Berichte des
Forschungsschwerpunktes
Informations- und
Kommunikationstechnik

Olaf Knüppel

PROFIL/BIAS V 2.0

Februar 1999

Bericht 99.1

Olaf Knüppel:
PROFIL/BIAS V 2.0

Februar 1999

Bericht 99.1

IMPORTANT NOTE:

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/report99.1.ps.Z`

PROFIL/BIAS V 2.0

Olaf Knüppel

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

email: `knueppel@tu-harburg.de`

Februar 1999

Abstract

This is a compact overview of PROFIL/BIAS, a portable C++ class library for developing and implementing interval algorithms in a comfortable but nevertheless efficient way. New data types as vectors, matrices, intervals, interval vectors and matrices, integer vectors and matrices, and a number of operations between them are introduced. Additionally, lots of commonly used packages (local and global optimization methods, a set of test matrices, automatic differentiation, etc.) are also included.

PROFIL V 2.0 is a completely revised version of PROFIL/BIAS [13, 15]. All packages (as e.g. [16, 14]) are no longer separately available but included in this distribution.

All interval operations of PROFIL are based on BIAS with the advantage that PROFIL is independent from the internal representation and the implementation of the interval types.

PROFIL has currently been tested on IBM RS/6000, Sparc, and 80x86 architectures. Other architectures, as DEC Alpha, HP-PA are also supported.

For readers coming from other programming languages and not being familiar with C, we will give a short introduction to the syntax of the language. We will restrict ourselves to the most simple and in numerical programs commonly needed constructs.

Acknowledgements

The development of PROFIL/BIAS V 2.0 was only possible with the support of many people around the world. Among them, the following ones need to be listed explicitly:

- Testing and helpful hints:
  - The staff from TI III
  - Henning Behnke
- DEC Alpha port:
  - Prof. Arnold Neumaier for providing information about the DEC Alpha instruction set
  - Dr. Minamoto for providing access to his DEC Alpha Station. It was sometimes not easy to work interactively on a connection which was half around the world (between Japan and Germany); however, in this case the different time zone was to our advantage.
  - Qiliang Zhu for providing the necessary SGI rounding mode switches.

We apologize for not mentioning everybody.

Contents

List of Tables

Chapter 1

Installation

1.1 Prerequisites

To compile and install PROFIL/BIAS, the following programs are required:

- GNU make (a standard make won't do it, unless all Makefiles are modified)
- a C and C++ compiler (sometimes an assembler is also needed). The GNU C/C++ compiler "gcc" is supported for all architectures.

1.2 Supported Hardware

The following hardware architectures are supported:

Architecture	Operating System	Compiler
DEC Alpha	Linux	gcc
HP PA	HP-UX[1]	gcc
IBM RS/6000	AIX 4.1	gcc
		xlc/xlC
MIPS	IRIX[1]	gcc
Sparc	Solaris 2.x	gcc
	SunOS 4.1.x	gcc
80x86	Linux	gcc
	Windows NT[2]	gcc
	Windows 95[2]	gcc

For some architectures, two different sets of rounding mode switches are available:

- A version which does not affect other floating point flags as floating point exceptions. This version is named "`compat`" an is generally slower than the second version.
- The second version uses the fastest way of switching the rounding mode. Other floating point flags may be set to a fixed value. Therefore, this version should not be used, if e.g. floating point exceptions are required.

1.3 Compiling and Installation

PROFIL/BIAS is shipped as compressed tape archive file (`Profil-2.0.tar.Z`). To install, change to the directory in which you want to install and copy the `Profil-2.0.tar.Z` file into this directory.

Unpack the compressed archive by

```
zcat Profil-2.0.tar.Z | tar xf -
```

After that, you can delete the `Profil-2.0.tar.Z` file to save space.

Change to the main PROFIL/BIAS directory by typing

```
cd Profil-2.0
```

Type

```
./Configure
```

to configure for your hardware configuration. A list of all supported architectures is presented. The names on the list are built by the following description strings separated by dashes:

- hardware
- operating system
- rounding control version ("`compat`" or "`fast`")
- required compiler

[1]This configurations is contained in the distribution but has not been tested yet.

[2]These configurations are contained in the distribution but are not very well tested yet. They require a `sh` compatible shell (`bash` for example) and a GNU make.

Choose the appropriate item by entering the corresponding number.

After that, type

```
gmake install
```

(or whatever you use to invoke GNU make with the target "`install`"). The compilation process should progress smoothly. After it finishes successfully, the header files and the libraries are copied into the subdirectories `include` and `lib` resp.

Now PROFIL/BIAS is ready to be used.

All test programs could be built by typing

```
gmake examples
```

The executable test programs all have the suffix "`exe`". Some of them only emit a message in the case of an error.

If you are using the GNU C/C++ compiler and the linker complains about missing iostream functions (as "`endl`") or could not find `libstdc++`, the linker libraries have to be changed. Probably, changing the line

```
SYSLIBS          = -lstdc++ -lm
```

into

```
SYSLIBS          = -lg++ -lm
```

in the file `Host.cfg` will do the job. This file is contained in the subdirectory belonging to your configuration, e.g. `config/rs6000-AIX4.1-compat-gcc`

1.4 Directory Structure

The PROFIL/BIAS directory is structured the following way:

Profil-2.0 main directory:

- **config** architecture dependent files (rounding mode switches, compiler/assembler/linker settings and options, etc.)
- **src** all source files:

- **BIAS** BIAS files
- **Base** PROFIL base files
- **Packages** extension packages:
  - **AutoDiff** automatic differentiation
  - **GlobalOpt** global optimization
  - **Lists** linear lists
  - **LocalOpt** local optimization
  - **Misc** some small support subroutines
  - **NLF** non linear functions
  - **TestMatrices** a set of testmatrices

include All include files after successful installation

lib All library files after successful installation

1.5 User Options

Some parts of PROFIL can be reconfigured by changing the appropriate definitions in the file `Configuration.h` and rebuilding the library. Currently, the following settings are possible (see also the description in the `Configuration.h` file):

- An optional index check can be performed for all vector and matrix types. If used, a better check for run-away indices is achieved, but the execution time grows. This setting should be used in the development phase of an algorithm and switched off in production version. The index check is performed, when a line

  ```
  #define __INDEXCHECK__
  ```

  is contained in the file `Configuration.h`. Otherwise the line has to be deleted or commented out.

- The memory management for vector and matrix types can be improved by avoiding unnecessary copies of vector or matrix elements. Unfortunately, due to some lacks in the C++ programming language, this may lead to some strange results in the following cases:
  - A vector or matrix expression is used as a parameter for a reference type.
  - A vector or matrix expression or a vector or matrix variable is used as a value parameter.
  - A vector or matrix is initialized without using the assignment operator. For example:

    ```
    VECTOR x = a + b;
    ```

  If at least one of the above cases can't be avoided, either the improved memory management should not be used or an explicit

```
MakePermanent(x);
```

where `x` is the name of the appropriate parameter or initialized vector or matrix variable must be used prior any access to the variable. In the developement phase, the improved memory management should be switched off. If switching the new memory management on, care should be taken, if the above assumptions are met. The improved memory management is performed, when a line

```
#define __DONTCOPY__
```

is contained in the file `Configuration.h`. Otherwise the line has to be deleted or commented out.

1.6 Adding New Packages

To add a new package, change to the main PROFIL/BIAS directory, unpack the new package and use

```
gmake install
```

to install the new package.

If example programs are available with the new package, they could be built with

```
make examples
```

1.7 Building Your Own Programs

To build your own program using the PROFIL/BIAS libraries and include files, add to the compiler's include path the PROFIL/BIAS include directory. The same should be done with the library path.

Additionally, all used libraries have to be mentioned. The available libraries are:

Library	Required
`lr`	mandatory
`Bias`	mandatory
`Profil`	mandatory
`ProfilPackages`	if required

Assume that you want to build your program `MyTest.C` using PROFIL/BIAS. Let us further assume that the PROFIL/BIAS include files and libraries reside in the following directories:

`/usr/local/Profil-2.0/include`	(include files)
`/usr/local/Profil-2.0/lib`	(library files)

Then the compiler call might be something like

```
CC -I/usr/local/Profil-2.0/include \
   -L/usr/local/Profil-2.0/lib \
   MyTest.C -lProfilPackages -lProfil -lBias -llr -lm
```

if "`CC`" is your C++ compiler.

1.8 Package Dependencies

Some PROFIL parts use other packages/parts. The dependencies are as follows (only direct dependencies are shown):

Package	depends on
`lr`	—
`Bias`	—
`Profil`	`Bias`, `lr`
`AutoDiff`	`Profil`
`GlobalOpt`	`LocalOpt`, `NLF`, `Profil`
`Lists`	—
`LocalOpt`	`Misc`
`Misc`	`Profil`
`NLF`	`AutoDiff`, `Profil`
`TestMatrices`	`Misc`, `Profil`

Chapter 2

BIAS

2.1 Introduction

This is a short description of the idea and the current implementation of BIAS (Basic Interval Arithmetic Subroutines). It is assumed that the reader is familiar with the basics of interval mathematics.

The developement of BIAS was guided by the ideas of BLAS, i.e. to provide an interface for basic vector and matrix operations with specific and fast implementations on various machines, the latter frequently provided by the manufacturers. The idea of BIAS is to give such an interface for interval operations with the objective:

- portability
- independency of a specific interval representation
- very efficient use of the unterlying hardware

First, we describe the data types and variables supported by BIAS, then after giving some notations we describe all routines from BIAS. The routines are described in the groups: scalar operations, vector operations, and matrix operations. At last, scalar interval standard functions are described. We give the header and a short description of each routine.

Currenly, three implementations of BIAS exist, all of them use the same interval representation as it will be described below. The three implementations are in particular:

1. Using the directed rounding. Because the switching of the rounding mode is machine dependent and not available on all architectures, this implementation is the most hardware dependent one.

Currently, it is available for IBM RS/6000 architectures, Sparc architectures, 80x86 (80386 and higher) processors, DEC Alpha processors, and HP-PA architectures. In all cases, assembler routines for switching the rounding mode are provided and it has been taken care to avoid unnecessary switching of the rounding mode.

2. Using an estimation of the error which occurs in performing a floating point operation. The error estimation is used to change the floating point calculated bounds in such a way that the isotonicity of the interval operations is always fulfilled.

   This implementation uses only standard floating point operations and because of this it can be used for a variety of machines.

   The only machine dependency lies in the maximal possible relative error of the floating point operations. In the current implementation it is assumed, that the following for the true result $\hat{x}$ and the computed result $\tilde{x}$ holds:

$$(1-\epsilon)\tilde{x} - \eta \le \hat{x} \le (1+\epsilon)\tilde{x} + \eta \text{ for } \tilde{x} \ge 0$$
$$(1+\epsilon)\tilde{x} - \eta \le \hat{x} \le (1-\epsilon)\tilde{x} + \eta \text{ for } \tilde{x} < 0$$

   Where ϵ is the machine precision and η the smallest positive machine number (see variables below).

3. Using the computed floating point result as bounds without any modification. This implementation is very fast, but doesn't always give correct bounds. The main application is in cases where a least significant bit accuracy is not needed, e.g. in global optimization, where only large intervals are used.

Only the first version (i.e. using directed rounding) ist contained in the distribution. The other two versions are just in an experimental stage.

2.2 Data Types

The following data types are defined and used by BIAS:

Data Type	Description
`REAL`	The real type (either `double` or `float`).
`INT`	The integer type
`BIASINTERVAL`	The interval type

In all current implementations of BIAS, the following definitions are given for the data types above:

```
typedef INT    int;
typedef REAL   double;
typedef struct {
  REAL inf;
  REAL sup;
} BIASINTERVAL;
```

Vectors are stored blockwise, beginning with the first component. A vector is described by two values: A pointer to the elements (`REAL *` for real and `BIASINTERVAL *` for interval vectors) and the number of elements (`INT`).

Matrices are stored rowwise in a block (that is the normal C storage, or the transposed FORTRAN storage). A matrix is described by three values: A pointer to the elements (`REAL *` for real and `BIASINTERVAL *` for interval matrices), the number of rows (`INT`), and the number of columns (`INT`).

2.2.1 Notation

In program text (e.g. in routine headers), we denote interval parameters by capital and real parameters by small letters. If preceeded by a small `p`, the parameters are pointers, which point to the objects without the leading `p`.

For simplification, the "`const`" attributes are missing in the specifications below. In case of interest, the header and implementation files should be considered.

If not stated explicitly otherwise, all parameters are scalar types.

The naming conventions for the routines in BIAS are as follows: All routine names start with `Bias`. After that, the purpose of the routine follows (e.g. `Mul` for multiplication). Last, the main parameter description is appended, where the letter `R` denotes a real and the letter `I` an interval parameter. Vectors and matrix parameters are given by using the letter `V` or `M` as a prefix to the data type. For example: The addition of a real and an interval vector is coded as `BiasAddVRVI`.

2.2.2 Implementation Dependencies

Depending on the implementation, some limitations may occur due to speed reasons. For example in the current implementation the operands and the result of an interval subtraction must not be the same.

In general, the following assumption holds:

> The result of any BIAS operation, i.e. any value that is changed by a routine from BIAS should *not* be the same as any parameter of that routine.

Another implementation dependency is the treatement of exceptions (e.g. division by zero). In the current implementation, one method out of two can be chosen at compile time:

1. A division by zero is always reported and the program is terminated.
2. If possible, the result is silently set e.g. to $[-\infty, +\infty]$. Otherwise the error is reported and the program is terminated.

Additionally, when using directed roundings, the rounding mode direction is undefined after execution of any routine from BIAS. In general, the rounding mode is set to upward rounding. If necessary (e.g. due to sensitive floating point expressions), BIAS can be configured to set the rounding mode to nearest after execution.

The controlling parameters for the configuration of the behaviour of BIAS are contained and explained in the file `BiasInt.h`.

