MATRICES AS A TENSOR PRODUCT AND THE *MAT* C++ CLASS TEMPLATE

ERNEST YEUNG ERNESTYALUMNI@GMAIL.COM

Contents

Abstract. 1. Executive Summary

I implement matrices, with a focus on implementing matrix multiplication and the taking the transpose for the matrices, in C++11. In designing the class templates for the matrices, I begin with the mathematical formulation of the space of all matrices of matrix size dimensions $M \times N$, $\mathbf{Mat}_{\mathbb{K}}(M,N)$, with $\mathbb{K}$ being the underlying field (e.g. $\mathbb{K} = \mathbb{R}, \mathbb{Z}$), as a *tensor product* of vector spaces, $\bigotimes_{j=1}^P \mathbb{K}^M$ and equivalently as a tensor product of *dual* vector spaces $\bigotimes_{i=1}^M \mathbb{K}^P$. By doing so, we can implement matrix multiplication and the transpose using new C++11 features in the library header 'algorithm', and efficiently read contiguous memory addresses along each of the `std::vectors`, representing rows or columns of a matrix.

2. Mathematical formulation: matrices as tensor products of vectors (or *dual* vectors)

Denote the set of all matrices of *matrix size dimensions* (i.e. number of rows $\times$ number of columns) (M,P) or i.e. $M \times N$, with the underlying field $\mathbb{K}$, to be $\text{Mat}_{\mathbb{K}}(M,P)$.

Consider two matrices $A \in \text{Mat}_{\mathbb{K}}(M,P)$, $B \in \text{Mat}_{\mathbb{K}}(P,N)$. We can multiply them together since they share equal "inner size dimensions."

Let's consider the entries of matrix A (even if only to set notation), $A_{ij}, \quad \forall\, i = 1,2,\dots M$, $j = 1,2,\dots P$. Consider the "rows" of the matrix, and the "columns" of the matrix A. For the ith row of A, denote it as A_{i*}, so that

$$A_{i*} = (A_{i1}, A_{i2}, \dots A_{iP}) \qquad \forall\, i = 1,2,\dots M \tag{1}$$

Date: 27 juillet 2017.
Key words and phrases. C++, C++11, Matrices, Tensor Products, dual basis, basis, numerical computation.

For the jth column of A, denote it as A_{*j}, so that

$$A_{*j} = (A_{1j}, A_{2j}, \dots A_{jM}) \qquad \forall\, j = 1, 2, \dots P \tag{2}$$

Let's treat the "row vector", A_{i*} as being an element of the dual vector space to $\mathbb{K}^P$, $(\mathbb{K}^P)^*$. Notice that matrix A has M of these "rows." Then $\forall\, A \in \text{Mat}_{\mathbb{K}}(M,P)$, $A \in \bigotimes_{i=1}^M (\mathbb{K}^P)^*$. In fact, we already described the isomorphism between these two spaces. So

$$\bigotimes_{i=1}^M (\mathbb{K}^P)^* \cong \text{Mat}_{\mathbb{K}}(M,P) \tag{3}$$

Let's treat the "column vector", A_{*j} as being an element of the vector space $\mathbb{K}^M$. Notice that matrix A has P of these "columns." Then $\forall\, A \in \text{Mat}_{\mathbb{K}}(M,P)$, $A \in \bigotimes_{j=1}^P (\mathbb{K}^M)^*$. In fact, we already described the isomorphism between these two spaces. So

$$\bigotimes_{j=1}^P (\mathbb{K}^M) \cong \text{Mat}_{\mathbb{K}}(M,P) \tag{4}$$

2.1. **Matrix Multiplication.** Again, given matrices $A \in \text{Mat}_{\mathbb{K}}(M,P)$, $B \in \text{Mat}_{\mathbb{K}}(P,N)$, we multiply them together to obtain $C \in \text{Mat}_{\mathbb{K}}(M,N)$:

$$C = AB$$

$$C_{ij} = \sum_{k=1}^P A_{ik} B_{kj} \equiv A_{ik} B_{kj} \quad \forall\, i = 1, 2, \dots M, \forall\, j = 1, 2, \dots N \tag{5}$$

where at the end, we use Einstein's summation notation of implicit summation of any repeated indices.

Keeping in mind that "row vector" and "column vector" (dual vector and vector) forms from Eq. 1, Eq. 2, respectively, let's try to rewrite the matrix multiplication $C = AB$:

$$C_{ij} = A_{ik} B_{kj} = A_{ik} (B^T)_{jk} = A_{i*} \cdot (B^T)_{j*} \qquad \forall\, i = 1, \dots M, \forall\, j = 1, \dots N \tag{6}$$

Thus, if we can get all the rows of A, and get all the columns of B, we can simply take the inner product of each row with each column (all possible row, column pairs), and we'll have obtained C!

