SMLIB v. 1.1
A Fortran 90 Library for Sparse Matrix Calculations

Ernst A. Meese

October 1998

MTF  Fluid Dynamics

Department of Applied Mechanics, Thermo- and Fluid Dynamics
Norwegian University of Science and Technology
Trondheim
Preface

This manual contains the documentation of a Fortran 90 subroutine library (SMLIB) aimed at the solution of large sparse systems of equations. The computer code is written and developed as part of my doctoral thesis work on finite volume methods for the incompressible Navier-Stokes equations on unstructured grids. The main motivation for writing the library was to study the performance of various iterative methods applied to matrices originating from the fluid flow equations.

As it is, SMLIB v. 1.1 is a rather small library, and contain routines you will find in the literature. I am, however, not aware of any other freely available implementation in Fortran 90. The development version of the library contains a larger amount of both abstract data types, solvers, and factorisations. As I get the time to clean up and validate the code, this will find it’s way into SMLIB. Highest on the priority list is a generalised strongly implicit factorisation routine [20], and some more abstract data types for sparse matrices.

The documentation of the library may still be somewhat scarce. I think it is sufficient for the ordinary user (please let me know if I’m wrong), but those who want to go deeper into it may want some more information.

I have tested the library code in the sense that the routines behave well together with my computer codes, but experience shows that bugs are very likely to still be around. So, if you find any, please tell me about it. I would also anticipate comments and critics. Enjoy!

October 23, 1998

Ernst A. Meese, Research Fellow
Department of Applied Mechanics, Thermo and Fluid Dynamics
Faculty of Mechanical Engineering
Norwegian University of Science and Technology
7034 Trondheim, Norway

Phone: +47 73 59 37 11 Fax: +47 73 59 37 13
E-mail: Ernst.A.Meese@mtf.ntnu.no
WWW: http://www.maskin.ntnu.no/mtf/people/eamtf/meese_eng.html

---

1 A few routines are third party freeware.
Acknowledgements

This work has received support from the Norwegian Supercomputing Committee (TRU) through a grant of computing time.

I also would like to thank my thesis supervisor, associate professor Skjalg Haaland, for support and interesting discussions.
Contents

Preface i
Acknowledgements ii
Table of Contents iii

1 Introduction 1
  1.1 Programming Ideology ........................................ 1
  1.2 How to compile and use the Library .......................... 2
  1.3 Where to Find SMLIB ............................................ 3

2 Miscellaneous 4
  2.1 Library Precision ............................................. 4
  2.2 Platform Dependent Routines ................................. 4
    CPU_TIME ...................................................... 4
    2.2.1 Floating Point Operations Counter ...................... 5
    Init_Flop ...................................................... 5
    Flop .......................................................... 5
    DFlop .......................................................... 5
    2.2.2 Input/Output Routines and Parameters .................... 5
    new_unit ....................................................... 5
  2.3 Miscellaneous ................................................. 6
    TimeStamp ...................................................... 6
    MachineStamp .................................................. 6
    PrecisionStamp ................................................ 6
    UpCase ........................................................ 7
    DownCase ....................................................... 7

3 Abstract Data Types for Sparse Matrices 8
  3.1 Compressed Sparse Row – CSR .................................. 8
    Nullify_Matrix .................................................. 9
    Allocate_Matrix ................................................ 9
    Reallocate_Matrix ............................................. 10
    Deallocate_Matrix ............................................ 10
    allocated_matrix ............................................. 10
    Entry .......................................................... 10
    SetRow ........................................................ 11
Matrix to array assignment; $A = S$ ..........................  11
Matrix-vector product; $S \times x$ ..........................  12
is_lower ..............................................  12
is_upper ..............................................  12
is_strictly_lower ..................................  13
is_strictly_upper ..................................  13
is_ok ..................................................  13
Save ..................................................  14
Load ..................................................  14

3.2 Modified Sparse Row - MSR ..................................  14
Nullify Matrix .........................................  15
Allocate Matrix ........................................  15
Reallocate Matrix .....................................  16
Deallocate Matrix .....................................  16
allocated_matrix ....................................  16
Entry ..................................................  17
SetRow ................................................  17
SetVal ................................................  18
AddVal ................................................  18
RowSum ................................................  18
RowNNZ .................................................  18
Matrix to array assignment; $A = S$ ..........................  19
Matrix-vector product; $S \times x$ ..........................  19
is_lower ..............................................  19
is_upper ..............................................  20
is_ok ................................................  20
Save ..................................................  20

3.3 Conversion Algorithms .........................................  21
MSR to CSR Assignment; $A_{CSR} = A_{MSR}$ .................  21

4 Approximate Factorisation .........................................  22
4.1 Abstract Data Type for Sparse Matrix Factorisations ...........  22
Nullify Matrix .........................................  22
Allocate Matrix ........................................  23
Reallocate Matrix .....................................  23
Deallocate Matrix .....................................  23
Entry_L .................................................  24
Entry_U .................................................  24
Solve ..................................................  24
is_ok ................................................  25

4.2 Incomplete $LU$ by Level of Fill-in .........................  25
Symbolic_Cancellation ..................................  25
ILU ...................................................  25
5 Iterative Solution Algorithms ............................................. 27
5.1 Common Optional Arguments and Default Values ................. 27
5.2 Iteration History Log .................................................... 28
   Nullify_ConvHist .................................................... 28
   Write_ConvHist ..................................................... 28
   No_of_Its .............................................................. 29
   Time_ConvHist ....................................................... 29
   Flop_ConvHist ...................................................... 30
   Residual_ConvHist .................................................. 30
5.3 Krylov Subspace Methods ............................................. 30
   BiCGSTAB ............................................................. 30
   CGS ................................................................. 31
   GMRES ............................................................... 31

6 Direct Solver LAPACK Interface ........................................ 33
6.1 Data Type for Band Matrix Factorisation .......................... 33
   Nullify_Matrix ...................................................... 33
   Allocate_Matrix .................................................... 34
   Deallocate_Matrix .................................................. 34
   allocated_matrix ................................................... 34
6.2 Interfacing the LAPACK Band Gauss Solver ........................ 34
   LU_factor ........................................................... 35
   Solve ............................................................... 35

A Brief Introduction to Krylov-Subspace Methods ...................... 36
A.1 Krylov Subspace Methods ............................................. 37
   A.1.1 Strategy 1 – Minimising the Norm over $K_m$ ............ 38
   A.1.2 Strategy 2 – Bi-orthogonality ............................. 40
   A.1.3 Hybrid Methods – Strategies 1 and 2 Combined ........ 41
A.2 Preconditioning ...................................................... 42
A.3 Matrix Splitting Methods ........................................... 43
   A.3.1 Incomplete LU Factorisation ............................... 45

B Example – Solving the Poisson equation .............................. 47

C Module Templates ....................................................... 49
   C.1 Abstract Data Type for Matrices ............................. 49
   C.2 Abstract Data Type for Factorisations ....................... 51
   C.3 Iterative Solvers ................................................. 52

Bibliography ........................................................................ 55
Chapter 1

Introduction

The Sparse Matrices Library (SMLIB) is written in the computer language Fortran 90, and is intended used with Fortran 90. Some very few routines are written in C, due to C's close connection to the operating system UNIX. These routines are, however, all wrapped inside Fortran 90 routines for easy reference from Fortran 90.

As the library is developed in an UNIX environment, the compilation of it may not be straightforward in other environments. However, the code follow the Fortran 90 standard ISO/IEC 1539:1991 and should compile well with any compiler, but file and module dependencies and operating system calls are likely to need special care.

At present, the library consists of the following parts

**Miscellaneous** Various handy routines and definitions. In particular it defines the numerical precision for the rest of the library.

**Matrices** Data structures for several sparse matrix and sparse matrix factorisation storage schemes, together with routines to operate on the data structures.

**Incomplete factorisations** Contain routines to perform incomplete factorisations for sparse matrices.

**Iterative solvers** Contain iterative methods for the solution of large sparse linear systems of equations.

**Direct solvers** Contains a Fortran 90 interface to the LAPACK subroutines for band matrix Gaussian elimination.

1.1 Programming Ideology

A common solution to the problem of making generally applicable code for iterative solvers in FORTRAN77 is reversed communication. What is generally not known at the time of implementation of a generic program library is the data storage format. With reversed communication, every time the solver needs to operate on the data, e.g. in a matrix by vector multiplication, the solver halts and send back a message to the main program, asking for the result of the needed operation. It is then up to the user of the library to perform this operation and to call the solver anew with the requested information.
CHAPTER 1. INTRODUCTION

Even though the library may contain the most common data storage formats and the operations on them, reversed communication make the user threshold quite high. On the other hand, it is a very flexible paradigm, allowing advanced users to make very efficient computer codes. In SMLIB, reversed communication is not used, and we try to exploit the new features of Fortran 90 to achieve both an ease of use and a flexibility not attainable in FORTRAN 77. Crucial features are dynamic memory, optional parameters, overloading and modules.

Abstract data types for sparse matrices are created by the aid of modules. These modules define both the storage schemes and the procedures to operate on the matrices. The procedures are given generic names by the Fortran 90 overloading facility. This makes it easy to write generic code for algorithms using linear algebra.

As the code is to a certain extent generic, it is not necessary to rewrite it when a new data storage scheme is implemented. The new procedures are simply given the same generic name as the older procedures, and voilà, the new data storage scheme is up and operating.

Or almost operating; due to Fortran 90's lack of the object oriented concept inheritance, we have to create an entirely new procedure for the new data storage. But, only the declaration heading will be different from the older procedure. The body of the solvers are therefore placed in include files in order to reduce redundancy of code.

These features make the library very easy to use if you use the matrix data structures that follow the library. However, as under the reversed communication paradigm, the advanced user will still have to implement her own data storage formats together with some procedures (see appendix C).

It must be mentioned, that in the development of the codes in SMLIB, my mind has been focused on approximate matrix factorisations as preconditioners. It is common also to use iterative methods for preconditioning. To make this possible in SMLIB, we could send the procedure name of the preconditioner to the solver via the argument list. This requires only small modifications of the existing code. However, the real problem is how to send parameters to the preconditioner. A later release of SMLIB will hopefully deal with this.

Each abstract data type is contained in a Fortran 90 module with it's data structures and procedures to operate on the structures. The data structures are all made public, and hence directly accessible to the user. This is not in accordance with the good programming principle of encapsulation, but efficient numerical routines usually require direct access to the data structure. However, to use this library, direct reference to the internal data structures are not necessary.

1.2 How to compile and use the Library

See the file release_notes for instructions on how to compile the library. Some effort are made in order to achieve portability between various UNIX platforms, and the installation should work well at least with NAG's and Cray's Fortran 90 compilers.

Once SMLIB is properly compiled, you can access the library from your own programs like any other library. The exception is that you must also tell the Fortran 90 compiler where to find module information during compilation. If the library directory is placed at our root, a typical compilation will look like; with the NAG compiler

```
f90 -c -I"/slib/nag_modules myprog.f90
```
and a typical invocation of the loader

```fortran
f90 -o myprog myprog.o -L~/libsm -lsm -lU77
```

Note that SMLIB is dependent on the FORTRAN77 `etime` subroutine which is usually found in the U77 library. The library U77 may have a different name on your particular platform.

On Crays the same steps are

```fortran
f90 -c -p ~/slib/libsm.a myprog.f90
```

and

```fortran
f90 -o myprog myprog.o -L~/libsm -lsm
```

The modules that the library consist of, are stacked in a hierarchical manner with the modules holding the solvers and incomplete factorisations on the top. This means, that if you use one of these modules, the modules from the miscellaneous and the matrices directories will be available too.

### 1.3 Where to Find SMLib

A tar'ed and compressed version of the library and a postscript version of this documentation are found at the IP address `ftp.maskin.ntnu.no` and may be downloaded with anonymous FTP. The catalogue is `/pub/mtf/smlib`. You may also find a small homepage for SMLib at `http://www.maskin.ntnu.no/mtf/people/eammtf/smlib.html`. 


Chapter 2

Miscellaneous

2.1 Library Precision

The module Precision Module define the three real kinds single_prec, double_prec, and prec. The two first refer to the common single and double precision. The working precision of the library prec is set to at least 12 significant digits, and with a range up to at least $10^{30}$. This will imply double precision on most machines, but on others, like on the Cray Y/MP, it is single precision.

All real variables send to the SM library should be of type

\[ \text{REAL}(\text{prec}) \]

The precision module is, with very few exceptions, used by all other modules in SMLIB, and the USE Precision Module statement is normally unnecessary, as it is included via other modules.

2.2 Platform Dependent Routines

- CPU\_TIME (time)

**Description.** Returns the CPU-time used by the program. It relies on the FORTRAN77 etime (stime on Cray) subroutine, and is the reason why the library U77 must be accessed in the linking process. etime is not standard FORTRAN77, which may cause trouble. At the advent on Fortran 95 compilers, CPU-TIME will become obsolescent as it is part of the new standard. The routine is not part of any module, and is called as an external procedure.

**Argument.** time will on output contain the CPU-time used by the program. time is of type REAL (not REAL(prec)). It is an INTENT(OUT) argument.

