Lecture 11: Libraries
BLAS, LAPACK & Eigen

Jonas Kusch and Martina Prugger
University of Innsbruck

May 2, 2023

Last goals: You are able to

- ☑ smart pointers
- ☑ header files
- ☑ makefiles
- ☑ cmake

Last goals: You are able to

- ☑ smart pointers
- ☑ header files
- ☑ makefiles
- ☑ cmake

Today's learning goals: Today we will learn about

- ☐ libraries

Last goals: You are able to

- ☑ smart pointers
- ☑ header files
- ☑ makefiles
- ☑ cmake

Today's learning goals: Today we will learn about

- ☐ libraries

Ask questions any time!

Motivation

- a lot of numerical algorithms are formulated in matrix notation
  - LU decomposition (direct solver)
  - conjugate gradients (iterative solver)
  - ...
- vectors and matrices are manipulated until a solution is obtained
- even if the algorithm is straight forward, an efficient implementation is not
  - the efficient implementation of a matrix transpose might require sophisticated optimizations in order to obtain good performance on modern computer systems
- there is a relatively small set of fundamental matrix and vector operations

$\longrightarrow$ a standardized set of routines has emerged

Motivation

$\longrightarrow$ a standardized set of routines has emerged

- various publicly available implementations
- most vendors such as Intel, AMD, and NVIDIA provide versions of this interface optimized for a particular architecture
- often relatively easy to switch from one implementation to another or even from one hardware platform to another
  - BLAS and LAPACK implementations are available on GPUs as well

Basic Linear Algebra Subprograms (BLAS)

- dense linear algebra
- collection of functions that can be used to perform low-level matrix-vector operations
- routines follow a naming scheme which is given as follows
  - first letter is the data type used
    - s ... single precision
    - d ... double precision
    - c ... single precision, complex number
    - z ... double precision, complex number
  - remainder: name of the routine (historically limited to four letters)
- BLAS routines are classified into three different levels

BLAS level 1: vector operations

- **axpy** computes $y = a \cdot x + y$
  - a ... scalar
  - x, y ... vectors
- **sdot** computes the dot product of two vectors
- **asum** computes the sum of absolute values
- **scal** computes $x = a \cdot x$
  - a ... scalar
  - x ... vector
- **nrm2** computes the l^2 norm of a vector
- **copy** computes $x = y$
  - x, y .. vectors

To call these functions, we have to prepend either s, d, c, or z to the function name to specify the desired data type. For example *dsdot* is the function used to compute the dot product using double precision floating point numbers.

BLAS level 2: matrix-vector operations

In this case the name of the function is subdivided into two parts

- first part: type of the matrix used
- second part: name of operation

For example *sgemv* operates on single precision data (s), expects a general full matrix (ge), and performs a matrix-vector multiplication (mv).

BLAS level 2: matrix-vector operations

BLAS level 2 the following matrix types can be used

- **ge** ... A generic full matrix that is stored as a single block of memory. Every matrix can be stored in this format.
- **sy** ... A generic symmetric matrix that is stored as a single block of memory. The difference with ge is that the algorithm can assume that the matrix is symmetric and that only the upper or the lower part (specified by an argument to the function) of the matrix (in both cases including the diagonal) has to be set. The remaining elements are not read by the algorithm. Note that using the sy function does **not** save storage compared to ge. Any symmetric matrix can be stored in this format
- **tr** ... A generic triangular matrix that is stored as a single block of memory. The algorithm can assume that the matrix is triangular (i.e. that all entries below or above the diagonal are zero). Note that using the tr type does **not** save storage compared to ge. Any triangular matrix can be stored in tr.

BLAS level 2: matrix-vector operations

- **gb** ... A $m \times n$ banded matrix with l subdiagonals and u superdiagonals is stored as a two-dimensional array with $l + u + 1$ rows and n columns. Columns of the matrix are stored in the corresponding columns of the array and (sub/sup) diagonals of the matrix are stored in rows of the array.

As an example, let us consider the following matrix

$$A = \begin{bmatrix} a_{11} & a_{12} & 0 & 0 & 0 \\ a_{21} & a_{22} & a_{23} & 0 & 0 \\ 0 & a_{32} & a_{33} & a_{34} & 0 \\ 0 & 0 & a_{43} & a_{44} & a_{45} \\ 0 & 0 & 0 & a_{54} & a_{55} \end{bmatrix}$$

which has $l = 1$ subdiagonals and $u = 1$ superdiagonals and is thus stored in an array with 3 rows and 5 columns.

