# Introduction to **make**, BLAS and LAPACK

李阳 周嘉鑫

March 19, 2020

# Contents

# Overview of **make**

- **make** is a program for directing recompilation, which automatically determines which pieces of a large program need to be recompiled, and issues commands to recompile them.
- Why **make**?
  1. Reduce unnecessary repeated work.
  2. **make** provides a way of efficiently codifying and organizing the instructions to be executed when building a program.
- You need a file called **Makefile** to tell **make** what to do. Most often, the makefile tells **make** how to compile and link a program.
- Issue the following command to install **make**:

```
sudo apt-get install make
```

# The Syntax of Makefiles

- A makefile consists of a set of dependencies and rules.

```
target : prerequisites
        rule
```

- *Note: put a Tab character at the beginning of the rule line!*
- A `target` is usually an executable or object file. It can also be the name of an action to carry out, such as '`clean`'
- A `prerequisite` is a file that is used as input to create the target. A target often depends on several files.
- The *rules* describe how to create the target from the prerequisites.

# A Simple Makefile

```
main : main.o add.o sub.o mul.o
    g++ -o main main.o add.o sub.o mul.o
main.o : main.cpp arithmetic.h
    g++ -c main.cpp
add.o : add.cpp arithmetic.h
    g++ -c add.cpp
sub.o : sub.cpp arithmetic.h
    g++ -c sub.cpp
mul.o : mul.cpp arithmetic.h
    g++ -c mul.cpp
```

# Variables in Makefile

- You define a variable in a makefile by writing `VARIABLE_NAME=value`, then accessing the value of `VARIABLE_NAME` by writing either `$(VARIABLE_NAME)` or `${VARIABLE_NAME}`.
- A comment in a makefile starts with # and continues to the end of the line.
- Some examples:

```
CC = g++ # Which compiler
# Options for development
CPPFLAGS = -g -std=c++11
# Options for linking dynamic libraries
LDFLAGS = -llapacke -lblas
LIBS = main.o add.o sub.o mul.o
```

# Automatic Variables

- $@ The name of the current target
- $< The name of the first prerequisite
- $^ The names of all the prerequisites, with spaces between them
- Some examples:

```
main :  main.o add.o sub.o mul.o
    $(CC) -o $@ $(CPPFLAGS) $^
main.o :  main.cpp arithmetic.h
    $(CC) $(CPPFLAGS) -c $<
```

# Built-in Rules

- **make** has a large number of built-in rules that can significantly simplify makefiles.
- To check this, create a traditional Hello World program called hello.cpp and compile it using

  ```
  make hello
  ```

- You can print the built-in rules by issuing

  ```
  make -p
  ```

# Putting it all together

```
CC = g++
CPPFLAGS = -g -std=c++11
LIBS = main.o add.o sub.o mul.o
RM = rm -rf

main : $(LIBS)
	$(CC) -o $@ $(CPPFLAGS) $(LIBS)

.PHONY : clean

clean :
	$(RM) *.o main
```

# Makefile for Your Homework

```
default: homeworkTemplate.pdf

%.pdf : %.tex
        pdflatex $<
        pdflatex $<

clean :
        $(RM) *.aux *.log

realclean :
        $(MAKE) clean
        $(RM) *.pdf
```

# Some useful options to **make**

- `-k` Continue as much as possible after an error.
- `-n` Print what **make** would have done without actually doing it.
- `-s` Do not print the rules as they are executed.
- `-f filename` Read the file named `filename` as a makefile.

# Libraries for Linear Algebra

- Linear algebra is a fundamental building block of nearly every scientific software package.
- Reasons for using libraries instead of coding an algorithm from scratch:
  1. **programmer productivity**: focus on **what** instead of **how**;
  2. **code quality**: a library is usually specifically tuned for your computer architecture.

# Overview of BLAS and CBLAS

- The BLAS(Basic Linear Algebra Subprograms) library contains high quality building block routines for performing basic vector and matrix operations.
- BLAS routines are written in Fortran.
- CBLAS provides a C/C++ language interface to BLAS routines.
- Install the package `libblas-dev` from Synaptic Package Manager.

# BLAS

The BLAS libraries are decomposed into problems at three levels:

- BLAS Level 1 operations work on vectors and have a typical complexity of $n$ operations for a vector of length $n$. Typical examples are vector addition, computing an inner product, or applying Givens rotations.
- BLAS Level 2 operations are of the matrix-vector type, with $n^2$ complexity, such as the matrix-vector product.
- BLAS Level 3 operations are between two matrices, the typical example being the matrix-matrix product, which has $O(n^3)$ complexity.