**Example.**

```fortran
REAL :: time

CALL CPU\_TIME ( time )
```
2.2. PLATFORM DEPENDENT Routines

2.2.1 Floating Point Operations Counter

To access the floating point operations counter, use the module `FlopModule`. If any of the iterative solvers are used, this module will already be available. Note that these routines depend on the computer to have a floating point operations counter. This is a heavily machine dependent facility, and is currently available only for Cray Y/MP and Cray J90. To check if the Cray counter is present, see if `Flop.Counter.Exists` is .TRUE.

- `Init_Flop ()`
  
  **Description.** Initiates the floating point operations counter if the computer has one. If no counter is present, this routine do nothing.
  
  **Example.** Call `Init_Flop ()`

- `Flop ()`
  
  **Description.** Returns the number of floating point operations used by the program. If a counter is not available, this routine will return -1.
  
  **Result type.** `INTEGER(Flop_Int)`
  
  **Example.** `PRINT *, Flop ()`

- `DFlop ()`
  
  **Description.** Returns the number of floating point operations used by the program since the last call to this function. If a counter is not available, this routine will return -1.
  
  **Result type.** `INTEGER(Flop_Int)`
  
  **Example.** `PRINT *, DFlop ()`

2.2.2 Input/Output Routines and Parameters

The parameters and routine in the input/output module are made available through the command `USE IO`. The NAG version are freeware developed at Unicomp, Inc. It is available at the world wide web URL [http://www.fortran.com/fortran/free.html](http://www.fortran.com/fortran/free.html), where also some other versions may be found. The Cray version of this module is a modification of the NAG version.

**Routine**

- `new_unit ()`
  
  **Description.** Returns an unused input/output unit number.
  
  **Return type.** `INTEGER`
  
  **Example.**
\[
a_{\text{unit}} = \text{new}_{\text{unit}}()
\]

OPEN \(a_{\text{unit}}, \text{FILE} = 'NS\text{-solution}', \text{STATUS} = 'READ'\)

Parameters

\textbf{DEFAULT\_INPUT\_UNIT} The unit number of the default input unit, normally the keyboard.

\textbf{DEFAULT\_OUTPUT\_UNIT} The unit number of the default output unit, normally the screen.

\textbf{NUMBER\_OF\_PRECONNECTED\_UNITS} The number of units occupied by the system.

\textbf{PRECONNECTED\_UNITS (NUMBER\_OF\_PRECONNECTED\_UNITS)} Array containing the units occupied by the system.

\textbf{END\_OF\_RECORD} The end of record marker.

\textbf{END\_OF\_FILE} The end of file marker.

\textbf{MAX\_UNIT\_NUMBER} The largest unit number allowed.

### 2.3 Miscellaneous

To make the miscellaneous routines available; \texttt{USE Misc\_Module}.

- **TimeStamp (\()**

  \textbf{Description.} Returns the current date and time in a formatted character string.

  \textbf{Result.} Character string of length 17.

  \textbf{Example.}

  \[
  \text{CHARACTER(LEN=17)} :: \text{time}
  \]

  \[
  \text{time} = \text{TimeStamp}() \\
  \text{print '(A)', TRIM(time)}
  \]

- **MachineStamp (\()**

  \textbf{Description.} Calls the UNIX \texttt{name} facility to identify the computer and the operating system.

  \textbf{Result.} It returns the information in an 80 character string.

  \textbf{Example.}

  \[
  \text{CHARACTER(LEN=80)} :: \text{platform}
  \]

  \[
  \text{platform} = \text{MachineStamp}() \\
  \text{print '(A)', TRIM(platform)}
  \]
• PrecisionStamp ()

**Description.** Returns a character string characterising the working precision of the library.
**Result.** Character string of length 80.
**Example.**

```plaintext
CHARACTER(LEN=80) :: precision_info

precision_info = PrecisionStamp ()
print '(A)', TRIM(precision_info)
```

• UpCase (String)

**Description.** Shifts all characters in String to upper case letters.
**Arguments.** String is a character string of any length.
**Example.**

```plaintext
CHARACTER(LEN=80) :: a_name

a_name = 'Ludwig Prandtl'
Call UpCase ( a_name )
print '(A)', TRIM(a_name)
```

• DownCase (String)

**Description.** Shifts all characters in String to upper case letters.
**Arguments.** String is a character string of any length.
**Example.**

```plaintext
CHARACTER(LEN=80) :: a_name

a_name = 'Geoffrey Ingram Taylor'
Call DownCase ( a_name )
print '(A)', TRIM(a_name)
```
Chapter 3

Abstract Data Types for Sparse Matrices

As mentioned in the introduction, abstract data types are defined by collecting data structures and the procedures operating on these structures, in a Fortran 90 module. All matrix modules use the precision module. Hence all real variables have the precision as defined by \texttt{prec}. E.g., in order to define an allocatable vector to the same precision as the matrices, use

\begin{verbatim}
REAL(prec), DIMENSION(:), ALLOCATABLE :: x
\end{verbatim}

As the rationale for this library is to solve the discretised equations of a differential equation, we restrict ourselves to square matrices.

All matrix data types will be made available by the command

\begin{verbatim}
USE Matrix_Arithmetic_Module
\end{verbatim}

It is however possible to use the data types one at a time by specifying the particular modules.

3.1 Compressed Sparse Row – CSR

A common format for storing general sparse matrices is the compressed sparse row format (CSR), see e.g. [27]. We restrict the generality to square matrices with \( N \) rows and columns.

The CSR data structure containing a matrix \( A = (a_{ij}) \) consists of three arrays. Firstly, a real array \( A \) containing the values \( a_{ij} \) stored row by row. The length of \( A \) equals the maximum possible number of non-zeros.

Secondly, an integer array \( JA \) containing the column indices \( j \) of the elements \( a_{ij} \) as stored in the array \( A \). The length of \( JA \) equals the length of \( A \). Lastly, an integer array \( IA \) containing pointers to the beginning of each row \( i \) in the arrays \( A \) and \( JA \). Thus, \( IA(i) \) points to the position in arrays \( A \) and \( JA \) where the \( i \)-th row starts. The length of \( IA \) is \( N+1 \) with \( IA(N+1) \) containing the number \( NNZ+1 \). \( NNZ \) denotes the number of non-zeros in the matrix.

No sorting of the elements within each row is assumed. An empty row \( i \) is handled by setting \( IA(i + 1) = IA(i) \).

The data structure is implemented in Fortran 90 as

\begin{verbatim}
TYPE CSR
  INTEGER :: N
\end{verbatim}
Figure 3.1: The CSR data format for a sparse matrix $A \in R^{N \times N}$.

REAL(prec), DIMENSION (:), POINTER :: A
INTEGER, DIMENSION (:), POINTER :: IA, JA
END TYPE CSR

The integer $N$ is strictly speaking not necessary. When it is needed, it may be found by the call size(S % IA)-1. However, this is rather cumbersome and non-intuitive, and since $S \% N$ is used extensively, it is included in the data structure.

To use the CSR data structure, the following two lines must be present among the declarations in the using routine:

USE CSR_Module
TYPE (CSR) :: S

Procedures

• **NullifyMatrix (S)**

  **Description.** Since Fortran 90 pointers have an initial association status that is undefined, this procedure must be run to guaranty that the memory allocation checks will perform correctly. It nullifies $A$, $IA$, and $JA$ and sets $N$ to zero. For safety, it should be called for all CSR matrices in the beginning of a program.

  **Argument.** $S$ is a matrix of CSR type. It is an INTENT(IN) argument.

  **Example.** CALL NullifyMatrix (S).

• **AllocateMatrix (S, N, NZMAX)**

  **Optional argument.** $NZMAX$.

  **Description.** Allocates memory sufficient to store $NZMAX$ elements in the matrix, and sets $S \% N$ to $N$. If $NZMAX$ is omitted, room for Block Size elements is created. Currently, Block Size is set to 50.

  **Arguments.**

  - $S$ is a matrix of CSR type.
  - $N$ is of type INTEGER. It is an INTENT(IN) argument.
  - $NZMAX$ is of type INTEGER. It is an INTENT(IN) argument.

  **Examples.**

  ${}^1$Remember that you may also write USE Matrix Arithmetic Module.
N = 10
CALL Allocate Matrix (S, N)
CALL Allocate Matrix (S, 100, 1000)

- Reallocation Matrix (S, Delta)

Optional argument. Delta.

Description. Adds/removes memory sufficient to store Delta matrix elements to/from the matrix. Negative Delta will remove memory. If Delta is omitted or set to a negative value that would leave too little memory for the non-zeros, allocated memory will shrink to fit the actual number of non-zeros. It perform a memory copy and may be relatively slow on large matrices.

Arguments.
- S is a matrix of CSR type.
- Delta is of type INTEGER. It is an INTENT(IN) argument.

Examples.
CALL Reallocation Matrix (S, 100)
CALL Reallocation Matrix (S)

- Deallocate Matrix (S)

Description. Frees memory allocated to the matrix and sets S % N to zero.

Arguments. S is a matrix of CSR type.

Example. CALL Deallocate Matrix (S)

- allocated matrix (S)

Description. Returns .TRUE. if the matrix is properly allocated, and .FALSE. if not. Never call this routine if the matrix allocation status is undefined, that is, there is no prior call to Allocate matrix or Nullify Matrix.

Arguments. S is a matrix of CSR type.

Example. IF (.NOT.allocated matrix(S)) CALL ALLOCATE MATRIX(S, N)

- Entry (S, i, j)

Description. Returns the value of element (i, j) in the matrix. This is not an efficient method to reference the matrix elements, but is included for debugging purposes.

Arguments.
- S is a matrix of CSR type.
- i is of type INTEGER. It is an INTENT(IN) argument.
- j is of type INTEGER. It is an INTENT(IN) argument.
Result. It returns a real of kind \texttt{prec}.

Example. This code prints a matrix of row dimension less or equal to ten.

\begin{verbatim}
DO i = 1, S % N
   PRINT ", (Entry (S, i, j), j = 1, S % N)
END DO
\end{verbatim}

- \texttt{SetRow (S, i, Cols, Row)}

Description. Sets the values of the elements of the matrix \( S \) at row \( i \) and the columns found in the array \( \text{Cols} \). The array \( \text{Row} \) holds the values corresponding to \( \text{Cols} \). Best efficiency is achieved by inserting rows by ascending row number, but any order is legal.

Arguments.

- \( S \) is a matrix of CSR type. It is an \texttt{INTENT(IN)} argument.
- \( i \) is of type \texttt{INTEGER}. It is an \texttt{INTENT(IN)} argument.
- \( \text{Cols} \) is a one dimensional array of type \texttt{INTEGER}. It’s size must equal the size of \( \text{Row} \). It is an \texttt{INTENT(IN)} argument.
- \( \text{Row} \) is a one dimensional array of type \texttt{REAL(prec)}. It’s size must equal the size of \( \text{Cols} \). It is an \texttt{INTENT(IN)} argument.

Example. The following code is creating a three diagonal matrix with 2 on the diagonal and -1 on the sub- and super diagonal.

\begin{verbatim}
DO I = 1, S % N
   cntr = 0
   IF (I > 1) THEN
      cntr = cntr + 1
      Cols(cntr) = I - 1; Row(cntr) = -1.0_prec
   END IF
   cntr = cntr + 1
   Cols (cntr) = I; Row (cntr) = 2.0_prec
   IF (I < S % N) THEN
      cntr = cntr + 1
      Cols(cntr) = I + 1; Row(cntr) = -1.0_prec
   END IF
   CALL SetRow (A, I, Cols(1:cntr), Row(1:cntr))
END DO
\end{verbatim}

- \texttt{Matrix to array assignment; A = S}

Description. Assign the entries of the two dimensional array \( A \) to the values of corresponding elements in the matrix \( S \).

Arguments.

- \( A \) 2-dimensional array of shape \( S \% N \) by \( S \% N \) and type \texttt{REAL(prec)}. It is an \texttt{INTENT(OUT)} argument.
CHAPTER 3. ABSTRACT DATA TYPES FOR SPARSE MATRICES

S is a matrix of CSR type. It is an INTENT(IN) argument.

Example.

REAL(prec), DIMENSION(N,N) :: A
TYPE (CSR) :: S

A = S

• Matrix-vector product; S * x

Description. Perform a product between a CSR matrix and a dense vector.

Arguments.

S is a matrix of CSR type. It is an INTENT(IN) argument.

x is a vector (1-dimensional array) of length S % N and type REAL(prec).

Result. Vector (1-dimensional array) of length S % N and type REAL(prec).

Example.

REAL(prec), DIMENSION(N) :: x, y
TYPE (CSR) :: S

y = S * x

• is_lower (S)

Description. Checks if the matrix represented by S is lower triangular. It only checks the column structure, hence if there is defined an possibly non-zero element in the upper triangle, this routine returns .FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of CSR type. It is an INTENT(IN) argument.

Result. .TRUE. when the structure of the matrix is lower triangular, .FALSE. if is not.

Example.

IF (is_lower (S)) &
   PRINT *, 'The matrix is lower triangular'

• is_upper (S)

Description. Checks if the matrix represented by S is upper triangular. It only checks the column structure, hence if there is defined an possibly non-zero element in the lower triangle, this routine returns .FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of CSR type. It is an INTENT(IN) argument.
3.1. COMPRESSED SPARSE ROW – CSR

Result. . TRUE. when the structure of the matrix is upper triangular, . FALSE. if is not.

