TECHNISCHE UNIVERSITÄT
TUHH
HAMBURG-HARBURG

Berichte des
Forschungsschwerpunktes
Informations- und
Kommunikationstechnik

Olaf Knüppel

PROFIL — Programmer's Runtime Optimized Fast Interval Library

Juli 1993

Bericht 93.4

Olaf Knüppel: Juli 1993

PROFIL — Programmer's Runtime Optimized Fast Interval Library

Bericht 93.4

IMPORTANT NOTE:

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

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

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

This report is available
via anonymous ftp from:
`ti3sun.ti3.tu-harburg.de`
in the file:
`/pub/reports/report93.4.ps.Z`

PROFIL — Programmer's Runtime Optimized Fast Interval Library

Olaf Knüppel*
Technische Informatik III
Technische Universität Hamburg-Harburg
email: knueppel@tu-harburg.d400.de

July 15, 1993

Abstract

This is a compact overview of PROFIL, a portable C++ class library. It introduces new data types as vectors, matrices, intervals, interval vectors and matrices, integer vectors and matrices, and lots of operations between them. Additionally, several commonly used routines as linear interval system solver, etc. are also included. Routines written in FORTRAN also can be included by using a FORTRAN to C converter (e.g. `f2c`, which is public domain), writing a small interface routine in PROFIL that calls the converted FORTRAN routine.

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 for IBM RS/6000 and Sparc architectures as well as for PCs.

For readers coming from other programming languages and being unfamiliar 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.

*D-21071 Hamburg

Contents

1 Introduction **4**

1.1 Installation 4

1.1.1 Unix Systems 4

1.1.2 PCs under DOS or OS/2 4

1.1.3 Common Installation Steps 4

1.2 Configuration 6

1.3 Supported Data Types 8

1.4 Further Remarks 8

2 Operations defined by any header file **9**

3 Error Handling (Error.h) **10**

4 Constants.h **10**

5 Complex.h **11**

6 Interval.h **12**

7 Vector.h **14**

8 IntervalVector.h **15**

9 IntegerVector.h **17**

10 Matrix.h **18**

11 IntervalMatrix.h **20**

12 IntegerMatrix.h **22**

13 Functions.h **23**

14 Utilities.h **24**

15 LSS.h **25**

16 Using FORTRAN routines with PROFIL **25**

17 Adapting Real Operations to a Specific Hardware **28**

18 A Quick Introduction To C **28**

18.1 The Subroutine Header 29

18.2 Statements 29

18.2.1 Executable Statements 30

1 Introduction

This documentation is structured as follows. First, we give a short description how to install PROFIL and how to run the test programs. Then we will give an introduction into the most commonly used C++ programming constructs.

After that follows the description of all supported data types and operations. In this reference part the operations are listed under the name of the header file in which they are defined.

1.1 Installation

1.1.1 Unix Systems

To install PROFIL, change to the directory in which you want to install and copy the `profil.tar.Z` file into this directory.

Unpack the compressed archive by

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

After that, you can delete the `profil.tar.Z` file to save space.

1.1.2 PCs under DOS or OS/2

To install PROFIL, you will need the public domain utilities `tar` and `gzip`. Change to the directory in which you want to install and copy the `profil.tgz` file into this directory.

Unpack the compressed archive by

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

After that, you can delete the `profil.tgz` file to save space.

1.1.3 Common Installation Steps

It is assumed that you have the `dmake` utility already installed. If not, you have to do it prior the following steps. Type

```
cd Profil/BIAS
```

to change to the BIAS directory. Depending on your compiler, you have to type the following in order to build the BIAS library:

Command	Architecture	Compiler used
`dmake xlc`	IBM RS/6000	IBM xlc
`dmake ibmgcc`	IBM RS/6000	gcc
`dmake sun4acc`	Sparc	SUN acc
`dmake sun4gcc`	Sparc	gcc
`dmake sun3gcc`	SUN 3 (680x0)	gcc
`dmake pcgcc`	PCs (80x86)	gcc

After the BIAS library has been built successfully, change to the PROFIL directory by typing

```
cd ..
```

(assuming that you have been in the BIAS directory). Here, you have to type the following in order to build the PROFIL library:

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

Additionally, a small test program (`Test.C`) can also built during this process, if `_test` is appended to the above build commands, e.g. `dmake xlC_test`.

Assuming that you develop you own programs, you have two choices:

1. You develop your programs in the PROFIL directory, then you compile and link your programm (assumed it is called `myprog.C`) the following way:

   ```
   CC -O -o myprog -I. -IBIAS myprog.C -L. -lProfil -LBIAS -lBias -lm
   ```

   where `CC` is the name of your C++ compiler. Other possibilities for the C++ compiler are: `gcc` or `xlC`.