In the routine description, expressions like $\mathtt{R} = \mathtt{A} + \mathtt{B}$ where $\mathtt{A}$,$\mathtt{B}$,$\mathtt{R}$ are intervals should be understand as

$$\mathtt{R} \supseteq \{\, a + b \mid a \in \mathtt{A},\ b \in \mathtt{B}\,\}$$

except for implementations where the computed bounds are not guaranteed to be correct. In this case, the above expression should be read as:

$$\mathtt{R} \approx \{\, a + b \mid a \in \mathtt{A},\ b \in \mathtt{B}\,\}$$

2.3 Initialization

All necessary initializations for BIAS are performed by a call to `BiasInit()`.

2.4 Variables

In this section we describe the variables with machine dependent constants which are supported by BIAS. Depending on the implementation, some of these variables may not be available. At least, `BiasEpsilon` and `BiasEta` should always contain reliable values.

Variable	Value
`BiasEpsilon`	The machine precision $\min\limits_{x \in \mathbb{F}}\{\, x \geq 0 \mid 1 + x > 1 \,\}$
`BiasEta`	The smallest positive floating point number $\min\limits_{x \in \mathbb{F}}\{x > 0\}$
`BiasNaN`	NaN (not a number)
`BiasPosInf`	$+\infty$
`BiasNegInf`	$-\infty$

We denote by $\mathbb{F}$ the set of floating point numbers.

2.5 Scalar Operations — `Bias0`

Headerfile: `Bias0.h`

`BiasInit()`

Initializes all internal values, must be called prior any other BIAS call.

`BiasRoundUp()`

Switches the rounding mode to upwards. Generates an error message, if no rounding mode switching is supported.

`BiasRoundDown()`

Switches the rounding mode to downwards. Generates an error message, if no rounding mode switching is supported.

`BiasRoundNear()`

Switches the rounding mode to nearest. Generates an error message, if no rounding mode switching is supported.

`REAL BiasPredR(REAL *pa)`

Returns the predecessor of `a`.

`REAL BiasSuccR(REAL *pa)`

Returns the successor of `a`.

`INT BiasPredI(BIASINTERVAL *pR, BIASINTERVAL *pA)`

On return, `R` is the largest interval which is contained in the interior of `A`. If the computation of `R` was impossible due to the small diameter of `A`, a value of 0 is returned and `R` is undefined. Otherwise, a value of 1 is returned.

`BiasSuccI(BIASINTERVAL *pR, BIASINTERVAL *pA)`

On return, `R` is the smallest interval, where `A` is contained in the interior.

`BiasAddRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Calculates the interval `R` which contains the true result of $\mathtt{a} + \mathtt{b}$.

`BiasAddRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \mathtt{a} + \mathtt{B}$.

`BiasAddIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Calculates $\mathtt{R} = \mathtt{A} + \mathtt{b}$.

`BiasAddII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \mathtt{A} + \mathtt{B}$.

`BiasNeg(BIASINTERVAL *pR, PINTERVAL pA)`

Calculates $\mathtt{R} = -\mathtt{A}$.

`BiasSubRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Calculates the interval R which contains the true result of $a - b$.

`BiasSubRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Calculates $R = a - B$.

`BiasSubIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Calculates $R = A - b$.

`BiasSubII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Calculates $R = A - B$.

`BiasMulRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Calculates the interval R which contains the true result of $a \cdot b$.

`BiasMulRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Calculates $R = a \cdot B$.

`BiasMulIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Calculates $R = A \cdot b$.

`BiasMulII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Calculates $R = A \cdot B$.

`BiasDivRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Calculates the interval R which contains the true result of $\frac{a}{b}$.

`BiasDivRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \frac{\mathtt{a}}{\mathtt{B}}$.

`BiasDivIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Calculates $\mathtt{R} = \frac{\mathtt{A}}{\mathtt{b}}$.

`BiasDivII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \frac{\mathtt{A}}{\mathtt{B}}$.

`BiasMacRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Adds to the interval `R` an interval which contains the true result of $\mathtt{a} \cdot \mathtt{b}$.

`BiasMacRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \mathtt{R} + \mathtt{a} \cdot \mathtt{B}$.

`BiasMacIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Calculates $\mathtt{R} = \mathtt{R} + \mathtt{A} \cdot \mathtt{b}$.

`BiasMacII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Calculates $\mathtt{R} = \mathtt{R} + \mathtt{A} \cdot \mathtt{B}$.

`REAL BiasInf(BIASINTERVAL *pA)`

Returns the lower bound of the interval `A`.

```
REAL BiasSup(BIASINTERVAL *pA)
```

Returns the upper bound of the interval `A`.

```
REAL BiasMid(BIASINTERVAL *pA)
```

Returns the midpoint of the interval `A`.

```
BiasMidRad(REAL *pm, REAL *pr, BIASINTERVAL *pA)
```

Caclulates the midpoint `m` and the radius `r` of the interval `A`.

```
REAL BiasDiam(BIASINTERVAL *pA)
```

Returns the diameter of the interval `A`.

```
REAL BiasAbs(BIASINTERVAL *pA)
```

Returns the absolute value of the interval `A`, i.e. $\max\{|\underline{A}|, |\overline{A}|\}$.

```
REAL BiasDistRI(REAL *pa, BIASINTERVAL *pB)
```

Returns the distance between `a` and `B`. If $a \in B$ then 0 is returned.

```
REAL BiasDistII(BIASINTERVAL *pA, BIASINTERVAL *pB)
```

Returns the distance between `A` and `B`.

```
INT BiasIntersection(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)
```

Calculates the intersection `R` between `A` and `B`. If the intersection is empty, a value of 0 is returned and the contents of `R` are undefined. Otherwise, a value of 1 is returned.

`BiasHullR(BIASINTERVAL *pR, REAL *pa)`

$\mathtt{R} = [\mathtt{a}, \mathtt{a}]$.

`BiasHullRR(BIASINTERVAL *pR, REAL *pa, REAL *pb)`

Initializes `R` with the convex hull of `a` and `b`.

`BiasHullRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB)`

Initializes `R` with the convex hull of `a` and `B`.

`BiasHullIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb)`

Initializes `R` with the convex hull of `A` and `b`.

`BiasHullII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)`

Initializes `R` with the convex hull of `A` and `B`.

`INT BiasInR(REAL *pa, BIASINTERVAL *pB)`

Returns 1, if $\mathtt{a} \in \mathtt{B}$. Otherwise, 0 is returned.

`INT BiasInI(BIASINTERVAL *pA, BIASINTERVAL *pB)`

Returns 1, if $\mathtt{A} \subseteq \mathtt{B}$. Otherwise, 0 is returned.

`INT BiasInInteriorR(REAL *pa, BIASINTERVAL *pB)`

Returns 1, if $\mathtt{a} \in \mathrm{Interior}(\mathtt{B})$. Otherwise, 0 is returned.

`INT BiasInInteriorI(BIASINTERVAL *pA, BIASINTERVAL *pB)`

Returns 1, if $\mathtt{A} \subseteq \mathrm{Interior}(\mathtt{B})$. Otherwise, 0 is returned.

`INT BiasIsEqual(BIASINTERVAL *pA, BIASINTERVAL *pB)`

Returns 1, if $\mathtt{A} = \mathtt{B}$. Otherwise, 0 is returned.

2.6 Vector Operations — Bias1

Headerfile: Bias1.h

In all vector routines the parameter `dim` denotes the number of vector elements. Throughout this description, elements are numbered beginning with the index 1. Vector parameters are given as pointer to the elements where the pointer points to the first element. We denote a single vector element by using indexing.

`VOID BiasPredVR(REAL *pr, REAL *pa, INT dim)`

The vector `r` is set to the predecessor of the vector `a`.

$$\mathtt{r}_i = \mathrm{pred}(\mathtt{a}_i), \quad i = 1, \ldots, \mathtt{dim}$$

`VOID BiasSuccVR(REAL *pr, REAL *pa, INT dim)`

The vector `r` is set to the successor of the vector `a`.

$$\mathtt{r}_i = \mathrm{pred}(\mathtt{a}_i), \quad i = 1, \ldots, \mathtt{dim}$$

`INT BiasPredVI(BIASINTERVAL *pR, BIASINTERVAL *pA, INT dim)`

Each component of the resulting interval vector `R` is set to the largest interval which is contained in the interior of the appropriate component of the interval vector `A`. If the computation of `R` was impossible due to a small diameter of `A`, a value of 0 is returned and `R` is undefined. Otherwise, a value of 1 is returned.

`VOID BiasSuccVI(BIASINTERVAL *pR, BIASINTERVAL *pA, INT dim)`

Each component of the resulting interval vector `R` is set to the smallest interval where the appropriate component of the interval vector `A` is contained in the interior.

```
VOID BiasAddVRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the sum of the vectors a and b.

$$\mathtt{R}_i = \mathtt{a}_i + \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasAddVRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R as the sum of the vector a and the interval vector B.

$$\mathtt{R}_i = \mathtt{a}_i + \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasAddVIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Calculates the interval vector R as the sum of the interval vector A and the vector b.

$$\mathtt{R}_i = \mathtt{A}_i + \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasAddVIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                 BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R as the sum of the interval vectors A and B.

$$\mathtt{R}_i = \mathtt{A}_i + \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasSubVRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the difference between the vectors a and b.

$$\mathtt{R}_i = \mathtt{a}_i - \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasSubVRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R as the difference between the vector a and the interval vector B.

$$\mathtt{R}_i = \mathtt{a}_i - \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasSubVIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Calculates the interval vector R as the difference between the interval vector A and the vector b.

$$\mathtt{R}_i = \mathtt{A}_i - \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasSubVIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                 BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R as the difference between the interval vectors A and B.

$$\mathtt{R}_i = \mathtt{A}_i - \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMulRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the product between the scalar a and the vector b.

$$\mathtt{R}_i = \mathtt{a} \cdot \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMulRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R which contains the true result of the product between the scalar a and the interval vector B.

$$\mathtt{R}_i = \mathtt{a} \cdot \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMulIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the product between the scalar interval A and the vector b.

$$\mathtt{R}_i = \mathtt{A} \cdot \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMulIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R which contains the true result of the product between the scalar interval A and the interval vector B.

$$\mathtt{R}_i = \mathtt{A} \cdot \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Adds to the interval vector R an interval vector which contains the true result of the product between the scalar a and the vector b.

$$\mathtt{R}_i = \mathtt{R}_i + \mathtt{a} \cdot \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Adds to the interval vector R an interval vector which contains the true result of the product between the scalar a and the interval vector B.

$$\mathtt{R}_i = \mathtt{R}_i + \mathtt{a} \cdot \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Adds to the interval vector R an interval vector which contains the true result of the product between the scalar interval A and the vector b.

$$\mathtt{R}_i = \mathtt{R}_i + \mathtt{A} \cdot \mathtt{b}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                BIASINTERVAL *pB, INT dim)
```

Adds to the interval vector R an interval vector which contains the true result of the product between the scalar interval A and the interval vector B.

$$\mathtt{R}_i = \mathtt{R}_i + \mathtt{A} \cdot \mathtt{B}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacRVIs(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                 INT dim, INT Bstep)
```

Same as BiasMacRVI but with different stepsize for the components of B.

$$\mathtt{R}_i = \mathtt{R}_i + \mathtt{a} \cdot \mathtt{B}_{(i-1)\mathtt{Bstep}+1}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacsRVIs(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                  INT dim, INT Rstep, INT Bstep)
```

Same as BiasMacRVIs but with different stepsize for the components of R.

$$\mathtt{R}_{(i-1)\mathtt{Rstep}+1} = \mathtt{R}_{(i-1)\mathtt{Rstep}+1} + \mathtt{a} \cdot \mathtt{B}_{(i-1)\mathtt{Bstep}+1}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasDivVRR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the quotient of the vector a and the scalar b.

$$\mathtt{R}_i = \frac{\mathtt{a}_i}{\mathtt{b}}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasDivVRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R which contains the true result of the quotient of the vector a and the scalar interval B.

$$\mathtt{R}_i = \frac{\mathtt{a}_i}{\mathtt{B}}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasDivVIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Calculates the interval vector R which contains the true result of the quotient of the interval vector A and the scalar b.

$$\mathtt{R}_i = \frac{\mathtt{A}_i}{\mathtt{b}}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasDivVII(BIASINTERVAL *pR, BIASINTERVAL *pA,
                BIASINTERVAL *pB, INT dim)
```

Calculates the interval vector R which contains the true result of the quotient of the interval vector A and the scalar interval B.

$$\mathtt{R}_i = \frac{\mathtt{A}_i}{\mathtt{B}}, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMacVRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Returns in the interval R an inclusion of the scalar product of the two vectors a and b.

```
VOID BiasMacVRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)
```

Returns in the interval R an inclusion of the scalar product of the vector a and the interval vector B.

```
VOID BiasMacVIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb, INT dim)
```

Returns in the interval R an inclusion of the scalar product of the interval vector A and the vector b.

```
VOID BiasMacVIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                 BIASINTERVAL *pB, INT dim)
```

Returns in the interval R an inclusion of the scalar product of the two interval vectors A and B.

```
VOID BiasInfV(REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector r containing the lower bounds of all components of the interval vector A.

$$\mathtt{r}_i = \inf \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasSupV(REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector r containing the upper bounds of all components of the interval vector A.

$$\mathtt{r}_i = \sup \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasNegV(BIASINTERVAL *pR, BIASINTERVAL *pA, INT dim)
```

Calculates the interval vector R containing the negative of the interval vector A.

$$\mathtt{r}_i = -\mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMidV(REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector r containing the midpoint of the interval vector A.

$$\mathtt{r}_i = \mathrm{mid}\, \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasMidRadV(REAL *pm, REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector m containing the midpoint of the interval vector A and the vector r containing the radius of A.

$$\begin{aligned} \mathtt{m}_i &= \operatorname{mid} \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim} \\ \mathtt{r}_i &= \operatorname{rad} \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim} \end{aligned}$$

```
VOID BiasDiamV(REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector r containing the diameters of all components of the interval vector A.

$$\mathtt{r}_i = \operatorname{diam} \mathtt{A}_i, \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasAbsV(REAL *pr, BIASINTERVAL *pA, INT dim)
```

Calculates the vector r containing the absolute values of all components of the interval vector A.

$$\mathtt{r}_i = |\mathtt{A}_i|, \quad i = 1, \ldots, \mathtt{dim}$$

```
INT BiasIntersectionV(BIASINTERVAL *pR, BIASINTERVAL *pA,
                      BIASINTERVAL *pB, INT dim)
```

Calculates the intersection vector R between the interval vectors A and B. If the intersection is empty, a value of 0 is returned and the contents of R are undefined. Otherwise, a value of 1 is returned.