Example.

IF (is_upper (S)) &
    PRINT *, 'The matrix is lower triangular'

• is_strictly_lower (S)

Description. Checks if the matrix represented by S is strictly lower triangular. It only checks the column structure, hence if there is defined a possibly non-zero element in the upper triangle or on the main diagonal, this routine returns . FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of CSR type. It is an INTENT(IN) argument.

Result. . TRUE. when the structure of the matrix is strictly lower triangular, . FALSE. if is not.

Example.

IF (is_strictly_lower (S)) &
    PRINT *, 'The matrix is strictly lower triangular'

• is_strictly_upper (S)

Description. Checks if the matrix represented by S is strictly upper triangular. It only checks the column structure, hence if there is defined a possibly non-zero element in the lower triangle or on the main diagonal, this routine returns . FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of CSR type. It is an INTENT(IN) argument.

Result. . TRUE. when the structure of the matrix is strictly upper triangular, . FALSE. if is not.

Example.

IF (is_strictly_upper (S)) &
    PRINT *, 'The matrix is strictly upper triangular'

• is_ok (S)

Description. Checks for consistency of the data structure in S.

Arguments. S is a matrix of CSR type. It is an INTENT(IN) argument.

Result. . TRUE. when the structure is consistent, . FALSE. if is not.

Example.
IF (is_ok (S)) THEN
    CALL The_Big_Calculation (S, b, x)
ELSE
    PRINT *, 'This matrix is bogus'
END IF

• Save (S, filename, form)

Optional argument. form.
Description. Save a CSR matrix to a disk file.
Arguments.
S is a matrix of CSR type. It is an INTENT(IN) argument.
filename Character string of any length. It is an INTENT(IN) argument.
form One of the two strings "formatted" or "unformatted", to store in either ascii or binary format respectively. Default is "formatted". It is an INTENT(IN) argument.

Examples.
CALL Save (S, "Matr.CSR", "unformatted")
CALL Save (S, "the_file")

• Load (S, filename, form)

Optional argument. form.
Description. Load a CSR matrix from a disk file.
Arguments.
S is a matrix of CSR type.
filename Character string of any length. It is an INTENT(IN) argument.
form One of the two strings "formatted" or "unformatted", to store in either ascii or binary format respectively. Default is "formatted". It is an INTENT(IN) argument.

Examples.
CALL Load (S, "Matr.CSR", "unformatted")
CALL Load (S, "the_file")

3.2 Modified Sparse Row – MSR

The modified sparse row format (MSR) for storing general sparse matrices is a quite common modification to the CSR format. The difference is that in MSR the value of the main diagonal elements is stored in a separate array. We restrict the generality to square matrices with \( N \) rows and columns.

The MSR data structure containing a matrix \( A = (a_{ij}) \) consists of two arrays. Firstly, a real array \( A \) where the first \( N \) elements hold the main diagonal \( (a_{ii}) \), element \( N+1 \) is not used,
and the remaining elements contain the off-diagonal elements \( a_{ij}, i \neq j \), stored row by row. The length of \( A \) equals the maximum possible number of non-zeroes plus one.

Secondly, an integer array \( JA \) where the first \( N \) elements points to the beginning of each row \( i \) for the off-diagonal elements in \( A \), \( N+1 \) points to the first unused element in \( A \), and the remaining elements contain the column indices \( j \) of the off-diagonal elements \( a_{ij}, i \neq j \) as stored in the array \( A \). The length of \( JA \) equals the length of \( A \). Note that \( JA(1:N+1) \) play the role of \( IA \) in the CSR data structure.

No sorting of the off-diagonal elements within each row is assumed. An empty row \( i \) is handled by setting \( IA(i+1)=IA(i) \).

The data structure is implemented in Fortran 90 as

```fortran
TYPE MSR
  INTEGER :: N
  REAL(prec), DIMENSION (:), POINTER :: A
  INTEGER, DIMENSION (:), POINTER :: JA
END TYPE MSR
```

To use the MSR data structure, the following two lines must be present among the declarations in the using routine\(^2\):

```fortran
USE MSR
TYPE (MSR) :: S
```

Procedures

- **Nullify Matrix** (\( S \))

**Description.** Since Fortran 90 pointers have an initial association status that is undefined, this procedure must be run to guarantee that the memory allocation checks will perform correctly. It nullifies \( A, IA, \) and \( JA \) and sets \( N \) to zero. For safety, it should be called for all MSR matrices in the beginning of a program.

**Argument.** \( S \) is a matrix of MSR type. It is an \textit{INTENT(IN)} argument.

**Example.** CALL Nullify Matrix (\( S \)).

\(^2\)Remember that you may also write USE Matrix Arithmetic Module.
Allocate Matrix (S, N, NZMAX)

Optional Argument. NZMAX.

Description. Allocates memory sufficient to store NZMAX elements for the matrix, and sets S % N to N. If NZMAX is omitted, room for Block Size elements is created. Currently, Block Size is set to 50. Note that there is always allocated memory for the main diagonal elements.

Arguments.

S is a matrix of MSR type.

N is of type INTEGER. It is an INTENT(IN) argument.

NZMAX is of type INTEGER. It is an INTENT(IN) argument.

Examples.

N = 10
CALL Allocate Matrix (S, N)
CALL Allocate Matrix (S, 100, 1000)

Reallocate Matrix (S, Delta)

Optional Argument. Delta.

Description. Adds/removes memory sufficient to store Delta matrix elements to/from the matrix. Negative Delta will remove memory. If Delta is omitted or set to a negative value that would leave too little memory for the non-zeros, allocated memory will shrink to fit the actual number of non-zeros. It performs a memory copy and may be relatively slow on large matrices.

Arguments.

S is a matrix of MSR type.

Delta is of type INTEGER. It is an INTENT(IN) argument.

Examples.

CALL Reallocate Matrix (S, 100)
CALL Reallocate Matrix (S)

Deallocate Matrix (S)

Description. Frees memory allocated to the matrix and sets S % N to zero.

Arguments. S is a matrix of MSR type.

Example. CALL Deallocate Matrix (S)
3.2. MODIFIED SPARSE ROW - MSR

- **allocated_matrix** (S)
  
  **Description.** Returns `.TRUE.` if the matrix is properly allocated, and `.FALSE.`, if not. 
  Never call this routine if the matrix allocation status is undefined, that is, there is no prior call to `Allocate_matrix` or `Nullify_Matrix`. 
  
  **Arguments.** S is a matrix of MSR type. 
  
  **Example.** IF `.NOT. allocated_matrix(S) CALL ALLOCATE_MATRIX(S, N)

- **Entry** (S, i, j)
  
  **Description.** Returns the value of element (i, j) in the matrix. This is not an efficient method to reference the matrix elements, but is included for debugging purposes. 
  
  **Arguments.** 
  
  S is a matrix of MSR type. It is an INTENT(IN) argument. 
  
  i is of type INTEGER. It is an INTENT(IN) argument. 
  
  j is of type INTEGER. It is an INTENT(IN) argument. 
  
  **Example.** This code prints a matrix of row dimension less or equal to ten.

```fortran
DO i = 1, S % N
  PRINT "(10F7.2)", (Entry (S, i, j), j = 1, S % N)
END DO
```

- **SetRow** (S, i, Cols, Row)
  
  **Description.** Sets the values of the elements of the matrix S at row i and the columns found in the array Cols. The array Row holds the values corresponding to Cols. 
  Best efficiency is achieved by inserting rows by ascending row number, but any order is legal.

  **Arguments.** 
  
  S is a matrix of MSR type. It is an INTENT(INOUT) argument. 
  
  i is of type INTEGER. It is an INTENT(IN) argument. 
  
  Cols is a one dimensional array of type INTEGER. It’s size must equal the size of Row. It is an INTENT(IN) argument. 
  
  Row is a one dimensional array of type REAL(prec). It’s size must equal the size of Cols. It is an INTENT(IN) argument. 
  
  **Example.** The following code is creating a three diagonal matrix with 2 on the diagonal and -1 on the sub- and super diagonal.

```fortran
DO I = 1, S % N
  cntr = 0 
  IF (I > 1) THEN 
    cntr = cntr + 1
    Cols(cntr)=I - 1; Row(cntr)=-1.0; prec 
  END IF
```
cntr = cntr + 1  
Cols (cntr) = I; Row (cntr) = 2.0_prec  
IF (I < S % N) THEN  
    cntr = cntr + 1  
    Cols(cntr) = I + 1; Row(cntr) = 1.0_prec  
END IF  
    CALL SetRow (A, I, Cols(1:cntr), Row(1:cntr))  
END DO

- **SetVal** (S, i, j, val)

  **Description.** Sets the value of the element $s_{ij}$ of the matrix S to val. The indices $i$, $j$ must be in the non-zero pattern of S, or else the routine will halt the program.

  **Arguments.**
  - S is a matrix of MSR type. It is an INTENT(INOUT) argument.
  - i is of type INTEGER. It is an INTENT(IN) argument.
  - j is of type INTEGER. It is an INTENT(IN) argument.
  - val is of type REAL(prec). It is an INTENT(IN) argument.

  **Example.** CALL SetVal(S, I, J, val)

- **AddVal** (S, i, j, val)

  **Description.** Adds the value val to the current value of element $s_{ij}$ in matrix S. The indices $i$, $j$ must be in the non-zero pattern of S, or else the routine will halt the program.

  **Arguments.**
  - S is a matrix of MSR type. It is an INTENT(INOUT) argument.
  - i is of type INTEGER. It is an INTENT(IN) argument.
  - j is of type INTEGER. It is an INTENT(IN) argument.
  - val is of type REAL(prec). It is an INTENT(IN) argument.

  **Example.** CALL AddVal(S, I, J, val)

- **RowSum** (S, i)

  **Description.** Returns the sum of the elements at row $i$ of the matrix S.

  **Arguments.**
  - S is a matrix of MSR type. It is an INTENT(IN) argument.
  - i is of type INTEGER. It is an INTENT(IN) argument.

  **Result.** is of type REAL(prec).

  **Example.** sum = RowSum(S, i)
3.2. **MODIFIED SPARSE ROW – MSR**

- **RowNNZ** ($S$, $i$)

  **Description.** Returns the number of the elements at row $i$ of the matrix $S$ that is in the non-zero sparsity pattern.

  **Arguments.**
  
  - $S$ is a matrix of MSR type. It is an INTENT(IN) argument.
  - $i$ is of type INTEGER. It is an INTENT(IN) argument.

  **Result.** An INTEGER greater than zero. Then MSR data structure has at least a non-zero main diagonal element.

  **Example.** $\text{sum} = \text{RowSum}(S, i)$

- **Matrix to array assignment; $A = S$**

  **Description.** Assign the entries of the two dimensional array $A$ to the values of corresponding elements in the matrix $S$.

  **Arguments.**
  
  - $A$ is a 2-dimensional array of shape $S \times N$ by $S \times N$ and type REAL(prec). It is an INTENT(OUT) argument.
  - $S$ is a matrix of MSR type. It is an INTENT(IN) argument.

  **Example.**
  
  ```
  REAL(prec), DIMENSION(N,N) :: A
  TYPE (MSR) :: S
  :
  A = S
  ```

- **Matrix-vector product; $S \ast x$**

  **Description.** Perform a product between a MSR matrix and a dense vector.

  **Arguments.**
  
  - $S$ is a matrix of MSR type. It is an INTENT(IN) argument.
  - $x$ is a vector (1-dimensional array) of length $S \times N$ and type REAL(prec). It is an INTENT(IN) argument.

  **Result.** Vector (1-dimensional array) of length $S \times N$ and type REAL(prec).

  **Example.**
  
  ```
  REAL(prec), DIMENSION(N) :: x, y
  TYPE (MSR) :: S
  :
  y = S \ast x
  ```
• is_lower (S)

Description. Checks if the matrix represented by S is lower triangular. It only checks the column structure, hence if there is defined an possibly non-zero element in the upper triangle, this routine returns .FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of MSR type. It is an INTENT(IN) argument.

Result. .TRUE. when the structure of the matrix is lower triangular, .FALSE. if is not.

Example.

IF (is_lower (S)) & 
   PRINT *, 'The matrix is lower triangular'

• is_upper (S)

Description. Checks if the matrix represented by S is upper triangular. It only checks the column structure, hence if there is defined an possibly non-zero element in the lower triangle, this routine returns .FALSE. even if this value is actually set to zero.

Arguments. S is a matrix of MSR type. It is an INTENT(IN) argument.

Result. .TRUE. when the structure of the matrix is upper triangular, .FALSE. if is not.

Example.

IF (is_upper (S)) & 
   PRINT *, 'The matrix is upper triangular'

• is_ok (S)

Description. Checks for consistency of the data structure in S.

Arguments. S is a matrix of MSR type. It is an INTENT(IN) argument.

Result. .TRUE. when the structure is consistent, .FALSE. if is not.

Example.

