BLAS and LAPACK

1 Introduction

Linear algebra is an essential tool in computational mathematics, and software for numerical linear algebra is a key component in many mathematical software packages. Matrix Laboratory, or MATLAB, is a well-known software package that makes it easy for the user to manipulate matrices and vectors, and it provides a comprehensive set of numerical linear algebra routines for various matrix decompositions and computations. Unlike C, MATLAB is an interpreted language, and it is great for fast prototyping, experimentation, and research. However, MATLAB is proprietary and has strict hardware requirements which means that it is not well-suited for all applications. However, implementing a set of high-quality routines for numerical linear that one can use in a stand-alone program is time-consuming and by all means nontrivial, but luckily there are a number of free libraries that a both free and of very high quality.

In this week's exercises, we will work with two external libraries that provide a set of numerical linear algebra routines, namely the BLAS (an abbreviation of *Basic Linear Algebra Subroutines*) and LAPACK (an abbreviation of *Linear Algebra PACKage* and pronounced *L-A-PACK*). The BLAS and LAPACK libraries were developed in late 1970s in FORTRAN, a programming language that was very popular at the time (and still is today in some communities). The reference implementations, which are available at netlib.org, are still FORTRAN code, but their technical specification has been standardized, so today there exists a number optimized implementations such as MKL (Intel's *Math Kernel Library*), ACML (*AMD Core Math Library*), ATLAS (*Automatically Tuned Linear Algebra Software*), and OpenBLAS. While these libraries may not be implemented in FORTRAN, they all adhere to the BLAS specification, and the compiled libraries can be used with many different programming languages simply by linking against the compiled library.

1.1 BLAS

The BLAS provides a number of routines that are specified in the BLAS Technical Forum Standard. The BLAS routines are divided into three levels/categories:

- BLAS level 1 routines implement basic vector operations (scaling a vector, adding two vectors, etc.)
- BLAS level 2 routines implement basic matrix-vector operations (matrix-vector multiplication, solving triangular systems of equations, etc.)
- BLAS level 3 routines implement matrix-matrix operations (matrix-matrix multiplication, etc.).

The BLAS routines have mnemonic names. For example, the routine for adding a scalar multiple of a vector to another vector is called **daxpy** which is a mnemonic for remembering *double (precision) a x plus y*. The same operation for single precision also exists, and it has the name **saxpy** (*single (precision) a x plus y*). You can find a complete list of the routines that are part of the BLAS library on the BLAS website.

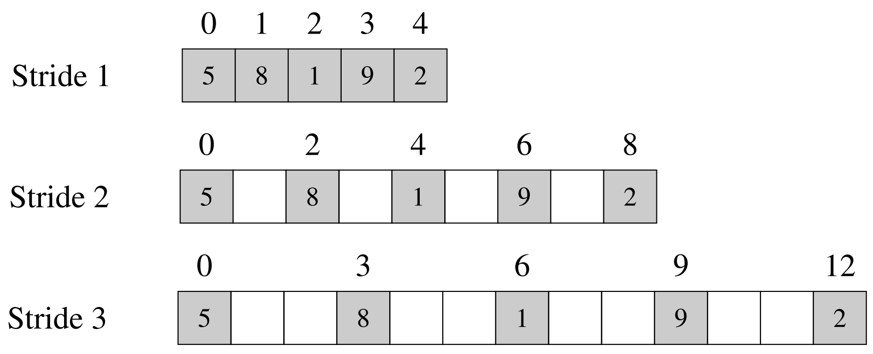
1.2 LAPACK

The LAPACK library provides a number of matrix factorization routines (LU, Cholesky, QR, etc.), routines for computing eigenvalues and singular values, routines for solving linear equations, and routines for solving linear least-squares problems. Many routines in the LAPACK library make use of routines from the BLAS library, and hence both libraries are necessary when using LAPACK (MKL, ACML, and OpenBLAS contain both BLAS and LAPACK routines). The LAPACK routines also have mnemonic names. For example, the **dgels** routine solves a least-squares problem (LS) with an unstructured or general (GE) coefficient matrix with input in double precision (D).

2 Storage schemes

2.1 Vectors

Vectors are represented as strided arrays. In C, this boils down to three things: the length of the vector, a pointer to its first element, and an integer stride. The following figure shows three different strided array representations of a vector of length 5.



