Linear Algebra Libraries

Ramses van Zon

PHY1610, Winter 2026

How to deal with linear algebra in code

As much as possible, rely on existing, mature software libraries to perform linear algebra computations.

By doing so you...

- Focus on your code details.
- Reduce the amount of code to write and debug
- Libraries are tuned and optimized, i.e., your code will run faster
- More options to switch methods if necessary.

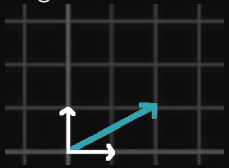
BLAS

Basic Linear Algebra Subroutines

- A well-defined standard interface for linear algebra routines.
- Many highly-tuned implementations exist for various platforms.
  (MKL, BLIS, Atlas, OpenBLAS, PLASMA, cuBLAS, ...)
- Interface vs. Implementation!
  Trick is designing a sufficiently general interface.
- Higher-order operations (e.g. factorizations, solving) defined in LAPACK, on top of BLAS.

Linear algebra recap: vectors

- Basic building block is the **vector**.

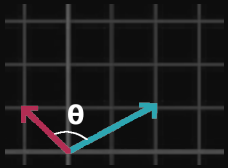


N numbers for a N-dimensional vector. Each number says how far along each axes $\hat{e}_i$.

- Magnitude of a vector by the square root of the sum of the squares of the components:

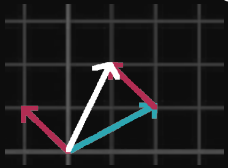
$$\|\vec{a}\| = \sqrt{\sum_i a_i^2}$$

- Inner product of two vectors:



$$\vec{a} \cdot \vec{b} = \sum_i a_i b_i = \|a\| \, \|b\| \, \cos\theta$$
$$a_i = \hat{e}_i \cdot \vec{a}$$

- Vectors addition means adding components.



- Scaling vectors means one scalar multiplying all components: $(\lambda\vec{a})_i = \lambda a_i$

Linear algebra recap: matrices

- We can look at the vector space in different coordinates.
- Individual new components of a vector in new coodinates can be found by inner product with unit vectors corresponding to the new axes:

$$(\vec{a}')_i = \hat{e}'_i \cdot \vec{a}$$

$$a'_i = \hat{e}'_i \cdot \vec{a}$$

$$a'_i = \hat{e}'_i \cdot \sum_j a_j \hat{e}_j$$

$$a'_i = \sum_j (\hat{e}'_i \cdot \hat{e}_j) a_j$$

- So the transformation from one to the other is given by an N x N matrix.

$$Q_{ij} = \hat{e}'_i \cdot \hat{e}_j$$

- And we need matrix vector multiplication to apply this matrix:

$$a'_i = \sum_j Q_{ij} a_j \quad \textit{i.e.} \quad \vec{a}' = \mathbf{Q} \cdot \vec{a}$$

- This is a special matrix called an orthogonal matrix, which preserves angles and sizes.
- General matrices can also scale and skew.

Finally, applying several matrices results in applying a matrix product:

$$(\mathbf{A} \cdot \mathbf{B})_{ij} = \sum_k A_{ik} B_{kj}$$

Typical BLAS routines

- Level 1: vector operations
  - `sdot` (dot product, single)
  - `zaxpy` ($ax + y$, dbl complex)
- Level 2: matrix-vector operations
  - `dgemv` (dbl matrix*vec)
  - `dsymv` (dbl symmetric matrix*vec)
- Level 3: matrix-matrix operations
  - `sgemm` (general matrix-matrix)
  - `ctrmm` (triangular matrix-matrix)

Somewhat cryptic names, interfaces.

Prefixes

S: Single
D: Double
C: Complex
Z: Double Complex Matrix

Types

GE: General
HY: Hermetian
SY: Symmetric
TR: Triangular
GB: General Banded
HB: Hermetian Banded
SB: Symmetric Banded
TB: Triangular Banded
TP: Triangular Packed

Why using Linear Algebra Packages?

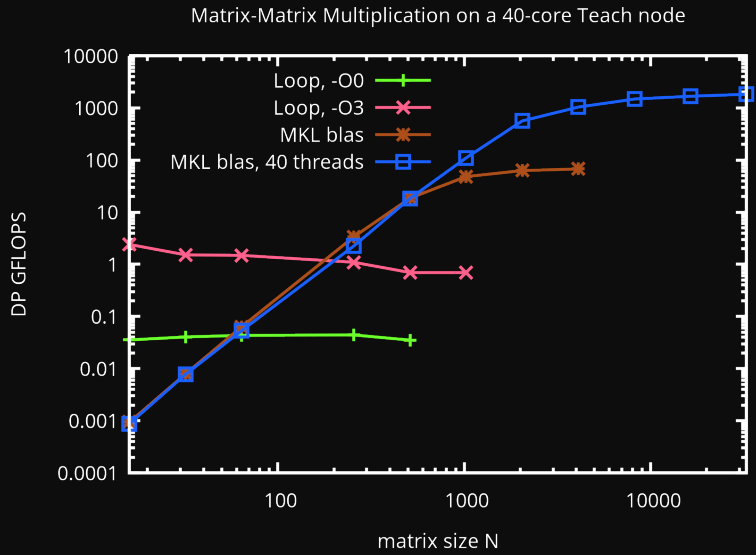
- Why bother?
- Finding, downloading, installing the library
- Figuring out how to link
- C/Fortran issues
- Why not just write it?
  (It's not rocket science)