IF (is_ok (S)) THEN 
   CALL The_Big_Calculation (S, b, x) 
ELSE 
   PRINT *, 'This matrix is bogus!' 
END IF
3.3. **CONVERSION ALGORITHMS**

- **Save (S, filename, form)**

  **Optional argument. form.**

  **Description.** Save a MSR matrix to a disk file in the same format as used for the CSR matrices.

  **Arguments.**
  
  \( S \) is a matrix of MSR type. It is an INTENT(IN) argument.
  
  \( \text{filename} \) Character string of any length. It is an INTENT(IN) argument.
  
  \( \text{form} \) One of the two strings "formatted" or "unformatted", to store in either ascii or binary format respectively. Default is "formatted". It is an INTENT(IN) argument.

  **Examples.**
  
  ```
  CALL Save (S, "Matr.CSR", "unformatted")
  CALL Save (S, "the_file")
  ```

3.3 **Conversion Algorithms**

- **MSR to CSR Assignment; \( A_{CSR} = A_{MSR} \)**

  **Description.** Transfers a space matrix in MSR format to a sparse matrix in CSR format.

  **Arguments.**
  
  \( A_{CSR} \) is a matrix of CSR type. It is an INTENT(INOUT) argument. The matrix is allocated or reallocated if it do not fit the matrix \( A_{MSR} \).
  
  \( A_{MSR} \) is a matrix of MSR type. It is an INTENT(IN) argument.

  **Example.**
  
  ```
  TYPE (CSR) :: A_CSR
  TYPE (MSR) :: A_MSR
  
  A_CSR = A_MSR
  ```
Chapter 4

Approximate Factorisation

As noted in Sec. 5.3 about Krylov subspace methods, approximate matrix factorisations are important for the preconditioning of the equation system. If we approximate the equation matrix $A$ with an $LU$ factorisation, the solution of $(LU)x = y$ is easily obtained by successively applying a forward and a backward substitution. For a good preconditioner, $LU$ should approximate $A$ well. However, for the substitution to be computationally cheap, $L$ and $U$ should also be sparse.

Matrix factorisations are frequent tools in numerical analysis. It is, of course, possible to use the abstract data types for the sparse matrices to hold each factor in a factorisation. However, we often know additional information about a factorisation than just what the factors are. E.g., we could have two, up to transpose, equal factors $A = L L^T$. To effectively store such information, we make abstract data types for the factorisation. Furthermore, this abstraction makes the programs more pleasant to read.

4.1 Abstract Data Type for Sparse Matrix Factorisations

For a general sparse matrix factorisation, we may use the data types CSR or MSR to hold the factors. We choose to store $L$ in the CSR format, and we do not store the main diagonal, which is all ones. For $U$ we choose the MSR format. To define a factorisation of this type, issue a command of the form

\[
\text{TYPE}(\text{LU}, \text{CSR}, \text{MSR}) :: \text{LU}
\]

To make the module available separately, use the module \text{LU\_CSR\_MSR\_MODULE}, but it will also be available by the command

\text{USE Matrix\_Arithmetic\_Module}

\textbf{Procedures}

- \text{Nullify\_Matrix} (LU)

\textbf{Description.} Since Fortran 90 pointers have an initial association status that is undefined, this procedure must be run to guarantee that the memory allocation checks will perform correctly. \textit{For safety, it should be called for all LU\_CSR\_MSR matrices in the beginning of a program.}
4.1. ABSTRACT DATA TYPE FOR SPARSE MATRIX FACTORISATIONS

**Argument.** LU is a matrix factorisation of LU_CSR_MSR type. It is an INTENT(IN) argument.

**Example.** CALL Nullify_Matrix (LU).

- **Allocate_Matrix** (LU, N, LNZMAX, UNZMAX)

  **Optional Arguments.** LNZMAX, UNZMAX.

  **Description.** Allocates memory sufficient to store LNZMAX elements for the lower triangular CSR matrix, and UNZMAX elements for the upper triangular MSR matrix. The matrices are of shape N by N. If the optional arguments are omitted, a default of 50 will be used for these.

  **Arguments.**
  
  LU is a matrix factorisation of LU_CSR_MSR type.
  
  N is of type INTEGER. It is an INTENT(IN) argument.
  
  LNZMAX is of type INTEGER. It is an INTENT(IN) argument.
  
  UNZMAX is of type INTEGER. It is an INTENT(IN) argument.

  **Examples.**
  
  ```fortran
  N = 10
  CALL Allocate_Matrix (LU, N)
  CALL Allocate_Matrix (LU, 100, LNZMAX = 1000, 500)
  ```

- **Reallocate_Matrix** (LU, LDelta, UDELTA)

  **Optional Arguments.** LDelta, UDelta.

  **Description.** Adds/removes memory sufficient to store LDelta and/or UDelta matrix elements to/from L and/or U respectively. Negative LDelta/UDelta will remove memory from the lower/upper triangular part. If UDelta/LDelta is omitted or set to a negative value that would leave too little memory for the non-zeros, allocated memory will shrink to fit the actual number of non-zeros. It perform a memory copy and may be relatively slow on large matrices.

  **Arguments.**
  
  LU Matrix factorisation of LU_CSR_MSR type.
  
  LDelta is of type INTEGER. It is an INTENT(IN) argument.
  
  UDelta is of type INTEGER. It is an INTENT(IN) argument.

  **Examples.**
  
  ```fortran
  CALL Reallocate_Matrix (LU, 100, 10)
  CALL Reallocate_Matrix (LU, UDelta = 40)
  ```
• Deallocate Matrix (LU)

Description. Frees memory allocated to the matrix factors.

Arguments. LU Matrix factorisation of LU_CSR_MSR type.

Example. CALL Deallocate Matrix (LU)

• Entry_L (LU, i, j)

Description. Returns the value of element \((i, j)\) in the lower triangular factor matrix.

This is not an efficient method to reference the matrix elements, but are included for debugging purposes.

Arguments.

LU Matrix factorisation of LU_CSR_MSR type. It is an INTENT(IN) argument.
i is of type INTEGER. It is an INTENT(IN) argument.
j is of type INTEGER. It is an INTENT(IN) argument.

Example. This code prints the lower triangular factor for a matrix of row dimension less or equal to ten.

```fortran
DO i = 1, S % N
    PRINT "(10F7.2)" , (Entry_L (LU, i, j), j = 1, S % N)
END DO
```

• Entry_U (LU, i, j)

Description. Returns the value of element \((i, j)\) in the upper triangular factor matrix.

This is not an efficient method to reference the matrix elements, but are included for debugging purposes.

Arguments.

LU Matrix factorisation of LU_CSR_MSR type. It is an INTENT(IN) argument.
i is of type INTEGER. It is an INTENT(IN) argument.
j is of type INTEGER. It is an INTENT(IN) argument.

Example. This code prints the upper triangular factor for a matrix of row dimension less or equal to 10.

```fortran
DO i = 1, S % N
    PRINT "(10F7.2)" , (Entry_U (LU, i, j), j = 1, S % N)
END DO
```

• Solve (LU, b, x)

Description. Solves the system of equations \(LUx = b\) by forward and backward substitution.

Arguments.
4.2. INCOMPLETE LU BY LEVEL OF FILL-IN

LU  Matrix factorisation of LU_CSR_MSR type. It is an INTENT(IN) argument.

b  Right hand side vector. It is an INTENT(IN) argument.

x  Unknown vector to solve for. It is an INTENT(OUT) argument.

Example. CALL Solve (LU, b, x)

- is_ok (LU)

Description. Checks for consistency of the data structure in LU.

Arguments. LU is a matrix factorisation of LU_CSR_MSR type. It is an INTENT(IN) argument.

Result. .TRUE. when the structure is consistent, .FALSE. if is not.

Example.

IF (is_ok (LU)) THEN
    CALL Solve (LU, b, x)
ELSE
    PRINT *, 'This matrix factorisation is bogus!'
END IF

4.2 Incomplete LU by Level of Fill-in

To compute sparse LU factors, some form of truncated Gaussian elimination may be used [26, 29, 37]. The code documented in this section is an implementation of the level of fill-in version of the truncated Gaussian elimination method for finding approximate LU factorisations proposed by Watts [37].

For a complete example on the use of these routines, refer to the appendix B. The procedures for incomplete LU factorisation is made available by the command USE ILU_MODULE.

Procedures

- Symbolic_Cancellation (p, S, LU)

Description. The process of finding the correct sparsity pattern for the factors \( L \) and \( U \) is called a symbolic cancellation. This routine finds this pattern for a particular matrix \( S \) and a certain maximal fill-in level \( p \).

On return LU has the correct sparsity pattern for a \( p \) level ILU factorisation of the matrix \( S \). It is not necessary to allocate memory for LU before the call, however, this may give a speed up. Remember to nullify LU in the beginning of your program.

Arguments.

\( p \) is of type INTEGER. It is an INTENT(IN) argument.

\( S \) is a matrix of type MSR. It is an INTENT(IN) argument.

LU is a factorisation of type LU_CSR_MSR. It is an INTENT(OUT) argument.

Example.

CALL Symbolic_Cancellation (p, S, LU)
• ILU (S, LU)

Description. Calculates the incomplete factors $L$ and $U$. The sparsity pattern of $LU$ must be created by Symbolic Cancellation before the call to ILU.

On return $LU$ contains the approximate factors of $S$.

Arguments.

$S$ is a matrix of type MSR. It is an INTENT(IN) argument.

$LU$ is a factorisation of type LU_CSR_MSR. It is an INTENT(INOUT) argument.

Example.

CALL ILU (S, LU)
Chapter 5

Iterative Solution Algorithms

5.1 Common Optional Arguments and Default Values

The following list contains optional arguments that may be used with any of the iterative solvers.

init is of type LOGICAL. If init=.TRUE. appears in the argument list, the values in the array for the indeterminate vector will be used as initial value for the iteration. If init=.FALSE. appears, the zero vector will be used as initial value. Default: init=.FALSE. It is an INTENT(IN) argument.

tol is of type REAL(prec). It defines the stop criterion tolerance. The iteration is terminated if the 2-norm of the residual is less than tol. Default: tol = 0.000001.prec. It is an INTENT(IN) argument.

reduction_factor is of type REAL(prec). It tells the solver to terminate the iteration when the residual is reduced by a certain factor. This option overrides the tol option. Usage e.g.: reduction_factor = 0.4.prec. It is an INTENT(IN) argument.

min_it is of type INTEGER. It defines the minimum number of iterations to perform. Default: min_it = 0. It is an INTENT(IN) argument.

max_it is of type INTEGER. It defines the maximum number of iteration to perform. Default: max_it = 50. It is an INTENT(IN) argument.

INFO is of type INTEGER. It defines the level of information to Show on the terminal during the iteration. INFO = NONE: Show no information (Default), INFO = SOME: Show summary information, INFO = ALL: Show information at every iteration. It is an INTENT(IN) argument.

no_of_its is of type INTEGER. On return, this argument will contain the number of iterations performed. It is an INTENT(OUT) argument.

residual_norm is of type REAL(prec). On return, this argument will contain the 2-norm of the residual. It is an INTENT(OUT) argument.

history is of type TYPE(ConvHist). It holds a data structure that on return will contain information on the iteration history. That is, for each iteration is listed iteration number,
residual and time used. On Cray Y-MP and Cray J90 the number of floating point operations used will be available. See next section for a detailed discussion. Usage e.g.: history = the history. It is an INTENT(OUT) argument.

The default values for the common optional arguments are held by the following variables, which may be changed any time in your program.

**Default_tolerance** Default value for tol (REAL(prec)).

**Default_minimum_iterations** Default value min it (INTEGER).

**Default_minimum_iterations** Default value max it (INTEGER).

**Default_infolevel** Default value info (INTEGER).

**Default_use_init_value** Default value init (LOGICAL).

### 5.2 Iteration History Log

When we study the performance of the iterative methods, it is interesting to follow the residual as a function of number of iterations and/or CPU-time. One way to do this is to send the optional argument INFO = ALL to the solver. But, as printing is a rather expensive operation, or just for avoiding a messy screen, we may want to write out convergence history information at some later instant.

The type definition for such an convergence history is

```
TYPE(ConvHist) :: the_history
```

This data structure should be nullified by CALL Nullify_ConvHist(History) at the beginning of the program. The iterative solver is called with History = the history among its optional arguments.

After the completion of the iterative solver, the history is either printed to screen by CALL Write_ConvHist(the history), to an open file unit by CALL Write_ConvHist(output_unit, the_history), or iteration by iteration to a string by CALL Write_ConvHist(ic, the_history, string).

**Procedures**

- **Nullify_ConvHist (History)**

  **Description.** Nullifies pointers and initialise variables in the history data structure. It should always be called before History is used.

  **Arguments.** History is an unused convergence history of type TYPE(ConvHist).

  **Example.** CALL Nullify_ConvHist (History)
5.2. **ITERATION HISTOR Y LOG**

- **Write_ConvHist** (History), (iounit, History), (ic, History, Line)

  **Optional forms.** Exactly one of the above calls must be chosen.

  **Description.** Writes the contents of a convergence history History. The first form writes to the screen, the second form to a file with output unit number iounit, and the third form write the information at an iteration number ic to a string Line.

  **Arguments.**
  - History is of type TYPE(ConvHist). It is an INTENT(IN) argument.
  - iounit is of type INTEGER. It is an INTENT(IN) argument.
  - ic is of type INTEGER. It is an INTENT(IN) argument.
  - Line is a character string of arbitrary length. It is an INTENT(IN) argument.