```
VOID BiasHullVR(BIASINTERVAL *pR, REAL *pa, INT dim)
```

Converts the vector a to the point interval vector R.

$$\mathtt{R}_i = [\mathtt{a}_i, \mathtt{a}_i], \quad i = 1, \ldots, \mathtt{dim}$$

```
VOID BiasHullVRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT dim)
```

Initializes the interval vector R with the convex hull of the vectors a and b.

`VOID BiasHullVRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB, INT dim)`

Initializes the interval vector `R` with the convex hull of the vector `a` and the interval vector `B`.

`VOID BiasHullVIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, PREAK pb, INT dim)`

Initializes the interval vector `R` with the convex hull of the interval vector `A` and the vector `b`.

```
VOID BiasHullVIVI(BIASINTERVAL *pR, BIASINTERVAL *pA,
                  BIASINTERVAL *pB, INT dim)
```

Initializes the interval vector `R` with the convex hull of the interval vectors `A` and `B`.

`INT BiasInVR(REAL *pa, BIASINTERVAL *pB, INT dim)`

Returns 1, if `a` $\in$ `B`, where `a` is a vector and `B` is an interval vector. Otherwise, 0 is returned.

`INT BiasInVI(BIASINTERVAL *pA, BIASINTERVAL *pB, INT dim)`

Returns 1, if `A` $\subseteq$ `B`, where `A`,`B` are interval vectors. Otherwise, 0 is returned.

`INT BiasInInteriorVR(REAL *pa, BIASINTERVAL *pB, INT dim)`

Returns 1, if `a` $\in$ Interior(`B`), where `a` is a vector and `B` is an interval vector. Otherwise, 0 is returned.

`INT BiasInInteriorVI(BIASINTERVAL *pA, BIASINTERVAL *pB, INT dim)`

Returns 1, if `A` $\subseteq$ Interior(`B`), where `A`,`B` are interval vectors. Otherwise, 0 is returned.

`INT BiasIsEqualV(BIASINTERVAL *pA, BIASINTERVAL *pB, INT dim)`

Returns 1, if `A` $=$ `B`, where `A`,`B` are interval vectors. Otherwise, 0 is returned.

`VOID BiasSetToZeroV(BIASINTERVAL *pR, INT dim)`

Initializes the interval vector with zeroes, i.e.

$$\mathtt{R}_i = 0, \quad i = 1, \ldots, \mathtt{dim}$$

`VOID BiasSetToZeroVs(BIASINTERVAL *pR, INT dim, INT step)`

Same as before, but with the stepsize `step`.

$$\mathtt{R}_{(i-1)\mathtt{step}+1} = 0, \quad i = 1, \ldots, \mathtt{dim}$$

2.7 Matrix Operations — `Bias2`

Headerfile: `Bias2.h`

In all matrix routines the parameter `rows` denotes the number of rows in the matrix and the parameter `cols` denotes the number of columns. Throughout this description, elements are numbered beginning with the index 1. Matrix parameters are given as pointer to the elements where the pointer points to the first element. We denote a single matrix element by using indexing giving the row first.

`VOID BiasPredMR(REAL *pr, REAL *pa, INT rows, INT cols)`

The matrix `r` is set to the predecessor of the matrix `a`.

$$\mathtt{r}_{ij} = \mathrm{pred}(\mathtt{a}_{ij}), \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

`VOID BiasSuccMR(REAL *pr, REAL *pa, INT rows, INT cols)`

The matrix `r` is set to the successor of the matrix `a`.

$$\mathtt{r}_{ij} = \mathrm{pred}(\mathtt{a}_{ij}), \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

`VOID BiasPredMI(BIASINTERVAL *pR, BIASINTERVAL *pA, INT rows, INT cols)`

Each component of the resulting interval matrix `R` is set to the largest interval which is contained in the interior of the appropriate component of the interval matrix `A`. If the computation of `R` was impossible due to a small diameter of `A`, a value of 0 is returned and `R` is undefined. Otherwise, a value of 1 is returned.

```
VOID BiasSuccMI(BIASINTERVAL *pR, BIASINTERVAL *pA, INT rows, INT cols)
```

Each component of the resulting interval matrix `R` is set to the smallest interval where the appropriate component of the interval matrix `A` is contained in the interior.

```
VOID BiasAddMRMR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                 INT rows, INT cols)
```

Calculates the interval matrix `R` which contains the true result of the sum of the matrices `a` and `b`.

$$\mathtt{R}_{ij} = \mathtt{a}_{ij} + \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasAddMRMI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval matrix `R` as the sum of the matrix `a` and the interval matrix `B`.

$$\mathtt{R}_{ij} = \mathtt{a}_{ij} + \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasAddMIMR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                 INT rows, INT cols)
```

Calculates the interval matrix `R` as the sum of the interval matrix `A` and the matrix `b`.

$$\mathtt{R}_{ij} = \mathtt{A}_{ij} + \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasAddMIMI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval matrix `R` as the sum of the interval matrices `A` and `B`.

$$\mathtt{R}_{ij} = \mathtt{A}_{ij} + \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasSubMRMR(BIASINTERVAL *pR, REAL *pa, REAL *pb, INT rows, INT cols)
```

Calculates the interval matrix `R` which contains the true result of the difference between the matrices `a` and `b`.

$$\mathtt{R}_{ij} = \mathtt{a}_{ij} - \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasSubMRMI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval matrix R as the difference between the matrix a and the interval matrix B.

$$\mathtt{R}_{ij} = \mathtt{a}_{ij} - \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasSubMIMR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                 INT rows, INT cols)
```

Calculates the interval matrix R as the difference between the interval matrix A and the matrix b.

$$\mathtt{R}_{ij} = \mathtt{A}_{ij} - \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasSubMIMI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval matrix R as the difference between the interval matrices A and B.

$$\mathtt{R}_{ij} = \mathtt{A}_{ij} - \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMulRMR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the product between the scalar a and the matrix b.

$$\mathtt{R}_{ij} = \mathtt{a} \cdot \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMulRMI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the product between the scalar a and the interval matrix B.

$$\mathtt{R}_{ij} = \mathtt{a} \cdot \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMulIMR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the product between the scalar interval A and the matrix b.

$$\mathtt{R}_{ij} = \mathtt{A} \cdot \mathtt{b}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMulIMI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the product between the scalar interval A and the interval matrix B.

$$\mathtt{R}_{ij} = \mathtt{A} \cdot \mathtt{B}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMulMRVR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                 INT rows, INT cols)
```

Calculates the interval vector R which contains the true result of the product between the matrix a and the vector b.

$$\mathtt{R}_{i} = \sum_{j=1}^{\mathtt{cols}} \mathtt{a}_{ij} \cdot \mathtt{b}_{j}, \quad i = 1, \ldots, \mathtt{rows}$$

```
VOID BiasMulMRVI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval vector R which contains the true result of the product between the matrix a and the interval vector B.

$$\mathtt{R}_{i} = \sum_{j=1}^{\mathtt{cols}} \mathtt{a}_{ij} \cdot \mathtt{B}_{j}, \quad i = 1, \ldots, \mathtt{rows}$$

```
VOID BiasMulMIVR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                 INT rows, INT cols)
```

Calculates the interval vector R which contains the true result of the product between the interval matrix A and the vector b.

$$\mathtt{R}_{i} = \sum_{j=1}^{\mathtt{cols}} \mathtt{A}_{ij} \cdot \mathtt{b}_{j}, \quad i = 1, \ldots, \mathtt{rows}$$

```
VOID BiasMulMIVI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                 INT rows, INT cols)
```

Calculates the interval vector R which contains the true result of the product between the interval matrix A and the interval vector B.

$$\mathtt{R}_i = \sum_{j=1}^{\mathtt{cols}} \mathtt{A}_{ij} \cdot \mathtt{B}_j, \quad i = 1, \ldots, \mathtt{rows}$$

```
VOID BiasMulMRMR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                 INT arows, INT acols, INT bcols)
```

Calculates the interval matrix R which contains the true result of the product between the matrices a and b.

$$\mathtt{R}_{ij} = \sum_{k=1}^{\mathtt{acols}} \mathtt{a}_{ik} \cdot \mathtt{b}_{kj}, \quad i = 1, \ldots, \mathtt{arows};\ j = 1, \ldots, \mathtt{bcols}$$

```
VOID BiasMulMRMI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                 INT arows, INT acols, INT Bcols)
```

Calculates the interval matrix R which contains the true result of the product between the matrix a and the interval matrix B.

$$\mathtt{R}_{ij} = \sum_{k=1}^{\mathtt{acols}} \mathtt{a}_{ik} \cdot \mathtt{B}_{kj}, \quad i = 1, \ldots, \mathtt{arows};\ j = 1, \ldots, \mathtt{Bcols}$$

```
VOID BiasMulMIMR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                 INT Arows, INT Acols, INT bcols)
```

Calculates the interval matrix R which contains the true result of the product between the interval matrix A and the matrix b.

$$\mathtt{R}_{ij} = \sum_{k=1}^{\mathtt{Acols}} \mathtt{A}_{ik} \cdot \mathtt{b}_{kj}, \quad i = 1, \ldots, \mathtt{Arows};\ j = 1, \ldots, \mathtt{bcols}$$

```
VOID BiasMulMIMI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                 INT Arows, INT Acols, INT Bcols)
```

Calculates the interval matrix R which contains the true result of the product between the interval matrices A and B.

$$\mathtt{R}_{ij} = \sum_{k=1}^{\mathtt{Acols}} \mathtt{A}_{ik} \cdot \mathtt{B}_{kj}, \quad i = 1, \ldots, \mathtt{Arows};\ j = 1, \ldots, \mathtt{Bcols}$$

```
VOID BiasDivMRR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the quotient of the matrix a and the scalar b.

$$\mathtt{R}_{ij} = \frac{\mathtt{a}_{ij}}{\mathtt{b}}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasDivMRI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the quotient of the matrix a and the scalar interval B.

$$\mathtt{R}_{ij} = \frac{\mathtt{a}_{ij}}{\mathtt{B}}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasDivMIR(BIASINTERVAL *pR, BIASINTERVAL *pA, REAL *pb,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the quotient of the interval matrix A and the scalar b.

$$\mathtt{R}_{ij} = \frac{\mathtt{A}_{ij}}{\mathtt{b}}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasDivMII(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                INT rows, INT cols)
```

Calculates the interval matrix R which contains the true result of the quotient of the interval matrix A and the scalar interval B.

$$\mathtt{R}_{ij} = \frac{\mathtt{A}_{ij}}{\mathtt{B}}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasInfM(REAL *pr, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the matrix r containing the lower bounds of all components of the interval matrix A.

$$\mathtt{r}_{ij} = \inf \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasSupM(REAL *pr, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the matrix r containing the upper bounds of all components of the interval matrix A.

$$\mathtt{r}_{ij} = \sup \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasNegM(BIASINTERVAL *pR, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the interval matrix R containing the negative of the interval matrix A.

$$\mathtt{r}_{ij} = -\mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMidM(REAL *pr, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the matrix r containing the midpoint of the interval matrix A.

$$\mathtt{r}_{ij} = \mathrm{mid}\, \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasMidRadM(REAL *pm, REAL *pr, BIASINTERVAL *pA,
                 INT rows, INT cols)
```

Calculates the matrix m containing the midpoint of the interval matrix A and the matrix r containing the radius of A.

$$\begin{aligned} \mathtt{m}_{ij} &= \mathrm{mid}\, \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols} \\ \mathtt{r}_{ij} &= \mathrm{rad}\, \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols} \end{aligned}$$

```
VOID BiasDiamM(REAL *pr, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the matrix r containing the diameters of all components of the interval matrix A.

$$\mathtt{r}_{ij} = \mathrm{diam}\, \mathtt{A}_{ij}, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
VOID BiasAbsM(REAL *pr, BIASINTERVAL *pA, INT rows, INT cols)
```

Calculates the matrix r containing the absolute values of all components of the interval matrix A.

$$\mathtt{r}_{ij} = |\mathtt{A}_{ij}|, \quad i = 1, \ldots, \mathtt{rows};\ j = 1, \ldots, \mathtt{cols}$$

```
INT BiasIntersectionM(BIASINTERVAL *pR, BIASINTERVAL *pA,
                      BIASINTERVAL *pB, INT rows, INT cols)
```

Calculates the intersection matrix `R` between the interval matrices `A` and `B`. If the intersection is empty, a value of 0 is returned and the contents of `R` are undefined. Otherwise, a value of 1 is returned.

```
VOID BiasHullMR(BIASINTERVAL *pR, REAL *pa, INT rows, INT cols)
```

Converts the matrix `a` to the point interval matrix `R`.

$$\mathtt{R}_{ij} = [\mathtt{a}_{ij}, \mathtt{a}_{ij}], \quad i = 1, \dots, \mathtt{rows};\ j = 1, \dots, \mathtt{cols}$$

```
VOID BiasHullMRMR(BIASINTERVAL *pR, REAL *pa, REAL *pb,
                  INT rows, INT cols)
```

Initializes the interval matrix `R` with the convex hull of the matrices `a` and `b`.

```
VOID BiasHullMRMI(BIASINTERVAL *pR, REAL *pa, BIASINTERVAL *pB,
                  INT rows, INT cols)
```

Initializes the interval matrix `R` with the convex hull of the matrix `a` and the interval matrix `B`.

```
VOID BiasHullMIMR(BIASINTERVAL *pR, BIASINTERVAL *pA, PREAK pb,
                  INT rows, INT cols)
```

Initializes the interval matrix `R` with the convex hull of the interval matrix `A` and the matrix `b`.

```
VOID BiasHullMIMI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB,
                  INT rows, INT cols)
```

Initializes the interval matrix `R` with the convex hull of the interval matrices `A` and `B`.