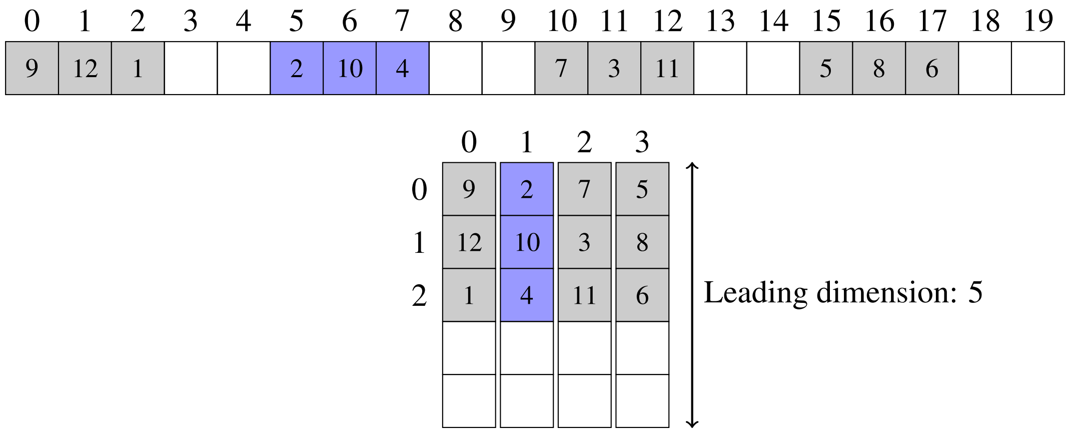
2.2 Matrices

The BLAS and LAPACK libraries contain routines for a variety of matrix storage schemes. These include the following:

- *full storage* (two-dimensional array, column-major storage order),
- *packed storage* (compact representation of triangular/symmetric/Hermitian matrices),
- *band storage* (compact representation of band matrices).

2.2.1 Full storage

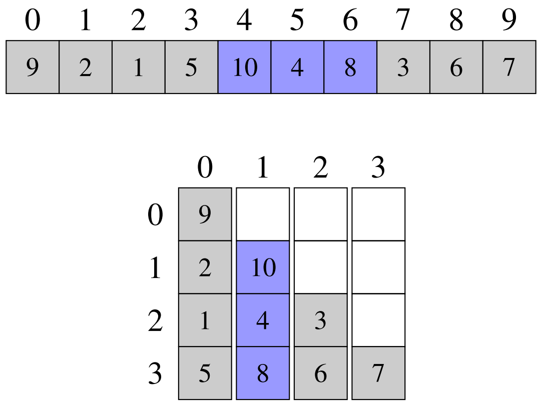
Full storage of a general matrix of size $m \times n$ is essentially a two-dimensional array using the column-major storage order. The m elements in each column are stored contiguously, but neighboring elements in a row may be more than m elements apart in memory, as illustrated in the following figure.



The matrix in the figure is of size 3×4, but the offset from one column to the next, which is referred to as the *leading dimension* of its representation, is equal to 5. Clearly, the leading dimension must be greater than or equal to m. Note that the leading dimension is also the stride of the rows. In other words, if $M \geq m$ is the leading dimension, then $i + jM$ is the one-dimensional index that corresponds to the (i, j)th entry.

2.2.2 Packed storage

Packed storage provides a memory efficient way to store triangular, symmetric, and Hermitian matrices, and it is basically the lower- or upper-triangular elements of the matrix in column-major storage order. The following figure illustrates the packed storage format for a lower-triangular matrix of size 4×4.



Note that this format requires an array of length $n(n+1)/2$ rather than n^2 to store a triangular/symmetric/Hermitian matrix.

2.2.3 Band storage

An $m \times n$ band matrix with l subdiagonals and u superdiagonals can be stored compactly in a two-dimensional array of size $(l+u+1) \times n$. This is a substantial reduction in storage when $l+u+1 \ll m$.

To illustrate the storage format, consider the following 5×6 matrix with $l = 2$ subdiagonals and $u = 1$ superdiagonal:

$$A = \begin{bmatrix} a_{11} & a_{12} & & & & \\ a_{21} & a_{22} & a_{23} & & & \\ a_{31} & a_{32} & a_{33} & a_{34} & & \\ & a_{42} & a_{43} & a_{44} & a_{45} & \\ & & a_{53} & a_{54} & a_{55} & a_{56} \end{bmatrix}.$$

The band storage representation of this matrix is show in the following figure:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
	a_{11}	a_{21}	a_{31}	a_{12}	a_{22}	a_{32}	a_{42}	a_{23}	a_{33}	a_{43}	a_{53}	a_{34}	a_{44}	a_{54}		a_{45}	a_{55}			a_{56}			

	0	1	2	3	4	5
0		a_{12}	a_{23}	a_{34}	a_{45}	a_{56}
1	a_{11}	a_{22}	a_{33}	a_{44}	a_{55}	
2	a_{21}	a_{32}	a_{43}	a_{54}		
3	a_{31}	a_{42}	a_{53}			

Note that symmetric or Hermitian band matrices of order n can be stored in a two-dimensional array of size $(l+1) \times n$ by storing either the upper-triangular or lower-triangular entries.