2. Your develop your programs in a separate directory and want to move the libraries and the include files to a common place. If this place is not in your home directory you probably have to ask your system administrator to do the appropriate steps for you

(including the setting of the correct file modes). Assuming that you library directory is `/home/lib` and your include directory is `/home/incl` then under Unix you type the following to move the files to their places, if your current directory is the PROFIL directory:

```
cp *.h /home/incl
cp BIAS/*.h /home/incl
cp libProfil.a /home/lib
cp BIAS/libBias.a /home/lib
ranlib /home/lib/libProfil.a
ranlib /home/lib/libBias.a
```

Under DOS or OS/2 you have to replace the `cp` by `copy`, the `ranlib` by `ar rvs` and the slashes by backslashes. Again assuming that your program is called `myprog.C`, you compile and link as follows:

```
CC -O -o myprog -I/home/incl myprog.C -L/home/lib -lProfil -lBias -lm
```

where `CC` is again the name of your C++ compiler.

If you use the `gcc`, you might have to precede the`-lm` by `-lg++` or `-lgcc`.

Due to limitations in file name length on some systems, some of the file names mentioned below are different on your system. If any differences occur, they can be found in the file `names` where the first name in each line is the name mentioned in this documentation and the second name is the file name on your system. On such systems the library `Profil` is called `prof` instead (i.e. the file name is `libprof.a`).

1.2 Configuration

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 developement 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 unneccessary 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.

1.3 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

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.

1.4 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.

2 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`
`a /= b`	Same as `a = a / b`

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

3 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

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:

$$\texttt{ErrorHandler::WarningLevel} = \texttt{ErrorHandler::Warning} = 1000$$
$$\texttt{ErrorHandler::SevereLevel} = \texttt{ErrorHandler::SevereError} = 2000$$

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()`.

4 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)

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.

5 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 `a`
`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

Operation	Description
`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`)

Additionally, the following functions are provided:

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

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

Comparision	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`)

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

6 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`)

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 inner of `a`
`Succ(a)`	`INTERVAL`	Smallest interval in which `a` is contained in the inner

Operation	Return Type	Description
`Hull(r)`	`INTERVAL`	Point interval [r, r]
`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.

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

Comparision	Description
`x <= a`	True, if `x` is contained in `a`
`x < a`	True, if `x` is contained in the interior `a`

7 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`
`v /= r`	Same as `v = v / r`

Operation	Description
`+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`

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`

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

Comparision	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

8 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

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

Operation	Return Type	Description
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	Convex hull of x and y
SymHull(v)	INTERVAL	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

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

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

9 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`

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,i)`	—	Initializes all elements of `v` with `i`

10 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 (result type: `VECTOR`)
`A * B`	Matrix product
`A(i,j)`	denotes the `i`-th row and `j`-th column of `A`

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,r)`	—	Initializes all elements of `A` with `r`
`Row(A,i)`	—	Returns the `i`-th row of `A` result type is `VECTOR`
`Col(A,i)`	—	Returns the `i`-th column of `A` result type is `VECTOR`
`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`

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

Comparision	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

11 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. 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
`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 vector subtraction (`X` and `Y` must not be both `MATRIX`)

Operation	Description
`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 (result type: `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`

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`
`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`
`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`	Convex hull of `X` and `Y`
`SymHull(A)`	`INTERVAL`	short form for `Hull(-A,A)`

Operation	Return Type	Description
`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.

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

Comparision	Description
`A <= IA`	True, if `A` is contained in `IA`
`A < IA`	True, if `A` is contained in the interior `IA`

12 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
`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 (result type: `INTEGER_VECTOR`)
`A(i,j)`	denotes the `i`-th row and `j`-th column of `A`

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`

13 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`	$e = 2.71828\ldots$
`Constant::Sqrt2`	$\sqrt{2}$
`Constant::InvSqrt2`	$\sqrt{0.5}$
`Constant::Ln10`	$\log_e 10$

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)`	$\sin$ `x`
`Cos(x)`	$\cos$ `x`
`Tan(x)`	$\tan$ `x`
`ArcSin(x)`	$\arcsin$ `x`
`ArcCos(x)`	$\arccos$ `x`
`ArcTan(x)`	$\arctan$ `x`
`Sinh(x)`	$\sinh$ `x`
`Cosh(x)`	$\cosh$ `x`
`Tanh(x)`	$\tanh$ `x`
`ArcSinh(x)`	$\operatorname{arcsinh}$ `x`
`ArcCosh(x)`	$\operatorname{arccosh}$ `x`
`ArcTanh(x)`	$\operatorname{arctanh}$ `x`
`Exp(x)`	$e^{\texttt{x}}$
`Log(x)`	$\log_e$ `x`
`Log10(x)`	$\log_{10}$ `x`
`Sqr(x)`	$\texttt{x}^2$
`Sqrt(x)`	$\sqrt{\texttt{x}}$
`Root(x,i)`	$\sqrt[i]{\texttt{x}}$
`Power(x,i)`	$\texttt{x}^{\texttt{i}}$
`Power(x,y)`	$\texttt{x}^{\texttt{y}}$

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

Function	Description
`Abs(r)`	$\lvert\texttt{r}\rvert$
`Min(r,s)`	$\min\{\texttt{r},\texttt{s}\}$
`Max(r,s)`	$\max\{\texttt{r},\texttt{s}\}$

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

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

14 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 $\mathtt{A}^{-1}$
`Transpose(A)`	`MATRIX`	Returns $\mathtt{A}^{T}$
`Id(i)`	`MATRIX`	Returns an identity matrix of dimension `i` $\times$ `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

15 LSS.h

This file defines (interval) linear system solvers computing an enclosure of the solution set. These methods are due to Prof. Dr. S.M. Rump. 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.