```
INT BiasInMR(REAL *pa, BIASINTERVAL *pB, INT rows, INT cols)
```

Returns 1, if $\mathtt{a} \in \mathtt{B}$, where `a` is a matrix and `B` is an interval matrix. Otherwise, 0 is returned.

`INT BiasInMI(BIASINTERVAL *pA, BIASINTERVAL *pB, INT rows, INT cols)`

Returns 1, if $\mathtt{A} \subseteq \mathtt{B}$, where `A`,`B` are interval matrices. Otherwise, 0 is returned.

`INT BiasInInteriorMR(REAL *pa, BIASINTERVAL *pB, INT rows, INT cols)`

Returns 1, if $\mathtt{a} \in \text{Interior}(\mathtt{B})$, where `a` is a matrix and `B` is an interval matrix. Otherwise, 0 is returned.

`INT BiasInInteriorMI(BIASINTERVAL *pA, BIASINTERVAL *pB, INT rows, INT cols)`

Returns 1, if $\mathtt{A} \subseteq \text{Interior}(\mathtt{B})$, where `A`,`B` are interval matrices. Otherwise, 0 is returned.

`INT BiasIsEqualM(BIASINTERVAL *pA, BIASINTERVAL *pB, INT rows, INT cols)`

Returns 1, if $\mathtt{A} = \mathtt{B}$, where `A`,`B` are interval matrices. Otherwise, 0 is returned.

`VOID BiasSetToZeroM(BIASINTERVAL *pR, INT rows, INT cols)`

Initializes the interval matrix with zeroes, i.e.

$$\mathtt{R}_{ij} = 0, \quad i = 1, \ldots, \mathtt{rows}; \; j = 1, \ldots, \mathtt{cols}$$

2.8 Scalar Standard Functions — BiasF

Headerfile: BiasF.h

The implementation of the standard functions contained in the distribution uses the floating point standard functions of the C library. It is assumed that these standard functions yield results, which are quite close to the true result. The macro `_BIASSTDFUNCINVALIDBITS_`, which is defined in the file `BiasRnd.h` in the architecture specific directory, contains the max. number of invalid bits obtained by the standard C library functions.

Thus, the implementation of the interval standard functions only yields true enclosures in every case, as long as the error of the C library functions is known.

`VOID BiasFuncInit()`

Initializes all internal values, must be called prior any other BIAS standard function call.

`VOID BiasSin(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \sin(\mathtt{A})$

`VOID BiasCos(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \cos(\mathtt{A})$

`VOID BiasTan(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \tan(\mathtt{A})$

`VOID BiasCot(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \cot(\mathtt{A})$

`VOID BiasArcSin(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \arcsin(\mathtt{A})$

`VOID BiasArcCos(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \arccos(\mathtt{A})$

`VOID BiasArcTan(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \arctan(\mathtt{A})$

`VOID BiasArcCot(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes $\mathtt{R} = \mathrm{arccot}(\mathtt{A})$

`VOID BiasSinh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = sinh(A)

`VOID BiasCosh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = cosh(A)

`VOID BiasTanh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = tanh(A)

`VOID BiasCoth(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = coth(A)

`VOID BiasArSinh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = arsinh(A)

`VOID BiasArCosh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = arcosh(A)

`VOID BiasArTanh(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = artanh(A)

`VOID BiasArCoth(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = arcoth(A)

`VOID BiasExp(BIASINTERVAL *pR, BIASINTERVAL *pA)`

Computes R = exp(A)

```
VOID BiasLog(BIASINTERVAL *pR, BIASINTERVAL *pA)
```

Computes $\mathtt{R} = \ln(\mathtt{A})$

```
VOID BiasLog10(BIASINTERVAL *pR, BIASINTERVAL *pA)
```

Computes $\mathtt{R} = \log_{10}(\mathtt{A})$

```
VOID BiasIAbs(BIASINTERVAL *pR, BIASINTERVAL *pA)
```

Computes $\mathtt{R} = \{\, |a| \mid a \in \mathtt{A} \,\}$

```
VOID BiasSqr(BIASINTERVAL *pR, BIASINTERVAL *pA)
```

Computes $\mathtt{R} = \{\, a^2 \mid a \in \mathtt{A} \,\}$

```
VOID BiasSqrt(BIASINTERVAL *pR, BIASINTERVAL *pA)
```

Computes $\mathtt{R} = \sqrt{\mathtt{A}}$

```
VOID BiasRoot(BIASINTERVAL *pR, BIASINTERVAL *pA, INT n)
```

Computes $\mathtt{R} = \sqrt[\mathtt{n}]{\mathtt{A}}$

```
VOID BiasPowerN(BIASINTERVAL *pR, BIASINTERVAL *pA, INT n)
```

Computes $\mathtt{R} = \mathtt{A}^{\mathtt{n}}$

```
VOID BiasPowerI(BIASINTERVAL *pR, BIASINTERVAL *pA, BIASINTERVAL *pB)
```

Computes $\mathtt{R} = \mathtt{A}^{\mathtt{B}}$

Chapter 3

PROFIL

This chapter describes the basic parts of PROFIL (Programmer's Runtime Optimized Fast Interval Library), a C++ class library for developing and implementing interval algorithms. More specific subroutines (e.g. local/global optimization) are described in later chapters.

The multiple precision arithmetic is now permanently included in PROFIL. Section 3.2 describes the input and output format for intervals in detail.

3.1 Supported Data Types

The following data types are currently supported by PROFIL:

Data Type	Defined by
`INT`	any PROFIL header file
`REAL`	any PROFIL header file
`INTERVAL`	`Interval.h`
`BOOL`	`Boolean.h`
`INTEGER_VECTOR`	`IntegerVector.h`
`VECTOR`	`Vector.h`
`INTERVAL_VECTOR`	`IntervalVector.h`
`BOOL_VECTOR`	`Boolean.h`
`INTEGER_MATRIX`	`IntegerMatrix.h`
`MATRIX`	`Matrix.h`
`INTERVAL_MATRIX`	`IntervalMatrix.h`
`BOOL_MATRIX`	`Boolean.h`

Table 3.1: Supported Data Types

Any data type can be written to output via the C++ streams. If `a` is a variable of any type above, it can be written to output by typing

```
cout << a;
```

A new line in output can be achieved by

```
cout << endl;
```

The input can be performed quite the same way:

```
cin >> a;
```

reads the contents of the variable `a` from input.

3.2 Input/Output of Intervals

With the multiple precision arithmetic, the interval input and output routines use the multiple precision interval I/O routines, all bounds on input and output are correctly rounded and the input format is the same as the output format. Additionally, there are some nice improvements when using intervals with a small relative diameter. For example, the interval $[12.345, 12.346]$ can be typed as:

```
[12.345,12.346]
12.34[5,6]
1.234[5,6]E+1
[1.2345,1.2346]E+1
[1.2345E+1,1.2346E+1]
```

For intervals like $[1.999, 2.002]$ the special notation

```
1.99[9,12]
```

can be used, where the `1` in the upper bound denotes a carry to be added to the least significant common digit. Point intervals can be read as one number, e.g. `0.1` and `[0.1,0.1]` are the same interval, both containing the real number 0.1.

The output format for intervals is controlled by the same commands used for the multiple precision reals/intervals (see files `LongReal.h` and `LongInterval.h`).

To get an enclosure of a real constant, the function `Hull` cannot be used as expected (e.g. by typing `Hull(0.1)`). This is because `Hull(0.1)` returns a point interval whose bounds are equal to a floating point approximation of 0.1. If you need an enclosure of 0.1, use either `DivBounds(1.0,10.0)` or `Enclosure("1.0")`. The latter is using the multiple precision package, where the first is independent of it.

3.3 Further Remarks

The indexing of all vector and matrix types starts with 1.

It may look surprising that there is no way to assign a value to the infimum or to the supremum of an interval type. This is due to the fact that all interval operations are performed by BIAS which hides the original interval representation. An explicit assignment would be possible, if C++ allows to make a difference between the left and the right side of an assignment. Unfortunately, this is not possible. If you really need to assign a value to a bound of an interval, e.g. you want to have

```
Inf(x) = 1.0;
```

use the following construct instead:

```
x = Hull(1.0, Sup(x));
```

Note, that there is a slight difference: The first case would allow intervals with a lower bound larger than the upper bound, which is impossible in the second case.

3.4 Operations defined by any header file

Let `a`,`b` be either `REAL` or `INT` types. Then the following operations are possible (The result is of type `REAL` if at least one operand is of type `REAL`. Otherwise, the result is of type `INT`):

Operation	Description
`a + b`	Addition
`a - b`	Subtraction
`a * b`	Multiplication
`a / b`	Division
`a += b`	Same as `a = a + b`
`a -= b`	Same as `a = a - b`
`a *= b`	Same as `a = a * b`

continued on next page ...

... continued from previous page

Operation	Description
`a /= b`	Same as `a = a / b`

Table 3.2: Operations defined by any header file

Let `p`,`q` be of type `BOOL`. Then the result of all following operations is also of type `BOOL`:

Operation	Description
`TRUE`	logical true
`FALSE`	logical false
`p || q`	logical or
`p && q`	logical and
`!p`	logical not

Table 3.3: Boolean Operations

3.5 Error Handling (Error.h)

PROFIL contains a simple set of functions used for error handling defined in the file `Error.h`. An error is reported using the function `Error`. The first parameter is a string and the second parameter is a number both describing the error. The last parameter p is a number which describes the severity of the error. The severity is divided into 4 classes:

Range of p	Class
-1	Fatal
0 ... `ErrorHandler::WarningLevel` $-$ 1	Ignored
`ErrorHandler::WarningLevel` ... `ErrorHandler::SevereLevel` $-$ 1	Warning
`ErrorHandler::SevereLevel` ...	Severe

Table 3.4: Error Severities

Fatal errors always terminate the program producing a core image dump. They can also be produced using the subroutine `FatalError` with only a string as parameter. In general, errors in the class “Ignored” are not printed and the program execution continues. Errors in the class “Warning” are printed and the program execution also continues. “Severe” errors are printed and the program terminates. The behaviour can be changed by modifying the levels for the error classes. On program start, the following settings are valid:

$$\begin{aligned}\texttt{ErrorHandler::WarningLevel} &= \texttt{ErrorHandler::Warning} = 1000\\ \texttt{ErrorHandler::SevereLevel} &= \texttt{ErrorHandler::SevereError} = 2000\end{aligned}$$

Further, the following values are defined:

$$\texttt{ErrorHandler::FatalError} = -1$$
$$\texttt{ErrorHandler::Ignore} = 0$$

The last error code (i.e. the last used second parameter of `Error`) is available in the variable `ErrorHandler::LastErrorCode`. If no error leads to program termination, but warnings or ignored errors occured, an error report is either generated automatically or can be called anywhere using `ErrorReport()`.

3.6 Constants.h

This file defines some useful machine constants, which are:

Name of Constant	Description
`Machine::Epsilon`	machine precision
`Machine::MinPositive`	smallest positive machine number
`Machine::PosInfinity`	$+\infty$ or at least a huge positive number
`Machine::NegInfinity`	$-\infty$ or at least a huge negative number
`Machine::NaN`	NaN (not a number)

Table 3.5: Machine Constants

Due to the automatic initialization of the machine constants given above, the object file belonging to `Constants.h` must be linked to the executable code. This is generally done automatically. If not, define in your program a global variable as e.g.