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$$

$$C_{ij} = \sum_k A_{ik} B_{kj}$$

```
for (i=0; i<N; i++) {
    for (j=0; j<N; j++) {
        c[i,j] = 0.0;
        for (k=0; k<N; k++) {
            c[i,j] += a[i,k]*b[k,j];
        }
    }
}
```

Never, ever, write your own...



Using BLAS


SciNet
ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

Using BLAS

- Netlib provides *reference* implementation
- Most vendors provide optimized versions
- Vendor-specific: Intel (MKL), AMD (AMD-BLIS), IBM (ESSL)
- Open Source: ATLAS, BLIS, OpenBLAS
- Fortran functions
- C interface using CBLAS and LAPACKE

Install OpenBLAS

```
$ module load gcc/14.3
$ git clone https://github.com/xianyi/OpenBLAS.git openblasbuild
$ cd openblasbuild
$ git checkout v0.3.31
$ make USE_LOCKING=1 USE_OPENMP=0 USE_THREAD=1 CC=gcc
$ make PREFIX=~/myopenblas install
```

Create a script to set variables:

```
echo 'LD_LIBRARY_PATH=$LD_LIBRARY_PATH:~/myopenblas/lib' >> ~/myopenblas/setup
echo 'LIBRARY_PATH=$LIBRARY_PATH:~/myopenblas/lib'       >> ~/myopenblas/setup
echo 'CPATH=$CPATH:~/myopenblas/lib'                     >> ~/myopenblas/setup
echo 'MANPATH=$MANPATH:~/myopenblas/man'                 >> ~/myopenblas/setup
```

Before using OpenBLAS, you must do

```
source ~/myopenblas/setup
```

Install manpages (optional, but useful)

```
$ wget http://www.netlib.org/lapack/lapack-3.1.1.tgz
$ tar -C ~/myopenblas --strip-components=3 -xaf lapack-3.1.1.tgz lapack-3.1.1/manpages/blas/man/manl
$ tar -C ~/myopenblas --strip-components=2 -xaf lapack-3.1.1.tgz lapack-3.1.1/manpages/man/manl
```

BLAS Examples

BLAS Example: DSCAL (x $\leftarrow \alpha$ x)

```cpp
// dscalex.cpp
#include <print>
#include <rarray>
#include <cblas.h>

int main() {
  rvector x = make_rarray({ 1.0, 2.0, 3.0 });

  cblas_dscal(x.size(), 4.323, &x[0], 1);

  std::println("{} ",x);
}
```

```
$ module load gcc/14.3 rarray/2.8.2
$ source ~/myopenblas/setup
$ g++ -c -std=c++23 dscalex.cpp -o dscalex.o
$ g++ dscalex.o -o dscalex -lopenblas
$./dscalex
{4.323,8.646,12.969}
```

BLAS Example: DSCAL (x $\leftarrow \alpha$ x)

Documentation

- https://www.netlib.org/blas/blast-forum
- `man dscal`

```
NAME
       DSCAL - a vector by a constant

SYNOPSIS
       SUBROUTINE DSCAL(N,DA,DX,INCX)

           DOUBLE PRECISION          DA
           INTEGER                   INCX,N
           DOUBLE PRECISION          DX(*)

PURPOSE
         scales a vector by a constant.
         uses unrolled loops for increment equal to one.
```

Function name + order and names of arguments

Says that DA is a double precision scalar

Says that INCX and N are integers.

Says that DX is a double precision array

Yep, that's the Fortran version, but the C/C++ versions are (nearly) the same.

BLAS Example: DSCAL (x $\leftarrow \alpha$ x)

```cpp
#include <print>
#include <rarray>
#include <cblas.h>

int main() {
  rvector x = make_rarray({ 1.0, 2.0, 3.0 });

  // DSCAL(N,DA,DX,INCX):

  // number of elements (x.extent(0) works too)
  int N = x.size();
  // scalar to multiply with
  double DA = 4.323;
  // point to first element (x.data() works too)
  double* DX = &x[0];
  // all elements are contiguous
  int    INCX = 1;
  cblas_dscal(N, DA, DX, INCX);

  std::println("{} ",x);
}
```

```
$ module load gcc/14.3 rarray/2.8.2
$ source ~/myopenblas/setup
$ g++ -c -std=c++23 dscalex2.cpp -o dscalex2.o
$ g++ dscalex2.o -o dscalex2 -lopenblas
$./dscalex2
{4.323,8.646,12.969}
```

Let's try something a bit more involved...