# Naming Conventions for BLAS Routines

$$\texttt{XYYZZZ}: \begin{cases} \texttt{X}: & \text{precision} \\ \texttt{YY}: & \text{matrix type} \\ \texttt{ZZZ}: & \text{operation} \end{cases}$$

- The "precision" character `X` can be `S` for single, `D` for double, `C` for single-precision complex, and `Z` for double-precision complex.
- The two characters `YY` reflects the matrix argument type. For instance, `GE` stands for general rectangular, `GT` for general triangular, `SY` for symmetric, `TR` for triangular, etc.
- The (up to three) characters `ZZZ` describe the operation abbreviated with shorthands, including `MV` for matrix-vector product or `MM` for matrix-matrix product.
- `SGEMV`: single precision, general matrix, matrix-vector product
- `ZTRMM`: double-precision complex, triangular matrix, matrix-matrix product

# CBLAS

- In CBLAS, the Fortran routine names are prefixed with `cblas_`. Names of all CBLAS functions are in lowercase letters.
- `SGEMV` becomes `cblas_sgemv`, and `ZTRMM` becomes `cblas_ztrmm`.
- All CBLAS function declarations are in the file `cblas.h`.

# BLAS Level 1 Example

Compute the inner product of two vectors:

```
double cblas_ddot(const int n, const double *x, const int
    incx, const double *y, const int incy)
```

`incx` and `incy` specify the increment for the elements of `x` and `y`, respectively.

Create a file named `ddot.cpp` with the following contents:

```
#include <iostream>
#include <cblas.h>

int main(int argc, char *argv[]) {
    double x[3] = {1.1, 1.2, 1.3};
    double y[3] = {1.0, 2.0, 3.0};
    std::cout << "The inner product of x and y is ";
    std::cout << cblas_ddot(3, x, 1, y, 1) << std::endl;
}
```

Issue the following command:

```
g++ -o ddot ddot.cpp -lblas
./ddot
```

# BLAS Level 2 Example

Compute a matrix-vector product using a general matrix: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$

```
void cblas_dgemv(const CBLAS_ORDER order,
          const CBLAS_TRANSPOSE trans,
          const int m, const int n,
          const double alpha, const double *A, const int lda,
          const double *x, const int incx,
          const double beta, double *y, const int incy)
```

# CBLAS_ORDER

- There are two general methods of storing a two dimensional matrix in linear (one dimensional) memory: column-wise (column major order) or row-wise (row major order).
- Consider a general matrix $A = (a_{ij})_{1 \le i \le M, 1 \le j \le N}$. In column major order, the matrix elements are located in memory according to this sequence:

$$a_{1,1}, a_{2,1}, \ldots, a_{M,1}, a_{1,2}, a_{2,2}, \ldots, a_{M,2}, \ldots, a_{1,N}, a_{2,N}, \ldots, a_{M,N},$$

  In row major order, the matrix elements are located in memory according to this sequence:

$$a_{1,1}, a_{1,2}, \ldots, a_{1,N}, a_{2,1}, a_{2,2}, \ldots, a_{2,N}, \ldots, a_{M,1}, a_{M,2}, \ldots, a_{M,N}$$

- `order=CblasRowMajor`: row-major. `order=CblasColMajor`: column-major.

# CBLAS_TRANSPOSE

- `trans=CblasNoTrans`: $\mathbf{y} \leftarrow \alpha A\mathbf{x} + \beta\mathbf{y}$.
- `trans=CblasTrans`: $\mathbf{y} \leftarrow \alpha A^T\mathbf{x} + \beta\mathbf{y}$.
- `trans=CblasConjTrans`: $\mathbf{y} \leftarrow \alpha A^H\mathbf{x} + \beta\mathbf{y}$.
- `trans=CblasConjNoTrans`: $\mathbf{y} \leftarrow \alpha \bar{A}\mathbf{x} + \beta\mathbf{y}$.