```
Machine MachineAutomaticInit;
```

to initialize the machine constants.

3.7 Complex.h

The file defines the complex data type and some basic operations on it. The data type is called `COMPLEX`. In the following, let `c`,`d` be of type `COMPLEX` and `r`,`s` be of type `REAL`. `x` and `y` denote expressions of either type `REAL` or `COMPLEX`. The result of all operations is of type `COMPLEX` if not stated otherwise. The following operations are provided:

Operation	Description
`COMPLEX c;`	Declaration of `c`
`COMPLEX c(1.0);`	Declaration of `c` and initialization with 1
`COMPLEX c(1.0,2.0);`	Declaration of `c` and initialization with $1 + 2i$
`c = x`	Assignment
`c += x`	Same as `c = c + x`
`c -= x`	Same as `c = c - x`
`c *= x`	Same as `c = c * x`
`c /= x`	Same as `c = c / x`
`+c`	monadic plus
`-c`	monadic minus
`x + y`	Complex addition (`x` and `y` must not be both `REAL`)
`x - y`	Complex subtraction (`x` and `y` must not be both `REAL`)
`x * y`	Complex product (`x` and `y` must not be both `REAL`)
`x / y`	Complex division (`x` and `y` must not be both `REAL`)

Table 3.6: Complex Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`Re(c)`	`REAL`	Real part of `c`
`Im(c)`	`REAL`	Imaginary part of `c`
`Conjg(c)`	`COMPLEX`	returns the conjugate of `c`
`Abs(c)`	`REAL`	Absolute value of `c`
`Sqrt(c)`	`COMPLEX`	Complex root of `c`

Table 3.7: Complex Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`x == y`	True, if `x` is equal to `y` (`x` and `y` must not be both `REAL`)
`x != y`	True, if `x` is not equal to `y` (`x` and `y` must not be both `REAL`)

Table 3.8: Comparisons

The variable `I` contains the value $(0 + 1i)$.

3.8 Interval.h

In this file the scalar interval operations and the data type `INTERVAL` are defined. In the following let `a`,`b`,`c` be of type `INTERVAL` and `r`,`s` be of type `REAL`. If `x` or `y` are used, either `INTERVAL` or `REAL` types are allowed. If not stated otherwise, all results are of type `INTERVAL`. The following basic operations are provided:

Operation	Description
`INTERVAL a;`	Declaration of `a`
`INTERVAL a(1.0);`	Declaration of `a` and initialization with $[1, 1]$
`INTERVAL a(1.0,2.0);`	Declaration of `a` and initialization with $[1, 2]$
`a = x`	Assignment
`a += x`	Same as `a = a + x`
`a -= x`	Same as `a = a - x`
`a *= x`	Same as `a = a * x`
`a /= x`	Same as `a = a / x`
`+a`	monadic plus
`-a`	monadic minus
`x + y`	Interval addition (`x` and `y` must not be both `REAL`)
`x - y`	Interval subtraction (`x` and `y` must not be both `REAL`)
`x * y`	Interval product (`x` and `y` must not be both `REAL`)
`x / y`	Interval division (`x` and `y` must not be both `REAL`)

Table 3.9: Basic Interval Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`AddBounds(r,s)`	`INTERVAL`	Evaluation of `r + s` yielding bounds for the true result
`SubBounds(r,s)`	`INTERVAL`	Same as above, but for `r - s`
`MulBounds(r,s)`	`INTERVAL`	Same as above, but for `r * s`
`DivBounds(r,s)`	`INTERVAL`	Same as above, but for `r / s`
`Inf(a)`	`REAL`	Lower bound of `a`
`Sup(a)`	`REAL`	Upper bound of `a`
`Pred(r)`	`REAL`	Largest floating-point number less than `r`
`Succ(r)`	`REAL`	Smallest floating-point number greater than `r`
`Pred(a)`	`INTERVAL`	Largest interval contained in the interior of `a`
`Succ(a)`	`INTERVAL`	Smallest interval in which `a` is contained in the interior
`Hull(r)`	`INTERVAL`	Point interval $[\mathtt{r}, \mathtt{r}]$

continued on next page ...

... *continued from previous page*

Operation	Return Type	Description
`Hull(x,y)`	`INTERVAL`	Convex hull of `x` and `y`
`SymHull(r)`	`INTERVAL`	short form for `Hull(-r,r)`
`Mid(a)`	`REAL`	Midpoint of `a`
`Diam(a)`	`REAL`	Diameter of `a`
`Abs(a)`	`REAL`	Absolute value of `a`
`Mig(a)`	`REAL`	Mignitude of `a`
`Distance(x,y)`	`REAL`	Distance between `x` and `y`
`Intersection(a,b,c)`	`INT`	If `b` and `c` intersect, a value of 1 is returned and `a` contains the intersection. Otherwise 0 is returned and the contents of `a` are undefined.

Table 3.10: Basic Interval Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`x <= a`	True, if `x` is contained in `a`
`x < a`	True, if `x` is contained in the interior `a`
`a == b`	True, if `a` and `b` are equal
`a != b`	True, if `a` and `b` are not equal

Table 3.11: Basic Interval Comparisons

3.9 Vector.h

This file defines all operations concerning real vectors. The data type is called `VECTOR`. In the following we assume `v`,`w` to be of type `VECTOR` and `r` to be of type `REAL`. The variable `i` denotes an expression of type `INT`. If not stated otherwise, the result type is `VECTOR`. The following basic operations are provided:

Operation	Description
`VECTOR v;`	Declaration of `v`. The dimension of `v` is determined at the first assignment to `v`.
`VECTOR v(20);`	Declaration of `v` as a vector with 20 components
`v = w`	Assignment
`v += w`	Same as `v = v + w`
`v -= w`	Same as `v = v - w`
`v *= r`	Same as `v = r * v`

continued on next page ...

. . . continued from previous page

Operation	Description
`v /= r`	Same as `v = v / r`
`+v`	monadic plus
`-v`	monadic minus
`v + w`	Vector addition
`v - w`	Vector subtraction
`r * w`	Multiplication with scalar
`v / r`	Division by scalar
`v * w`	Scalar product (result type: `REAL`)
`v(i)`	denotes the `i`-th element of `v`

Table 3.12: Basic Vector Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`Dimension(v)`	`INT`	Returns the number of components in `v`
`Resize(v,i)`	—	Discards the old contents of `v` and resizes it to contain `i` elements
`MakeTemporary(v)`	—	Sets `v` to be a temporary variable
`MakePermanent(v)`	—	Sets `v` to be a non-temporary variable
`Clear(v)`	—	Initializes all elements of `v` with 0
`Initialize(v,r)`	—	Initializes all elements of `v` with `r`
`Sqr(v)`	`REAL`	same as `v * v`
`Norm(v)`	`REAL`	Computes the 2-Norm of `v`
`Max(v)`	`REAL`	returns the maximum of all elements of `v`
`Min(v)`	`REAL`	returns the minimum of all elements of `v`

Table 3.13: Vector Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`v < w`	True, if each element of `v` is less than the appropriate element in `w`
`v <= w`	same as above, but less or equal
`v > w`	same as above, but greater than
`v >= w`	same as above, but greater or equal

Table 3.14: Vector Comparisons

3.10 IntervalVector.h

This file defines all operations concerning interval vectors. The appropriate data type is called `INTERVAL_VECTOR`. In the following we assume `iu`,`iv`,`iw` to be of type `INTERVAL_VECTOR` and `r` to be of type `REAL`. We further assume `v`,`w` to be of type `VECTOR`. The variable `i` denotes an expression of type `INT`. If `x` or `y` are used, either `INTERVAL_VECTOR` or `VECTOR` types are allowed. `z` denotes a variable of either `INTERVAL` or `REAL`. `a` denotes a variable of type `INTERVAL`. If not stated otherwise, the result type is `INTERVAL_VECTOR`. The following basic operations are provided:

Operation	Description
`INTERVAL_VECTOR iv;`	Declaration of `iv`. The dimension of `iv` is determined at the first assignment to `iv`.
`INTERVAL_VECTOR iv(20);`	Declaration of `iv` as a vector with 20 components
`iv = x`	Assignment
`iv += x`	Same as `iv = iv + x`
`iv -= x`	Same as `iv = iv - x`
`iv *= z`	Same as `iv = z * iv`
`iv /= z`	Same as `iv = iv / z`
`+iv`	monadic plus
`-iv`	monadic minus
`x + y`	Interval vector addition (`x` and `y` must not be both `VECTOR`)
`x - y`	Interval vector subtraction (`x` and `y` must not be both `VECTOR`)
`z * x`	Multiplication with scalar. The case `VECTOR x` and `REAL z` is not allowed
`x / z`	Division by scalar. The case `VECTOR x` and `REAL z` is not allowed
`x * y`	Scalar product (result type: `REAL`) (`x` and `y` must not be both `VECTOR`)
`iv(i)`	denotes the `i`-th element of `iv`

Table 3.15: Interval Vector Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`Dimension(iv)`	`INT`	Returns the number of components in `iv`
`Resize(iv,i)`	—	Discards the old contents of `iv` and resizes it to contain `i` elements
`MakeTemporary(iv)`	—	Sets `iv` to be a temporary variable
`MakePermanent(iv)`	—	Sets `iv` to be a non-temporary variable
`Clear(iv)`	—	Initializes all elements of `iv` with 0
`Initialize(iv,a)`	—	Initializes all elements of `iv` with `a`
`AddBounds(v,w)`	`INTERVAL_VECTOR`	Evaluation of `v + w` yielding bounds for the true result
`SubBounds(v,w)`	`INTERVAL_VECTOR`	Same as above, but for `v - w`
`MulBounds(r,v)`	`INTERVAL_VECTOR`	Same as above, but for `r * v`
`DivBounds(v,r)`	`INTERVAL_VECTOR`	Same as above, but for `v / r`
`Inf(iv)`	`VECTOR`	Lower bounds of `iv`
`Sup(iv)`	`VECTOR`	Upper bounds of `iv`
`Hull(v)`	`INTERVAL_VECTOR`	Point interval vector [`v`, `v`]
`Hull(x,y)`	`INTERVAL_VECTOR`	Interval hull of `x` and `y`
`SymHull(v)`	`INTERVAL_VECTOR`	short form for `Hull(-v,v)`
`Mid(iv)`	`VECTOR`	Midpoint of `iv`
`Diam(iv)`	`VECTOR`	Diameter of `iv`
`Abs(iv)`	`VECTOR`	Absolute value of `iv`
`Intersection(iu,iv,iw)`	`INT`	If `iv` and `iw` intersect, a value of 1 is returned and `iu` contains the intersection. Otherwise 0 is returned and the contents of `iu` are undefined.
`Sqr(iv)`	`INTERVAL`	Calculates generally narrower bounds than `iv * iv`
`Norm(iv)`	`INTERVAL`	Computes the 2-Norm of `iv`

Table 3.16: Interval Vector Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`x <= iv`	True, if `x` is contained in `iv`
`x < iv`	True, if `x` is contained in the interior `iv`

continued on next page ...

... continued from previous page

Comparison	Description
`iv == iw`	True, if `iv` and `iw` are equal
`iv != iw`	True, if `iv` and `iw` are not equal

Table 3.17: Interval Vector Comparisons

3.11 IntegerVector.h

This file defines all operations concerning integer vectors. The appropriate data type is called `INTEGER_VECTOR`. In the following we assume `v`,`w` to be of type `INTEGER_VECTOR`. The variable `i` denotes an expression of type `INT`. The following basic operations are provided:

Operation	Description
`INTEGER_VECTOR v;`	Declaration of `v`. The dimension of `v` is determined at the first assignment to `v`.
`INTEGER_VECTOR v(20);`	Declaration of `v` as an integer vector with 20 components
`v = w`	Assignment
`v += w`	Same as `v = v + w`
`v -= w`	Same as `v = v - w`
`v *= i`	Same as `v = i * v`
`v /= i`	Same as `v = v / i`
`v + w`	Vector addition
`v - w`	Vector subtraction
`i * w`	Multiplication with scalar
`v / i`	Division by scalar
`v * w`	Scalar product (result type: `INT`)
`v(i)`	denotes the `i`-th element of `v`

Table 3.18: Integer Vector Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`Dimension(v)`	`INT`	Returns the number of components in `v`
`Resize(v,i)`	—	Discards the old contents of `v` and resizes it to contain `i` elements
`MakeTemporary(v)`	—	Sets `v` to be a temporary variable
`MakePermanent(v)`	—	Sets `v` to be a non-temporary variable
`Clear(v)`	—	Initializes all elements of `v` with 0

continued on next page ...

... continued from previous page

Operation	Return Type	Description
`Initialize(v,i)`	—	Initializes all elements of `v` with `i`

Table 3.19: Integer Vector Functions

3.12 Matrix.h

This file defines all operations concerning real matrices. The data type is called `MATRIX`. In the following we assume `A`,`B` to be of type `MATRIX` and `v`,`w` to be of type `VECTOR`. `r` denotes an expression of type `REAL`. The variable `i` denotes an expression of type `INT`. If not stated otherwise, the result type is `MATRIX`. The following basic operations are provided:

Operation	Description
`MATRIX A;`	Declaration of `A`. The dimension of `A` is determined at the first assignment to `A`.
`MATRIX A(20,30);`	Declaration of `A` as a matrix with 20 rows and 30 columns
`A = B`	Assignment
`A += B`	Same as `A = A + B`
`A -= B`	Same as `A = A - B`
`A *= r`	Same as `A = r * A`
`A /= r`	Same as `A = A / r`
`+A`	monadic plus
`-A`	monadic minus
`A + B`	Matrix addition
`A - B`	Matrix subtraction
`r * A`	Multiplication with scalar
`A / r`	Division by scalar
`A * v`	Matrix vector product (the result type is: `VECTOR`)
`A * B`	Matrix product
`A(i,j)`	denotes the `i`-th row and `j`-th column of `A`

Table 3.20: Matrix Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`RowDimension(A)`	`INT`	Returns the number of rows in `A`
`ColDimension(A)`	`INT`	Returns the number of columns in `A`
`Resize(A,i,j)`	—	Discards the old contents of `A` and resizes it to contain `i` rows and `j` columns

continued on next page ...

... continued from previous page

Operation	Return Type	Description
`MakeTemporary(A)`	—	Sets `A` to be a temporary variable
`MakePermanent(A)`	—	Sets `A` to be a non-temporary variable
`Clear(A)`	—	Initializes all elements of `A` with 0
`Initialize(A,r)`	—	Initializes all elements of `A` with `r`
`Row(A,i)`	`VECTOR`	Returns the `i`-th row of `A`
`Col(A,i)`	`VECTOR`	Returns the `i`-th column of `A`
`SetRow(A,i,v)`	—	Sets the `i`-th row of `A` to `v`
`SetCol(A,i,v)`	—	Sets the `i`-th column of `A` to `v`

Table 3.21: Matrix Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`A < B`	True, if each element of `A` is less than the appropriate element in `B`
`A <= B`	same as above, but less or equal
`A > B`	same as above, but greater than
`A >= B`	same as above, but greater or equal

Table 3.22: Matrix Comparisons

3.13 IntervalMatrix.h

This file defines all operations concerning interval matrices. The appropriate data type is called `INTERVAL_MATRIX`. In the following we assume `IA`,`IB`,`IC` to be of type `INTERVAL_MATRIX` and `iv` to be of type `INTERVAL_VECTOR` and `v` to be of type `VECTOR`. The variable `i` denotes an expression of type `INT`. If `X` or `Y` are used, either `INTERVAL_MATRIX` or `MATRIX` types are allowed. `A` or `B` denotes an expression of type `MATRIX`. If we use `x` or `y`, either `INTERVAL_VECTOR` or `VECTOR` is allowed. `z` denotes a variable of either `INTERVAL` or `REAL` and `a` denotes a variable of type `INTERVAL`. We further assume `r` to be of type `REAL`. If not stated otherwise, the result type is `INTERVAL_MATRIX`. The following basic operations are provided:

Operation	Description
`INTERVAL_MATRIX IA;`	Declaration of `IA`. The dimension of `IA` is determined at the first assignment to `IA`.
`INTERVAL_VECTOR IA(20,30);`	Declaration of `IA` as a matrix with 20 rows and 30 columns
`IA = X`	Assignment

continued on next page ...

... continued from previous page

Operation	Description
`IA += X`	Same as `IA = IA + X`
`IA -= X`	Same as `IA = IA - X`
`IA *= z`	Same as `IA = z * IA`
`IA /= z`	Same as `IA = IA / z`
`+IA`	monadic plus
`-IA`	monadic minus
`X + Y`	Interval matrix addition (`X` and `Y` must not be both `MATRIX`)
`X - Y`	Interval matrix subtraction (`X` and `Y` must not be both `MATRIX`)
`z * X`	Multiplication with scalar. The case `MATRIX X` and `REAL z` is not allowed
`X / z`	Division by scalar. The case `MATRIX X` and `REAL z` is not allowed
`X * x`	Matrix vector product (the result type is: `INTERVAL_VECTOR`). The case `MATRIX X` and `VECTOR x` is not allowed
`X * Y`	Matrix product (`X` and `Y` must not be both `MATRIX`)
`IA(i,j)`	denotes the `i`-row and `j`-th column of `IA`

Table 3.23: Interval Matrix Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`RowDimension(IA)`	`INT`	Returns the number of rows in `IA`
`ColDimension(IA)`	`INT`	Returns the number of columns in `IA`
`Resize(IA,i,j)`	—	Discards the old contents of `IA` and resizes it to contain `i` rows and `j` columns
`MakeTemporary(IA)`	—	Sets `IA` to be a temporary variable
`MakePermanent(IA)`	—	Sets `IA` to be a non-temporary variable
`Clear(IA)`	—	Initializes all elements of `IA` with 0
`Initialize(IA,a)`	—	Initializes all elements of `IA` with `a`
`Row(IA,i)`	`INTERVAL_VECTOR`	Returns the `i`-th row of `IA`
`Col(IA,i)`	`INTERVAL_VECTOR`	Returns the `i`-th column of `IA`
`SetRow(IA,i,iv)`	—	Sets the `i`-th row of `IA` to `iv`
`SetCol(IA,i,iv)`	—	Sets the `i`-th column of `IA` to `iv`
`AddBounds(A,B)`	`INTERVAL_MATRIX`	Evaluation of `A + B` yielding bounds for the true result
`SubBounds(A,B)`	`INTERVAL_MATRIX`	Same as above, but for `A - B`
`MulBounds(r,A)`	`INTERVAL_MATRIX`	Same as above, but for `r * A`

continued on next page ...

... continued from previous page

Operation	Return Type	Description
`DivBounds(A,r)`	`INTERVAL_MATRIX`	Same as above, but for `A / r`
`Inf(IA)`	`MATRIX`	Lower bounds of `IA`
`Sup(IA)`	`MATRIX`	Upper bounds of `IA`
`Hull(A)`	`INTERVAL_MATRIX`	Point interval matrix [A, A]
`Hull(X,Y)`	`INTERVAL_MATRIX`	Interval hull of `X` and `Y`
`SymHull(A)`	`INTERVAL`	short form for `Hull(-A,A)`
`Mid(IA)`	`MATRIX`	Midpoint of `IA`
`Diam(IA)`	`MATRIX`	Diameter of `IA`
`Abs(IA)`	`MATRIX`	Absolute value of `IA`
`Intersection(IA,IB,IC)`	`INT`	If `IB` and `IC` intersect, a value of 1 is returned and `IA` contains the intersection. Otherwise 0 is returned and the contents of `IA` are undefined.

Table 3.24: Interval Matrix Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`A <= IA`	True, if `A` is contained in `IA`
`A < IA`	True, if `A` is contained in the interior `IA`
`IA == IB`	True, if `IA` and `IB` are equal
`IA != IB`	True, if `IA` and `IB` are not equal

Table 3.25: Interval Matrix Comparisons

3.14 IntegerMatrix.h

This file defines all operations concerning integer matrices. The appropriate data type is called `INTEGER_MATRIX`. In the following we assume `A`,`B` to be of type `INTEGER_MATRIX` and `v` to be of type `INTEGER_VECTOR`. The variables `i`,`j` denote an expression of type `INT`. The following basic operations are provided:

Operation	Description
`INTEGER_MATRIX A;`	Declaration of `A`. The dimensions of `A` is determined at the first assignment to `A`.
`INTEGER_MATRIX A(20,30);`	Declaration of `A` as an integer matrix with 20 rows and 30 columns
`A = B`	Assignment

continued on next page ...

... continued from previous page

Operation	Description
`A += B`	Same as `A = A + B`
`A -= B`	Same as `A = A - B`
`A *= i`	Same as `A = i * A`
`A /= i`	Same as `A = A / i`
`A + B`	Matrix addition
`A - B`	Matrix subtraction
`i * A`	Multiplication with scalar
`A / i`	Division by scalar
`A * v`	Matrix vector product (the result type is: `INTEGER_VECTOR`)
`A(i,j)`	denotes the `i`-th row and `j`-th column of `A`

Table 3.26: Integer Matrix Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`RowDimension(A)`	`INT`	Returns the number of rows in `A`
`ColDimension(A)`	`INT`	Returns the number of columns in `A`
`Resize(A,i,j)`	—	Discards the old contents of `A` and resizes it to contain `i` rows and `j` columns
`MakeTemporary(A)`	—	Sets `A` to be a temporary variable
`MakePermanent(A)`	—	Sets `A` to be a non-temporary variable
`Clear(A)`	—	Initializes all elements of `A` with 0
`Initialize(A,i)`	—	Initializes all elements of `A` with `i`

Table 3.27: Integer Matrix Functions

3.15 Functions.h

This file defines some common standard functions for real and interval arguments. Additionally the following constants are defined:

Name of Constant	Description
`Constant::Pi`	$\pi = 3.14159\ldots$
`Constant::TwoPi`	2π
`Constant::PiHalf`	$\pi/2$
`Constant::PiQuarter`	$\pi/4$
`Constant::e`	$e = 2.71828\ldots$

continued on next page ...

... continued from previous page

Name of Constant	Description
`Constant::Sqrt2`	$\sqrt{2}$
`Constant::InvSqrt2`	$\sqrt{0.5}$
`Constant::Ln10`	$\log_e 10$

Table 3.28: Some Common Constants

Due to the automatic initialization of the constants given above, the object file belonging to `Functions.h` must be linked to the executable code. This is generally done automatically. If not, define in your program a global variable as e.g.