BLAS level 2: matrix-vector operations; gb continuation

The matrix is stored in an array with 3 rows and 5 columns as follows

$$\begin{bmatrix} & a_{12} & a_{23} & a_{34} & a_{45} \\ a_{11} & a_{22} & a_{33} & a_{44} & a_{55} \\ a_{21} & a_{32} & a_{43} & a_{54} & \end{bmatrix}$$

- The blank elements are not used but have to be stored in memory
  - they are wasted in some sense
  - for large matrices they contribute only a small fraction of the total storage space needed
- In order to reconstruct the matrix A we also have to know the distance of the different bands from the diagonal
  - which sub- and superdiagonal of the matrix is not equal to zero
- Banded matrices are particularly important in applications
  - most stencil codes can be formulated as a banded matrix-vector multiplication
  - in this case the storage required is drastically reduced
  - however, this is still inefficient as for a two-dimensional problem we have only three non-zero superdiagonals but have to choose $l = u = N$.

BLAS level 2: matrix-vector operations

- **tb** ... A triangular banded matrix is stored in the same way as a banded matrix. However, only half of the matrix has to bet set.
- **sb** ... The symmetric banded storage scheme is similar to gb but since the matrix is symmetric only half of the matrix has to be set.
- **sp** ... The symmetric packed matrix storage scheme is similar to sy but stores the symmetric matrix in a one-dimensional array. Thus,

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} & a_{15} \\ a_{12} & a_{22} & a_{23} & a_{24} & a_{25} \\ a_{13} & a_{23} & a_{33} & a_{34} & a_{35} \\ a_{14} & a_{24} & a_{34} & a_{44} & a_{45} \\ a_{15} & a_{25} & a_{35} & a_{45} & a_{55} \end{bmatrix}$$

is stored as

$$[a_{11} \quad a_{12} \quad a_{13} \quad a_{14} \quad a_{15} \quad a_{12} \quad \ldots \quad a_{45} \quad a_{55}].$$

This reduces the amount of memory required by one-half as compared to sy.

BLAS level 2: matrix-vector operations

- **tp** ... The triangular packed matrix storage scheme is equivalent to sp but instructs the algorithm to assumea triangular matrix. It reduces the amount of memory required by one-half as compared to tr.

By the description of the matrix types we have seen that the BLAS interface is mostly geared towards dense matrix algebra. However, it also includes banded storage schemes which are often useful for the discretization of PDEs. Now, let us turn to the actual matrix operations that are implemented in BLAS level 2.

- **mv** ... Multiplies a matrix with a vector resulting in a vector.
- **sv** ... solves a linear system of equations.
  - For this operation only the matrices of type tr, tb, and tp can be used!

BLAS level 3: matrix-matrix operations

- **mm** ... Computes a matrix-matrix multiplication.
- **sm** ... Solve a triangular system of equations with multiple right hand sides.

We should note that especially matrix-matrix multiplication is quite difficult to implement efficiently and large gains in performance are expected by using a tuned BLAS library as compared to a hand written implementation.

BLAS and CMAKE: prerequisites

- folder structure in working directory:
  - directory: build
  - directory: src
  - file: CMakeLists.txt
- install ccmake
  - CMake curses interface
  - Project configuration settings may be specified interactively through this GUI
  - Brief instructions are provided at the bottom of the terminal when the program is running

```
1 sudo apt install cmake-curses-gui
```

- download *blas.cpp* and *timer.hpp* from OLAT and put into *src* directory

BLAS and CMAKE: CMakeLists.txt

```
cmake_minimum_required( VERSION 3.16 )

project( C++_lecture )

find_package( BLAS REQUIRED )

add_executable( blas src/blas.cpp)
target_include_directories( blas PUBLIC ${BLAS_INCLUDE_DIR} )
target_link_libraries( blas PUBLIC ${BLAS_LIBRARIES} )
```

BLAS and CMAKE: cmake the build

ccmake the CMakeLists.txt i.e., if you are in your build directory:

```
ccmake ..
```

- An interface opens $\longrightarrow$ use *c* to configure (sometimes needs to be pressed twice)
- No warnings $\longrightarrow$ use *g* to generate the appropriate make file

Now, you have multiple files (including a Makefile) in your build directory. Now, in the build directory *make* the executable by typing

```
make
```

If you have OpenMP installed on your computer and know how to set it up, you can also uncomment the respective lines and *make* again. Run the newly created executable

```
./blas
```

and compare the runtimes of the code.

Note If you want to use the BLAS library on the command line without *cmake*, you can link it via

```
g++ blas.cpp -O3 -o blas -lblas
```

the Linear Algebra Package (LAPACK)

- Implementations are usually based on BLAS.
- Provides more complicated matrix operations (e.g., solution of a linear system of equations).
- Naming conventions similar to the BLAS level 2 and 3 routines in that we have to prepend the algorithm with the desired data type and the matrix type we wish to use.
- The most important operations:
  - **sv** ... Solves a linear system of equations (possibly with multiple right-hand sides). This operation is supported for both dense (for example, ge) as well as sparse (for example, gb) storage schemes.
  - **ls** ... Computes the least square solution of an over-determined system of linear equations.
  - **ev** ... Computes all eigenvalues and eigenvectors of a matrix.
  - **vd** ... Computes the singular value decomposition (SVD) of a matrix.