3 Using routines from the BLAS library in a C program

We will now illustrate how to making use a routine from the BLAS library in a C program. To this end, we will use the `dscal` routine to scale the entries of a double-precision vector by a constant. The following snippet from the dscal reference implementation may be viewed as the Fortran equivalent to a function prototype in C:

```
SUBROUTINE dscal(N,DA,DX,INCX)
* .. Scalar Arguments ..
DOUBLE PRECISION DA
INTEGER INCX,N
* .. Array Arguments ..
DOUBLE PRECISION DX(*)
```

This translates to the following C prototype:

```
/* DSCAL (scale array)                              */
void dscal_(
  const int * n,           /* length of array       */
  const double * a,         /* scalar a              */
  double * x,               /* pointer to first elem. */
  const int * incx          /* array stride          */
);
```

First, note that all the arguments are pointers. This has to do with the fact that, in general, Fortran passes arguments by reference whereas C passes arguments by value. Note also that the function name in the C prototype has a trailing underscore. This is something that Fortran compiler normally appends, and hence it may be necessary in a C program in order to link against an external BLAS library.

The following program scales the elements of an array of length 5 using `dscal` from the BLAS library:

```
/* example_blas.c */
#include <stdio.h>
void dscal_(const int *n, const double *a, double *x, const int *incx);

int main(void) {

  double a=3.0, x[]={1.0, 2.0, 3.0, 4.0, 5.0};
  int incx=1, n=sizeof(x)/sizeof(double);

  dscal_(&n,&a,x,&incx);

  for (int i=0;i<n;i++)
    printf("x[%d] = %.2g\n",i,x[i]);

  return 0;
}
```

We will use the following makefile to compile our program and link against the BLAS library:

```
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=-lblas
```

Notice that the variable `LDLIBS` has the value `-lblas`. This instructs the linker to look for undefined functions (such as `dscal_`) in a library called `blas`. The library name depends on the available BLAS library on your system (if any). The linker will fail if it is unable to find a library with the specified name. In this case, we will need to check that the library name is correct and/or to provide the linker with the path to the library. The path can be specified by adding a flag of the form `-Lpath/to/blas/` to the `LDFLAGS` variable in the makefile.

4 CBLAS — a C-style interface to the BLAS routines

As we saw in the previous section, we needed a function prototype to compile a program that calls a routine from an external BLAS library. Moreover, all the input arguments are pointers, and this can be somewhat awkward when programming in C. An alternative is to link against "CBLAS" which is a C-style interface to the BLAS routines. CBLAS comes with a header file with prototypes and various enumeration types. The following example illustrates how to call `dscal` via its CBLAS interface `cblas_dscal`:

```c
/* example_cblas.c */
#include <stdio.h>
#if defined(__APPLE__) && defined(__MACH__)
#include <Accelerate/Accelerate.h>
#else
#include <cblas.h>
#endif

int main(void) {

  double a=3.0, x[]={1.0, 2.0, 3.0, 4.0, 5.0};
  int incx=1, n=sizeof(x)/sizeof(double);

  cblas_dscal(n,a,x,incx);

  for (int i=0;i<n;i++)
    printf("x[%d] = %.2g\n",i,x[i]);

  return 0;
}
```

We will use the following makefile to compile the program and link against the CBLAS library:

```makefile
CC=gcc
CPPFLAGS=
CFLAGS=-Wall -std=c99
LDFLAGS=
LDLIBS=-lcblas
```

Notive that `LDLIBS` now has the value `-lcblas`. If the preprocessor cannot find the header file `cblas.h`, then we can instruct the preprocessor to look for header files in some directory, say, `path/to/header/files/`, by adding the flag `-Ipath/to/header/files/` to the `CPPFLAGS` variable in our makefile. Moreover, if linking fails, it may be necessary to add the path to the CBLAS library to `LDFLAGS` so that the linker can find it.