  **Example.**
  ```
  INTEGER :: ic
  CHARACTER(LEN=80) :: string
  TYPE(ConvHist) :: history
  
  ! write to screen
  CALL Write_History (history)
  
  ! write to file
  OPEN (8, FILE="output.dat", STATUS="WRITE")
  CALL Write_History (8, history)
  
  ! write to file with additional information
  DO ic = 1, No_of_Its (history)
      CALL Write_History (ic, history, string)
      WRITE (8, '(A)') TRIM(string) // 'Whatever you want'
  END DO
  ```

- **No_of_Its** (History)

  **Description.** Finds the total number of iterations in the convergence history.

  **Arguments.** is of type TYPE(ConvHist).

  **Result.** is of type INTEGER.

  **Example.** tot_ic = No_of_Its(History)

- **Time_ConvHist** (ic, History)

  **Description.** Returns the CPU-time spent by the iteration at some iteration number ic.

  **Arguments.**
  - ic is of type INTEGER. It is an INTENT(IN) argument.
History is of type TYPE(ConvHist). It is an INTENT(IN) argument.

Result. is of type REAL(prec).

Example. cpu_time_spent = Time_ConvHist (i, history)

• Flop_ConvHist (ic, History)

Description. On some Cray machines this function returns the number of floating point operations the iteration has spent at some iteration number ic.

Arguments.

ic is of type INTEGER. It is an INTENT(IN) argument.

History is of type TYPE(ConvHist). It is an INTENT(IN) argument.

Result. is of type INTEGER(flop_int).

Example. flops_spent = Flop_ConvHist (i, history)

• Residual_ConvHist (ic, History)

Description. Returns the residual obtained by the iteration after some number of iterations ic.

Arguments.

ic is of type INTEGER. It is an INTENT(IN) argument.

History is of type TYPE(ConvHist). It is an INTENT(IN) argument.

Result. A residual (REAL(prec)).

Example.

residual = Residual_ConvHist (i, history)

5.3 Krylov Subspace Methods

The discretisation of systems of partial differential equations often leads to very large sparse systems of linear equations,

\[ A x = b, \]

(5.1)

\( A \in \mathbb{R}^{n \times n}, \quad x, b \in \mathbb{R}^n. \) One of the most prominent classes of methods for the solution of these equations, is the Krylov-subspace methods. However, crucial for the efficiency of these methods is the preconditioning of the system (5.1).

If we represent a right preconditioner by the matrix \( M \in \mathbb{R}^{n \times n} \), the Krylov-subspace method should solve the system

\[ A M^{-1} y = b, \quad x = M^{-1} y, \]

(5.2)

instead of system (5.1). The matrix M is often realised in the form of an iterative method, e.g. Jacobi iteration, SSOR, et c., or even a member of the Krylov-subspace methods.

Another common approach, which is the only one implemented in this version of SMLIB, is to express \( M \) as a product \( LU \), where \( L \) and \( U \) are lower and upper triangular matrices respectively. This factorisation may be computed by the procedure in chapter 4 and stored in the factorisation storage scheme from chapter 4.
5.3. KRYLOV SUBSPACE METHODS

- BiCGSTAB \((x, a, b, PC, [\text{common optional arguments}])\)

  **Description.** This routine performs the Bi-Conjugate Gradients Stabilised algorithm (BiCGSTAB) [36] for the iterative solution of a non-symmetric linear system of equations. However, the BiCGSTAB algorithm is known to have breakdown problems particularly in connection with convection dominated flows [33]. A remedy to improve the situation [32] is implemented.

  **Arguments.**
  
  - \(x\): On entry, this array is used as an initial value for the iteration if either `Default_use_init_value` is set to `.TRUE.` or the optional argument `init = .TRUE.` is present. On return it contains the solution. It is a one dimensional array of size \(A \times N\) and type `REAL(prec)`. It is an `INTENT(INOUT)` argument.
  - \(A\): The matrix representing the linear equation system. At present, only the MSR data storage format is implemented. It is an `INTENT(IN)` argument.
  - \(b\): Represents the right hand side of the equation system. It is a one dimensional array of size \(A \times N\) and type `REAL(prec)`. It is an `INTENT(IN)` argument.
  - \(PC\): The preconditioner. At present, only matrix factorisations of the LU, CSR, MSR type are implemented. It is an `INTENT(IN)` argument.

  **Example.**

  ```fortran
  CALL BiCGSTAB (x, a, b, PC, init = .TRUE.)
  ```

- CGS \((x, a, b, PC, [\text{common optional arguments}])\)

  **Description.** This routine performs the Conjugate Gradients Squared algorithm (CGS) [34] for the iterative solution of a non-symmetric linear system of equations.

  **Arguments.**
  
  - \(x\): On entry, this array is used as an initial value for the iteration if either `Default_use_init_value` is set to `.TRUE.` or the optional argument `init = .TRUE.` is present. On return it contains the solution. It is a one dimensional array of size \(A \times N\) and type `REAL(prec)`. It is an `INTENT(IN)` argument.
  - \(A\): The matrix representing the linear equation system. At present, only the MSR data storage format is implemented. It is an `INTENT(IN)` argument.
  - \(b\): Represents the right hand side of the equation system. It is a one dimensional array of size \(A \times N\) and type `REAL(prec)`. It is an `INTENT(IN)` argument.
  - \(PC\): The preconditioner. At present, only matrix factorisations of the LU, CSR, MSR type are implemented. It is an `INTENT(IN)` argument.

  **Example.**

  ```fortran
  CALL CGS (x, a, b, PC, reduction_factor = 0.2_prec)
  ```
• **GMRES (k, x, A, b, PC, [common optional arguments])**

**Description.** This routine performs the truncated generalised minimal residuals algorithm (GMRES(k)) [30] for the iterative solution of a non-symmetric linear system of equations.

**Arguments.**

- **k** is the krylov subspace dimension and must be positive. Typical values are from 1 to 30. Large values of k will improve convergence, but will on average use more CPU-time per iteration and use more memory. It is of type INTEGER. It is an **INTENT(IN)** argument.

- **x** On entry, this array is used as an initial value for the iteration if either **Default_use_init_value** is set to .TRUE. or the optional argument **init** = .TRUE. is present. On return it contains the solution. It is a one dimensional array of size A % N and type REAL(prec). It is an **INTENT(IN)** argument.

- **A** The matrix representing the linear equation system. At present, only the MSR data storage format is implemented. It is an **INTENT(IN)** argument.

- **b** represent the right hand side of the equation system. It is a one dimensional array of size A % N and type REAL(prec). It is an **INTENT(IN)** argument.

- **PC** is the preconditioner. At present, only matrix factorisations of the LU_CSR_MSR type is implemented. It is an **INTENT(IN)** argument.

**Example.**

```fortran
CALL GMRES (k, x, A, b, PC, min_it = 1, max_it = 3*k)
```
Chapter 6

Direct Solver LAPACK Interface

Matrices of interest often have bandwidth that are an order of magnitude smaller than the matrix dimension. For moderate sized matrices, a band Gaussian elimination solver may be a good choice. SMLib offers an interface to the LAPACK FORTRAN77 subroutines needed to use this non-iterative solver.

On many computers, the vendors have provided implementations of LAPACK. In this case, this LAPACK version should be linked into your program at the final loading stage. If LAPACK is not installed on your computer, a public domain version is available at NETLIB\(^1\). There you may find the subroutines SGBFA, DGBFA, SGBSL, and DGBSL which are called by SMLib. Download these routines and the support routines (BLAS; Basic Linear Algebra Subroutines) needed and compile them separately from SMLib. Link the resulting object files into your program at the final loading stage.

6.1 Data Type for Band Matrix Factorisation

The LAPACK band matrix Gaussian elimination procedures GBFA and GBSL use a two-dimensional array abd to hold the matrix coefficients of both the matrix and its factors. Furthermore, a pivot array ipvt is used. For details are referred to the LAPACK documentation. In SMLib this is implemented in the datastructure

```
TYPE Band_LU
    INTEGER :: N, Upper_Band_Width, Lower_Band_Width
    REAL(prec), DIMENSION(:,:), POINTER :: abd
    INTEGER, DIMENSION(:,), POINTER :: ipvt
END TYPE Band_LU
```

It is not intended that the user of SMLib shall modify any component of this datastructure by himself. The matrix to solve should be in MSR format, and the conversion to a banded LU format is hidden.

Procedures

- **NullifyMatrix (LU)**

\(^1\)An internet URL for NETLIB is http://www.netlib.no/.
CHAPTER 6. DIRECT SOLVER LAPACK INTERFACE

Description. Since Fortran 90 pointers have an initial association status that is undefined, this procedure must be run to guarantee that the memory allocation checks will perform correctly. It nullifies LU % abd and LU % ipvt and sets LU % N, LU % Lower_Band_Width, and LU % Upper_Band_Width to zero. For safety, it should be called for all Band_LU matrix factorisations in the beginning of a program.

Argument. LU is of Band_LU type. It is an INTENT(INOUT) argument.

Example. CALL Nullify_Matrix (LU).

Allocate_Matrix (LU, Lower_Band_Width, Upper_Band_Width, N)

Description. Allocates memory sufficient to a matrix with of dimension N × N with specified lower and upper bandwidth.

Arguments.
  LU is a matrix factorisation Band_LU of type.
  Lower_Band_Width is of type INTEGER. It is an INTENT(IN) argument.
  Upper_Band_Width is of type INTEGER. It is an INTENT(IN) argument.
  N is of type INTEGER. It is an INTENT(IN) argument.

Examples.
  N = 10
  CALL Allocate_Matrix (LU, 4, 3, N)

Deallocate_Matrix (LU)

Description. Frees memory allocated to the matrix and sets LU % N, LU % Lower_Band_Width, and LU % Upper_Band_Width to zero.

Arguments. LU is a matrix of Band_LU type.

Example. CALL Deallocate_Matrix (LU)

allocated_matrix (LU)

Description. Returns .TRUE. if the matrix is properly allocated, and .FALSE. if not. Never call this routine if the matrix allocation status is undefined, that is, there is no prior call to Allocate_matrix or Nullify_Matrix.

Arguments. LU is a matrix of Band_LU type.

Example. IF (.NOT.allocated_matrix(LU) CALL ALLOCATE_MATRIX(LU, 4, 3, 10)

6.2 Interfacing the LAPACK Band Gauss Solver

The SMLib Fortran 90 interface to LAPACK’s band matrix Gaussian elimination procedures consists of two subroutines. Firstly, LU_factor calls the factorisation routine GBFA, and secondly, Solve calls the forward and backward substitution routine GBSL to obtain the solution.
6.2. INTERFACING THE LAPACK BAND GAUSS SOLVER

Procedures

• LU_factor (S, LU, inform)

Optional Argument. inform.

Description. The routine finds the band widths for the sparse matrix S. It then set up
the datastructure for LU, and call LAPACK’s GBFA to find the factors by banded
Gaussian elimination. Remember to nullify LU in the beginning of your program.

Arguments.

S is a matrix of type MSR. It is an INTENT(IN) argument.
LU is a factorisation of type Band_LU. It is an INTENT(OUT) argument.
inform equals LAPACK’s info variable from the routine GBFA. It is of type IN-
TEGER.

Example.

CALL LU_factor(S, LU, info)

• Solve (LU, b, x)

Description. Solves the equation system LUx=b where LU has been found previously
by a call to LU_factor. On return x contains the solution.

Arguments.

LU is a factorisation of type Band_LU. It is an INTENT(IN) argument.
b is a one-dimensional array of type REAL(prec). It is an INTENT(IN) argument.
x is a one-dimensional array of type REAL(prec). It is an INTENT(OUT) argu-
ment.

Example.

TYPE(MSR) :: S
TYPE(Band_LU) :: LU
REAL(prec), DIMENSION(:), ALLOCATABLE :: x, b

CALL Nullify_Matrix(S) CALL Nullify_Matrix(LU)
CALL LU_factor(S, LU)
CALL Solve(LU, b, x)
Appendix A

Brief Introduction to Krylov-Subspace Methods

The equations solved by SMLib may be formulated as

\[ \mathbf{A} \mathbf{x} = \mathbf{b} \]  

(A.1)

with \( \mathbf{A} \in \mathbb{R}^{n \times n} \) and \( \mathbf{x}, \mathbf{b} \in \mathbb{R}^n \). Proofs for convergence of iterative methods for the solution of Eq. (A.1) usually require the matrix \( \mathbf{A} \) to be symmetric, definite, and/or diagonal dominant, and to ensure the existence of an ILU factorisation the matrix has to be an \( M \)-matrix (see [29, p. 103]). Unfortunately real life matrices will generally not be of any of these types. What we do know about them is their sparsity patterns. Further, since the matrices often result from the discretisation of elliptic differential equations, we often know the diagonal to be non-zero. These facts we will of course try to exploit.