# Leading Dimension Parameter

- A leading dimension parameter allows use of BLAS routines on a submatrix of a larger matrix.
- Consider a submatrix $B$ extracted from the original matrix $A$:

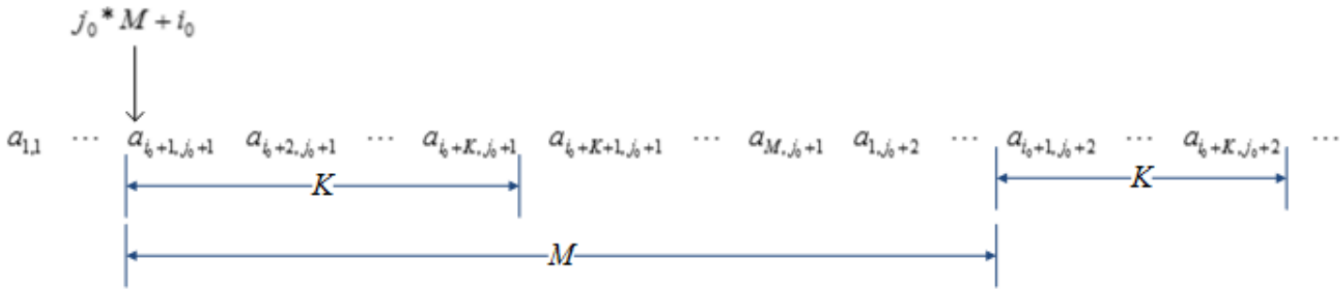
$$B = \begin{bmatrix} a_{i_0+1,j_0+1} & a_{i_0+1,j_0+2} & \cdots & a_{i_0+1,j_0+L} \\ a_{i_0+2,j_0+1} & a_{i_0+2,j_0+2} & \cdots & a_{i_0+2,j_0+L} \\ \vdots & \vdots & \ddots & \vdots \\ a_{i_0+K,j_0+1} & a_{i_0+K,j_0+2} & \cdots & a_{i_0+K,j_0+L} \end{bmatrix}$$

- To specify matrix $B$, BLAS routines require four parameters:
  1. the number of rows $K$ ;
  2. the number of columns $L$;
  3. a pointer to the start of the array containing elements of $B$;
  4. the leading dimension of the array containing elements of $B$.

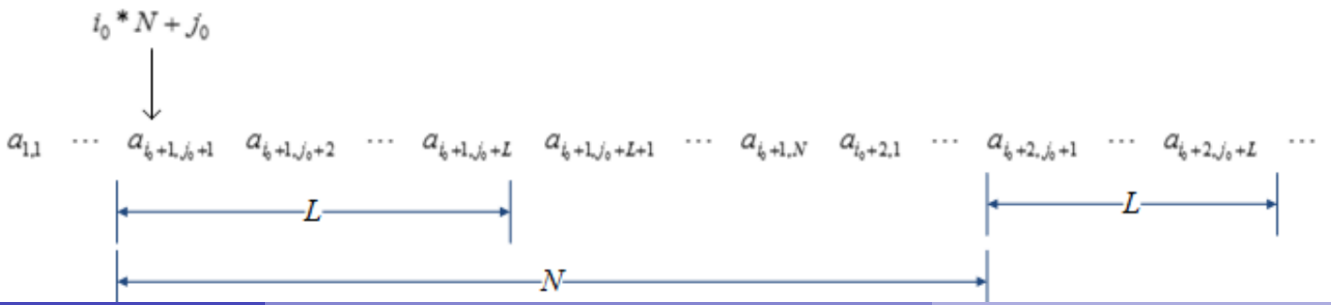
# Leading Dimension Parameter

The leading dimension depends on the storage order of the matrix $A$:

- Column major order
  Leading dimension is $M$, i.e., the number of rows of matrix $A$.
  Starting address: offset by $i_0 + j_0 M$ from $a_{1,1}$.




- Row major order
  Leading dimension is $N$, i.e., the number of columns of the matrix $A$.
  Starting address: offset by $i_0 N + j_0$ from $a_{1,1}$.



```
#include <iostream>
#include <algorithm>
#include <iterator>
#include <cblas.h>
using namespace std;

int main(int argc, char *argv[]) {
    double mat[4*3] = {1, 2, 0, 1,
                       2, 0, 1, 1,
                       0, 3, 2, 2};
    double x[3] = {1, 2, 3};
    double y[3] = {0, 0, 0};
    cblas_dgemv(CblasColMajor, CblasNoTrans,
                    3, 3, 1.0, mat, 4, x, 1, 0.0, y, 1);
    copy(begin(y), end(y),
           ostream_iterator<double>(cout, " "));
    cout << endl;
}
```

# BLAS Level 3 Example

Compute a matrix-matrix product with general matrices:

```
void cblas_dgemm(const CBLAS_ORDER order, const
CBLAS_TRANSPOSE transa, const CBLAS_TRANSPOSE transb,
const int m, const int n, const int k,
const double alpha, const double *a, const int lda,
const double *b, const int ldb, const double beta,
double *c, const int ldc)
```

Try it yourself!
For more information, refer to `http://netlib.org/blas/`.

