Project: Simple numerical operations with vectors and matrices

Tomasz Chwiej

25th February 2025

1 Introduction

1.1 Scalar product of two vectors

One of the most involved numerical operation is computing the scalar product of two vectors, mathematically is defined as follows

$$s = \vec{x} \cdot \vec{y} = \vec{x}^T \vec{y} = \begin{bmatrix} x_0 & x_1 & \dots & x_{n-1} \end{bmatrix} \begin{bmatrix} y_0 \\ y_1 \\ \vdots \\ y_{n-1} \end{bmatrix} = \sum_{i=0}^{n-1} x_i y_i \tag{1}$$

which can be e.g. used in calculations of the vector Euclidean norm (length)

$$norm(\vec{x}) = \|\vec{x}\|_2 = \sqrt{\vec{x}^T \vec{x}} \tag{2}$$

Computer implementation of scalar product is straightforward, assuming the vectors $\vec{x}$ and $\vec{y}$ are stored in 1D arrays $x[]$ and $y[]$ this would be like following

```
s ← 0
do i=0 to n-1
s ← s + x_i · y_i
end do
```

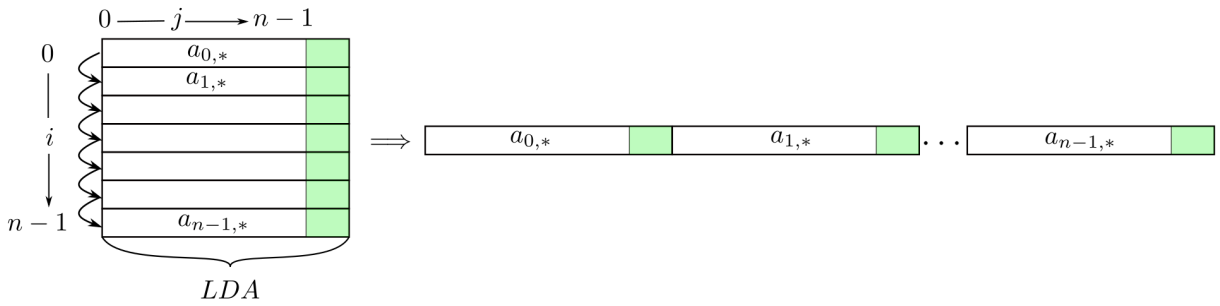
Operations on vectors might be conducted with BLAS1 procedures (part of BLAS package).

1.2 Matrix storage in 1D array

At present the most popular numerical package used for linear algebra operations is LAPACK supported by BLAS library. BLAS conducts the most basic operations on vectors and matrices while the LAPACK performs more advanced tasks (matrix decompositions, solves systems of linear equations, diagonalizes the matrices, etc.) Both are written in Fortran but C/C++ wrappers were created which utilizes the row-major and column-major storage of two-dimensional arrays. To use these packages efficently one needs to understand the peculiar way the matrix elements are stored.

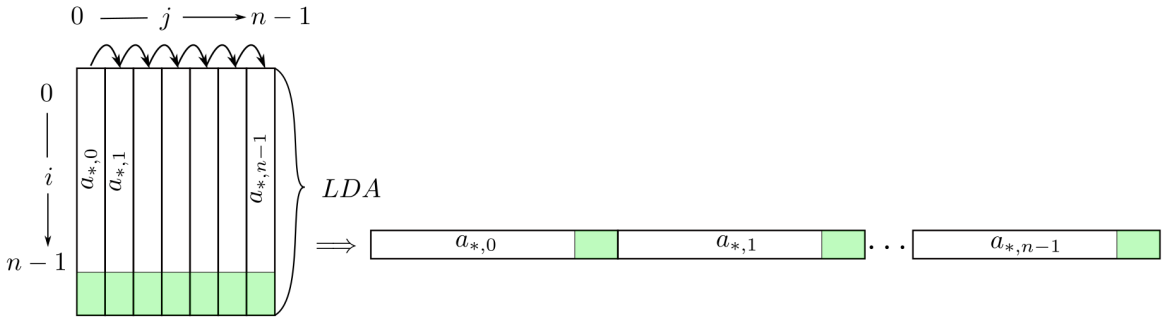
In C/C++ the matrix elements are stored in row-major order as shown in figure 1(a) while in Fortran these are natively stored in columns [see Fig.1(b)] what in some numerical applications is more natural. Sometimes it can happen that the matrix we wish to operate with is a part of larger structure (white color in figure, green color denote empty sites), then we must inform the numerical procedure how the elements are stored, how many elements are stored in each row/column and how large phisically are the rows/columns (white and green color elements in figure). The last quantity is called **leading**

(a) row-major-order



$$a_{i,j} \quad \Longrightarrow \quad a[i \cdot LDA + j]$$

(b) column-major-order



$$a_{i,j} \quad \Longrightarrow \quad a[i + j \cdot LDA]$$

Figure 1: Scheme of matrix elements storage in (a) row-wise and (b) column-wise order in 1D array (right side). Note that the rows and columns indices start from 0, number of elements in row/column is **n** while the 2D data structure may have larger size **LDA** (white+green elements) along the chosen direction of filling the array.

dimension of matrix A (LDA), obviously it might happen that we declare the storage matrix with parameter $LDA = n$ then there are no empty (green) sites in rows or in columns.