```
Constant ConstantAutomaticInit;
```

to initialize the constants. The standard functions provided are (where `x` and `y` are of type `REAL` or `INTERVAL` and the result is of the same type as `x` and `y`; further, let `i` be of type `INT`):

Function	Description
`Sin(x)`	$\sin \mathtt{x}$
`Cos(x)`	$\cos \mathtt{x}$
`Tan(x)`	$\tan \mathtt{x}$
`ArcSin(x)`	$\arcsin \mathtt{x}$
`ArcCos(x)`	$\arccos \mathtt{x}$
`ArcTan(x)`	$\arctan \mathtt{x}$
`Sinh(x)`	$\sinh \mathtt{x}$
`Cosh(x)`	$\cosh \mathtt{x}$
`Tanh(x)`	$\tanh \mathtt{x}$
`ArSinh(x)`	$\operatorname{arsinh} \mathtt{x}$
`ArCosh(x)`	$\operatorname{arcosh} \mathtt{x}$
`ArTanh(x)`	$\operatorname{artanh} \mathtt{x}$
`Exp(x)`	$e^{\mathtt{x}}$
`Log(x)`	$\log_e \mathtt{x}$
`Log10(x)`	$\log_{10} \mathtt{x}$
`Sqr(x)`	$\mathtt{x}^2$
`Sqrt(x)`	$\sqrt{\mathtt{x}}$
`Root(x,i)`	$\sqrt[i]{\mathtt{x}}$
`Power(x,i)`	$\mathtt{x}^{\mathtt{i}}$
`Power(x,y)`	$\mathtt{x}^{\mathtt{y}}$

Table 3.29: Standard Functions

Additionally, the following functions for `REAL` arguments are provided:

Function	Description
`Abs(r)`	$\|\mathtt{r}\|$
`Min(r,s)`	$\min\{\mathtt{r},\mathtt{s}\}$
`Max(r,s)`	$\max\{\mathtt{r},\mathtt{s}\}$

Table 3.30: Additional Real Functions

For interval arguments `ix` the function `IAbs(ix)` is defined as:

$$\mathtt{IAbs(x)} := \{\, |x| \mid x \in \mathtt{x} \,\}$$

3.16 Utilities.h

This file defines some commonly used functions. We assume `A` to be of type `MATRIX`, `iv` be of type `INTERVAL_VECTOR` and `i` of type `INT`. Then the following functions are provided:

Operation	Return Type	Description
`Inverse(A)`	`MATRIX`	Returns an approximation of $\mathtt{A}^{-1}$
`Transpose(A)`	`MATRIX`	Returns $\mathtt{A}^T$
`Id(i)`	`MATRIX`	Returns an identity matrix of dimension $\mathtt{i} \times \mathtt{i}$
`Lower(iv,i)`	`INTERVAL_VECTOR`	Splits `iv` normal to the `i`-th direction and returns the lower half
`Upper(iv,i)`	`INTERVAL_VECTOR`	Splits `iv` normal to the `i`-th direction and returns the upper half

Table 3.31: Additional Functions

3.17 LSS.h

This file defines (interval) linear system solvers computing an enclosure of the solution set. These methods are due to S.M. Rump [20, 21]. We assume `A` to be of type `MATRIX`, `IA` to be of type `INTERVAL_MATRIX`, `iv` be of type `INTERVAL_VECTOR`, `v` to be of type `VECTOR` and `i` of type `INT`. The return type is in both cases `INTERVAL_VECTOR`. The following functions are provided:

Operation	Description
`LSS(A,v,i)`	Computes an enclosure of the solution set x of the linear system `A`x = `v`. If the enclosure is possible, it is returned and `i` is set to 1. Otherwise `i` is set to 0 and the return value is undefined.
`ILSS(IA,iv,i)`	Computes an enclosure of the solution set x of the interval linear system `IA`x = `iv`. If the enclosure is possible, it is returned and `i` is set to 1. Otherwise `i` is set to 0 and the return value is undefined.

Table 3.32: Linear System Solvers

3.18 LongReal.h

This file defines all operations concerning multiple precision real numbers (in the following refered as long real numbers). All operations are based on the long real package from D. Husung [3].

We assume that `a`,`b` are of type `LONGREAL` and `v`,`w` are of type `VECTOR`. With `x` and `y` we denote expressions either of type `LONGREAL` or `REAL`. By `s` we denote a string expression and by `n` an `INT` expression. If not explicitly mentioned, the return type of the routines below is `LONGREAL`.

The current working precision (i.e. the initial precision of new declared variables and the precision of any calculated results) is controlled by the following routines:

Operation	Return Type	Description
`SetDigits(n)`	—	Sets the current working precision to approx. `n` decimal digits
`SetBaseDigits(n)`	—	Sets the current working precision to `n` base 2^{16} digits
`GetDigits()`	`INT`	Returns the current working precision in approx. decimal digits
`GetBaseDigits()`	`INT`	Returns the current working precision in base 2^{16} digits

Table 3.33: Precision Control

The following basic operations are provided:

Operation	Description
`LONGREAL a;`	Declaration of `a`
`LONGREAL a("1.234");`	Declaration of `a` and initialization with the constant 1.234. In this case, the length of the string constant determines the precision of `a` in decimal digits
`LONGREAL a("1.234",n);`	Same as above, but the precision of `a` is given by `n` (in decimal digits)
`a = x`	Assignment
`a += x`	Same as `a = a + x`
`a -= x`	Same as `a = a - x`
`a *= x`	Same as `a = a * x`
`a /= x`	Same as `a = a / x`
`a++`	successor of `a`
`a--`	predecessor of `a`
`-a`	monadic minus
`x + y`	Long real addition (`x` and `y` must not both be `REAL`)
`x - y`	Long real subtraction (`x` and `y` must not both be `REAL`)
`x * y`	Long real multiplication (`x` and `y` must not both be `REAL`)
`x / y`	Long real division (`x` and `y` must not both be `REAL`)

Table 3.34: Multiple Precision Arithmetic Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`MakeTemporary(a)`	—	Sets `a` to be a temporary variable
`MakePermanent(a)`	—	Sets `a` to be a non-temporary variable
`RoundToReal(a)`	`REAL`	Returns the value of `a` rounded to the nearest `REAL` value
`RoundToReal(a,LR_RND_DOWN)`	`REAL`	same as above, but rounded downwards
`RoundToReal(a,LR_RND_UP)`	`REAL`	same as above, but rounded upwards
`Accumulate(a,v,w)`	—	Calculates the accurate scalar product of `v` and `w` and adds it to `a`. Important: The precision of `a` is unchanged!
`StringToLongReal(s)`	`LONGREAL`	Converts the string `s` to a long real number

Table 3.35: Multiple Precision Arithmetic Functions

The following comparison operators are provided, returning a value of 1 if the comparison has a true result and 0 otherwise.

Comparison	Description
`a < b`	True, if `a` is less than `b`
`a <= b`	True, if `a` is less or equal `b`
`a > b`	True, if `a` is greater than `b`
`a >= b`	True, if `a` is greater or equal `b`

Table 3.36: Multiple Precision Comparisons

The input and output of long real numbers is written as for all other data types. Currently, long real numbers are always output using an exponential format. Only the number of digits and the output rounding mode can be controlled. The following functions are used to control the output format for long real numbers and for the long intervals, which are described in the next section:

Operation	Return Type	Description
`SetOutFracDigits(n)`	—	Sets the number of fractional digits on output
`SetOutIntDigits(n)`	—	Sets the maximum number of integer digits. If the true number of integer digits is larger, the exponential format will be used. (Only the output of long intervals is affected)
`SetOutRndMode(n)`	—	Sets the output rounding mode for long real numbers to `n`, where the following values are permitted: **0** set rounding mode to nearest **1** set rounding mode to downwards **2** set rounding mode to upwards **3** set rounding mode to chop
`GetOutFracDigits()`	`INT`	Returns the current number of fractional digits
`GetOutIntDigits()`	`INT`	Returns the current number of integer digits
`GetOutRndMode()`	`INT`	Returns the current output rounding mode

Table 3.37: Input/Output Control Functions

3.19 LongInterval.h

This file defines all operations concerning multiple precision intervals (in the following refered as long intervals). It is based on the long interval arithmetic from [12]. We assume that `a`,`b` are of type `LONGINTERVAL` and `v`,`w` are of type `VECTOR` or `INTERVAL_VECTOR`. An expression of type `LONGREAL` is denoted by `p` or `q`. With `x` and `y` we denote expressions either of type `LONGINTERVAL`, `INTERVAL`, `LONGREAL` or `REAL`. Note that if both operands are allowed to be one of the 4 types, at least one operand must be of type `LONGINTERVAL`. By `s` we denote a string expression and by `n` an `INT` expression. If not explicitly mentioned, the return type of the routines below is `LONGINTERVAL`.

The current working precision is controlled by the same commands used for the long real numbers.

The following basic operations are provided:

Operation	Description
`LONGINTERVAL a;`	Declaration of `a`
`LONGINTERVAL a("[1.2,1.3]");`	Declaration of `a` and initialization with the constant interval $[1.2, 1.3]$.
`LONGINTERVAL a("[1.2,1.3]",n);`	Same as above, but the precision of `a` is given by `n` (in decimal digits)
`a = x`	Assignment
`a += x`	Same as `a = a + x`
`a -= x`	Same as `a = a - x`
`a *= x`	Same as `a = a * x`
`a /= x`	Same as `a = a / x`
`-a`	monadic minus
`x + y`	Long interval addition
`x - y`	Long interval subtraction
`x * y`	Long interval multiplication
`x / y`	Long interval division

Table 3.38: Multiple Precision Interval Arithmetic Operations

Additionally, the following functions are provided:

Operation	Return Type	Description
`Hull(p)`	`LONGINTERVAL`	An enclosure of `p` (cf. section 3.2)
`Hull(p,q)`	`LONGINTERVAL`	Convex hull of `p` and `q`
`Inf(a)`	`LONGREAL`	Lower bound of `a`
`Sup(a)`	`LONGREAL`	Upper bound of `a`

continued on next page ...

... *continued from previous page*

Operation	Return Type	Description
`Diam(a)`	`LONGREAL`	Diameter of `a`
`Mid(a)`	`LONGREAL`	Midpoint of `a`
`MakeTemporary(a)`	—	Sets `a` to be a temporary variable
`MakePermanent(a)`	—	Sets `a` to be a non-temporary variable
`RoundToInterval(a)`	`INTERVAL`	Returns the value of `a` rounded to the smallest enclosing `INTERVAL` value
`Accumulate(a,v,w)`	—	Calculates the accurate scalar product of `v` and `w` and adds it to `a`. Important: The precision of `a` is unchanged!
`Enclosure(s)`	`INTERVAL`	Returns an interval which encloses the interval given by the string `s`
`LongIntervalEnclosure(s)`	`LONGINTERVAL`	Same as above, but returns a long interval

Table 3.39: Multiple Precision Interval Arithmetic Functions

The output format of long intervals is controlled by the same commands used for the long real numbers. Additionally, the appearance of the output can be changed by the command

