type is defined. After the generation step, you can define a new list `li` of vectors by

```
VECTORLIST li;
```

6.2 Sorted Lists

The general linear list package also supports sorted lists where the sorting criterion is given by a comparison function provided by the user. The sorting is performed when inserting a new list element, i.e. the new element is inserted into an already sorted list.

The comparison function used for the sorted insert must have the following prototype

```
INT SortCompare (LIST_ELEMENT & e1, LIST_ELEMENT & e2);
```

where the name of the function can be chosen arbitrarily. The return value of the comparison function must be 1, if the element given by `e1` should precede the element given by `e2` in the list. Otherwise, a value of 0 must be returned. To create the list considered in the above example as sorted list with the comparison function called `SortComp`, the list is defined as:

```
VECTORLIST li(SortComp);
```

The elements are inserted into this list by using the `<<=` operator (see below).

6.3 Supported Functions

The following functions are supported, where it is assumed, that the data type of `li` is the same as defined with `#define LIST` and the data type of `e` is the same as used with `#define LIST_ELEMENT`.

Operation	Return Type	Description
`First(li)`	`LIST_ELEMENT`	Returns the first element of the list and sets the first list element as current.
`Next(li)`	`LIST_ELEMENT`	Returns the next element of the list and sets this element as current list element.
`Last(li)`	`LIST_ELEMENT`	Returns the last element of the list.
`Current(li)`	`LIST_ELEMENT`	Returns the current list element.
`RemoveCurrent(Li)`	—	Removes the current list element from the list and sets the new current element to the following element.
`Finished(li)`	`INT`	Returns 1, if the current element is undefined, i.e. if the end of the list is reached. Otherwise, 0 is returned.
`IsEmpty(li)`	`INT`	Returns 1, if the list is empty. Otherwise, 0 is returned.
`Length(li)`	`INT`	Returns the current length of the list, i.e. the number of list elements.
`MaxLength(li)`	`INT`	Returns the maximal length of the list since the last call of `ResetLength` or the definition of the list variable.
`ResetLength(li)`	—	Resets the maximal list length to the current length.
`li += e`	—	Appends the element `e` at the end of the list `li`.
`li *= e`	—	Prepends the element `e` at the beginning of the list `li`.
`li <<= e`	—	Inserts the element `e` into the sorted list `li`.
`li--`	—	Removes the first element of the list.

A complete list can be output by using the `<<` operator, e.g.

```
cout << li;
```

7 Automatic Differentiation

For functions of the type

$$f : \mathbb{R}^n \to \mathbb{R} \qquad \text{or} \qquad f : \mathbb{IR}^n \to \mathbb{IR}$$

where $\mathbb{IR}$ denotes the set of intervals, an automatic computation of the gradient and optionally the Hessian value can be performed during the evaluation of the functional expression in parallel to the computation of the function value.

If only the first derivative (the gradient) has to be computed, the necessary header files are (depending whether you have a real or interval valued function):

```
Gradient.h
IntervalGradient.h
```

Is the Hessian also needed, the header files are:

```
AutoDiff.h
IntervalAutoDiff.h
```

With automatic differentiation, new data types which contain the current function value as well as the current derivative values, do exist. Their names depend on the header file used:

Type Name	Header File
`GRADIENT`	`Gradient.h`
`INTERVAL_GRADIENT`	`IntervalGradient.h`
`AUTODIFF`	`AutoDiff.h`
`INTERVAL_AUTODIFF`	`IntervalAutoDiff.h`

In the following, we will use always `AUTODIFF` as new type name.

To use automatic differentiation, you first have to define your independent variable by creating a new variable of the type `AUTODIFF` containing the values of your independent variable. After that, you write your functional expression as you would do in the real case, using variables of the type `AUTODIFF` instead of `REAL` or `INTERVAL` for your intermediate results. At least, you can assign your result (either the function value, the gradient value, or the Hessian value).

If `a` if a variable of type `AUTODIFF` and contains the tupel with the function, gradient, and optionally Hessian value, you can access the components as follows:

Access		Component
`a.fkt()`	`FunctionValue(a)`	Value of the function
`a.grd()`	`GradientValue(a)`	Value of the gradient
`a.hessian()`	`HessianValue(a)`	Value of the Hessian

Using the types `AUTODIFF` or `INTERVAL_AUTODIFF`, the evaluation of the Hessian is performed if the variable `AUTODIFF::ComputeHessian` (`INTERVAL_AUTODIFF::ComputeHessian`) has a non-zero value. Otherwise, the Hessian is not computed.