1.3 Matrix-vector product

Besides the scalar product also matrix-vector product is often met especially in the iterative algorithms, mathematically it is written as follows

$$\vec{y} = A\vec{x} \qquad \Longrightarrow \qquad y_i = \sum_{j=0}^{n-1} A_{i,j}x_j, \quad (i = 0, 1, 2, \ldots, n-1) \tag{3}$$

translating it into numerical code with assumption of matrix storage as 1D array **a**[] gives

- for row-major order

```
do i=0 to n-1
  y_i ← 0
 do j=0 to n-1
  y_i ← a_{i·LDA+j} · x_j
 end do
end do
```

- for column-major order

```
do i=0 to n-1
  y_i ← 0
 do j=0 to n-1
  y_i ← a_{i+j·LDA} · x_j
 end do
end do
```

Both examples convince that we must carefully check which index (row or column) changes the fastest. Mathematical operations involving matrix and vector as operands can be conducted with BLAS2 procedures (part of BLAS package).

1.4 Matrix-matrix product

As the last example we consider the product of two matrices, to ease the calculations we assume all three matrices are stored in the same order as 1D arrays. First let's write the definition of the product of two square matrices

$$C = A \cdot B \qquad \Longrightarrow \qquad C_{i,k} = \sum_{j=0}^{n-1} A_{i,j} \cdot B_{j,k}, \quad (i, k = 0, 1, \ldots, n-1) \tag{4}$$

and its computer implementation for A, B and C stored in 1D arrays in

- row-major order

```
do i=0 to n-1
 do k=0 to n-1
  s ← 0
  do j=0 to n-1
   s ← s + a_{i·LDA+j} · b_{j·LDB+k}
  end do
```

```
    c_{i·LDC+k} ← s
  end do
end do
```

- column-major order

```
do i=0 to n-1
 do k=0 to n-1
  s ← 0
  do j=0 to n-1
   s ← s + a_{i+j·LDA} · b_{j+k·LDB}
  end do
   c_{i+k·LDC} ← s
 end do
end do
```

2 Practical part

Tasks to do

1. Write the computer program which

   - performs $m = 30$ times the matrix-vector multiplication, namely

   $$\vec{x}^{(m)} = A^m \vec{x} = \underbrace{A \cdot A \cdot \ldots \cdot A}_{\text{m times}} \vec{x} \tag{5}$$

   In calculations assume the following parameters: number of rows/columns $LDA = n = 2000$, starting vector is filled with values $x_i = 1, \quad (i = 0, 1, \ldots, n-1)$ while the matrix elements are defined below

   $$A_{i,j} = \frac{1}{|i-j|^4 + \gamma}, \quad (\gamma = 2, \quad i, j = 0, 1, 2, \ldots, n-1) \tag{6}$$

   Store the matrix A as 1D array using row-major or column-major storage type (chose one). After each multiplication calculate the norm of resultant vector $\vec{x}^{(i)}$ and write the norm and the number i to the file (save data on disk). Repeat the calculations for $\gamma = 3$. For $\gamma = 2$ and $\gamma = 3$ you shall observe increasing and decreasing value of vector norm, respectively.

   - calculates matrix-matrix product: $C = A \cdot A$ using code given in one of the listings presented in Sec.1.4
   - calculates the matrix-matrix product $C = A \cdot A$ using the BLAS3 procedure **cblas_dgemm(...)** - see the hint in Sec.3

2. At home prepare a short report. Make separate figures showing the changes in vector norm $\vec{x}^{(i)}$ for consecutive matrix-vector multiplications for both $\gamma's$ values. Try to assess the time of matrix-matrix product calculated with your own procedure and the procedure from BLAS3. To accelerate the speed of your numerical code you may use optimization flags: **-O1**, **-O2**, **-O3**, **-Ofast** during compilation in Linux system.

3 Computational hints

In project we wish to use one of the BLAS3 routines to perform the matrix-matrix multiplication. For this purpose in C/C++ we need to include the proper headers of the **cblas** wrapper and of the math functions

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

and remember to link appropriate numerical library (**blas** and **math**) during the compilation for C++

```
g++ *.cpp -lblas -lm
```

or for C

```
gcc *.c -lblas -lm
```

If the compiler can not find the library then locate the header file and compiled library with the command **locate cblas** on Taurus server (linux) and provide the paths to both using the flags: **-I/directory** for header and **-L/directory** for library.

The use of CBLAS library packages is well explained in Math Kernel Library documentation (link to the pdf version is at the bottom of the web page of this module). Here we use the **cblas_dgemm()** for matrix-matrix product

$$C = \alpha \cdot A \cdot B + \beta \cdot C, \quad (\alpha, \beta \in \mathcal{R}) \tag{7}$$

```
void cblas_dgemm (const CBLAS_LAYOUT Layout,
                  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);
```

where: **Layout=CblasRowMajor or CblasColMajor** denotes the storage order the same for matrices A, B and C, **transa,transb = CblasNoTrans** means the matrix operands are not transposed, **m** is the number of rows in matrix A, **n** is the number of columns in matrix B, **k** is the number of columns in A and the number of rows in B (must be the same), **lda**, **ldb** and **ldc** are integers denoting the leading dimensions of A, B and C, respectively. The values of constants are $\alpha = 1$ and $\beta = 0$. The **a**, **b** and **c** are the 1D arrays which store the elements of matrices A, B and C. In **C/C++** these arrays can be allocated using **malloc** function, e.g.

```
int n=value
double *a=(double*) malloc(n*n*sizeof(double))
```

or in **C++** as vector

```
#include<vector>

int n=value
vector<double> a(n*n, 0.0);
```

but then we must pass the pointer to the first element in array: **&a[0]**, **&b[0]** and **&c[0]**.