Matrix Multiply Documentation

- man dgemm

```
NAME
       DGEMM - performs one of the matrix-matrix operations   C := alpha*op( A )*op( B ) + beta*C,
SYNOPSIS
       SUBROUTINE DGEMM(TRANSA,TRANSB,M,N,K,ALPHA,A,LDA,B,LDB,BETA,C,LDC)

           DOUBLE PRECISION     ALPHA,BETA
           INTEGER              K,LDA,LDB,LDC,M,N
           CHARACTER            TRANSA,TRANSB
           DOUBLE PRECISION     A(LDA,*),B(LDB,*),C(LDC,*)

PURPOSE
       DGEMM  performs one of the matrix-matrix operations

       where  op( X ) is one of
          op( X ) = X   or   op( X ) = X',

       alpha and beta are scalars, and A, B and C are matrices, with op( A ) an m by k matrix,
       op( B ) a  k by n matrix and C an m by n matrix.
```

LDA? Leading Dimension?

Miscellaneous Details

- CBLAS calls involving matrices have an additional first argument `CblasRowMajor`, `CblasColMajor`
- LDA - *Leading dimension of* `A` used to access subblocks of the matrix.
- The *leading dimension* is the number of elements to skip to get to the next row (when row major) or column (when column major).
- For full matrices, this is the number of columns (for row major)
  or the number of rows (for column major).
- This allows you to use submatrices without making a copy.

BLAS Example: DGEMM (C $\leftarrow \alpha$ A · B + β C)

```
$ g++ -std=c++23 dgemmex.cpp -o dgemmex -lopenblas
```

```
#include <cblas.h>
#include <print>
#include <rarray>
void printmatrix(const rmatrix<double>& a) {
   std::println("{} by {}\n{}", a.extent(0), a.extent(1), a);
}
int main() {
   int m = 3, k = 5, n =4;
   double alpha = 1.0, beta = 0.0;
   rmatrix<double> A(m,k), B(k,n);
   rmatrix<double> C(m,n);
   for (int i=0; double& a: A) a = ++i;
   for (int i=0; double& b: B) b = --i;
   C.fill(0.0);
   cblas_dgemm(CblasRowMajor, CblasNoTrans, CblasNoTrans,
               m, n, k, alpha, &A[0][0], k, &B[0][0], n,
               beta, &C[0][0], n);
   std::print("A:"); printmatrix(A);
   std::print("B:"); printmatrix(B);
   std::print("C:"); printmatrix(C);
}
```

```
$ ./dgemmex
```

```
A:3 by 5
{
{1,2,3,4,5},
{6,7,8,9,10},
{11,12,13,14,15}
}
B:5 by 4
{
{-1,-2,-3,-4},
{-5,-6,-7,-8},
{-9,-10,-11,-12},
{-13,-14,-15,-16},
{-17,-18,-19,-20}
}
C:3 by 4
{
{-175,-190,-205,-220},
{-400,-440,-480,-520},
{-625,-690,-755,-820}
}
```

LAPACK

Linear Algebra PACKage (LAPACK)

LAPACK contains a variety of subroutines for solving linear systems, matrix decompositions, and factorizations.

- Internally uses BLAS calls
- Supports the same data types (single/double precision, real/complex and matrix structure types (symmetric, banded, etc.) as BLAS
- Three categories: auxiliary routines, computational routines, and driver routines
- C interface with prefix `LAPACKE_`
  https://www.netlib.org/lapack/lapacke.html

Linear Algebra PACKage (LAPACK)

Computational routines are designed to perform single, specific computational tasks:

- factorizations:
  - LU , LL^T /LL^H, LDL^T /LDL^H ,
  - QR, LQ, QRZ generalized QR and RQ
- symmetric/Hermitian and nonsymmetric eigenvalue decompositions
- singular value decompositions
- generalized eigenvalue and singular value decompositions

LAPACK Example: DGESV (Solve A x = b)

```
NAME
       DGESV - computes the solution to a real system of linear equations  A * X = B,

SYNOPSIS
       SUBROUTINE DGESV( N, NRHS, A, LDA, IPIV, B, LDB, INFO )

           INTEGER          INFO, LDA, LDB, N, NRHS
           INTEGER          IPIV( * )
           DOUBLE PRECISION A( LDA, * ), B( LDB, * )
PURPOSE
       DGESV computes the solution to a real system of linear equations
          A  *  X = B, where A is an N-by-N matrix and X and B are N-by-NRHS matrices.
       The LU decomposition with partial pivoting and row interchanges is used to factor A as
          A = P * L * U,
       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 A * X = B.

ARGUMENTS
      N       (input) INTEGER
              The number of linear equations, i.e., the order of the matrix A.  N >= 0.

      NRHS    (input) INTEGER
              The number of right hand sides, i.e., the number of columns of the matrix B.
```