All methods implemented are targeted on the solution of a system of linear equations to a high accuracy, and to get this high accuracy at a low cost is indeed the quest of most authors on this subject. However, the ultimate goal for the use of SMLib is commonly to solve a set of non-linear equations, and SMLib is then used to repeatedly solve a set of linearised equations. To this end we may not need high-accuracy solutions of the linearised equations, and the method’s behaviour within the first few iterations is of particular importance. Furthermore, the initial guesses will in some cases be rather close to the solution, so it is vital that the method do not loose performance as only the low frequency errors remain.

The class of Krylov subspace solvers is well suited for such purposes. They typically perform better as the residual gets smaller, and given sufficient preconditioning, convergence is good during the first few iterations as well. Furthermore, they are easily implemented for unstructured sparse matrices. The weak point lie in the need for a good preconditioner. The incomplete factorisations implemented in SMLib has very good properties as preconditioners, but are quite complex. Incomplete factorisations have significant overhead in terms of CPU-time and memory storage. However, once implemented, ILU(\( p \)) is applicable, without modification, to a very wide range of problems.

For supercomputing purposes, it might be argued that these incomplete factorisations are poorly parallelisable. However, a good parallel code keeps interprocessor communication at a minimum. That is, the problem has to be divided into quite large parts. For this purpose

\footnote{An \( M \)-matrix satisfies \( a_{ij} \leq 0 \) for \( i \neq j \), and is non-singular with \( (\mathbf{A}^{-1})_{ij} \geq 0 \) for all \( (i, j) \).}
the domain decomposition methods are well suited [24]. Incomplete factorisations may then be used on each subdomain, which is run serially on one processor.

The Krylov subspace methods are roughly said to be of $O(N^{1.25})$ in complexity. In this respect, the Krylov subspace methods do not match the multigrid methods which optimally perform at $O(N)$. This has spurred large activity in the application of multigrid solvers. However, the mere task of implementing a multigrid solver on an unstructured grid is formidable, however attempts are made (see [19] for a review). Furthermore, great care has to be taken in the transfer of the variables between the meshes to achieve near optimal convergence. For this reason, multigrid methods are not considered here.

## A.1. Krylov Subspace Methods

To find an approximation $x_m$ to the solution $x$ of Eq. (A.1), we estimate $A^{-1}$ by a polynomial $q_{m-1}$ of degree $m - 1$. For the sake of simplicity, we choose an initial value $x_0 = 0$. This gives us

$$A^{-1} b \approx x_m = q_{m-1}(A) b$$

where $q_{m-1}$ should be good in some sense. This polynomial is recursively generated. That is, initially we use the zero'th order polynomial $q_0 = 1$. The higher order polynomials are on the form

$$q_m = p_m q_{m-1}$$

where $p_m$ usually is a first order polynomial, determined by the choice of method, which we will discuss later. For each new polynomial the method generates, we get a new approximation $x_m$. This amounts to one iteration of the algorithm.

If we spell out the expression for $x_m$, we get

$$x_m = b + c_1 A b + c_2 A^2 b + \ldots + c_{m-1} A^{m-1} b$$

where at least $c_{m-1} \neq 0$. Hence, the approximation $x_m$ is a linear combination of vectors from $\mathbb{R}^m$, and $x_m$ is found in the $m$ dimensional subspace

$$x_m \in \text{span } \{ b, A b, A^2 b, \ldots, A^{m-1} b \} \overset{\text{def}}{=} K_m(A, b)$$

of $\mathbb{R}^n$. This is called the $m$-dimensional Krylov subspace generated by $A$ and $b$. If $A$ and $b$ is understood, we will use the notation $K_m$. The Krylov subspace methods operates in this space, and extension to a higher dimension equals one iteration. However, the polynomials $q_{m-1}$ are not explicitly generated.

As $m$ increases, $K_m$ will fill out more of $\mathbb{R}^n$ (with exception of special cases where the process breaks down), and for $m = n$ we have $K_m = \mathbb{R}^n$. From Eq. (A.4) we see that the set $\{ x_m \}$ is linear independent, and hence, $x_n = x$.

That is, Krylov subspace methods theoretically converge after at most $n$ iterations. This is however valid only in exact arithmetics. In practical applications, rounding errors will destroy the linear independence for large $n$. What makes the methods attractive is that the

$^2$A complexity of $O(N^x)$ signifies that the work needed to obtain a solution increases with the number of unknowns $N$ to the power of $x$. 

37
number of iterations to reach a sufficiently accurate $x_m$ is usually considerably smaller than $n$.

Nevertheless, the early methods, like Hestenes and Stiefel's [13] conjugate gradients from 1952, was primarily viewed upon as direct solvers. In fact, Hestenes and Stiefel proved both conjugate gradients and Gaussian elimination to be special cases of the very general method. The possibility of applying the method as an iterative solver was recognised at this early stage, but the results was too poor to spur any large activity. It was not until Reid [25] in 1971 pointed out conjugate gradient's efficiency on large, sparse, and well conditioned systems, the whole topic lifted off.

The challenge in constructing a Krylov subspace method is to select an, in some sense, best possible $x_m$ in $K_m$. Different strategies for this selection have given rise to a myriad of methods. However, two main strategies exist, and all methods convail either one or both. Furthermore, the need for a well conditioned system of equations make the use of preconditioners necessary. In the algorithms presented in this section the preconditioner is simply represented by a matrix $M$. A more detailed discussion is left to Sect. A.2.

For the work presented in this thesis, several Krylov subspace methods was considered for the solution of the linear equations appearing in an incompressible Navier-Stokes solver. The basic principles behind these Krylov subspace solvers will be reviewed in the next sections, and pseudo-code will be presented for all but one. However, as no algorithmic development has been attempted in the current work, a rigid treatment of each method is considered to be outside the scope of this text. Hence, for the finer details, the reader is referred to the literature.

A.1.1 Strategy 1 − Minimising the Norm over $K_m$

The first main strategy is to choose $x_m$ so that either

1. $r_m \perp K_m$. For symmetric and positive definite $A$ this is equivalent with minimising
   $$\|x-x_m\|_A$$
   with $x_m \in K_m$.

2. $r_m \perp A K_m$. This is equivalent to minimising the residual norm $\|r_m\|_2$ with $x_m \in K_m$.

where $r_m$ equals the residual $b-Ax_m$ of Eq. (A.1). The orthogonality conditions are called Galerkin and Petrov-Galerkin conditions respectively. These orthogonality conditions are very general mathematical constructions and may be found in several branches of mathematics, most notably perhaps in finite element theory.

Even though the minimising in condition 1 is valid for symmetric and positive definite matrices only, it is used for general matrices as well. The convergence is then assured by the fact that the iterates achieved by the application of condition 1 is proportional to the iterates achieved by the application of condition 2.

Some algorithms based on condition 1 are Conjugate Gradients (CG), Generalised Conjugate Gradients (GENCG), Full Orthogonalisation Method (FOM), and ORTHORES. Condition 2 lead to the algorithms Generalised Conjugate Residuals (GCR), ORTHOMIN, ORTHODIR, one more Generalised Conjugate Residuals (GENCG), and Generalised Minimal Residuals (GMRES).

A very appealing property of these methods is that the residual is reduced in some norm for every iteration. This is not the case for methods using the next main strategy. Unfortunately, a condition for this is that all basis vectors of $K_m$ is stored. As the number of iterations
increase, this become very costly, primarily in terms of computer storage, but also in terms of numerical calculations since new basis vectors has to be found orthogonal to all the previous ones.

To avoid this aggregating number of basis vectors, the algorithm is either truncated or restarted. If the algorithm is truncated, only the last \( m \) basis vectors of \( K_m \) is saved. If the algorithm is restarted, it is restarted after \( m \) iterations. The solution then achieved is fed into a new run of the algorithm as the initial value. E.g., ORTHOMIN(\( k \)) is a truncated method, whereas GMRES(\( m \)) is a restarted method.

Judging from the literature, GMRES(\( m \)) [30] seems to be the most robust of these methods. Like several other methods, it is headed directly towards the minimising of the residual at each iteration. To begin with, GMRES(\( m \)) constructs a more convenient basis for the Krylov subspace than the one in the definition (A.5). An orthonormal basis for \( K_m \) is generated by an Arnoldi process, which is in turn based on Gram-Schmidt orthogonalisation. For the sake of numerical robustness modified Gram-Schmidt is used. The Arnoldi process generates a basis for the Krylov subspace that makes it possible to formulate the minimising of the residual as a solution to a least squares problem where the defining matrix \( H \in \mathbb{R}^{m+1 \times m} \) has been obtained during the Arnoldi process. The matrix \( H \) is kept upper triangular by performing plane Givens rotations on it at each Arnoldi process iteration step. The solution of this upper triangular form of the least squares problem is found at very low computational cost.

Finally, we note that when the matrix \( A \) is symmetric, the GMRES orthogonalisation process will simplify. The new vector needs only be calculated orthogonal to the two last vectors in order to be orthogonal to the full set of basis vectors. If this simplification is applied together with condition 1, the classical conjugate gradient method for symmetric, positive definite matrices is the result.

GMRES is the most complicated method to implement of the Krylov subspace methods considered in this thesis. A pseudo-code listing of the algorithm has therefore not been included, and the reader is referred to Saad [29] for implementational details. A possible alternative to GMRES is the older GCR method as developed by Elman [6]. GCR is later proved [30] to be equivalent in exact arithmetic to the far more popular GMRES method, but GMRES is claimed to be more robust under finite precision arithmetic. GCR is however much more straightforward to implement than GMRES.

**Algorithm A.1. Restarted Generalised Conjugate Residuals – GCR(\( \ell \))**

\[
\begin{align*}
\mathbf{x}_0 & \text{ is an initial guess; } \mathbf{r}_0 = \mathbf{b} - \mathbf{A} \mathbf{x}_0; \\
\text{Solve } \mathbf{r}_1 \text{ for } \mathbf{M} \mathbf{r} = \mathbf{b} - \mathbf{A} \mathbf{x}_0 \\
m & = 0 \\
\text{For } i = 1, 2, \ldots \\
m & = m + 1 \\
\text{Solve } \mathbf{u} \text{ for } \mathbf{M} \mathbf{u} = \mathbf{A} \mathbf{r}_i \\
\beta_j & = (p_j, \mathbf{u}) \phi_j \text{ for } j = 1, \ldots, m - 1 \\
\mathbf{q}_m & = \mathbf{r}_i - \sum_{j=1}^{m-1} \beta_j \mathbf{q}_j \\
\mathbf{p}_m & = \mathbf{u} - \sum_{j=1}^{m-1} \beta_j \mathbf{p}_j \\
\phi_m & = 1/(\mathbf{p}_m^T \mathbf{p}_m) \\
\alpha & = (\mathbf{r}_i, \mathbf{p}_m) \phi_m \\
\mathbf{x}_i & = \mathbf{x}_{i-1} + \alpha \mathbf{q}_m
\end{align*}
\]
If $x_i$ is accurate enough then quit
\[ r_i = r_{i-1} - \alpha p_m \]
If $m = \ell + 1$ then set $m = 0$
End For

Note that when $m = 1$, the convention $\sum_{j=1}^{0} a_j = 0$ is used. The residual $r$ in GCR($\ell$) is the residual of the preconditioned system, which is usually much less than the residual of the original system. If the residual should be used in a stopping criterion, the real residual should be calculated as $\|b - A x_i\|$.

A.1.2 Strategy 2 – Bi-orthogonality

For this class of methods the Petrov-Galerkin condition
\[ r_m \perp K_m(A^T, \hat{b}), \tag{A.6} \]
where $\hat{b}$ is an auxiliary vector, is used to select the approximation $x_m \in K_m(A, b)$. For $A$ symmetric and $\hat{b} = b$ this condition equals condition 1 from strategy 1, from which it is originally motivated. The Petrov-Galerkin condition (A.6) has however no minimising property, and the error may increase from one iteration to the next.

In addition to the generation of the vectors $\{r_m\}$, the vectors $\hat{r}_m \perp K_m(A, b)$ are generated with the same polynomial as for the $r_m$. It is then possible to show that

1. $(r_i, \hat{r}_j) = 0$ for $i \neq j$ (bi-orthogonality), and that

2. $r_{j+1}$ dependent upon only $r_j$ and $r_{j-1}$.

It is hence not necessary to store a load of basis vectors, and this is the most immediate advantage with this strategy.

A straightforward implementation of the above conditions together with $\hat{b} = b$ is the bi-conjugate gradient method [7] (Bi-CG). A further development of the method, however, give a more interesting method for practical purposes. The residual of Bi-CG has the form
\[ r_m = b - A x_m = b - A q_{m-1}(A) b = p_m(A) b. \tag{A.7} \]

Furthermore, for $\hat{r}_m$,
\[ \hat{r}_m = p_m(A^T) b. \tag{A.8} \]

This yields
\[ (r_i, \hat{r}_j) = (p_i(A) b, p_j(A^T) b) = (p_j(A) p_i(A) b, b) = 0 \tag{A.9} \]
for $j < i$. The iteration parameters of Bi-CG are calculated from such inner products. However, Sonneveld [34, 35] observed that it is possible to construct residuals on the form $\hat{r}_j = p_j(A) b$ by using only the last form of the inner product. This way he avoided the construction of the vectors $\hat{r}_m$ and the multiplication with the transpose of $A$.