Note that to call the sv operation we have to actually use *dgesv* to perform a double precision generic matrix linear solve.

LAPACK and CMAKE: preliminary information

- LAPACK is a FORTRAN library.
- Using the command line to link the LAPACK library works similar as for the BLAS library.
- Due to the intercommunication with FORTRAN, linking LAPACK with *cmake* is a bit trickier.
- The documentation can be found on `https://netlib.org/lapack`.
- Note that in FORTRAN, every function parameter is a pointer
  - *real* translates to float
  - *dimension* denotes an array and the corresponding size of the array
- In case, *make* goes wrong, one can check, whether *cmake* put the links correctly by setting the verbosity to 1

```
1    VERBOSE=1 make
```

- use the *ccmake* interface to figure out, what to set for *target_link_libraries*, by pressing *t* and looking for the LAPACK installation

LAPACK and CMAKE v1: prerequisites

- Download the file *lapack_v1.cpp* into your *src* directory.
- Make sure that LAPACK is installed on your computer (it is very likely, that another program has already put it on your computer, but it is not necessarily the case!)
- You can check for LAPACK, e.g., by using

```
2    ls /usr/lib/x86_64-linux-gnu/liblapack.so
```

- If it is not installed and you are not sure how it might be called, you can use

```
3    apt-cache search lapack
```

- set up the *CMakeLists.txt* file accordingly

```
find_package( LAPACK REQUIRED )

add_executable( lapack1 src/lapack_v1.cpp)
target_link_libraries( lapack1  ${LAPACK_lapack_LIBRARY} )
```

LAPACK and CMAKE v1

- *ccmake*, *make*, and execute code

If we look at the file, we see the following code:

```
extern "C"{
        void dgesv_(int *N, int *RHS, double *A, int *LDA,
                    int *IPIV, double *B, int *LDB, int *INFO);
}
```

This is the interface to tell C++, which variables it needs to expect for the function that it sends to FORTRAN. To avoid this translation, there is also the possibility of a C++ interface, using the library *LAPACKE*.

LAPACK and CMAKE v2: prerequisites

- Download the file *lapack_v2.cpp* into your *src* directory.
- Create an additional folder on the same level as *src* and *build*, called *cmake − modules*.
- The library needs to be found for cmake. This has been done already for you: go to `https://raw.githubusercontent.com/optimad/bitpit/master/external/LAPACKE/cmake/FindLAPACKE.cmake` and put the file into the cmake-modules folder.
- Make sure that LAPACKE is installed on your computer. You can use

```
4    apt-cache search lapacke
```

LAPACK and CMAKE v2: CMakeLists file

- set up the *CMakeLists.txt* file accordingly: after the project specification, set the cmake-modules folder as module path

```
set(CMAKE_MODULE_PATH ${CMAKE_MODULE_PATH}
                      "${CMAKE_SOURCE_DIR}/cmake-modules")

find_package( LAPACK REQUIRED )
find_package( LAPACKE REQUIRED )

add_executable( lapack2 src/lapack_v2.cpp)
target_link_libraries( lapack2  ${XXX} )
target_include_directories( lapack2 PUBLIC ${YYY} )
```

- use *ccmake* to determine what to put instead of *XXX* and *YYY*. (Hint: look at the source code)

Eigen: the C++ way to do it

- Pro
  - Eigen has no other dependency than the C++ standard library.
  - Offers some features, BLAS and LAPACK do not (built-in support for sparse matrices and vectors, provides a lot of convenience features (see Geometry module, Array module, etc), which are also very widely used)
  - Using only one thread, Eigen compares very well performance-wise against the existing BLAS implementations.
  - The syntax is as close to numpy/matlab, as C++ permits.
- Con:
  - Eigen is a very heavy library $\rightarrow$ not necessarily clear, what it is doing. It can happen, that Eigen performs terrible compared to BLAS, but most of the time it performs similar.
  - LAPACK is a standardized API, while Eigen is very x86-architecture specific. You might not have Eigen available on, e.g, GPUs
- Summary: you can probably use Eigen for 90% of use cases. If you find out, however, that a matrix operation is the bottleneck in your application and you need performance, you should probably switch to BLAS/LAPACK.

Eigen

- Eigen can be easily installed via the Linux installer

```
sudo apt update && sudo apt upgrade
sudo apt install libeigen3-dev

dpkg -L libeigen3-dev
```

- In CMakeLists.txt, include Eigen:

```
find_package(Eigen3 3.3 REQUIRED NO_MODULE)

add_executable (example example.cpp)
target_link_libraries (example Eigen3::Eigen)
```

- Use Eigen in your code via includes:

```
#include <Eigen/Dense>

using namespace Eigen;
```