For simplicity, the following conversions are automatically performed when assigning a variable `a` of an automatic differentiation type to another data type `b`:

Type of `a`	Type of `b`	Conversion
`GRADIENT`	`REAL`	`a.fkt()`
`GRADIENT`	`VECTOR`	`a.grd()`
`INTERVAL_GRADIENT`	`INTERVAL`	`a.fkt()`
`INTERVAL_GRADIENT`	`INTERVAL_VECTOR`	`a.grd()`
`AUTODIFF`	`REAL`	`a.fkt()`
`AUTODIFF`	`VECTOR`	`a.grd()`
`AUTODIFF`	`MATRIX`	`a.hessian()`
`INTERVAL_AUTODIFF`	`INTERVAL`	`a.fkt()`
`INTERVAL_AUTODIFF`	`INTERVAL_VECTOR`	`a.grd()`
`INTERVAL_AUTODIFF`	`INTERVAL_MATRIX`	`a.hessian()`

For example, the gradient of the function $f : x \to 2x_1x_2$, $x \in \mathbb{R}^2$ at $x = (1, 2)$ can be computed with:

```
VECTOR x(2), y(2);

x(1) = 1.0; x(2) = 2.0;

GRADIENT xg(x); // x resp. xg is the independent variable

y = 2.0 * xg(1) * xg(2);
```

Important Note: This automatic conversion will fail if the expression does not depend on the independent variable. In this case, the compiler will generate an error message about incompatible types. Using an intermediate result of an automatic differentiation type or the explicit assignment of zero to the derivatives will be a work-around. This is not a bug of the automatic differentiation package but due to the limited capabilities of C++.

The file `AutoDiffExample.C` contains a simple example of how to use the automatic differentiation package. If you want to use other than the predefined set of functions with automatic differentiation, you have to modify the files `AutoDiff.hgen` and `AutoDiff.Cgen`. These files contain the generic parts for all automatic differentiation files.

More information about automatic differentiation can be found in e.g. [7].

8 Sample Programs

The subdirectory `Examples` contains sample programs for some of the new features discussed in this documentation. The examples are kept quite simple to make the main ideas of the new features clear.

All sample programs can simply be build by appending `_example` to the `dmake` command discussed in the installation section. For example,

```
dmake xlC_examples
```

builds all libraries (if not already done) and all sample programs.

9 Adaptation of BIAS to non-supported Architectures

In this section, we will give some information of how to adapt the distributed BIAS version to other non-supported architectures which support the IEEE-standard with directed roundings. For other architectures, a different version of BIAS is available on request.

The adaptation is performed in 4 steps:

- The file `Portab.h` in the `BIAS` subdirectory must be modified to define some general information about your system.
- Subroutines for switching the rounding mode must be written or taken from the system.
- If the routines for switching the rounding mode are given in an extra file, the `Makefile` has to be modified to include this file into the BIAS library.
- If the rounding mode routines are taken from the system or you can use inline assembler statements, the files `BiasInt.h` and `Bias0.c` have to be modified.

For the correct automatic compilation of BIAS, your system architecture must be identified at compilation time. For this purpose, most compilers support pre-defined macros which can be used for that purpose. For example, the `xlc` on RS/6000 architectures running under AIX defines the macro `_AIX`.

To identify your architecture, you have to modify the file `Portab.h`. The necessary insertions should be behind the comment starting with

```
Insert settings for new compilers / machines here!
```

The most important setting is whether you have a big or little endian architecture on your machine. Big endian architectures are e.g. Sparc, RS/6000. Little endian architectures are e.g. PCs with 80x86. As a help, the following C program may be used to identify your machine:

```
#include <stdio.h>

union {
  struct { char a; char b; } bytes;
  short word;
} un;

main()
{
  un.bytes.a = 1; un.bytes.b = 2;
  if (un.word == 258)
    printf ("big endian machine\n");
  else if (un.word == 513)
    printf ("little endian machine\n");
  else
    printf ("unknown machine\n");
}
```

Define the macro `__HIGHBYTEFIRST__` for big endian and `__LOWBYTEFIRST__` for little endian machines. Additionally, you have to define `__32BIT__` or `__16BIT__` if your system word length is either 32 or 16 bit. Define `__ANSI__` if your compiler supports the ANSI keyword `volatile`. If your compiler supports the library function `memmove` with overlapping memory fields and does not define the macro `__STDC__`, you should define it yourself.

As an example, consider the xlc compiler on RS/6000 architectures running AIX. For this case, the settings to be inserted into `Portab.h` should be:

```
#if defined (_AIX)
#ifndef __ANSI__
#define __ANSI__
#endif
/* __STDC__ predefined */
#define __32BIT__
#define __HIGHBYTEFIRST__
#endif
```

After that, you have to define the subroutines for switching the rounding mode. These subroutines should be called

```
BiasRoundUp()
BiasRoundDown()
BiasRoundNear()
```

for the different rounding mode settings. Then the assemblation of the rounding mode switching routines must be inserted into the `Makefile` of BIAS. For this purpose, you need the name of your compiler, the necessary command line flags and the name of the assembler file, where the rounding mode setting routines are contained. If you use C routines instead of assembler subroutines, you have to change in the `Makefile` in the line

```
$(ROUNDOBJ)     : $(ROUND).s
```

the Extension "`.s`" into "`.c`".