The resulting method, conjugate gradients squared (CGS), work generally very well for many asymmetric linear problems, and it often converges much better than Bi-CG (up to twice as fast). However, CGS may converge very irregular. This behaviour may even lead to cancellation and destroyed solution.
A.1. KRYLOV SUBSPACE METHODS

Algorithm A.2. Conjugate Gradients Squared – CGS

\[ \mathbf{x}_0 \text{ is an initial guess; } \mathbf{r}_0 = \mathbf{b} - \mathbf{A} \mathbf{x}_0; \]
\[ \mathbf{r}_0 \text{ is an arbitrary vector, such that } (\mathbf{r}, \mathbf{r}_0) \neq 0, \text{ e.g. } \mathbf{r}_0 = \mathbf{r}_0 \]
\[ \rho = 1; \quad \mathbf{p} = \mathbf{q} = \mathbf{0} \]

For \( i = 1, 2, \ldots \)

\[ \rho' = (\mathbf{r}_0, \mathbf{r}_i) \]
\[ \beta = \rho'/\rho; \rho = \rho' \]
\[ \mathbf{u} = \mathbf{r}_i + \beta \mathbf{q} \]
\[ \mathbf{p} = \mathbf{u} + \beta (\mathbf{q} + \beta \mathbf{p}) \]

Solve \( \mathbf{v} \) for \( \mathbf{M} \mathbf{v} = \mathbf{p} \)
\[ \mathbf{v} = \mathbf{A} \mathbf{v} \]
\[ \alpha = \rho/(\mathbf{r}_0, \mathbf{v}) \]
\[ \mathbf{q} = \mathbf{u} - \alpha \mathbf{v} \]

Solve \( \mathbf{z} \) for \( \mathbf{M} \mathbf{z} = \mathbf{u} + \mathbf{q} \)
\[ \mathbf{z} = \alpha \mathbf{z} \]
\[ \mathbf{x}_i = \mathbf{x}_{i-1} + \mathbf{z} \]

If \( \mathbf{x}_i \) is accurate enough then quit
\[ \mathbf{r}_i = \mathbf{r}_{i-1} - \mathbf{A} \mathbf{z} \]

End For

There also exists another group of methods belonging to strategy 2. We may use a Petrov-Galerkin condition relating to condition (A.6) as condition 1 related to condition 2 in main strategy 1,

\[ \mathbf{r}_m \perp \mathbf{A} K_m(\mathbf{A}^T, \mathbf{b}). \quad \text{(A.10)} \]

Using this condition, with an orthogonalisation similar to the one found in GMRES, and a quasi-minimising, the quasi-minimal residuals algorithm [9] (QMR) appear. This algorithm may be developed further, much in the same lines as from Bi-CG to CGS, to give version free of multiplication with the transpose of \( \mathbf{A} \). The method is called the transpose free quasi-minimal residuals [8] (TFQMR). These methods have not been tested in the present work.

A.1.3 Hybrid Methods – Strategies 1 and 2 Combined

In addition to the methods that stick to one of the main strategies, there are methods that combine the two. In the stabilised bi-conjugate gradients [36] (BiCGSTAB), a residual of the form \( \mathbf{r} = \mathbf{p}_j(\mathbf{A}) \mathbf{p}_j(\mathbf{A}) \mathbf{b} \) is used instead of the square used in CGS. The polynomials \( p_j \) are the same as those calculated by Bi-CG. However, the polynomials \( \mathbf{p}_j \) are chosen as to minimise the residual. This is done by selecting a polynomial from a residual minimising method (strategy 1, condition 2), e.g. GMRES.

In this text, we take little care about the finer details of breakdown situations for Krylov subspace methods. However, the BiCGSTAB algorithm is known to have breakdown problems particularly in connection with convection dominated flows [33]. This is indeed our experience too, and we therefore use a modified formula for \( \omega \) suggested by Sleijpen and Van der Vorst [32].
Algorithm A.3. Bi-Conjugate Gradients Stabilised – BiCGSTAB

\( x_0 \) is an initial guess; \( r_0 = b - Ax_0 \)
\( \tilde{r}_0 \) is an arbitrary vector, such that \( (r_0, \tilde{r}_0) \neq 0 \), e.g., \( \tilde{r}_0 = r_0 \)
\( \rho = \alpha = \omega = 1 \); \( v = p = 0 \)

For \( i = 0, 1, 2, \ldots \)

\( \rho' = (\tilde{r}_0, r_i) \)
\( \beta = \alpha (\rho' / \rho) / \omega; \rho = \rho' \)
\( p = r_i + \beta (p - \omega v) \)
Solve \( y \) from \( M y = p \)
\( v = Ay \)
\( \alpha = \rho / (\tilde{r}_0, v) \)
\( s = r_i - \alpha v \)
Solve \( z \) from \( M z = p \)
\( t = Az \)
\( c = (s, t) / (||s|| ||t||) \)
\( \omega = \text{sign}(c) \max(|r|, 0.7) ||s|| / ||t|| \)
\( x_i = x_{i-1} + \alpha y + \omega z \)
If \( x_i \) is accurate enough then quit
\( r = s - \omega t \)

End For

BiCGSTAB may be viewed as a product between Bi-CG and GCR(1). The study of such products are a field of great activity. This has resulted in methods like QMR, which is a product between QMR and GMRES(1), and to BiCGSTAB(\( \ell \)) \cite{31} which is a product between Bi-CG and GCR(\( \ell \)).

A.2 Preconditioning

It is well known that preconditioning is of crucial importance for the efficiency of Krylov subspace methods. That is, we modify our Eq. (A.1) by a matrix \( M \) in an attempt to reduce the spectral radius of the system, and hence increase convergence speed. There are basically three strategies for applying \( M \) as a preconditioner:

Left preconditioning \( M^{-1} A x = M^{-1} b \)

Right preconditioning \( A M^{-1} y = b \) with \( x = M^{-1} y \)

Two-sided preconditioning \( L^{-1} A U^{-1} y = L^{-1} b \) with \( M = L U \) and \( x = U^{-1} y \)

In this work, the quite common right preconditioning is used.

One way of implementing the preconditioner is to calculate one of the above systems, use a Krylov subspace method on the transformed system, and, in the end, transform the answer back to the original system. This approach is a waste of calculations. To compute a matrix product like \( B = M^{-1} A \) we will need to solve the \( n \) systems \( M b_{i,j} = a_{i,j} \).

If instead, we implement the preconditioned systems into the Krylov subspace methods, we get statements like \( w = M^{-1} A v \) to be computed a few times every iteration. As the
number of iterations is typically much smaller than $n$, we get away with a number of solves of $Mw = Av$ that is much smaller than $n$.

A good preconditioner $M$ should satisfy the following points:

1. $M$ should be a good approximation to $A$.
2. $M$ should be inexpensive to construct.
3. $M$ should be sparse to save memory.
4. Equations with $M$ as coefficient matrix should be efficiently solved in $O(N)$ operations.

Note that the first of these points as in opposition to the others. We remember from Sect A.3 that the matrix $M$ from a matrix splitting method is designed to cluster the spectral radius of $M^{-1}A$ around 1. Furthermore, these matrices are usually cheap to store and easy to invert. This is properties we evaluate for a preconditioner. It is therefore very common to use matrices from these methods as preconditioners, e.g. Jacobi iteration or symmetric successive over-relaxation (SSOR)\(^3\) [10]. Note that we do not necessarily need to explicitly formulate the matrix $M$. It may very well be implicitly defined by the application of an iterative method to $Mw = Av$. It is even proposed to use another Krylov subspace method as a preconditioner [28]. Furthermore, it has been noted that the viewpoint of the Krylov subspace method being preconditioned by a matrix splitting method may very well be turned around to view the Krylov subspace method as an accelerator for the matrix splitting method [35]. This reversed viewpoint may be beneficial for the implementation of Krylov subspace methods into existing computer codes.

For non-symmetric non-definite matrices, it is far from obvious what preconditioner to choose. A rule of thumb may be that the better reputation a matrix splitting method has in it's own right, the better we may hope for it to work as a preconditioner. As SIP is a well reputed solver for fluid dynamical problems we have chosen to take a closer look at it and it's antecessors. The use of SIP as a preconditioner is also discussed by Jacobs [14, 15], Khosla and Rubin [18], and Berg [3]. Both Jacobs and Berg uses full cancellation, that is $\alpha = 1$.

### A.3 Matrix Splitting Methods

Following Johnson and Riess [16] we obtain the iterative method by splitting the matrix $A$ into $A = M - S$. We assume $M$ to be non-singular. Equation (A.1) may then be written as

$$Mx = Sx + b.$$  \hfill (A.11)

From this we create an iterative method by letting the right hand side $x$ be an old or guessed value, and the left hand side $x$ be the new value. Hence, we create a sequence of iterates $\{x^k\}$ by

$$Mx^{k+1} = Sx^k + b$$  \hfill (A.12)

\(^3\)But not SOR. To quote the Templates [2, p. 41], “The SOR and Gauss-Seidel matrices are never used as preconditioners, for a rather technical reason. SOR-preconditioning with optimal $\omega$ maps the eigenvalues of the coefficient matrix to a circle in the complex plane; see Hageman and Young [12, §9.3]. In this case no polynomial acceleration is possible, i.e., the accelerating polynomial reduces to the trivial polynomial $P_n(x) = x^n$, and the resulting method is simply the stationary SOR method."
which obviously satisfies the original system if the sequence converges. For the iteration (A.12) to be practical, it must be “easy” to solve a linear system with $M$ as the matrix.

All classical iteration methods belong to this group of methods, e.g., Jacobi iteration, Gauß-Seidel iteration, and Successive Over-Relaxation (SOR). More recent are the block correction methods like alternating direction iteration (ADI) [23] and Line Gauß-Seidel (LGS). Even some multigrid methods may be viewed in this context. In these latter methods the splitting is rarely formulated explicitly. But, the alternating directions performed may be viewed as a reordering of the equation system. This reordering is formulated in matrix context as permutation matrices. E.g., the LGS method may be formulated as $M = \sum_{i=1}^{4} M_i P_i$ where the $P_i$’s see to the reordering accordingly to up, down, left, and right sweeps, and the $M_i$s are the respective block Gauß-Seidel splitting matrices. The formulation of this matrix $M$, however, gets more complex than is enjoyable, and hence the usual formulation is certainly preferable (see e.g., Patankar [22]).

Whether or not (A.12) converges to $x = A^{-1} b$ depends upon the eigenvalues of $M^{-1} S$. In particular, the necessary and sufficient condition for this iteration to converge is that the spectral radius $\rho(M^{-1} S)$ has to be less than one (for a proof see [10, p. 508]).

Furthermore, the smaller $\rho(M^{-1} S)$ is, the faster is the convergence. In this sense we may say that we want $\rho(M^{-1} S) \approx 0$. For the eigenvalues of $M^{-1} S$ we have

$$M^{-1} S x = \lambda x$$

which by use of the splitting turns into

$$M^{-1} (M - A) x = I x - M^{-1} A x = \lambda x$$

$$\downarrow$$

$$M^{-1} A x = (I - \lambda I) x = (1 - \lambda) x.$$  

Hence, the eigenvalues of $\rho(M^{-1} A)$ should be clustered around 1, and the success of a splitting method is therefore dependent on how close $M^{-1} A$ approximates $I$. Or better put, how well $M^{-1}$ approximates $A^{-1}$. From this we formulate the following heuristic: *for a successful splitting-method the matrix $M$ should not deviate too much from $A$.*

A convenient reformulation of the iteration (A.12) appears if we substitute with $x^{k+1} = x^k + \delta^k$. We then get

$$M \delta^k = b + S x^k - M x^k = b - A x^k$$

by the definition of the matrix splitting (A.11). With the residual defined as $r^k = b - A x^k$ we now get the following iteration,

$$M \delta_k = r^k$$

$$x^{k+1} = x^k + \delta^k$$

which we will call a *delta formulation*. Other names on this formulation is deferred correction or iterative defect correction (see e.g., Böhmer et.al. [4]). However these names are often found in other contexts.

Note that the matrix $S$ is not explicit in the iteration (A.18).

---

4Let $\lambda(G)$ denote the set of all eigenvalues of an $n \times n$ matrix $G$. Then, the spectral radius of $G$ is defined by $\rho(G) = \max \{|\lambda| : \lambda \in \lambda(G)|$.}
A.3. MATRIX SPLITTING METHODS

### A.3.1 Incomplete LU Factorisation

One might imagine to use Gaussian elimination to find the factors $L$ and $U$ of $M$. The iterations (A.12) or (A.18) would then converge after one iteration. This, of course, would be equivalent to the use Gaussian elimination as a direct solver on $Ax = b$ which is well known to demand a prohibitive amount of work for large equation systems.

Watts [37] therefore proposed to truncate the elimination process. The Gaussian elimination induces new elements, denoted as fill-in elements, in the $L$ and $U$ factors. That is, elements with indices not in $\text{NZ}(A)$. The truncation is done by ignoring some of these new elements, and the tool we will use for selecting which elements to discard is the notion of the level of an element’s index. This method is not used much as a solver, but is mostly viewed as a preconditioner for other methods.