```
SetOutOptions(n)
```

where `n` is one of the following values.

An optionally added `LI_STR_SINGLE` denotes that all spaces used for tabbing are removed.

Value	Description
`LI_FSTR_INT`	If the interval bounds are integer values and the number of decimal digits is less than the value returned by `GetIntDigits()`, the bounds are written to output without any fractional digits. Otherwise, the next format is tried.
`LI_FSTR_FRACCOM`	If there are common leading digits for the interval bounds, they are extracted and displayed only once. E.g. the interval $[1.234, 1.235]$ is displayed as `1.23[4,5]`. Otherwise, the next format is tried.
`LI_FSTR_FRACSEP`	If the interval bounds are not too large (less than `GetIntDigits()` integer digits), they are displayed as two floating point numbers. Otherwise, the exponential format is used.

Table 3.40: Output Options

The default output format is `(LI_FSTR_INT | LI_STR_SINGLE)`.

3.20 HighPrec.h

This file provides several vector/matrix operations using an accurate scalar product instead of a normal floating point or interval accumulator. We assume `v`,`w` to be of type `VECTOR` and `A`,`B` be of type `MATRIX`. If `x` or `y` are used, either `INTERVAL_VECTOR` or `VECTOR` is allowed. If we use `X` or `Y`, either `INTERVAL_MATRIX` or `MATRIX` types are allowed.

The following subroutines are provided:

Operation	Return Type	Description
`HighPrecMul(v,w)`	`REAL`	`v` · `w`
`HighPrecMul(A,v)`	`VECTOR`	`A` · `v`
`HighPrecMul(A,B)`	`MATRIX`	`A` · `B`
`HighPrecMulBounds(v,w)`	`INTERVAL`	An enclosure of `v` · `w`
`HighPrecMulBounds(A,v)`	`INTERVAL_VECTOR`	An enclosure of `A` · `v`
`HighPrecMulBounds(A,B)`	`INTERVAL_MATRIX`	An enclosure of `A` · `B`
`HighPrecMul(x,y)`	`INTERVAL`	`x` · `y` (`x` and `y` must not be both `VECTOR`.)
`HighPrecMul(X,x)`	`INTERVAL_VECTOR`	`X` · `x` (The case `MATRIX X` and

continued on next page . . .

... continued from previous page

Operation	Return Type	Description
		`VECTOR x` is not allowed)
`HighPrecMul(X,Y)`	`INTERVAL_MATRIX`	`X` · `Y` (`X` and `Y` must not be both `MATRIX`.)

Table 3.41: High Precision Subroutines

With `IdMinusHighPrecMul(X,Y)`, high accuracy enclosures of $I - X \cdot Y$ are computed.

3.21 Adapting Real Operations to a Specific Hardware

All vector and matrix real operations which may be adapted to a specific hardware supporting special features of this machine (e.g. vector computers, cache strategies) are collected in the file `RealOp.c` with the headers in `RealOp.h`. This file is compiled with a C compiler, not with a C++ compiler, because the C compiler generally generates slightly better code.

3.22 A Quick Introduction To C

In this section we give a very short introduction to the most commonly used language constructs of C, which are needed for numerical programming. An experienced reader should excuse the simplifications we make.

A PROFIL program is structured as follows:

- inclusion of definitions. For almost any data type or operation you need, you have to include the appropriate definitions into your program. For example to include the real vector operations, precede your program with

  ```
  #include "Vector.h"
  ```

- global data definitions (the syntax is the same as the data definition in the subroutines).
- subroutines
- main program (syntactically a subroutine, but with the special name “main”).

Comments are either enclosed in `/* ... */` or prepended by `//`. The latter starting a comment until the end of the current line.

A subroutine is structured as follows:

- header (consisting of return type, subroutine name, and parameter list)
- body (statements enclosed in curly braces)

3.22.1 The Subroutine Header

You can use any of PROFIL data types as return types or as parameter types. If you use the special name “`VOID`” as return type, you have a normal subroutine without any return value. Otherwise you have a function with a return value of the specified data type.

The parameters in the parameter list are separated by commas and consist of parameter data type and parameter name. If a “`&`” is between the data type and the name, the parameter is a reference parameter. The whole parameter list is enclosed in parentheses which are neccessary and must not be omitted even if the parameter list is empty.

Example: A header for a function computing the norm of an vector:

```
REAL Norm(VECTOR & x)
```

3.22.2 Statements

A statement can be either a single statement (terminated with a semicolon) or a group of statements enclosed in curly braces. Please note that all declarations which occur between curly braces are local to this block.

A single statement is either a declaration or an executable statement. Every variable must be declared before it is used, but you may mix declarations and executable statements if you like to. There is no need to collect all declarations at the beginning of a subroutine, but it is a good practice to do so.

A declaration statement consists of the data type and a list a variable names separated by commas. For vectors and matrices optional arguments are supported to define the size of these structures.

Examples:

```
REAL a, b;              // defines 2 variables "a" and "b" of type REAL
INTERVAL_VECTOR v(10);  // defines an interval vector v with 10 elements
                        // v(1), v(2), ..., v(10)
MATRIX M(10,20);        // defines a real matrix M with 10 rows and
                        // 20 columns: m(1,1), ..., m(10,20)
```

Executable Statements

Assignments: Syntax: *variable* `=` *expression*

The possible functions and operators depend on the expression type and are listed below. Vector and matrix element access is done with the index (indices) following the variable in parentheses.

Example: `v(1) = m(i+1,j-1)`

If clause: Syntax: `if (`*expression*`)` *statement*

The expression must be of type `BOOL` or `INT`.

Example: `if (i == j) v(1) = 0.0;`

For loop: Syntax: `for (`*initialization*`;` *termination*`;` *incrementation*`)` *statement*

Examples:

1. increment of one (e.g. for cases like $i = a, \ldots, b$):
   `for (i = a; i <= b; i++)`
2. increment of s:
   `for (i = a; i <= b; i += s)`
3. decrement of one:
   `for (i = a; i >= b; i--)`

While loop: Syntax: `while (`*expression*`)` *statement*

Example: `while (i <= j) { j = j + 1; v(j) = j; }`

`return` **statement:** The return statement followed by an expression is used to leave a subroutine and set the return value of a function. Example:

```
REAL Square(REAL x) { return x*x; }
```

Chapter 4

Testmatrices

A set of test matrices is provided with the package `TestMatrices`. Those test matrices will mainly 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` $\times$ `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

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

The general structure of the matrix is:

$$A = \begin{pmatrix} n & n-1 & n-2 & \cdots & 2 & 1 \\ n-1 & n-1 & n-2 & \cdots & 2 & 1 \\ n-2 & n-2 & n-2 & \cdots & 2 & 1 \\ \multicolumn{6}{c}{\dotfill} \\ 2 & 2 & 2 & \cdots & 2 & 1 \\ 1 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix}.$$

`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:

$$A = \begin{pmatrix} 1 & 2 & 3 & \ldots & n-1 & n \\ 2 & 2 & 3 & \ldots & n-1 & n \\ 3 & 3 & 3 & \ldots & n-1 & n \\ \multicolumn{6}{c}{\dotfill} \\ n-1 & n-1 & n-1 & \ldots & n-1 & n \\ n & n & n & \ldots & n & n \end{pmatrix}.$$

`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 $n \geq 5$:

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

`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 $a_1, a_2, \ldots, a_{n-1}$ in vector U.

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

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

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 $A = (a_{ij})$:

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

It holds $0 < x_i \leq 1$, $0 < y_i \leq 1$ for $i = 1, 2, \dots, n$.
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, $\dots$, x_n in vector x, and y_1, y_2, $\dots$, y_n in vector y.

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

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

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

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

`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 $[\texttt{lb}, \texttt{ub}]$ 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 $[0, 1]$.

Chapter 5

Local Optimization Methods

Some local optimization methods are provided with the package `LocalOpt`.

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 [17]. It is a robust but rather slow method in terms of required function evaluations. The implementation has been taken from [18] 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.

Chapter 6

General Lists

The package `Lists` contains support for 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;
```

Chapter 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.

This functionality is provided by the package `AutoDiff`. The automatic differentiation is included in the non-linear function package (see 8). The information about the implementation of the automatic differentiation is given below and should be only necessary for advanced users.

In general, the non-linear function package will suffice for mostly all needs.

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. [19].

Chapter 8

Non-Linear Function Support

The non-linear function package `NLF` provides support for functions of the type $f : \mathbb{R}^n \to \mathbb{R}$ as well as for functions like $f : \mathbb{R}^n \to \mathbb{R}^m$. Derivatives are supported up to order 2 (Hessian). Function and derivative values are available as floating point (approximation of the true value) or as an interval enclosing the true value. These function types where chosen to meet the requirements imposed from global optimization tasks.

It is up to the user to decide which derivatives are computed automatically by using automatic forward differentiation and which are implemented by hand (e.g. to speed up the computation of a Hessian or to implement a better inclusion function for the derivatives). The interface to calling routines which use the function or its derivatives is always the same. Thus the true implementation of the function is completely hidden.

At least, only one function implementation is necessary to get the full set of derivatives and inclusion functions needed for many purposes as e.g. global optimization.

The package introduces the new data type `FUNCTION` which contains the complete information about the function.

If $f : \mathbb{R}^n \to \mathbb{R}$ denotes a defined and implemented function of type `FUNCTION`, the following routines provide the necessary parts:

`Function(`f`,`x`)` returns $f(x)$.

`Gradient(`f`,`x`)` returns $\nabla f(x)$.

`Hessian(`f`,`x`)` returns $\nabla^2 f(x)$.

for vector valued functions, i.e. $f : \mathbb{R}^n \to \mathbb{R}^m$, the following routines are available:

`VectorFunction(`f`,`x`)` returns $f(x) = \big(f_1(x), \dots , f_m(x)\big)$.

`Gradient(`f`,`x`,`k`)` returns $\frac{\partial f_k(x)}{\partial x}$.

`Hessian(`f`,`x`,`k`)` returns $\frac{\partial^2 f_k(x)}{\partial x \partial x}$.

`Jacobian(`f`,`x`)` returns $J_f(x)$ for functions $f : \mathbb{R}^n \to \mathbb{R}^n$

The distinction between "`Function`" and "`VectorFunction`" is due to the fact that C++ does not permit subroutines with same argument lists but different return types.

The routines above take care to use the most appropriate implementation of the required type. For example, if a floating-point Hessian is required, a true floating point Hessian implementation is checked first. If it does not exist, the midpoint value of an interval Hessian implementation is tried next. If this is also unavailable, floating point and interval implementations of the function using automatic differentiation are tried to get the required value. If everything fails, an error message is emitted and the program is aborted.

This search imposes a little overhead on every function and derivative call, but that could be neglected for almost all problems.

Additionally, the number of function and derivative calls are counted to provide statistic information.

8.1 Defining a Function

Let us now consider how to use the non-linear function package we discussed above. As an example, we take the following function "f" with $f : \mathbb{R}^2 \to \mathbb{R}^2$:

$$f(x) = \begin{pmatrix} f_1(x) \\ f_2(x) \end{pmatrix} = \begin{pmatrix} 1 - 2x_2 + \frac{1}{20}\sin(4\pi x_2) - x_1 \\ x_2 - \frac{1}{2}\sin(2\pi x_1) \end{pmatrix}$$

The most simple way of implementing this function is by using automatic differentiation:

```
static INTERVAL_AUTODIFF f_Impl (CONST INTERVAL_AUTODIFF & x, int n)
{
  switch (n) {
    case 1:
      return 1.0 - 2.0 * x(2) +
             Sin (4.0 * Constant::Pi * x(2)) / 20.0 - x(1);
    case 2:
      return x(2) - 0.5 * Sin (2.0 * Constant::Pi * x(1));
  }
}
```

The parameter x denotes the argument and n denotes the component of "f" to be evaluated. Constant::Pi denotes the value $\pi = 3.14159\ldots$, which is a predefined constant in PROFIL.