The BIAS `Makefile` is prepared for one new architecture/compiler to be simply added. Search for the line

```
.ELIF $(YOURCOMP)
```

and modify the following lines for your purposes, if you consider that the compiler name is defined by the macro `CC`, the command line flags by the macro `CFLAGS` and the file with the new rounding mode subroutines is defined by `ROUND`.

The compilation then can be started by

```
dmake yourcomp
```

or

```
dmake yourcomp_test
```

If you can use assembler inline sequences to switch the rounding mode, more modifications which are beyond the scope of this report are necessary. Take a look at the files `BiasInt.h` and `Bias0.c`. Look for the keyword `__I386__` together with `__GNUC__` for a machine which only uses inline assembler sequences. If you provide subroutines as well as inline sequences, look for the keyword `_AIX` or `sparc` together with `__GNUC__` as example.

If you have adapted the BIAS IEEE version to any non-supported architecture and you think that your adaptation is of general interest, you may send it with all modified and/or new files via email with the subject field `PROFIL/BIAS` to the author. Please include a declaration, that your adaptation may be distributed for non-commercial use free of charge.

References

[1] R. P. Brent. *Algorithms for Minimization without Derivatives.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

[2] R. T. Gregory, D. L. Karney. *A Collection of Matrices for Testing Computational Algorithms.* Wiley-Interscience, New York, 1969.

[3] O. Knüppel. BIAS — Basic Interval Arithmetic Subroutines. Bericht 93.3 des Forschungsschwerpunktes Informations- und Kommunikationstechnik der TU Hamburg-Harburg, TU Hamburg-Harburg, Technische Informatik III, July 1993.

[4] O. Knüppel. PROFIL — Programmer's Runtime Optimized Fast Interval Library. Bericht 93.4 des Forschungsschwerpunktes Informations- und Kommunikationstechnik der TU Hamburg-Harburg, TU Hamburg-Harburg, Technische Informatik III, July 1993.

[5] J. A. Nelder, R. Mead. A simplex method for function minimization. *Computer Journal*, 7:308–313, 1965.

[6] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[7] L. B. Rall. *Automatic Differentiation, Techniques and Applications.* Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1981.

In dieser Reihe sind bisher erschienen:

Bericht 90.1 (Juni 1990)
S. M. Rump: Approximate inverses of almost singular matrices still contain useful information

Bericht 90.2 (September 1990)
C. Jansson: A Posteriori Error Bounds for the Solution of Linear Systems with Simplicial Techniques

Bericht 90.3 (September 1990)
S. M. Rump: A class of arbitrarily ill-conditioned floating-point matrices

Bericht 90.4 (Oktober 1990)
C. Jansson: A Fast Direct Method for Computing Verified Inclusions

Bericht 90.5 (November 1990)
S. M. Rump: On the Solution of Interval Linear Systems

Bericht 91.1 (April 1991)
D. Husung: TPX Precompiler für Turbo Pascal

Bericht 91.2 (November 1991)
C. Jansson: A Global Minimization Method: The One-Dimensional Case

Bericht 91.3 (Dezember 1991)
S. M. Rump: Inclusion of the Solution for Large Linear Systems with M-Matrix

Bericht 92.1 (Januar 1992)
C. Jansson, O. Knüppel: A Global Minimization Method: The Multi-Dimensional Case

Bericht 92.2 (Februar 1992)
H.-P. Jüllig: Algorithmen mit Ergebnisverifikationen mit C++/2.0

Bericht 92.3 (Februar 1992)
S. M. Rump: Recent Results in Interval Mathematics

Bericht 92.4 (Februar 1992)
D. Husung: TPX Version 1.1 US Precompiler for Turbo Pascal 5.0

Bericht 92.5 (März 1992)
D. M. Claudio, S. M. Rump: Inclusion methods for real and complex functions in one variable

Bericht 92.6 (März 1992)
H.-P. Jüllig: BIBINS/2.0 C++-Bibliotheken für Vektoren und Matrizen über beliebigem skalaren Datentyp unter Berücksichtigung spezieller spärlicher Strukturen sowie Intervalldatentypen

Bericht 93.1 (Januar 1993)
S. M. Rump: Validated Solution of Large Linear Systems

Bericht 93.2 (Januar 1993)
D. Husung: TPX Version 1.1 US Precompiler for Turbo Pascal 5.0

Bericht 93.3 (Juli 1993)
O. Knüppel: BIAS — Basic Interval Arithmetic Subroutines

Bericht 93.4 (Juli 1993)
O. Knüppel: PROFIL — Programmer's Runtime Optimized Fast Interval Library