LAPACK Example: DGESV (Solve A x = b)

```cpp
// dgesvex.cpp
#include <print>
#include <lapacke.h>
#include <rarray>
int main() {
   const int N=3, NRHS=2, LDA=N, LDB=NRHS;
   rvector<int> ipiv(N);
   int info;
   rmatrix<double> A(N, N);
   A.fill({{6.80, -6.05, -0.45},
           {-2.11, -3.30,  2.58},
           {5.66,  5.36, -2.70}});
   rmatrix<double> b(N,NRHS);
   b.fill({{4.02, -1.56},
           {6.19,  4.00},
           {-8.22, -8.67}});
   info = LAPACKE_dgesv(LAPACK_ROW_MAJOR, N, NRHS,
                        &A[0,0], A.extent(1), &ipiv[0], &b[0,0], b.extent(1));
   std::println("Solutions x: {}", b);
   std::println("Details of LU factorization: {}", A);
   std::println("Pivot indices: {}", ipiv);
}
```

LAPACK Example: DGESV (Solve A x = b)

```
$ g++ -std=c++23 dgesvex.cpp -o dgesvex -lopenblas
$ ./dgesvex
```

```
Solutions x: {
{-0.0517981,-0.892398},
{-0.819976,-0.736171},
{1.30806,-0.121056}
}
Details of LU factorization: {
{6.8,-6.05,-0.45},
{0.832353,10.3957,-2.32544},
{-0.310294,-0.49802,1.28225}
}
Pivot indices: {1,3,3}
```

What about non-dense matrices?

E.g.:

$$\begin{pmatrix}
-2 & 1 & & & & & \\
1 & -2 & 4 & & & & \\
 & 4 & -2 & 4 & & & \\
 & & 4 & -2 & 4 & & \\
 & & & 4 & -2 & 4 & \\
 & & & & & \ddots & \\
 & & & & 4 & -2 & 1 \\
 & & & & & 1 & -2
\end{pmatrix}$$

For different types, use different functions

- Banded: `DGBSV`
- Tri-diagonal: `DGTSV`
- Symmetric positive definite: `DPOSV`

LAPACK Example: DGTSV (Solve Ax=b)

```cpp
// dgtsvex.cpp
#include <print>
#include <lapacke.h>
#include <rarray>
int main() {
    const int N=5, NRHS=3;
    rvector<double> dl(N-1); dl.fill({1, 4, 4, 1});
    rvector<double> dc(N);   dc.fill({-2, -2 ,-2 ,-2, -2});
    rvector<double> du(N-1); du.fill({1, 4, 4, 1});
    rmatrix<double> b(N, NRHS);
    b.fill({{3, -1.56,  9.81},
            {5,  4.00, -4.09},
            {5, -8.67, -4.57},
            {5,  1.75, -8.61},
            {3,  2.86,  8.99}});
    int info = LAPACKE_dgtsv(LAPACK_ROW_MAJOR, N, NRHS,
                             &dl[0], &dc[0], &du[0], &b[0,0], b.extent(1));
    std::print("Solutions x: {}", b);
    return info;
}
```

LAPACK Example: DGTSV (Solve Ax=b)

```
$ g++ -std=c++23 dgtsvex.cpp -o dgtsvex -lopenblas
$ ./dgtsvex
```

```
Solutions x:
{
{-0.931034,0.285747,-6.09874},
{1.13793,-0.988506,-2.38747},
{2.05172,0.43431,-0.691552},
{1.13793,-0.961839,0.899195},
{-0.931034,-1.91092,-4.0454}
}
```

Sparse BLAS ?

Unfortunately there is not just one mature, standard sparse matrix BLAS library.

Some potential options:

- "Official'' Sparse BLAS: a reference implementation is not yet available
  https://www.netlib.org/blas/blast-forum
- NIST Sparse BLAS: An alternative BLAS system; a reference implementation is available
  https://math.nist.gov/spblas
- MKL sparse linear algebra routines:

  https://www.intel.com/content/www/us/en/docs/onemkl/developer-reference-c/2025-2/inspector-executor-sparse-blas-routines.html
- AMD sparse linear algebra routines:

  https://www.amd.com/en/developer/aocl/sparse.html
- Various linear algebra packages may offer support for sparse matrices.

LAPACK References

- https://www.netlib.org/lapack
- LAPACK Internet Interface and Search Engine
  https://www.cs.colorado.edu/~jessup/lapack
- https://web.cs.ucdavis.edu/~bai/publications/baidemmeletal06.pdf