16 Using FORTRAN routines with PROFIL

In this section we describe how to use FORTRAN routines with PROFIL. FORTRAN `double precision` and `integer` types can be used without any modification. For single precision, a conversion is neccessary for vector and matrix types, because PROFIL uses double precision internally. This conversion is beyond the scope of this short documentation and because of this left out here. It is assumed that the reader is familiar with building and using own libraries.

The general steps to be performed are:

1. Convert all used FORTRAN routines to C (e.g. using `f2c`, which is freely available via NetLib).
2. Compile the converted C files with the `f2c.h` file which is distributed with PROFIL.
3. Build a library from the compiled files.
4. Write the PROFIL interface routines to the compiled C routines.
5. Use the PROFIL interface routines in your program and link the new library routines with your program.

Throughout this section we use the computation of the eigenvalues of a symmetric real matrix as an example (the FORTRAN routine is `RSM` from EISPACK). We assume, that all FORTRAN routines belonging to `RSM` are converted to C and a library called `libRSM.a` has been built from this set of routines.

The parameter description of the FORTRAN `RSM` subroutine is given below (taken from the EISPACK `rsm.f` file).

```
      subroutine rsm(nm,n,a,w,m,z,fwork,iwork,ierr)
c
      integer n,nm,m,iwork(n),ierr
      integer k1,k2,k3,k4,k5,k6,k7
      double precision a(nm,n),w(n),z(nm,m),fwork(1)
c
c     this subroutine calls the recommended sequence of
c     subroutines from the eigensystem subroutine package (eispack)
c     to find all of the eigenvalues and some of the eigenvectors
c     of a real symmetric matrix.
c
c     on input
c
c        nm  must be set to the row dimension of the two-dimensional
c        array parameters as declared in the calling program
c        dimension statement.
c
c        n  is the order of the matrix  a.
c
c        a  contains the real symmetric matrix.
c
c        m  the eigenvectors corresponding to the first m eigenvalues
```

```
c           are to be computed.
c           if m = 0 then no eigenvectors are computed.
c           if m = n then all of the eigenvectors are computed.
c
c     on output
c
c        w  contains all n eigenvalues in ascending order.
c
c        z  contains the orthonormal eigenvectors associated with
c           the first m eigenvalues.
c
c        ierr  is an integer output variable set equal to an error
c           completion code described in the documentation for tqlrat,
c           imtqlv and tinvit.  the normal completion code is zero.
c
c        fwork  is a temporary storage array of dimension 8*n.
c
c        iwork  is an integer temporary storage array of dimension n.
c
```

This header is converted by `f2c` into the following C header, which can be used from PROFIL routines:

```
extern "C" {
  int rsm_(PINT nm, PINT n, PREAL a, PREAL w,
           PINT m, PREAL z, PREAL fwork,
           PINT iwork, PINT ierr);
}
```

The leading `P` of the types denotes a pointer to the appropriate type, because FORTRAN always uses reference parameters.

Assuming that we want to create a PROFIL subroutine which returns a vector containing the eigenvalues of its matrix argument, the header would be

```
  VECTOR Eigenvalues (MATRIX & A)
```

To use the PROFIL vectors and matrices as FORTRAN subroutine parameters, the extension `theElements` must be appended with a separating dot to the vector or matrix name. Additionally, all vector and matrix types must be correctly sized before calling the converted

FORTRAN routine. For the small example discussed here, the complete implementation of the subroutine `Eigenvalues` might look like:

```
VECTOR Eigenvalues (MATRIX & A)
// Computes and returns all eigenvalues of the symmetric matrix A
{
  INT nm = RowDimension (A);
  INT n = ColDimension (A);
  VECTOR e(n);
  VECTOR fwork(8 * n);
  INTEGER_VECTOR iwork(n);
  INT ierr, m = 0;

  // No symmetry check is performed!!
  rsm_ (& nm, & n, A.theElements, e.theElements, & m, NULL,
        fwork.theElements, iwork.theElements, & ierr);
  if (ierr) Error ("Something is wrong in eigenvalue computation",
                   ierr, ErrorHandler::SevereError);
  return e;
}
```

17 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.

18 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)

18.1 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)
```

18.2 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)
```

18.2.1 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; }
```

In dieser Reihe sind bisher erschienen:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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