In Gaussian elimination, an element $a_{ik}$ in the lower triangular part is zeroed by the product $l_{ik}u_{kk}$. This elimination will however create fill-ins of the form $l_{ik}u_{kj}$ (see algorithm A.4 below). We let indices in $\text{NZ}(A)$ be of level zero and define the level of fill-in as

$$\text{lev}(i, j) = \text{lev}(i, k) + \text{lev}(k, j) + 1.$$  \hspace{1cm} (A.19)

when a new fill-in element is created or the level is previously set to a larger value.

Typically, for diagonal dominant matrices, the higher the level of fill-in of an element, the smaller its magnitude, and this suggests dropping any fill-in element whose level is higher than a certain integer $p$. However, for more general matrices, the size of an element is not necessarily related to it’s level of fill-in. Stated below is a variant of ILU(p).

**Algorithm A.4. Incomplete LU factorisation – ILU(p)**

1. Define $u_{ij} = a_{ij}$ and for all non-zero elements $a_{ij}$ set $\text{lev}(i, j) = 0$.
2. For $i = 2, \ldots, N$ do
   - For each $k = 1, \ldots, i - 1$ and if $u_{ik} \neq 0$ do
     - Compute $l_{ik} = u_{ik}/u_{kk}$.
     - Set $u_{ik} \leftarrow u_{ik} - l_{ik}u_{kk} \equiv 0$.
     - For $j = k + 1, \ldots, n$ and if $u_{kj} \neq 0$ do
       - Compute $\text{lev}(i, j)$ using (A.19).
       - if $\text{lev}(i, j) \leq p$ Compute $u_{ij} \leftarrow u_{ij} - l_{ik}u_{kj}$.
     - End For
   - End For
End For

One drawback of this algorithm is that the computational work for $p > 0$ is generally not predictable. But, most notable are the problems inherited from the Gaussian elimination for non diagonal-dominant matrices, with the possible growth of elements with $p$. The remedy in full Gaussian elimination for this problem is the use of pivoting, and pivoting versions of the incomplete elimination is also found in the literature [26]. Another, more recent, strategy to avoid discarding large elements is the ILU approach [29]. Here the size of the fill-in elements are checked, and only elements below some threshold is zeroed.

The ILU(0) is a very popular preconditioner due to its simplicity and efficient factorisation. However, as shown by Joly and Eymard [17], it may be well worth the effort to include more fill-ins. Using ILU(p) to precondition GMRES in reservoir simulations they found $p = 2$ to be optimal. Watts [37] himself, also doing reservoir simulations, set $p = 6$ in 2-dimensional
simulations and found this to be a better preconditioner than SIP. Watts was however in a quite particular situation since he was using Hestenes and Stiefels [13] classical conjugate gradient method, and hence had to put a major effort in making his matrices symmetric and positive definite.

Incomplete versions of other classical factorisations from the linear algebra theory has been used too. Most notably is perhaps the incomplete Cholesky factorisation by Meijerink and Van der Vorst [21]. The Cholesky factorisation is only defined for symmetric matrices, but application of the factorisation on the matrix’s symmetric part has been used. It’s success rest on an insignificant antisymmetric part which is generally not the case for us.

Another approach [26] is to use a truncated Gram-Schmidt process to find an incomplete LQ factorisation of $A$ ($L$ is lower triangular and $Q$ is orthonormal). A collection of articles devoted to incomplete decompositions in numerical fluid mechanics may be found in Hackbusch and Wittum [11].
Appendix B

Example – Solving the Poisson equation

PROGRAM Poisson_Solver
!
! This example program is part of SMLIB v. 1.0b. It solves the
! the Axelsson and Polman Poisson equation test case from "A robust
! preconditioner based on algebraic substructuring and two-level grids",
!
! Copyright (C) 1996 Ernst A. Meese
! Refer to the file copyright.doc for details and important disclaimer.
!
! Created: January 1996
!
! ----------------------------------------------------------------------
!
USE Matrix_Arithmetic_Module
USE ILU_Module
USE CGS_Module
!
IMPLICIT NONE
!
This sample Poisson equation is discretised with nine
internal nodes in each direction; that is, the mesh
width is 0.1.
!
INTEGER, PARAMETER :: N = 9
!
TYPE(MSR) :: S
TYPE(ILU_CSR_MSR) :: PC
REAL(prec), DIMENSION(:), ALLOCATABLE :: x, b
TYPE (ConvHist) :: History
INTEGER :: I, J, K
INTEGER, DIMENSION(5) :: Cols
REAL(prec), DIMENSION(5) :: Row
LOGICAL, DIMENSION(5) :: Mask
!
Remmer always to nullify the pointers in these data structures.
APPENDIX B. EXAMPLE — SOLVING THE POISSON EQUATION

CALL NullifyMatrix ( S )
CALL NullifyMatrix ( PC )
CALL NullifyConvHist ( History )
!
! Define the matrix S, and allocate some memory for it. Allocate
! memory for the indeterminate (x) and the right hand side (b) vectors too.
!
CALL AllocateMatrix ( S, N**2, NZMAX = 5*N**2 )
ALLOCATE ( x(N**2), b(N**2) )
!
! Set up the five point stencil central differences discrete
! Laplace operator, and the right hand side vector b.
!
Row = (/ -1, -1, 4, -1, -1 /)*REAL(N+1, prec)**2
K = 0
DO J = 1, N
  DO I = 1, N
    K = K + 1
    Mask = (/ J > 1, I > 1, .TRUE., I < N, J < N /)
    Cols = (/ K - N, K - 1, K, K + 1, K + N /)
    CALL SetRow ( S, K, PACK(Cols, Mask), PACK(Row, Mask))
    b(K) = F( REAL(I, prec)/REAL(N+1, prec), REAL(J, prec)/REAL(N+1, prec) )
  END DO
END DO
!
! Solve the equation system and print iteration history and solution
!
CALL SymbolicCancellation (1, S, PC) ! Set up the data structure for PC
CALL ILU (S, PC) ! Incomplete factorisation by ILU
CALL CGS (x, S, b, PC, history = History) ! Solution by CGS
CALL WriteConvHist( History ) ! Print convergence history
!
PRINT ’(/” Y"X ".9F7.4”)’, (REAL(i, prec)/REAL(N+1, prec), i = 1, N)
PRINT ’(9(10F7.4/)///)’, (REAL(j, prec)/REAL(N+1, prec)),
& (x((j-1)*N+i), i=1, N), j=1, N)

CONTAINS

FUNCTION F(x, y)
!
! This function defines the right hand side of the Poisson equation
!
REAL(prec) :: F, x, y

F = (-y*y(-1.0, prec)*x**4+(2.0, prec-5.0, prec*y+y**2)*x**3 &
  & -(y-1.0, prec)*y**3+4.0, prec*y**2+y**4+2.0, prec)*x &
  & +(-5.0, prec*y**3+4.0, prec*y**2+y**4+2.0, prec)*x &
  & -4.0, prec*y**2+2.0, prec*y**3+2.0, prec)*EXP(x*y)

END FUNCTION F

END PROGRAM PoissonSolver
Appendix C

Module Templates

Users who want to implement their own data storage formats and/or solvers should do this in a module containing at least the procedures shown in the templates in this appendix. Actually, for a very quick and dirty implementation, only the matrix-vector product is necessary for the Krylov-subspace methods, but for a final implementation, the memory management routines should be included.

C.1 Abstract Data Type for Matrices

To include the new module into the library, it should reside in the matrices directory. It should furthermore be used by the matrix_arithmetic module. That is, the new module must be use referenced from the file matrix_arithmetic.f90. Lastly, due to Fortran 90's lack of inheritance, you will have to write variable declarations for your data structure in the modules for the iterative solvers you want to use with your module.

MODULE Matrix_Template_Module

USE Precision_Module

IMPLICIT NONE

TYPE MatrTempl
  INTEGER :: N
END TYPE MatrTempl

INTERFACE nullify_matrix
  MODULE PROCEDURE nullify_MatrTempl
END INTERFACE

INTERFACE allocate_matrix
  MODULE PROCEDURE allocate_MatrTempl
END INTERFACE

INTERFACE reallocate_matrix
  MODULE PROCEDURE reallocate_MatrTempl
END INTERFACE
END INTERFACE
INTERFACE deallocate_matrix
  MODULE PROCEDURE deallocate_MatrTempl
END INTERFACE

INTERFACE ALLOCATED
  MODULE PROCEDURE allocated_MatrTempl
END INTERFACE

INTERFACE operator (*)
  MODULE PROCEDURE MatrTempl_vector_product
END INTERFACE

CONTAINS

SUBROUTINE nullify_MatrTempl (A)
  TYPE(MatrTempl) :: A
  :
END SUBROUTINE nullify_MatrTempl

SUBROUTINE allocate_MatrTempl (A [, ...])
  TYPE (MatrTempl) :: A
  :
END SUBROUTINE allocate_MatrTempl

SUBROUTINE reallocate_MatrTempl (S, DELTA)
  TYPE (MatrTempl) :: S
  INTEGER, OPTIONAL, INTENT (in) :: DELTA
  :
END SUBROUTINE reallocate_MatrTempl

SUBROUTINE deallocate_MatrTempl (A)
  TYPE(MatrTempl) :: A
  :
END SUBROUTINE deallocate_MatrTempl

FUNCTION allocated_MatrTempl ( A ) result (res)
  TYPE(MatrTempl), INTENT (in) :: A
  LOGICAL :: res
  :
END FUNCTION allocated_MatrTempl

FUNCTION MatrTempl_vector_product ( A, x ) Result (y)
  TYPE(MatrTempl), INTENT(in) :: A
  REAL(prec), DIMENSION (:), INTENT(in) :: x
  REAL(prec), DIMENSION (SIZE(x)) :: y
  :
END FUNCTION MatrTempl_vector_product
C.2 Abstract Data Type for Factorisations

To include the new module into the library, it should reside in the matrices directory. It should furthermore be used by the matrix arithmetic module. That is, the new module must be used referenced from the file matrix_arithmetic.f90. Lastly, due to Fortran 90’s lack of inheritance, you will have to write variable declarations for your data structure in the modules for the iterative solvers you want to use with your module.

MODULE Matrix_Factorisation_module

USE Precision_Module

IMPLICIT NONE

TYPE Matrix_Factorisation

END TYPE Matrix_Factorisation

INTERFACE Nullify_Matrix
    MODULE PROCEDURE Nullify_Matrix_Factorisation
END INTERFACE

INTERFACE Allocate_Matrix
    MODULE PROCEDURE Allocate_Matrix_Factorisation
END INTERFACE

INTERFACE Reallocate_Matrix
    MODULE PROCEDURE Reallocate_Matrix_Factorisation
END INTERFACE

INTERFACE Deallocate_Matrix
    MODULE PROCEDURE Deallocate_Matrix_Factorisation
END INTERFACE

INTERFACE ALLOCATED
    MODULE PROCEDURE Allocated_Matrix_Factorisation
END INTERFACE

INTERFACE Solve
    MODULE PROCEDURE Matrix_Factorisation_Solve
END INTERFACE

CONTAINS

SUBROUTINE Nullify_Matrix_Factorisation (LU)
C.3 Iterative Solvers

To create an iterative solver that adheres to the standard user interface suggested in SMLIB, the following template should be used for the module. Variables not declared in the template are declared in the iteration-defaults.f90 file.

```fortran
MODULE Solver_module
  USE Matrix_Arithmetic_Module
END MODULE Solver_module
```
USE Iteration Defaults Module

IMPLICIT NONE

INTERFACE Solver
MODULE PROCEDURE Solver A Matrix
END INTERFACE

CONTAINS

SUBROUTINE Solver A Matrix &
( x, A, B, PC, init, tol, reduction_factor, &
  min_it, max_it, INFO, no_of_its, &
  residual_norm, history)
TYPE(MSR), INTENT (IN) :: A
TYPE(Matrix_Factorisation), INTENT (IN) :: PC
REAL(prec), DIMENSION(:), INTENT (INOUT) :: x
REAL(prec), DIMENSION(A % N), INTENT (IN) :: b
LOGICAL, OPTIONAL, INTENT (IN) :: init
REAL(prec), OPTIONAL, INTENT (IN) :: tol, reduction_factor
INTEGER, OPTIONAL, INTENT (IN) :: min_it, max_it, INFO
INTEGER, OPTIONAL, INTENT (OUT) :: no_of_its, residual_norm
TYPE(ConvHist), OPTIONAL, INTENT (OUT) :: History
!---
INTEGER :: ic, min_iter, max_iter, infolevel
REAL(prec) :: toler, norm
REAL(prec), DIMENSION ( A % N ) :: r
!---

CALL Check_Options ()

IF (infolevel/=NONE) PRINT F1, 'Solver', toler, min_iter, max_iter
  ic = 0; r = b - A*x
DO
  norm = SQRT(DOT_PRODUCT (r,r))
  IF (PRESENT(history)) CALL SetIterationHistory (history,norm)
  IF (infolevel=ALL) PRINT F2, ic, norm
  IF ((norm<toler .or. ic>=max_iter) .and. ic>=min_iter) EXIT
    ic = ic + 1
END DO

IF (infolevel=SOME) PRINT F3, ic, norm, SQRT(SUM((b-A*x)**2))

INCLUDE "check-options.inc"
END SUBROUTINE Solver_MSR

END MODULE Solver_A_Matrix
Bibliography