# Overview of LAPACK

- LAPACK(Linear Algebra PACKage) is written in Fortran 90 and provides routines for solving systems of linear equations, linear least-square problems, eigenvalue and singular value problems, and performing a number of related computational tasks.
- LAPACK routines are written so that as much as possible of the computation is performed by calls to the BLAS.
- Naming convention of LAPACK follows from that of BLAS.

# Overview of LAPACKE

- LAPACKE provides a C/C++ language interface to LAPACK routines.
- In LAPACKE, the Fortran routine names are prefixed with `LAPACKE_`.
- All LAPACKE function declarations are in the file `lapacke.h`.
- Install the package `liblapacke-dev` from Synaptic Package Manager.

# LAPACKE_dgesv

```
lapack_int LAPACKE_dgesv(int matrix_layout,
        lapack_int n, lapack_int nrhs,
        double* A, lapack_int lda,
        lapack_int* ipiv,
        double* B, lapack_int ldb)
```

This routines solves for $X$ the system of linear equations $AX = B$, where $A$ is a square matrix, the columns of $B$ are individual right-hand sides, and the columns of $X$ are the corresponding solutions.
The LU decomposition with partial pivoting and row interchanges is used to factor $A$ as $A = PLU$, where $P$ is a permutation matrix, $L$ is unit lower triangular, and $U$ is upper triangular. The factored form of $A$ is then used to solve the system of equations $AX = B$.

# LAPACKE_dgesv

- `matrix_layout` specifies whether matrix storage is row major(`LAPACK_ROW_MAJOR`) or column major(`LAPACK_COL_MAJOR`).
- Side effects:
  1. $A$ is overwritten by the factors $L$ and $U$ from the factorization $A = PLU$; the unit diagonal elements of $L$ are not stored.
  2. $B$ is overwritten by the solution matrix $X$.
  3. `ipiv` is the pivot vector that defines the permutation matrix.
- `LAPACKE_dgesv` returns a value `info`.
  1. If `info`=0, the executation is successful.
  2. If `info`=-$i$, parameter $i$ had an illegal value.
  3. If `info`=$i$, $U_{i,i}$ is exactly zero. The factorization has been completed, but the factor $U$ is exactly singular, so the solution could not be computed.

```
#include <iostream>
#include <lapacke.h>
#include <algorithm>
#include <iterator>
using namespace std;

int main(int argc, char *argv[]) {
    double A[9] = {2, -2, -1,
                   -2, 4, 8,
                    6, 3, 4};
    double b[3] = {16, 0, -1};
    int ipiv[3];
    LAPACKE_dgesv(LAPACK_COL_MAJOR, 3, 1, A, 3, ipiv, b, 3);
    cout << "The solution is : \n";
    copy(begin(b), end(b),
             ostream_iterator<double>(cout, " "));
    cout << endl;
}
```

# LAPACKE_dsterf

Compute all eigenvalues of a real symmetric tridiagonal matrix.

```
lapack_int LAPACKE_dsterf(lapack_int n, double *d, double *e)
```

Input Parameters:

- *n*: The order of the matrix
- **d**: **d** contains the diagonal elements
- **e**: **e** contains the off-diagonal elements

Side Effects:

- **d**: The *n* eigenvalues in ascending order.

```
#include <iostream>
#include <lapacke.h>
#include <algorithm>
#include <iterator>
using namespace std;

int main(int argc, char *argv[]) {
    double d[4] = {2, 2, 2, 2};
    double e[3] = {-1, -1, -1};
    auto info = LAPACKE_dsterf(4, d, e);
    if(info==0) {
        copy(begin(d), end(d),
                ostream_iterator<double>(cout, " "));
    }
    else {
        cout << "LAPACKE_dsterf failed\n";
    }
}
```

# References


- GNU Make Manual
- Beginning Linux Programming
- Introduction to Scientific and Technical Computing
- `http://netlib.org/blas/`
- `http://www.netlib.org/lapack/`