To define "f" to be of type FUNCTION, with an argument dimension of 2, a function value dimension of 2, and an implementation which uses automatic differentiation, the following simple line is used:

```
FUNCTION f (2, 2, f_Impl);
```

After that, "f" could be used as a new function: For example, we obtain the Jacobian matrix of "f" at x by Jacobian (f, x). If x is of type VECTOR, we get an approximation of the Jacobian matrix at x. To get an enclosure of the Jacobian, the parameter x just needs to be an interval vector. Therefore, the enclosure of the Jacobian of "f" at point x, where x is again of type VECTOR, is simply obtained by Jacobian (f, Hull (x)).

8.2 An Example: Newton's Method

A simple application example (the file TestNLF.C contains the complete example) should conclude the discussion about the non-linear function package. We take Newton's method for approximating a root of a function as an example. The algorithm is:

Approximation of a root of f starting at x:
repeat
$x_{n+1} = x_n - J(f, x) \cdot f(x)$
until $\|x_{n+1} - x_n\| \leq \epsilon$

where ϵ is the error bound and $J(f, x)$ denotes the Jacobian matrix of f at x.

Using $\epsilon = 10^{-6}$, the implementation of the algorithm above is as follows:

```
VECTOR Root (FUNCTION & f, VECTOR & x)
{
  VECTOR xOld;

  do {
    xOld = x;
    x = x - Inverse (Jacobian (f, x)) * VectorFunction (f, x);
  } while (Norm (x - xOld) > 1e-6);
  return x;
}
```

Starting at $x_s = (0, 0)$, we could easily get an approximation of a root of the function "f" we implemented before by using:

```
int main ()
{
  VECTOR xs(2);
  xs(1) = 0.0;
  xs(2) = 0.0;
  cout << "Approximation of a root of f = "
       << Root (f, xs)
       << endl;
  return 0;
}
```

Chapter 9

Global Optimization

This documentation describes in a compact way the usage of the global minimization algorithm whose mathematical background together with many applications is described in detail in [9, 11, 6, 5, 8, 4, 7, 10].

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.

The global minimization algorithm is provided by the package `GlobalOpt`.

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

9.1 The Method

In order to keep this description small, we assume that the reader is familiar with the basics of interval mathematics as well as with the principles behind the new method (see, e.g. [9, 11, 6, 5, 8, 4, 7, 10]). The implementation combines the derivative-free method together with the version using derivatives. It is up to the user which version is used.

The global optimization method uses the non-linear function package to access the function. Thus it is up to the user to implement derivatives or to use the automatic differentiation feature.

9.1.1 Description of Modules

The PROFIL/BIAS implementation of the global minimization method is divided into several modules which also might be helpful for other purposes. The modules are:

`AppList:` Implementation of lists containing approximations of local and/or global minima along with additional information. This list is called "A" in the mathematical description of the method. The list data type is called `APPROXIMATIONLIST`.

`VecList:` Implementation of lists containing boxes to be processed later. This list is called "S" in the mathematical description of the method. The list data type is called `SOLUTIONLIST`.

`Expand:` The expansion strategy.

`UnconstrainedOpt:` The main driver for the method. As local descent method, the method of Brent taken from the PROFIL/BIAS local optimization package is used. The call of the descent method is hardcoded into the file `UnconstrainedOpt.C`, but can be changed, if another method is desired.

`VecUtils:` Some necessary utility subroutines.

In the following, we describe the two versions of the interface to the global minimization method in detail, starting with the simple driver.

9.1.2 The Main Driver

The main driver of the global minimization method consists of 4 subroutines:

`StartUnconstrainedOptimization` This subroutine is called when initiating the minimization method. It sets up all internal data structures and calls the next subroutine.

`UnconstrainedOptimization` This subroutine assumes that all internal data structures have been set up and performs the demanded number of iteration steps. After it has finished, additional iterations can be achieved by calling this subroutine again (perhaps with modified parameters).

`CleanUpLists` As a finishing step, after all requested iterations have been performed, the lists can be cleaned from unnecessary elements by calling this subroutine.

`UpdateLowerBound` A new lower bound is computed from the solution list.

In the following, we will describe the parameters of those 4 subroutines in detail:

```
VOID StartUnconstrainedOptimization (
  SOLUTIONLIST & SolutionList,
  APPROXIMATIONLIST & ApproximationList,
  INT Iterations, INT BranchLevels, REAL Eps,
  REAL RaiseFactor, REAL HeuristicExpandFactor,
  REAL & LowerBound, REAL & UpperBound,
```

```
  INTERVAL_VECTOR & TheDomain,
  FUNCTION & TheFunction,
  BOOL UseDerivatives)
```

`SolutionList`: Contains all unprocessed boxed on output. The list must be empty on input.

`ApproximationList`: Contains all computed approximations together with additional information (e.g. enclosures of the minima). The list must be empty on input.

`Iterations`: The maximal number of iterations. This parameter is called n_{it} in the mathematical description of the method.

`BranchLevels`: The maximal number of bisection performed in each coordinate direction in each iteration. This parameter is called n_{d} in the mathematical description of the method.

`Eps`: Relative tolerance for the local descent method.

`RaiseFactor`: Parameter of the method. Larger values will increase the number of calls of the local descent method.

`HeuristicExpandFactor`: Parameter of the method. Larger values will enlarge the heuristic area around computed local minima. Larger heuristic areas might decrease the number of calls of the local descent method.

`LowerBound`: Contains the lower bound for the function value on output.

`UpperBound`: Contains the upper bound for the function value on output.

`TheDomain`: The domain of the function.

`TheFunction`: The function to be minimized.

`UseDerivatives`: Must be set to `TRUE`, if derivatives should be used. This parameter has a default value of `FALSE`.

To perform further steps after the initial iterations, the following subroutine is used:

```
VOID UnconstrainedOptimization (
  SOLUTIONLIST & SolutionList,
  APPROXIMATIONLIST & ApproximationList,
  INT Iterations, INT BranchLevels, REAL Eps,
  REAL RaiseFactor, REAL HeuristicExpandFactor,
  REAL & LowerBound, REAL & UpperBound,
  BOOL UseDerivatives);
```

Any parameters not described here have the same meaning as in `StartUnconstained-Optimization`.

`SolutionList`: Contains all unprocessed boxed on input and returns the new list on output.

`ApproximationList`: Contains all computed approximations together with additional information (e.g. enclosures of the minima) on input and returns the new list on output.

`LowerBound`: Contains a lower bound for the function value on input and yields the new computed lower bound on output.

`UpperBound`: Contains an upper bound for the function value on input and yields the new computed lower bound on output.

`UseDerivatives`: Must be set to `TRUE`, if derivatives should be used. This parameter has a default value of `FALSE`.

To clean up the lists, the following subroutine could be used:

```
VOID CleanUpLists (
  SOLUTIONLIST & SolutionList,
  APPROXIMATIONLIST & ApproximationList,
  REAL Eps, REAL LowerBound, REAL UpperBound);
```

`SolutionList`: Contains all unprocessed boxed on input and returns the new list on output.

`ApproximationList`: Contains all computed approximations together with additional information (e.g. enclosures of the minima) on input and returns the new list on output.

`Eps`: Relative tolerance for the local descent method.

`LowerBound`: Contains a lower bound for the function value on input.

`UpperBound`: Contains an upper bound for the function value on input.

The following subroutine updates the lower bound by using the solution list:

```
VOID UpdateLowerBound (
  SOLUTIONLIST & SolutionList,
  REAL & LowerBound);
```

`SolutionList`: Contains all unprocessed boxed on input and returns the new list on output.

`LowerBound`: Contains the current lower bound on input and the updated lower bound on output.

9.2 Sample Programs

The program TestOpt.C contains a sample program as an application of the subroutines discussed above.

It is listed below for information purposes:

```
/************************************************************************
 *
 * Simple Test Program for Minimization Algorithm
 * ----------------------------------------------
 *
 * Copyright (c) 1997 Olaf Knueppel
 *
 * $Id: TestSimpleOpt.C,v 1.1 1997/09/23 11:52:10 knueppel Exp $
 *
 ************************************************************************/

#include <GlobalOpt/UnconstrainedOpt.h>
#include <Functions.h>

//
// The Branin function is used as test function.
// This function has 5 global minima.
//

INTERVAL_AUTODIFF fTestFunction (CONST INTERVAL_AUTODIFF & x)
{
  return Sqr (1.0 - 2.0 * x(2) + 0.05 * Sin (4.0 * Constant::Pi * x(2)) - x(1))
       + Sqr (x(2) - 0.5 * Sin (2.0 * Constant::Pi*x(1)));
}

FUNCTION TestFunction (2, fTestFunction);

VOID SetTestDomain (INTERVAL_VECTOR & x)
// The domain of the test function
{
  Resize (x, 2);
  x(1) = Hull (-10.0, 10.0);
  x(2) = Hull (-10.0, 10.0);
}


INT main()
{
  INT Iterations, BranchLevels;
  SOLUTIONLIST SolutionList;
```

```
  APPROXIMATIONLIST ApproximationList;
  INTERVAL_VECTOR TestDomain;
  INTERVAL ValueBounds;
  REAL LowerBound, UpperBound;

  SetTestDomain (TestDomain);

  cout << "Computing all global minima of the Branin function" << endl << endl;

  cout << "To get all minima, use e.g. nit = 6, nd = 3" << endl << endl;

  cout << "nit = "; cin >> Iterations;
  cout << "nd  = "; cin >> BranchLevels;

  StartUnconstrainedOptimization (SolutionList, ApproximationList,
                                  Iterations, BranchLevels, 1e-6,
                                  0.2, 0.2,
                                  LowerBound, UpperBound, TestDomain,
                                  TestFunction, TRUE);

  CleanUpLists (SolutionList, ApproximationList, 1e-6, LowerBound, UpperBound);
  ValueBounds = Hull (LowerBound, UpperBound);

  cout << "f min in " << ValueBounds << endl;
  cout << "Solution List:" << endl << SolutionList << endl;
  cout << "Approximation List:" << endl << ApproximationList << endl;
  cout << "Calls needed:" << endl;
  cout << "  Function calls (Real,Interval): "
       << TestFunction.RealFunctionCalls << ','
       << TestFunction.IntervalFunctionCalls << endl;
  cout << "  Gradient calls (Real,Interval): "
       << TestFunction.RealGradientCalls << ','
       << TestFunction.IntervalGradientCalls << endl;
  cout << "   Hessian calls (Real,Interval): "
       << TestFunction.RealHessianCalls << ','
       << TestFunction.IntervalHessianCalls << endl;
  return 0;
}
```

Chapter 10

Miscellaneous Functions

The Package `Misc` 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. They are all defined in the file `MiscFunctions.h`. 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 long`	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.

Bibliography

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

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

[3] HUSUNG, D. Eine Langzahlarithmetik in C. TU Hamburg-Harburg, Technische Informatik III, 1989.

[4] JANSSON, C. A global minimization method: The one-dimensional case. Bericht 91.2, TU Hamburg-Harburg, Technische Informatik III, Nov. 1991.

[5] JANSSON, C. A global optimization method using interval arithmetic. In *Computer Arithmetic and Enclosure Methods.* Elsevier Science Publishers B.V., North-Holland, 1992, pp. 259–268.

[6] JANSSON, C. On self-validating methods for optimization problems. In *Topics in Validated Computations* (Amsterdam, 1994), J. Herzberger, Ed., Studies in Computational Mathematics, North Holland, To appear.

[7] JANSSON, C., AND KNÜPPEL, O. A global minimization method: The multi-dimensional case. Bericht 92.1, TU Hamburg-Harburg, Technische Informatik III, Jan. 1992.

[8] JANSSON, C., AND KNÜPPEL, O. A minimization method for systems with singular value constraints. In *Operations Research '92.* Physica-Verlag, Heidelberg, 1992, pp. 190–192.

[9] JANSSON, C., AND KNÜPPEL, O. Eine intervallanalytische Methode für globale Optimierungsprobleme. *Zeitschrift für angewandte Mathematik und Mechanik 73*, 6 (1993), T741–T743.

[10] JANSSON, C., AND KNÜPPEL, O. Numerical results for a self-validating global optimization method. Bericht 94.1, TU Hamburg-Harburg, Technische Informatik III, Feb. 1994.

[11] JANSSON, C., AND KNÜPPEL, O. A branch and bound algorithm for bound constrained optimization problems without derivatives. *Journal of Global Optimization 7* (1995), 297–331.

[12] Knüppel, O. Implementierung einer dynamischen Langzahlintervallarithmetik. Master's thesis, Technische Informatik III, Technische Universität Hamburg-Harburg, 1990.

[13] Knüppel, O. 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.

[14] Knüppel, O. A multiple precision arithmetic for PROFIL. Bericht 93.6 des Forschungsschwerpunktes Informations- und Kommunikationstechnik der TU Hamburg-Harburg, TU Hamburg-Harburg, Technische Informatik III, Nov. 1993.

[15] Knüppel, O. 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.

[16] Knüppel, O., and Simenec, T. PROFIL/BIAS extensions. Bericht 93.5 des Forschungsschwerpunktes Informations- und Kommunikationstechnik der TU Hamburg-Harburg, TU Hamburg-Harburg, Technische Informatik III, Nov. 1993.

[17] Nelder, J. A., and Mead, R. A simplex method for function minimization. *Computer Journal 7* (1965), 308–313.

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

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

[20] Rump, S. M. *Kleine Fehlerschranken bei Matrixproblemen.* Dissertation, Inst. f. Angew. Math., Universität Karlsruhe, 1980.

[21] Rump, S. M. Solving algebraic problems with high accuracy. In *A New Approach to Scientific Computation.* Academic Press, New York, 1983, pp. 51–120.

In dieser Reihe sind bisher erschienen: