

LIBLAC — Structured Data Types and Basic Operations for Numerical Linear Algebra in an ANSI C/Fortran 77 Environment

Daniel B. Leineweber Jochen A. Jost

September 20, 1996

Abstract

The C/Fortran library LIBLAC provides a set of convenient, structured data types and basic operations for numerical linear algebra. The library is intended to facilitate the ANSI C implementation of high-level applications which involve hierarchically structured data like partitioned vectors or block matrices. By maintaining compatibility with Fortran 77 conventions (e.g., matrices are stored by columns, and vectors may have arbitrary memory strides), the task of sharing data with existing Fortran 77 code is greatly simplified. Basic operations for data access, data copying/conversion, and input/output are supported in fully generic form. In addition, safe and user-friendly interfaces to Level 1–3 BLAS [7, 3, 4] and LAPACK [1] are provided, i.e., the use of existing libraries of highly efficient Fortran 77 subroutines is encouraged for performing common lower level computations. LIBLAC is easily portable to platforms supporting ANSI C and Standard Fortran 77, and it fully encapsulates the platform-dependent details of the C/Fortran interface, thereby resolving the inherent portability problems of mixed-language programs. The performance of typical applications is competitive with Fortran 77/BLAS.

1 Introduction

The success of large, complex numerical software projects critically depends on the chosen design methodology and programming paradigm as well as on their practical support through suitable languages and tools. Fortran 77 is not an ideal implementation language for larger projects, in particular if they involve the development of high-level applications. Basically, this is due to the well-known major limitations of Fortran 77,

- no module support (i.e., no names with *file scope*),
- no dynamic memory allocation,
- no user-defined, structured data types.

ANSI C, on the other hand, does provide all of this, but it has not been originally designed as a language for numerical applications—hence, there are a number of other shortcomings to deal with (e.g., no built-in complex arithmetic, less scope for code optimization by the compiler). Furthermore, there is only limited compatibility with the millions of lines of code in existing Fortran libraries, and redoing all of this in C is quite impractical.

For these reasons, it would be certainly best to combine the strengths of both languages in a new language, as has been attempted in the design of Fortran 90.

All major limitations of Fortran 77 have been addressed, and Fortran 90 provides records, dynamic storage, and even better module support than ANSI C. However, the transition from Fortran 77 to Fortran 90 has been painfully slow, and there is still a long way to go. On many platforms, there does not even exist a compiler supporting the new standard. Therefore, Fortran 90 is not yet considered a real alternative to Fortran 77 and ANSI C by many developers in the field.

In order to circumvent these problems, it is clearly advantageous to use *both* languages ANSI C and Fortran 77 in combination: the modular framework and the high-level parts of the code involving complex data can then be implemented in ANSI C, while calls to highly efficient Fortran 77 subroutines are made to perform the lower-level number crunching. However, the difficulty with this approach is that different conventions are used for the storage of arrays in the two languages, and that the C/Fortran interface is not officially standardized. The library LIBLAC has been developed to bridge this gap between ANSI C and Fortran 77 by providing a set of Fortran-compatible, structured data types for numerical linear algebra and by encapsulating the platform-dependent details of the C/Fortran interface. Hence based on LIBLAC it is possible to write truly portable C/Fortran programs. Of course, full compatibility with existing Fortran code (especially libraries) is maintained. Additional high-level support is provided for Level 1–3 BLAS [7, 3, 4] and LAPACK [1] routines through safe and user-friendly macro interfaces.

The design of LIBLAC has been much influenced by the requirements of the MUSCOD-II project [8, 9] dealing with the development of a modular optimal control package. Within this project, LIBLAC has soon become an indispensable tool for high-level development, and it has helped to quickly create more than 30 000 lines of clean, fast, and portable C code, thus significantly contributing to the success of MUSCOD-II. From this favourable experience we would expect LIBLAC to be useful in other large numerical software projects as well.

All LIBLAC types and operations can of course be used equally well within C++ programs. In addition, the flexibility and adaptability of LIBLAC and its ease of use could be further enhanced by implementing suitable object-oriented extensions. Note also that there already exist C++/Fortran tools supporting the hybrid design of numerical software, e.g. the LAPACK++ package [5].

2 Overview of LIBLAC

2.1 Vector and Matrix Types

LIBLAC supports the standard scalar types listed in Table 1. In the following, we will use the generic prefix X (and occasionally also Y) to denote the element type of

Table 1: Scalar types supported by LIBLAC.

prefix	description	corresponding scalar type in	
		C	Fortran 77
S	real single precision	float	REAL
D	real double precision	double	DOUBLE PRECISION
C	complex single precision	[Complex]	COMPLEX
Z	complex double precision	[ZComplex]	[COMPLEX*16]
I	short integer	short	[INTEGER*2]
L	long integer	long	INTEGER

LIBLAC vectors and matrices. Unless stated differently, *any* of the prefixes S, D, C, Z, I, and L can be substituted for X or Y. When it is required to refer to the scalar types themselves, we will use the type specification XType (implying that SType is equivalent to float, etc.). The complex types Complex and ZComplex are defined in the following way:

```
typedef struct { float real, imag; } Complex;
typedef struct { double real, imag; } ZComplex;
```

Note that on certain platforms it may be required to replace long by int in order to obtain the proper integer size. (This is automatically done by LIBLAC—therefore, long should always be used as the type corresponding to INTEGER.)

The vector and matrix types provided by LIBLAC are shown in Table 2. No further explanation should be necessary for *simple vectors* and *matrices*. A *partitioned vector* is a vector composed of subvectors which can be used just like simple vectors, e.g.,

$$a = \begin{pmatrix} a_0 \\ a_1 \\ a_2 \end{pmatrix}.$$

Of course, the partitioned vector itself also behaves exactly like a simple vector (i.e., it can be accessed as a whole). In other words, the partitioned vector inherits all the features of a simple vector and adds some of its own. Consequently, a partitioned vector can be used as an argument to LIBLAC operations anywhere a simple vector can.

Similarly, a *block matrix* is a matrix composed of submatrices which can be used just like simple matrices, e.g.,

$$A = \begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \\ A_{20} & A_{21} & A_{22} \end{pmatrix}.$$

(Note that all submatrices of a block row share the same row dimension and all submatrices of a block column share the same column dimension.) Again, the block matrix itself behaves exactly like a simple matrix, and within LIBLAC operations it can be used anywhere a simple matrix can.

The *list types* are intended for maintaining variable-size collections of individual (partitioned) vectors or (block) matrices. It is important to understand the difference between a vector list and a partitioned vector: while a partitioned vector can be accessed as a whole, the subvectors of a vector list can only be accessed individually.

Table 2: Vector and matrix types provided by LIBLAC.

name	description
XVec	simple vector
XPartVec	partitioned vector
XMat	simple matrix
XBlockMat	block matrix
XVecList	list of vectors
XPartVecList	list of partitioned vectors
XMatList	list of matrices
XBlockMatList	list of block matrices

2.2 Basic Objects, References, and Templates

In the following, we will use the term *object* for a properly initialized variable of one of the vector and matrix types discussed above. LIBLAC objects may be classified in basic objects, references, and templates, and we will now briefly define each kind of object in turn. (A remark for people knowing C++: LIBLAC templates have nothing to do with the concept of templates in C++.)

Basic objects are created by the LIBLAC allocation operations which set aside fresh memory for the elements (and in case of partitioned objects also for the required pointers to subobjects). The elements are automatically initialized by zero. After use, a basic object should be destructed by the appropriate LIBLAC deallocation operation in order to free the associated memory.

References just refer to (parts of) already existing LIBLAC objects or even to C/Fortran standard arrays. They can be created by the LIBLAC reference operations or simply by assignment of an existing object to a new variable. For instance, it is possible to obtain a vector reference to the main diagonal of an existing matrix. Since references do not have any memory of their own, they need not be destructed after use.

Templates also refer to already existing LIBLAC objects, but in addition they provide a partitioning of the existing object into subobjects. For instance, an existing simple vector can be partitioned into subvectors by defining a suitable vector template. Templates are created by the LIBLAC template allocation operations which set aside fresh memory for the pointers to the newly introduced subobjects (but not for the elements which are taken from the original object). After use, a template should be destructed by the appropriate LIBLAC template deallocation operation in order to free the associated memory. Of course, the original object is not affected by the creation or destruction of a template.

It is important to note that the *elements* of a basic object may be modified through any reference or template which refers to this object—there is no difference in usage between a reference and a basic simple object or between a template and a basic partitioned object. List objects are always basic objects but may *hold* any appropriate kind of object. For instance, a list of vectors may hold basic vector objects as well as vector references.

2.3 Creating and Using LIBLAC Objects

Now we will give some illustrating examples which show how to actually create and use LIBLAC objects in practice. All LIBLAC types and operations are defined in the main header `lac_util.h` which must be always included by

```
#include "lac_util.h"
```

when using the library. (More information on LIBLAC header files can be found in Appendix A of this manual.) For instance, having declared the variables

```
DVec x, y, z;
DMat A, B, C;
```

we could now perform the allocation and reference operations

```
x = alloc_DVec(5);
A = alloc_DMat(5, 5);
y = ref_DSubVec(x, 1, 3);
z = ref_DMatDiag(A);
B = ref_DSubMat(A, 1, 1, 3, 3);
C = A;
```

to create a set of simple LIBLAC objects (two basic objects and four references). Then, after adding

```
LVec d;
DPartVec u, v;
DBlockMat D, E;
```

to our list of declarations, we could write

```
d = alloc_LVec(2);
V_ELE(d,0) = 2; /* store subobject dimensions in d */
V_ELE(d,1) = 3;
u = alloc_DPartVec(d);
D = alloc_DBlockMat(d, d);
v = alloc_DVecTempl(x, d);
E = alloc_DMatTempl(A, d, d);
```

to create a set of partitioned LIBLAC objects (two new basic objects and two templates for already existing objects) based on the partitioning information we have stored in vector `d`. Note that LIBLAC objects can be defined and initialized at the same time, i.e., a list of declarations may contain a construct like

```
DVec w = alloc_DVec(10);
```

which is sometimes more handy than using a separate initialization.

Next, let us read some data from a text file. After preparing an input file `data.txt` containing

```
x
 1  2  3  4  5

A
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
21 22 23 24 25
```

and declaring a file pointer `ifile`, we could use

```
ifile = fopen("data.txt", "r");
search_key(ifile, "x");
v_fscan(ifile, x, RSEQ_MODE);
search_key(ifile, "A");
m_fscan(ifile, A, RSEQ_MODE);
fclose(ifile);
```

to read the element values from `data.txt` into vector `x` and matrix `A`. Then, in order to display the contents of some of our objects, we could write

```
printf("y =\n");
v_print(y, ROW_MODE);
printf("z =\n");
v_print(z, ROW_MODE);
printf("B =\n");
m_print(B, MAT_MODE);
printf("v =\n");
v_print(v, ROW_MODE);
```

```

printf("PV_SVEC(v,0) =\n");
v_print(PV_SVEC(v,0), ROW_MODE);
printf("BM_SMAT(E,0,1) =\n");
m_print(BM_SMAT(E,0,1), MAT_MODE);

```

which would produce the following screen output (using the default print format parameters which can be changed by `set_pformat`):

```

y =
[      2,      3,      4 ]

z =
[      1,      7,     13,     19,     25 ]

B =
[      7,      8,      9 ]
[     12,     13,     14 ]
[     17,     18,     19 ]

v =
[      1,      2,      3,      4,      5 ]

PV_SVEC(v,0) =
[      1,      2 ]

BM_SMAT(E,0,1) =
[      3,      4,      5 ]
[      8,      9,     10 ]

```

Of course, various other kinds of formatting are available for the printing of LIBLAC objects, and it is also possible to save and restore objects in binary form.

Finally, in order to actually *do* something with our data, let us try some of the LIBLAC copy operations:

```

v_copy(x, u);
v_copy(y, PV_SVEC(u,1));
m_copy(A, D);
m_copy(B, BM_SMAT(D,1,1));
m_swapt(BM_SMAT(D,0,1), BM_SMAT(D,1,0));

```

(What are the contents of `u` and `D` after these operations?) In addition, after including the LIBLAC BLAS header by

```
#include "lac_blas.h"
```

we could also test a few BLAS routines, for instance:

```

DAXPY(D_ONE, x, u);
DAXPY(D_MINUS_ONE, y, PV_SVEC(u,1));
DGEMV("No Transpose", D_ONE, D, x, D_ONE, u);
DGEMV("Transpose", D_ONE, BM_SMAT(E,0,1), PV_SVEC(v,0),
      D_ONE, PV_SVEC(u,1));

```

(It is *not* a required exercise to calculate the resulting `u` by hand!) Observe that internal LIBLAC operations like `v_print` or `v_copy` are *generic* (i.e., one and the same operation works for all different element types `S`, `D`, `C`, `Z`, `I`, `L`), while in case of the BLAS routines, the element type of the objects passed (here `D`) must be explicitly specified as prefix of the routine name.

We close our small tour of LIBLAC with some general remarks on index ranges, element access modes, empty objects, and automatic dimension checks.

1. All LIBLAC objects strictly follow the C convention for index ranges: indices always start from zero, and the last valid index is dimension minus one. (For performance reasons and to avoid an unnecessary source of confusion, no “sugar” has been added here.) Thus, a standard way to write a loop over all elements of an object would be:

```
for (jj=0; jj<PV_DIM(pvec); jj++) {
    /* do something with PV_SVEC(pvec,jj) */
}
```

There are some rare occasions where it is important to remember that in Fortran, array indices start from one by default.

2. LIBLAC provides two different modes for the access of individual vector and matrix elements: *logical* element access using `V_ELE()` and `M_ELE()` (safe but, at least for vectors, relatively slow), and *physical* element access using `V_EACC()` and `M_EACC()` (fast but potentially dangerous). For stride-one vectors, straightforward physical element access is the method of choice, but it must be carefully checked whether general strides needing special treatment may occur in a given context. Of course, the internal LIBLAC operations as well as the BLAS/EBLAS routines will handle general (even negative) strides correctly. As far as possible, element-oriented operations should be performed by specialized low-level routines written either in Fortran 77 or in plain ANSI C, because LIBLAC has not been designed with this kind of operations in mind.
3. It is perfectly legal to create empty objects, i.e., objects having at least one zero dimension. Such empty objects provide a marvellous way to write very general code without having to use many if-statements. As long as the dimensions still match properly, internal LIBLAC operations as well as BLAS/EBLAS and LAPACK routines will behave well.
4. By default, LIBLAC automatically performs dimension checks on the objects passed to internal operations and BLAS routines. Suitable error messages are generated in case of mismatches. If desired, most of these checking procedures can be eliminated by recompiling the application with the flag `-DNDEBUG`. For instance, BLAS calls will then be directly inlined in the code without using intermediate wrappers in order to create faster executables.

Hopefully, the above examples and remarks have given a first impression of the ways in which LIBLAC objects can be created and used. We will present a more realistic code example below after discussing the use of the C/Fortran interface. For a complete description of all LIBLAC functions and macros, the reader is referred to Appendix B of this manual. The BLAS/EBLAS and LAPACK interfaces are described in Appendices C and D, respectively.

2.4 Using the C/Fortran Interface

Virtually every platform supporting ANSI C and Fortran 77 provides some way to call Fortran routines from C functions and vice versa, and to share global data among C and Fortran modules. However, this C/Fortran interface is not standardized, and consequently, mixed-language programs suffer from inherent portability

problems. In order to resolve these problems, LIBLAC provides its own *portable* C/Fortran interface which fully encapsulates the platform-dependent details.

The C/Fortran interface of LIBLAC is implemented through a set of special macros whose names start with F77. For each of the platforms supported by LIBLAC, a suitable definition of these macros is available. In order to select the correct definition, the target platform must be specified by compiling the library as well as the applications with the flag `-Dmachine`. Currently, *machine* may be one of the following:

- SGI (SiliconGraphics Indy or Indigo running IRIX)
- IBM (IBM RS/6000 running AIX)
- SUN (Sun Sparc running SunOS or Solaris)
- PARSYTEC (Parsytec running PARIX)
- DEC (DEC Alpha running OpenVMS-AXP)
- CRAY (CRAY running UNICOS)

Adding further platform definitions to LIBLAC is a simple matter (only the files `f77interface.h` and possibly `intern.c` have to be modified accordingly).

The best way to explain the use of the F77-macros for constructing a portable C/Fortran interface is to look at a simple example. Therefore, let us assume that we have to call the following Fortran subroutine from C:

```

SUBROUTINE BAR(MODE, UP, M, N, A, LDA, X, MSG)
CHARACTER*1 MODE
LOGICAL UP
INTEGER M, N, LDA
DOUBLE PRECISION A(LDA,N), X(N)
CHARACTER*(*) MSG
DOUBLE PRECISION R, S
INTEGER P
COMMON /UGLY/ R(10), S, P(4)
.
.
.
RETURN
END

```

Note that BAR communicates with the outside world through its argument list as well as through common block UGLY. A portable C program to call BAR could be written as follows:

```

#include <string.h>
#include "lac_util.h"

void F77NAME(bar)(
    F77SDESC *mode,
    long *up,
    long *m,
    long *n,
    double *a,
    long *lda,
    double *x,

```

```

    F77SDESC *msg
    F77IARG(long len1)
    F77IARG(long len2)
);

struct {
    double r[10];
    double s;
    long p[4];
} F77NAME(ugly);

main()
{
    long up = F77TRUE;
    DMat a;
    DVec x;
    char msg[] = "just for fun";

    F77NAME(ugly).r[0] = 1.41;
    F77NAME(ugly).s = 3.14;
    F77NAME(ugly).p[3] = 42;
    a = alloc_DMat(10, 15);
    x = alloc_DVec(15);

    F77NAME(bar)(
        F77STRING1("TEST"),
        &up,
        &M_ROWDM(a),
        &M_COLDM(a),
        M_START(a),
        &M_LEADM(a),
        V_START(x),
        F77STRING(msg)
        F77IARG1
        F77IARG(strlen(msg))
    );

    free_DVec(x);
    free_DMat(a);
}

```

From this example, we can now extract five important rules for writing portable C/Fortran code based on LIBLAC:

1. Within C code, the names of Fortran subroutines, functions, and common blocks must be written in *lower case*, and they must be enclosed by the F77NAME() macro. In our example, BAR and UGLY become F77NAME(bar) and F77NAME(ugly).
2. When calling a Fortran subroutine or function from C, all explicit arguments are passed *by reference*, i.e., through the use of appropriate pointers. Character strings are declared as F77SDESC*. (It should be remembered that it is not possible in C to take the address of constants or expressions—these must be stored into variables first.)

3. Actual character string arguments must be enclosed by the `F77STRING()` macro (instead, the macro `F77STRING1()` can be used in the special case of length-one character strings). Observe that it is allowed to directly use string literals as actual arguments, e.g., `F77STRING1("TEST")`.
4. For each character string argument, an implicit length argument of type `long` (not `long*`) has to be added at the end of the argument list in the order the corresponding string appeared in the list. These implicit length arguments must be enclosed by the `F77IARG()` macro (`F77IARG1` is equivalent to `F77IARG(1)`). There must be *no* comma separators before and between the implicit length arguments.
5. Fortran arguments of type `LOGICAL` have to be replaced by arguments of type `long*` in C. The values `.TRUE.` and `.FALSE.` correspond to `F77TRUE` and `F77FALSE`, respectively.

Although this is not the complete story of the C/Fortran interface, the majority of Fortran calls from C can be handled in a safe and portable way according to these simple rules.

2.5 A Small Code Example

As an example for the use of `LIBLAC`, we discuss a small C program which solves quadratic programming (QP) problems by calling the Fortran subroutine `E04NAF` from the NAG library [12]. The QP problem is assumed to be stated in the form

$$\min_{x \in \mathbb{R}^n} c^T x + \frac{1}{2} x^T H x$$

subject to

$$l \leq \begin{pmatrix} x \\ Ax \end{pmatrix} \leq u,$$

where c is a constant n -vector, H is a constant n by n symmetric matrix, and A is a constant m by n matrix (where m may be zero). The constant vectors l and u have dimension $n + m$.

The C program listed below reads the data n , m , c , H , A , l , and u from an input file, calls `E04NAF`, and writes the solution x along with some other results to an output file. The diagnostic printout of `E04NAF` goes to standard output. On a SiliconGraphics Indy workstation running IRIX, the program can be compiled and linked using the following commands:

```
cc -c example.c -DSGI
f77 -o qpsol example.o -lnag -llac -lblas
```

Note that the platform SGI must be selected by compiling with `-DSGI` as explained above. For the linking step, the `f77` driver command is used instead of `cc` to ensure that the Fortran 77 link libraries are automatically scanned (with `cc`, these libraries would have to be specified manually). Assuming that a suitable input file `input.dat` has been prepared, the command

```
qpsol input.dat results.dat
```

writes the solution of the QP problem to file `results.dat`, see the example program input/output given below. A detailed description of subroutine `E04NAF` and its arguments can be found in the NAG documentation [12].

Program Text (example.c)

```

#include <stdio.h>
#include <stdlib.h>

#include "lac_util.h"
#include "lac_blas.h"
#include "lac_eblas.h"

void F77NAME(e04naf)(
    const long    *itmax,
    const long    *msglvl,
    const long    *n,
    const long    *nclin,
    const long    *nctotl,
    const long    *nrowa,
    const long    *nrowh,
    const long    *ncolh,
    const double  *bigbnd,
    const double  *a,
    const double  *bl,
    const double  *bu,
    const double  *cvec,
    const double  *featol,
    const double  *hess,
    void    (*qp Hess)(
        const long    *n,
        const long    *nrowh,
        const long    *ncolh,
        const long    *jthcol,
        const double  *hess,
        const double  *x,
        double        *hx
    ),
    const long    *cold,
    const long    *lp,
    const long    *orthog,
    double        *x,
    long          *istate,
    long          *iter,
    double        *obj,
    double        *clamda,
    long          *iwork,
    const long    *liwork,
    double        *work,
    const long    *lwork,
    long          *ifail
);

static void hess_mv(
    const long    *n,
    const long    *nrowh,
    const long    *ncolh,

```

```

    const long  *jthcol,
    const double *hess,
    const double *x,
    double      *hx
);

main(int argc, char *argv[])
/*
 * command-line arguments:
 *   argv[1] ... name of input file
 *   argv[2] ... name of output file
 */
{
    const long ITMAX = 20;
    const long MSGLVL = 1;
    const long COLD = F77TRUE;
    const long LP = F77FALSE;
    const long ORTHOG = F77TRUE;
    const double BIGBND = 1.0E10;
    const double FTOL = 1.0E-8;

    FILE *ifile, *ofile;
    long n, m, nctotl, iter, ifail;
    double obj;

    DVec cvec, lvec, uvec, xvec, featol, clamda, work;
    DMat hmat, amat;
    LVec istate, iwork;

    ifile = fopen(argv[1], "r");

    search_key(ifile, "n");
    fscanf(ifile, "%ld", &n);
    search_key(ifile, "m");
    fscanf(ifile, "%ld", &m);

    nctotl = n + m;

    cvec = alloc_DVec(n);
    lvec = alloc_DVec(nctotl);
    uvec = alloc_DVec(nctotl);
    hmat = alloc_DMat(n, n);
    amat = alloc_DMat(m, n);

    search_key(ifile, "cvec");
    v_fscan(ifile, cvec, CSEQ_MODE);
    search_key(ifile, "lvec");
    v_fscan(ifile, lvec, CSEQ_MODE);
    search_key(ifile, "uvec");
    v_fscan(ifile, uvec, CSEQ_MODE);
    search_key(ifile, "hmat");
    m_fscan(ifile, hmat, RSEQ_MODE);

```

```

search_key(ifile, "amat");
m_fscan(ifile, amat, RSEQ_MODE);

fclose(ifile);

xvec = alloc_DVec(n);
featol = alloc_DVec(nctotl);
clamda = alloc_DVec(nctotl);
work = alloc_DVec(2*n*n + 4*n + 2*m + M_LEADIM(amat));
istate = alloc_LVec(nctotl);
iwork = alloc_LVec(2*n);

DVINIT(FTOL, featol);
ifail = 1;

F77NAME(e04naf)(
    &ITMAX,
    &MSGLVL,
    &n,
    &m,
    &nctotl,
    &M_LEADIM(amat),
    &M_LEADIM(hmat),
    &M_COLDIM(hmat),
    &BIGBND,
    M_START(amat),
    V_START(lvec),
    V_START(uvec),
    V_START(cvec),
    V_START(featol),
    M_START(hmat),
    hess_mvp,
    &COLD,
    &LP,
    &ORTHOG,
    V_START(xvec),
    V_START(istate),
    &iter,
    &obj,
    V_START(clamda),
    V_START(iwork),
    &V_DIM(iwork),
    V_START(work),
    &V_DIM(work),
    &ifail
);

ofile = fopen(argv[2], "a");

fprintf(ofile, "obj\n %lg\n\n", obj);
fprintf(ofile, "xvec\n");
v_fprint(ofile, xvec, CSEQ_MODE);
fprintf(ofile, "clamda\n");
v_fprint(ofile, clamda, CSEQ_MODE);

```

```

fclose(ofile);

free_LVec(iwork);
free_LVec(istate);
free_DVec(work);
free_DVec(clamda);
free_DVec(featol);
free_DVec(xvec);
free_DMat(amat);
free_DMat(hmat);
free_DVec(uvec);
free_DVec(lvec);
free_DVec(cvec);
}

static void hess_mvp(
    const long    *n,
    const long    *nrowh,
    const long    *ncolh,
    const long    *jthcol,
    const double  *hess,
    const double  *x,
    double        *hx
)
{
    DMat hessmat = ref_DStdMat(*n, *n, (double *) hess, *nrowh);
    DVec xvec    = ref_DStdVec(*n, (double *) x, 1);
    DVec hxvec   = ref_DStdVec(*n, hx, 1);

    if (*jthcol > 0)
        v_copy(ref_DMatCol(hessmat, *jthcol-1), hxvec);
    else
        DGEMV("No Transpose", D_ONE, hessmat, xvec, D_ZERO, hxvec);
}

```

Program Input (input.dat)

```

n
 7

m
 7

cvec
-0.02
-0.2
-0.2
-0.2
-0.2
 0.04
 0.04

```

hmat

```

  2  0  0  0  0  0  0
  0  2  0  0  0  0  0
  0  0  2  2  0  0  0
  0  0  2  2  0  0  0
  0  0  0  0  2  0  0
  0  0  0  0  0 -2 -2
  0  0  0  0  0 -2 -2

```

amat

```

  1      1      1      1      1      1      1
  0.15  0.04  0.02  0.04  0.02  0.01  0.03
  0.03  0.05  0.08  0.02  0.06  0.01  0
  0.02  0.04  0.01  0.02  0.02  0      0
  0.02  0.03  0      0      0.01  0      0
  0.70  0.75  0.80  0.75  0.80  0.97  0
  0.02  0.06  0.08  0.12  0.02  0.01  0.97

```

lvec

```

-0.01
-0.1
-0.01
-0.04
-0.1
-0.01
-0.01
-0.13
-1.0E12
-1.0E12
-1.0E12
-1.0E12
-0.0992
-0.003

```

uvec

```

  0.01
  0.15
  0.03
  0.02
  0.05
  1.0E12
  1.0E12
 -0.13
 -0.0049
 -0.0064
 -0.0037
 -0.0012
  1.0E12
  0.002

```

Program Output (results.dat)

```

obj
  0.0370316

xvec
  -0.01
 -0.06986
  0.01826
 -0.02426
 -0.06201
  0.01381
  0.004066

clamda
  0.47
    0
    0
    0
    0
    0
    0
  -1.908
    0
 -0.3144
    0
    0
  1.955
  1.972

```

3 Summary

In this manual, we have given a brief description of LIBLAC, a C/Fortran library which has been developed as a tool for the implementation of large, complex numerical software applications in an ANSI C/Fortran 77 mixed-language environment. LIBLAC provides a set of fully Fortran-compatible, structured types and the corresponding basic operations for numerical linear algebra. In addition, by encapsulating the platform-dependent details of the C/Fortran interface, LIBLAC allows to write truly portable C/Fortran programs which can draw on the particular strengths of *both* programming languages. The library is intended primarily for the implementation of high-level applications which involve hierarchically structured data like partitioned vectors or block matrices. LIBLAC has been very successfully used within the MUSCOD-II project.

One final word of caution: like any tool, LIBLAC can easily be *misused*. The following three general guidelines should be kept in mind when developing applications with LIBLAC:

1. Keep it simple. Use the advanced concepts of LIBLAC (references, templates) rather sparingly.
2. As far as possible, avoid performing element-oriented operations with LIBLAC (these are better handled by subroutines written in Fortran or plain C).

- When calling Fortran subroutines from C, always use the F77-macros provided by LIBLAC to ensure portability of the resulting code.

Regardless of the sophistication of the tools employed in a development effort, there is still no substitute for good programming style.

Acknowledgements. We wish to thank Oliver Bösl and Andreas Ströder for suggesting many improvements to the library and its documentation. Also, we would like to thank Hartmut Kapp for proofreading this manual and extensively testing the use of LIBLAC from C++. Financial support from the *Deutsche Forschungsgemeinschaft (DFG)* within the graduate program “Modellierung und Wissenschaftliches Rechnen in Mathematik und Naturwissenschaften” and the DFG-Schwerpunktprogramm “Anwendungsbezogene Optimierung und Steuerung” is gratefully acknowledged.

A LIBLAC Header Files

The LIBLAC data types and basic operations are defined in the main header `lac_util.h` which must be always included when using the library. The interfaces to BLAS, BLAS Extensions, and LAPACK are defined in the the special headers `lac_blas.h`, `lac_eblas.h`, and `lac_lapack.h`, respectively. All other LIBLAC header files are “internal”, i.e., they need not be explicitly included by the application program. However, some of these “internal” headers may be interesting in other contexts, e.g., for making existing mixed-language programs portable:

- `f77interface.h` (definition of the C/Fortran interface)
- `blas.h` (definitions of the original Level 1–3 BLAS routines)
- `lapack.h` (definitions of most of the original LAPACK routines)

Of course, LIBLAC can be used also by C++ applications; in order to achieve correct external linkage, a *linkage specification* like

```
extern "C" {
    #include "lac_util.h"
}
```

must be used in this case.

B Specification of Functions and Macros

B.1 Creation and Cleanup of Objects

Basic Vector and Matrix Objects

```
XVec alloc_XVec(long dim)
void free_XVec(XVec v)
```

`alloc_XVec` allocates and returns a zero-initialized simple vector with element type `XType` and dimension `dim`. `free_XVec` deallocates the simple vector `v` previously allocated by `alloc_XVec`.

```
XPartVec alloc_XPartVec(LVec pdim)
void free_XPartVec(XPartVec pv)
```

`alloc_XPartVec` allocates and returns a zero-initialized partitioned vector with element type `XType`. The number of partitions is given by the dimension of `pdim`, and the dimension of each partition is specified by the corresponding element of `pdim`. `free_XPartVec` deallocates the partitioned vector `pv` previously allocated by `alloc_XPartVec`.

`XMat alloc_XMat(long rowdim, long coldim)`
`void free_XMat(XMat m)`
`alloc_XMat` allocates and returns a zero-initialized simple matrix with element type `XType` and dimensions `rowdim`, `coldim`. `free_XMat` deallocates the simple matrix `m` previously allocated by `alloc_XMat`.

`XBlockMat alloc_XBlockMat(LVec browdim, LVec bcoldim)`
`void free_XBlockMat(XBlockMat bm)`
`alloc_XBlockMat` allocates and returns a zero-initialized block matrix with element type `XType`. The numbers of blocks in row direction and column direction are given by the dimensions of `browdim` and `bcoldim`, respectively, and the dimensions of each block are specified by the corresponding pair of elements of `browdim` and `bcoldim`. `free_XBlockMat` deallocates the block matrix `bm` previously allocated by `alloc_XBlockMat`.

References and Templates

`XVec ref_XSubVec(XVec v, long istart, long sdim)`
`ref_XSubVec` returns a reference to a subvector of an existing vector `v`. The subvector starts at element `istart` of `v` and has dimension `sdim`.

`XVec ref_XStdVec(long vdim, XType *vp, long vinc)`
`ref_XStdVec` returns a reference to an existing standard vector with dimension `vdim`, start pointer `vp`, and stride `vinc`.

`XVec ref_XMatRow(XMat m, long i)`
`ref_XMatRow` returns a reference to the `i`th row vector of an existing matrix `m`.

`XVec ref_XMatCol(XMat m, long j)`
`ref_XMatCol` returns a reference to the `j`th column vector of an existing matrix `m`.

`XVec ref_XMatDiag(XMat m)`
`ref_XMatDiag` returns a reference to the main diagonal of an existing matrix `m`.

`XPartVec alloc_XVecTempl(XVec v, LVec tdim)`
`void free_XVecTempl(XPartVec vt)`
`alloc_XVecTempl` allocates and returns a vector template for an existing vector `v`. The number of vector partitions and their dimensions are specified by `tdim` in the same manner as described for `alloc_XPartVec`. `free_XVecTempl` deallocates the vector template `vt` previously allocated by `alloc_XVecTempl` (the original vector `v` is unaffected).

`XMat ref_XSubMat(XMat m, long istart, long jstart, long srowdim, long scoldim)`
`ref_XSubMat` returns a reference to a submatrix of an existing matrix `m`. The submatrix starts at row `istart` and column `jstart` of `m` and has dimensions `srowdim`, `scoldim`.

`XMat ref_XStdMat(long mrowdim, long mcoldim, XType *mp, long mleadim)`
`ref_XStdMat` returns a reference to an existing standard matrix with dimensions `mrowdim` and `mcoldim`, start pointer `mp`, and leading dimension `mleadim`. It is important to note that the standard matrix must conform to the Fortran 77 convention, i.e., the matrix elements must be stored by columns.

```
XBlockMat alloc_XMatTempl(XMat m, LVec trowdim, LVec tcoldim)
void free_XMatTempl(XBlockMat mt)
```

`alloc_XMatTempl` allocates and returns a matrix template for an existing matrix `m`. The number as well as the dimensions of block rows and block columns are specified by `trowdim` and `tcoldim`, respectively, in the same manner as described for `alloc_XBlockMat`. `free_XMatTempl` deallocates the matrix template `mt` previously allocated by `alloc_XMatTempl` (the original matrix `m` is unaffected).

List Objects

```
XVecList alloc_XVecList(long ldim)
void realloc_XVecList(XVecList *vlp, long newldim)
void free_XVecList(XVecList vl)
```

`alloc_XVecList` allocates and returns a list which can hold `ldim` simple vectors of type `XVec`. `realloc_XVecList` changes the size of the list `*vlp` to `newldim` (`vlp` must point to a list previously allocated by `alloc_XVecList`; the contents will be unchanged up to the minimum of the old and new sizes). `free_XVecList` deallocates the list `vl` which has been previously allocated by `alloc_XVecList`. The user is responsible for the proper initialization and, if required, also for the deallocation of individual list elements.

```
XPartVecList alloc_XPartVecList(long ldim)
void realloc_XPartVecList(XPartVecList *pvlp, long newldim)
void free_XPartVecList(XPartVecList pvl)
```

These functions are used to maintain lists of partitioned vectors `XPartVec`. The basic functionality is the same as described above for lists of simple vectors.

```
XMatList alloc_XMatList(long ldim)
void realloc_XMatList(XMatList *mlp, long newldim)
void free_XMatList(XMatList ml)
```

These functions are used to maintain lists of simple matrices `XMat`. The basic functionality is the same as described above for lists of simple vectors.

```
XBlockMatList alloc_XBlockMatList(long ldim)
void realloc_XBlockMatList(XBlockMatList *bmlp, long newldim)
void free_XBlockMatList(XBlockMatList bml)
```

These functions are used to maintain lists of block matrices `XBlockMat`. The basic functionality is the same as described above for lists of simple vectors.

Allocation Diagnostics

```
AFTally set_tally(void)
void check_tally(AFTally tally, const char *label)
```

`set_tally` returns a structure of type `AFTally` containing the current allocation counts of LIBLAC objects. `check_tally` compares the variable `tally` previously set by `set_tally` with the current internal tally. In case of disagreement, an appropriate warning message is printed to `stderr` along with the character string `label`. (A positive allocation count means that objects have been allocated but not deallocated since the last call of `set_tally`.) This pair of functions can be used during program development to detect allocation or deallocation errors; the following C code fragment provides a simple example of usage.

```

void foo(long m, long n, ...)
{
#ifdef NDEBUG
    AFTally tally = set_tally();
#endif
    DVec vec = alloc_DVec(n);
    DMat mat = alloc_DMat(m, n);
    .
    .
    .

    free_DMat(mat);
    free_DVec(vec);
#ifdef NDEBUG
    check_tally(tally, "foo()");
#endif
}

```

B.2 Macros for Object Access

General Vector Access

`XType *V_START(XVec v)`

`V_START` returns the start address (pointer to first physical element) of vector `v`.

`XType V_ELE(XVec v, long i)`

`V_ELE` gives access to the element with logical index `i` of vector `v`. It should be noted that physical element access using `V_EACC` is considerably more efficient than logical element access using `V_ELE`. (However, `V_ELE` is “safer”.)

`XType V_EACC(XVec v, long idx)`

`V_EACC` gives access to the element with physical index `idx` of vector `v`. Observe that `V_EACC(v, idx)` is equivalent to `*(V_START(v)+idx)`.

`long V_DIM(XVec v)`

`V_DIM` returns the dimension (number of logical elements) of vector `v`.

`long V_INC(XVec v)`

`V_INC` returns the stride (increment of physical index between two logical elements) of vector `v`.

`long V_SIDX(XVec v)`

`V_SIDX` returns the start index (physical index of first logical element) of vector `v`.

Subvector Access

`XVec PV_SVEC(XPartVec pv, long ii)`

`PV_SVEC` gives access to the `ii`th subvector of partitioned vector `pv`.

`long PV_DIM(XPartVec pv)`

`PV_DIM` returns the number of subvectors of partitioned vector `pv`.

`XVec VL_SVEC(XVecList vl, long ii)`

VL_SVEC gives access to the *i*ith element of vector list *v1*.

long VL_DIM(XVecList *v1*)

VL_DIM returns the dimension of vector list *v1*.

General Matrix Access

XType *M_START(XMat *m*)

M_START returns the start address of matrix *m* (pointer to element in the “northwest” corner).

XType *M_ROWADDR(XMat *m*, long *i*)

M_ROWADDR returns the address of the first element of the *i*th row vector of matrix *m*.

XType *M_COLADDR(XMat *m*, long *j*)

M_COLADDR returns the address of the first element of the *j*th column vector of matrix *m*.

XType M_ELE(XMat *m*, long *i*, long *j*)

M_ELE gives access to the matrix element with row index *i* and column index *j* of matrix *m*. Note that physical element access using M_EACC might be more efficient than logical element access using M_ELE in certain cases (depending on how well the code is optimized by the compiler).

XType M_EACC(XMat *m*, long *idx*)

M_EACC gives access to the element with physical index *idx* of matrix *m*. Observe that M_EACC(*m*,*idx*) is equivalent to *(M_START(*m*)+*idx*).

long M_ROWDIM(XMat *m*)

M_ROWDIM returns the row dimension of matrix *m*.

long M_COLDIM(XMat *m*)

M_COLDIM returns the column dimension of matrix *m*.

long M_LEADIM(XMat *m*)

M_LEADIM returns the leading dimension of matrix *m* (row dimension of the physical array in which *m* is stored). The leading dimension of a matrix is always greater than or equal to its row dimension and never smaller than one.

Submatrix Access

XMat BM_SMAT(XBlockMat *bm*, long *ii*, long *jj*)

BM_SMAT gives access to the submatrix with block row index *ii* and block column index *jj* of block matrix *bm*.

long BM_ROWDIM(XBlockMat *bm*)

BM_ROWDIM returns the number of block rows of block matrix *bm*.

long BM_COLDIM(XBlockMat *bm*)

BM_COLDIM returns the number of block columns of block matrix *bm*.

XMat ML_SMAT(XMatList *ml*, long *ii*)

ML_SMAT gives access to the *i*ith element of matrix list *ml*.

long ML_DIM(XMatList *ml*)

ML_DIM returns the dimension of matrix list *ml*.

B.3 Copy and Conversion Operations

Vector Copying and Conversion

`void v_copy(XVec x, XVec y)`

`v_copy` copies the elements of vector `x` to vector `y`. Both vectors must exist and agree in dimension and element type.

`void v_pcopy(XVec x, long *ixstart, XVec y)`

`v_pcopy` copies a sequence of elements starting at position `*ixstart` of vector `x` to vector `y` (the number of elements copied is given by the dimension of `y`). Both vectors must exist and agree in element type. In addition, the dimension of `y` must be compatible with `*ixstart` and the dimension of `x`. On successful return, `*ixstart` is set to `*ixstart + V_DIM(y)`.

`void v_copyp(XVec x, XVec y, long *iystart)`

`v_copyp` copies vector `x` to position `*iystart` of vector `y`. Both vectors must exist and agree in element type. In addition, the dimension of `x` must be compatible with `*iystart` and the dimension of `y`. On successful return, `*iystart` is set to `*iystart + V_DIM(x)`.

`void v_swap(XVec x, XVec y)`

`v_swap` interchanges the elements of vectors `x` and `y`. Both vectors must exist and agree in dimension and element type.

`void v_ext(LVec iv, XVec x, XVec y)`

`v_ext` copies individual elements of vector `x` into vector `y`. The elements to be “extracted” are specified by index vector `iv`. Note that `y` must exist and have the same element type as `x`. In addition, `iv` must have the same dimension as `y`, and the element values of `iv` must represent valid indices for `x`.

`void v_ins(LVec iv, XVec x, XVec y)`

`v_ins` copies the elements of vector `x` into certain positions of vector `y`. The elements of `y` to be overwritten by the elements of `x` (“insert positions”) are specified by index vector `iv`; the remaining elements of `y` are unaffected. Note that `y` must exist and have the same element type as `x`. In addition, `iv` must have the same dimension as `x`, and the element values of `iv` must represent valid indices for `y`.

`void v_convc(XVec x, YVec y)`

`v_convc` converts and copies the elements of vector `x` to vector `y`. Both vectors must exist and agree in dimension. The type conversions supported are listed in Table 3.

Matrix Copying and Conversion

`void m_copy(XMat a, XMat b)`

`m_copy` copies the elements of matrix `a` to matrix `b`. Both matrices must exist and agree in dimensions and element type.

`void m_copyt(XMat a, XMat b)`

`m_copyt` copies the rows of matrix `a` to the columns of matrix `b`. Both matrices must exist, have compatible dimensions, and agree in element type.

`void m_swap(XMat a, XMat b)`

Table 3: Type conversions supported by LIBLAC.

source type	target types			
S	(S)	D	I	L
D	S	(D)	I	L
C		(C)	Z	
Z		C	(Z)	
I	S	D	(I)	L
L	S	D	I	(L)

`m_swap` interchanges the elements of matrices `a` and `b`. Both matrices must exist and agree in dimensions and element type.

```
void m_swapt(XMat a, XMat b)
```

`m_swapt` interchanges the rows of matrix `a` with the columns of matrix `b`. Both matrices must exist, have compatible dimensions, and agree in element type.

```
void m_rowext(LVec ivr, XMat a, XMat b)
```

`m_rowext` copies individual rows of matrix `a` into matrix `b`. The rows to be “extracted” are specified by index vector `ivr`. Note that `b` must exist and have the same element type and column dimension as `a`. In addition, `ivr` must have a dimension equal to the row dimension of `b`, and the element values of `ivr` must represent valid row indices for `a`.

```
void m_rowins(LVec ivr, XMat a, XMat b)
```

`m_rowins` copies the rows of matrix `a` into certain row positions of matrix `b`. The rows of `b` to be overwritten by the rows of `a` (“insert positions”) are specified by index vector `ivr`; the remaining rows of `b` are unaffected. Note that `b` must exist and have the same element type and column dimension as `a`. In addition, `ivr` must have a dimension equal to the row dimension of `a`, and the element values of `ivr` must represent valid row indices for `b`.

```
void m_colext(LVec ivc, XMat a, XMat b)
```

`m_colext` copies individual columns of matrix `a` into matrix `b`. The columns to be “extracted” are specified by index vector `ivc`. Note that `b` must exist and have the same element type and row dimension as `a`. In addition, `ivc` must have a dimension equal to the column dimension of `b`, and the element values of `ivc` must represent valid column indices for `a`.

```
void m_colins(LVec ivc, XMat a, XMat b)
```

`m_colins` copies the columns of matrix `a` into certain column positions of matrix `b`. The columns of `b` to be overwritten by the columns of `a` (“insert positions”) are specified by index vector `ivc`; the remaining columns of `b` are unaffected. Note that `b` must exist and have the same element type and row dimension as `a`. In addition, `ivc` must have a dimension equal to the column dimension of `a`, and the element values of `ivc` must represent valid column indices for `b`.

```
void m_conv(XMat a, YMat b)
```

`mconv` converts and copies the elements of matrix `a` to matrix `b`. Both matrices must exist and agree in dimensions. The type conversions supported are listed in Table 3.

Table 4: Print modes (p) and read modes (r) supported by LIBLAC.

name	type	description
ROW_MODE	p	row vector mode (with delimiters)
COL_MODE	p	column vector mode (with delimiters)
MAT_MODE	p	matrix mode (with delimiters)
STD_MODE	p/r	standard mode (with element indices)
RSEQ_MODE	p/r	plain row-sequential mode (“tabular mode”)
CSEQ_MODE	p/r	plain column-sequential mode

B.4 Input and Output Operations

Print Format Specification

```
void set_pformat(long width, long prec, long imax, long jmax)
```

`set_pformat` sets new values for the format parameters `width`, `prec`, `imax`, and `jmax`, which are used by the print routines `v_fprint` and `m_fprint`. (See the corresponding documentation for further explanation of these parameters.) If no call is made to `set_pformat`, the default values `width=13`, `prec=6`, `imax=5`, and `jmax=5` are used.

Keyword Search in Text Files

```
long search_key(FILE *fp, char *key, ...)
```

`search_key` searches text file `fp` for a keyword specified by `key` (and possibly one or more further arguments). In the text file, each keyword string must start at the beginning of a line and must be terminated by at least one white space character; the rest of the corresponding line is ignored. `search_key` always searches from the beginning of the file, not from the current position of the file pointer. If the search is successful, the file pointer is positioned at the beginning of the line after the first occurrence of the keyword, and `search_key` returns the value 1. Otherwise, the file pointer is positioned at EOF, and the value 0 is returned. (See Section 2.5 for an example of usage.)

Vector Input and Output

```
long v_fprint(FILE *fp, XVec v, PRMode pm)
```

`v_fprint` prints vector `v` to text file `fp`. The output is formatted according to the print mode `pm`; valid print modes are `ROW_MODE`, `COL_MODE`, `STD_MODE`, `RSEQ_MODE`, and `CSEQ_MODE` (see Table 4). Individual elements are printed using the minimum field width and precision parameters `width` and `prec`, respectively. In `ROW_MODE` and `COL_MODE`, no more than `imax` elements are printed. (The print format parameters `width`, `prec`, and `imax` can be set by calling `set_pformat`.) `v_fprint` returns the number of vector elements printed.

```
long v_print(XVec v, PRMode pm)
```

`v_print` prints vector `v` to standard output, i.e., `v_print(v, pm)` is equivalent to `v_fprint(stdout, v, pm)`.

```
long v_fscan(FILE *fp, XVec v, PRMode rm)
```

`v_fscan` reads vector `v` from text file `fp`. The input file must be in a format equivalent to the format generated by `v_fprint` with print mode either `STD_MODE`, `RSEQ_MODE`, or `CSEQ_MODE` (see Table 4). This input format is specified as read mode `rm`. `v_fscan` returns the number of vector elements read.

`long v_fsscan(FILE *fp, XVec v, long start, long inc)`

`v_fsscan` reads selected data values from text file `fp` into vector `v`. `start` and `inc` denote the file position of the first vector element to be read and the file increment to be used between subsequent vector elements, respectively. Each value of type `S`, `D`, `I`, or `L` corresponds to one file position; therefore, *two* file positions are needed to represent a value of type `C` or `Z`. Observe that `v_fsscan(fp,v,1,1)` reads `v` in the usual manner without skipping data values. The input file must be in plain sequential format and may contain values of different types. For instance, assuming that `fp` contains five columns of data and all columns are of “simple” type (`S`, `D`, `I`, or `L`), the third data column can be read into `v` by `v_fsscan(fp,v,3,5)`. `v_fsscan` returns the number of vector elements read.

`long v_fsave(FILE *fp, XVec v)`

`v_fsave` saves vector `v` to binary file `fp`; the number of vector elements written is returned.

`long v_frestore(FILE *fp, XVec v)`

`v_frestore` restores vector `v` from binary file `fp`. The binary file must have been previously written by `v_fsave`. `v_frestore` returns the number of vector elements read.

Matrix Input and Output

`long m_fprint(FILE *fp, XMat m, PRMode pm)`

`m_fprint` prints matrix `m` to text file `fp`. The output is formatted according to the print mode `pm`; valid print modes are `MAT_MODE`, `STD_MODE`, `RSEQ_MODE`, and `CSEQ_MODE` (see Table 4). Individual elements are printed using the minimum field width and precision parameters `width` and `prec`, respectively. In `MAT_MODE`, no more than `imax` rows and `jmax` columns are printed. (The print format parameters `width`, `prec`, `imax`, and `jmax` can be set by calling `set_pformat`.) `m_fprint` returns the number of matrix elements printed.

`long m_print(XMat m, PRMode pm)`

`m_print` prints matrix `m` to standard output, i.e., `m_print(m,pm)` is equivalent to `m_fprint(stdout,m,pm)`.

`long m_fscan(FILE *fp, XMat m, PRMode rm)`

`m_fscan` reads matrix `m` from text file `fp`. The input file must be in a format equivalent to the format generated by `m_fprint` with print mode either `STD_MODE`, `RSEQ_MODE`, or `CSEQ_MODE` (see Table 4). This input format must be specified as read mode `rm`. `m_fscan` returns the number of matrix elements read.

`long m_fsave(FILE *fp, XMat m)`

`m_fsave` saves matrix `m` to binary file `fp`; the number of matrix elements written is returned.

`long m_frestore(FILE *fp, XMat m)`

`m_frestore` restores matrix `m` from binary file `fp`. The binary file must have been previously written by `m_fsave`. `m_frestore` returns the number of matrix elements read.

C BLAS Interface and BLAS Extensions

Important note: all arguments passed to BLAS and EBLAS routines must be variables in addressable memory (i.e., *lvalues*). An exception to this rule are the character strings used by Level 2–3 BLAS, where it is possible to pass string literals as in Fortran. For convenience, the predefined constants `X_ZERO`, `X_ONE`, and `X_MINUS_ONE` are available for the frequently needed scalar factors 0, 1, and -1 , respectively ($X = S, D, I, L$). No interfaces are provided for the Level 2 BLAS routines involving banded or packed matrices.

C.1 Level 1 BLAS Interface

```

void XROTG(XType a, XType b, XType c, XType s)           (X = S, D)
void XROT(XVec x, XVec y, XType c, XType s)           (X = S, D)
void XSWAP(XVec x, XVec y)                             (X = S, D, C, Z)
void XSCAL(XType alpha, XVec x)                       (X = S, D, C, Z)
void XYSCAL(YType alpha, XVec x)                     (XY = CS, ZD)
void XCOPY(XVec x, XVec y)                            (X = S, D, C, Z)
void XAXPY(XType alpha, XVec x, XVec y)              (X = S, D, C, Z)
XType XDOT(XVec x, XVec y)                            (X = S, D)
XType XDOTU(XVec x, XVec y)                           (X = C, Z)
XType XDOTC(XVec x, XVec y)                           (X = C, Z)
XType XNRM2(XVec x)                                    (X = S, D)
XType XYNRM2(YVec x)                                   (XY = SC, DZ)
XType XASUM(XVec x)                                    (X = S, D)
XType XYASUM(YVec x)                                   (XY = SC, DZ)
long IXAMAX(XVec x)                                    (X = S, D, C, Z)

```

C.2 Level 1 BLAS Extensions

(initialization and scaling of vectors)

```

void XVZERO(XVec x)                                     (X = S, D, C, Z, I, L)
    XVZERO sets all elements of vector x to zero (i.e.,  $x_i \leftarrow 0$ ).

void XVINIT(XType gamma, XVec x)                       (X = S, D, C, Z, I, L)
    XVINIT initializes all elements of vector x by value gamma (i.e.,  $x_i \leftarrow \gamma$ ).

void XVSEQ(XType gamma, XType delta, XVec x)          (X = S, D, C, Z, I, L)
    XVSEQ initializes the elements of vector x by a sequence of values starting at
    gamma and having increment delta (i.e.,  $x_i \leftarrow \gamma + i\delta$ ).

void XVMULT(XVec x, XVec y)                             (X = S, D, C, Z)
    XVMULT multiplies each element of vector y by the corresponding element of
    vector x (i.e.,  $y_i \leftarrow x_i y_i$ ).

void XAVMULT(XType alpha, XVec x, XVec y)            (X = S, D, C, Z)
    XAVMULT multiplies each element of vector y by alpha and the corresponding
    element of vector x (i.e.,  $y_i \leftarrow \alpha x_i y_i$ ).

```

void XVDIV(XVec x, XVec y) (X = S, D, C, Z)
 XVDIV divides each element of vector y by the corresponding element of vector x (i.e., $y_i \leftarrow y_i/x_i$). Vector x must not contain any zero elements.

void XAVDIV(XType alpha, XVec x, XVec y) (X = S, D, C, Z)
 XAVDIV divides each element of vector y by the corresponding element of vector x and in addition multiplies by α (i.e., $y_i \leftarrow \alpha y_i/x_i$). Vector x must not contain any zero elements.

C.3 Level 2 BLAS Interface

void XGEMV(char *trans, XType alpha, XMat a, XVec x, XType beta, XVec y) (X = S, D, C, Z)

void XHEMV(char *uplo, XType alpha, XMat a, XVec x, XType beta, XVec y) (X = C, Z)

void XSYMV(char *uplo, XType alpha, XMat a, XVec x, XType beta, XVec y) (X = S, D)

void XTRMV(char *uplo, char *trans, char *diag, XMat a, XVec x) (X = S, D, C, Z)

void XTRSV(char *uplo, char *trans, char *diag, XMat a, XVec x) (X = S, D, C, Z)

void XGER(XType alpha, XVec x, XVec y, XMat a) (X = S, D)

void XGERU(XType alpha, XVec x, XVec y, XMat a) (X = C, Z)

void XGERC(XType alpha, XVec x, XVec y, XMat a) (X = C, Z)

void XHER(char *uplo, XType alpha, XVec x, XMat a) (X = C, Z)

void XHER2(char *uplo, XType alpha, XVec x, XVec y, XMat a) (X = C, Z)

void XSYR(char *uplo, XType alpha, XVec x, XMat a) (X = S, D)

void XSYR2(char *uplo, XType alpha, XVec x, XVec y, XMat a) (X = S, D)

C.4 Level 2 BLAS Extensions

(initialization and scaling of matrices, matrix addition)

void XMZERO(XMat a) (X = S, D, C, Z, I, L)
 XMZERO sets all elements of matrix a to zero (i.e., $a_{ij} \leftarrow 0$).

void XMUNIT(XMat a) (X = S, D, C, Z, I, L)
 XMUNIT sets the diagonal elements of matrix a to one, all remaining elements to zero (i.e., $a_{ii} \leftarrow 1$ and $a_{ij} \leftarrow 0$ for $i \neq j$).

void XMDIAG(XType gamma, XMat a) (X = S, D, C, Z, I, L)
 XMDIAG initializes the diagonal elements of matrix a with value γ , all remaining elements are unaffected (i.e., $a_{ii} \leftarrow \gamma$).

void XMINIT(XType gamma, XMat a) (X = S, D, C, Z, I, L)
 XMINIT initializes all elements of matrix a with value γ (i.e., $a_{ij} \leftarrow \gamma$).

void XMSCAL(XType alpha, XMat a) (X = S, D, C, Z)
 XMSCAL multiplies all elements of matrix a by α (i.e., $a_{ij} \leftarrow \alpha a_{ij}$).

```

void XMRMULT(XVec x, XMat a) (X = S, D, C, Z)
    XMRMULT multiplies each row of matrix a by the corresponding element of
    vector x (i.e.,  $a_{ij} \leftarrow x_i a_{ij}$ ).

void XMRDIV(XVec x, XMat a) (X = S, D, C, Z)
    XMRDIV divides each row of matrix a by the corresponding element of vector x
    (i.e.,  $a_{ij} \leftarrow a_{ij}/x_i$ ). Vector x must not contain any zero elements.

void XMCMULT(XVec x, XMat a) (X = S, D, C, Z)
    XMCMULT multiplies each column of matrix a by the corresponding element of
    vector x (i.e.,  $a_{ij} \leftarrow x_j a_{ij}$ ).

void XMCDIV(XVec x, XMat a) (X = S, D, C, Z)
    XMCDIV divides each column of matrix a by the corresponding element of
    vector x (i.e.,  $a_{ij} \leftarrow a_{ij}/x_j$ ). Vector x must not contain any zero elements.

void XMADDDTO(XMat a, XMat b) (X = S, D, C, Z)
    XMADDDTO adds matrix a elementwise to matrix b (i.e.,  $b_{ij} \leftarrow a_{ij} + b_{ij}$ ).

void XMTADDDTO(XMat a, XMat b) (X = S, D, C, Z)
    XMTADDDTO adds the transpose of matrix a elementwise to matrix b (i.e.,  $b_{ij} \leftarrow$ 
     $a_{ji} + b_{ij}$ ).

```

C.5 Level 3 BLAS Interface

```

void XGEMM(char *transa, char *transb, XType alpha, (X = S, D, C, Z)
    XMat a, XMat b, XType beta, XMat c)

void XSYMM(char *side, char *uplo, XType alpha, (X = S, D, C, Z)
    XMat a, XMat b, XType beta, XMat c)

void XHEMM(char *side, char *uplo, XType alpha, (X = C, Z)
    XMat a, XMat b, XType beta, XMat c)

void XSYRK(char *uplo, char *trans, XType alpha, (X = S, D, C, Z)
    XMat a, XType beta, XMat c)

void XHERK(char *uplo, char *trans, XType alpha, (X = C, Z)
    XMat a, XType beta, XMat c)

void XSYR2K(char *uplo, char *trans, XType alpha, (X = S, D, C, Z)
    XMat a, XMat b, XType beta, XMat c)

void XHER2K(char *uplo, char *trans, XType alpha, (X = C, Z)
    XMat a, XMat b, XType beta, XMat c)

void XTRMM(char *side, char *uplo, char *transa, (X = S, D, C, Z)
    char *diag, XType alpha, XMat a, XMat b)

void XTRSM(char *side, char *uplo, char *transa, (X = S, D, C, Z)
    char *diag, XType alpha, XMat a, XMat b)

```

D LAPACK Interface

Important note: all arguments passed to LAPACK routines must be variables in addressable memory (i.e., *values*). An exception to this rule are character strings—string literals can be directly passed as in Fortran. Arguments of type `long*` (e.g., `info`) correspond to output parameters and therefore must be passed by address. The same is true for some arguments of type `char*` or `XType*`, see the LAPACK manual [1]. No interfaces are provided for routines involving tridiagonal, banded, or packed matrices.

D.1 Driver Routines for Linear Equations

Simple Driver Routines

```
void XGESV(XMat a, LVec ipiv, XMat b, long *info)      (X = S, D, C, Z)
void XPOSV(char *uplo, XMat a, XMat b, long *info)   (X = S, D, C, Z)
void XSYSV(char *uplo, XMat a, LVec ipiv, XMat b,    (X = S, D, C, Z)
    long *info)
void XHESV(char *uplo, XMat a, LVec ipiv, XMat b,    (X = C, Z)
    long *info)
```

Expert Driver Routines

```
void XGESVX(char *fact, char *trans, XMat a, XMat af, (X = S, D)
    LVec ipiv, char *equed, XVec r, XVec c, XMat b,
    XMat x, XType *rcond, XVec ferr, XVec berr,
    long *info)
void XGESVX(char *fact, char *trans, XMat a, XMat af, (XY = CS, ZD)
    LVec ipiv, char *equed, YVec r, YVec c, XMat b,
    XMat x, YType *rcond, YVec ferr, YVec berr,
    long *info)
void XPOSVX(char *fact, char *uplo, XMat a, XMat af, (X = S, D)
    char *equed, XVec s, XMat b, XMat x, XType *rcond,
    XVec ferr, XVec berr, long *info)
void XPOSVX(char *fact, char *uplo, XMat a, XMat af, (XY = CS, ZD)
    char *equed, YVec s, XMat b, XMat x, XType *rcond,
    YVec ferr, YVec berr, long *info)
void XSYSVX(char *fact, char *uplo, XMat a, XMat af, (X = S, D)
    LVec ipiv, XMat b, XMat x, XType *rcond, XVec ferr,
    XVec berr, long *info)
void XSYSVX(char *fact, char *uplo, XMat a, XMat af, (XY = CS, ZD)
    LVec ipiv, XMat b, XMat x, YType *rcond, YVec ferr,
    YVec berr, long *info)
void XHESVX(char *fact, char *uplo, XMat a, XMat af, (XY = CS, ZD)
    LVec ipiv, XMat b, XMat x, YType *rcond, YVec ferr,
    YVec berr, long *info)
```

D.2 Driver Routines for Linear Least Squares Problems

Simple Driver Routines

```
void XGELS(char *trans, XMat a, XMat b, long *info)   (X = S, D, C, Z)
void XGGLSE(XMat a, XMat b, XVec c, XVec d, XVec x, (X = S, D, C, Z)
    long *info)
void XGGGLM(XMat a, XMat b, XVec d, XVec x, XVec y, (X = S, D, C, Z)
    long *info)
```

Expert Driver Routines

```

void XGELSX(XMat a, XMat b, LVec jpv, XType rcond,      (X = S, D)
           long *rank, long *info)
void XGELSX(XMat a, XMat b, LVec jpv, YType rcond,    (XY = CS, ZD)
           long *rank, long *info)
void XGELSS(XMat a, XMat b, XVec s, XType rcond,      (X = S, D)
           long *rank, long *info)
void XGELSS(XMat a, XMat b, YVec s, YType rcond,     (XY = CS, ZD)
           long *rank, long *info)

```

D.3 Driver Routines for Standard Eigenvalue and Singular Value Problems**Simple Driver Routines**

```

void XSYEV(char *jobz, char *uplo, XMat a, XVec w,      (X = S, D)
           long *info)
void XHEEV(char *jobz, char *uplo, XMat a, YVec w,     (XY = CS, ZD)
           long *info)
void XSYEVD(char *jobz, char *uplo, XMat a, XVec w,    (X = S, D)
           long *info)
void XHEEVD(char *jobz, char *uplo, XMat a, YVec w,    (XY = CS, ZD)
           long *info)
void XGEES(char *jobvs, char *sort,                    (X = S, D)
           long (*select)(XType *wrj, XType *wij), XMat a,
           long *sdim, XVec wr, XVec wi, XMat vs, long *info)
void XGEES(char *jobvs, char *sort,                    (X = C, Z)
           long (*select)(XType *wj), XMat a, long *sdim,
           XVec w, XMat vs, long *info)
void XGEEV(char *jobvl, char *jobvr, XMat a, XVec wr,  (X = S, D)
           XVec wi, XMat vl, XMat vr, long *info)
void XGEEV(char *jobvl, char *jobvr, XMat a, XVec w,  (X = C, Z)
           XMat vl, XMat vr, long *info)
void XGESVD(char *jobu, char *jobvt, XMat a, XVec s,   (X = S, D)
           XMat u, XMat vt, long *info)
void XGESVD(char *jobu, char *jobvt, XMat a, YVec s,  (XY = CS, ZD)
           XMat u, XMat vt, long *info)

```

Expert Driver Routines

```

void XSYEVX(char *jobz, char *range, char *uplo,      (X = S, D)
           XMat a, XType vl, XType vu, long il, long iu,
           XType abstol, long *m, XVec w, XMat z, LVec ifail,
           long *info)
void XHEEVX(char *jobz, char *range, char *uplo,      (XY = CS, ZD)
           XMat a, YType vl, YType vu, long il, long iu,
           YType abstol, long *m, YVec w, XMat z, LVec ifail,
           long *info)

```

```

void XGEESX(char *jobvs, char *sort,                (X = S, D)
    long (*select)(XType *wrj, XType *wij), char *sense,
    XMat a, long *sdim, XVec wr, XVec wi, XMat vs,
    XType *rconde, XType *rcondv, long *info)

void XGEESX(char *jobvs, char *sort,                (XY = CS, ZD)
    long (*select)(XType *wj), char *sense,
    XMat a, long *sdim, XVec w, XMat vs,
    YType *rconde, YType *rcondv, long *info)

void XGEEVX(char *balanc, char *jobvl, char *jobvr, (X = S, D)
    char *sense, XMat a, XVec wr, XVec wi, XMat vl, XMat vr,
    long *ilo, long *ihi, XVec scale, XType *abnrm,
    XVec rconde, XVec rcondv, long *info)

void XGEEVX(char *balanc, char *jobvl, char *jobvr, (XY = CS, ZD)
    char *sense, XMat a, XVec w, XMat vl, XMat vr,
    long *ilo, long *ihi, YVec scale, YType *abnrm,
    YVec rconde, YVec rcondv, long *info)

```

D.4 Driver Routines for Generalized Eigenvalue and Singular Value Problems

```

void XSYGV(long itype, char *jobz, char *uplo,      (X = S, D)
    XMat a, XMat b, XVec w, long *info)

void XHEGV(long itype, char *jobz, char *uplo,      (XY = CS, ZD)
    XMat a, XMat b, YVec w, long *info)

void XGEGS(char *jobvsl, char *jobvsr,             (X = S, D)
    XMat a, XMat b, XVec alphas, XVec alphas_i, XVec betas,
    XMat vsl, XMat vsr, long *info)

void XGEGS(char *jobvsl, char *jobvsr,             (X = C, Z)
    XMat a, XMat b, XVec alpha, XVec beta,
    XMat vsl, XMat vsr, long *info)

void XGEGV(char *jobvl, char *jobvr,               (X = S, D)
    XMat a, XMat b, XVec alphas, XVec alphas_i, XVec betas,
    XMat vl, XMat vr, long *info)

void XGEGV(char *jobvl, char *jobvr,               (X = C, Z)
    XMat a, XMat b, XVec alpha, XVec beta,
    XMat vl, XMat vr, long *info)

void XGGSVD(char *jobu, char *jobv, char *jobq,     (X = S, D)
    long *k, long *l, XMat a, XMat b, XVec alpha, XVec beta,
    XMat u, XMat v, XMat q, long *info)

void XGGSVD(char *jobu, char *jobv, char *jobq,     (XY = CS, ZD)
    long *k, long *l, XMat a, XMat b, YVec alpha, YVec beta,
    XMat u, XMat v, XMat q, long *info)

```

References

- [1] Anderson, E., Z. Bai, C. Bischof, J. W. Demmel, J. J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen: LAPACK Users' Guide. Second Edition. SIAM, Philadelphia, PA, 1995.

- [2] Coleman, T. F., and C. Van Loan: Handbook for Matrix Computations. SIAM, Philadelphia, PA, 1988.
- [3] Dongarra, J. J., J. Du Croz, S. Hammarling, and R. J. Hanson: An extended set of Fortran Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 14(1):1–17, 1988.
- [4] Dongarra, J. J., J. Du Croz, S. Hammarling, and I. Duff: A set of Level 3 Basic Linear Algebra Subprograms. *ACM Trans. Math. Softw.* 16(1):1–17, 1990.
- [5] Dongarra, J. J., R. Pozo, and D. W. Walker: LAPACK++: A Design Overview of Object-Oriented Extensions for High-Performance Linear Algebra. Computer Science Technical Report, University of Tennessee, 1993.
- [6] Kernighan, B. W., and D. M. Ritchie: The C Programming Language. Second Edition. Prentice Hall, Englewood Cliffs, NJ, 1988.
- [7] Lawson, C. L., R. J. Hanson, D. R. Kincaid, and F. T. Krogh: Basic Linear Algebra Subprograms for Fortran usage. *ACM Trans. Math. Softw.* 5(3):308–323, 1979.
- [8] Leineweber, D. B.: The Theory of MUSCOD in a Nutshell. Diploma thesis, University of Heidelberg, 1995. (Appeared also as IWR-Preprint 96-19. University of Heidelberg, 1996.)
- [9] Leineweber, D. B., H. G. Bock, J. P. Schlöder, A. Schäfer, and P. Jansohn: Efficient Techniques for the Optimization of Complex Chemical Processes. IWR-Preprint 96–40, University of Heidelberg, 1996.
- [10] Metcalf, M.: Effective Fortran 77. Clarendon Press, Oxford, 1985.
- [11] Stroustrup, B.: The C++ Programming Language. Second Edition. Addison-Wesley, Reading, MA, 1993.
- [12] The NAG FORTRAN Libraray Manual. Mark 14. Wilkinson House, Jordan Hill Road, Oxford OX2 8DR.