2.1.1. *The case for thinking of Matrix Multiplication as the inner product of a dual vector with a vector from a computational optimization point of view.* Consider a "naive" implementation of matrix multiplication, which, regardless of your choice of programming language, mathematically is essentially the following:

$$\begin{aligned} &\forall\, i = 1, 2, \dots M, \\ &\quad \forall\, j = 1, 2, \dots N, \\ &\quad\quad \forall\, k = 1, 2, \dots P, \\ &\text{sum}+ = A_{ik} B_{kj}, \quad \textit{i.e.} \quad \sum_{k=1}^P A_{ik} B_{kj} \end{aligned} \tag{7}$$

A (very) cursory look at the read requirements in this matrix multiplication operation would tell us that the read, memory requirements are of order $O(M * P^2 * N)$ (imagine multiply a row from A with a "fixed" column from B. That same "fixed" column from B will have to be multiplied by the other $M-1$ rows, but we've read this particular column from B already $M-1$ redundantly). Nevertheless, it is polynomial in time.

Also, notice that if we assume *row-major ordering* for how the values of the matrix entries are inserted or stored, contiguously, in memory addresses, then in accessing the values for the columns of matrix B, we are not efficiently accessing those values wince we're "jumping" over P memory addresses to grab the next adjacent entry in a column.

However, Eq. 6 gives us a prescription:

$$C_{ij} = A_{i*} \cdot (B^T)_{j*} \quad \forall\, i = 1 \dots M, \forall\, j = 1 \dots N \tag{8}$$

If we can store the "columns" of B as rows, $(B^T)_{j*}$, then we can access columns in a way that their respective memory addresses are contiguous. Also, if we have an optimized routine for taking *inner products* or, i.e., *dot products*, we can employ that as well, according to Eq. 8. Indeed, we have this in C++11 standard library `<numeric>`, with `std::inner_product`. At this point, take a look at the code in `Mat/Mat.h`, for the implementation of matrix multiplication given in the operator overloading of `*` operator for class `Mat`,

```
Mat<Type> operator*(const Mat<Type>& rhs)
```

. Each row from matrix A, ith row `A_i` and each column from $B \equiv$ `rhs`, `col`, has its inner product computed by `std::inner_product`.

2.2. **Transpose operation on matrices, as tensor products of vector spaces and dual vector spaces.** Consider the mathematical formulation of the transpose operation:

$$\begin{gathered} \text{Mat}_{\mathbb{K}}(M,P) \xrightarrow{T} \text{Mat}_{\mathbb{K}}(P,M) \\ \bigotimes_{i=1}^{P} (\mathbb{K}^M) \xrightarrow{T} \bigotimes_{j=1}^{P} (\mathbb{K}^M)^* \\ A = (A_{*1}, A_{*2}, \dots A_{*P}) \overset{T}{\longmapsto} A^T = (A^T_{1*}, A^T_{2*}, \dots A^T_{M*}) \end{gathered} \tag{9}$$

Thus, it's clear from Eq. 9 that if we can simply store the "columns" or "column" vectors of matrix A as "rows" or dual vectors, we can simply read off these "rows" or dual vectors as the new rows of a new matrix A^T, the transpose of A.

This is what was exactly implemented in `Mat.h` in the class template method `Mat<Type> T()`. All we do is take the "columns" of a matrix, stored in private member variable `Columns_` and assign them to the new matrix for the transpose, `new_Rows`, and then initialize a new matrix with the class template `Mat`, `Atranspose`.

3. Further isomorphisms between math and the code

Take a look at the `private` member variables of the class template `Mat` in `Mat/Mat.h`. They include

- `std::array<unsigned int,2> Size_Dims_;`
- `std::vector<std::vector<Type>> Rows_`
- `std::vector<std::vector<Type>> Columns_`
- `std::vector<Type> Entries_`

These are isomorphic to the following:

- (M, P) in $\text{Mat}_{\mathbb{K}}(M, P) \ni A$
- $(A_{1*}, A_{2*}, \dots A_{i*} \dots A_{M*}) \in \bigotimes_{i=1}^{M} (\mathbb{K}^P)^*$
- $(A^T_{1*}, A^T_{2*}, \dots A^T_{j*} \dots A^T_{P*}) \in \bigotimes_{j=1}^{P} (\mathbb{K}^M)^*$
- $(A_{ij}) = A \in \mathbb{K}^{M \cdot P}$ (there is an obvious isomorphism between matrices and a vector space, i.e. $\text{Mat}_{\mathbb{K}}(M, P) \cong \mathbb{K}^{M \cdot P}$.

, respectively.

Understanding these relations and the availability of these private member variables, one can implement further useful methods, including getting the values of the entries, getting a particular row (that's contiguously laid out sequentially on adjacent memory addresses), setting a specific row, and then going from `Rows_` and turning it into a new `Entries_`. Also, addition for matrices (since matrices are non-commutative rings) and scalar multiplication could also be implemented in memory efficient (i.e. saving the number of reads into memory that is needed and the utilization of optimized algorithms in the standard library of C++11) with this understanding.

4. Concluding Remarks

I explicitly show an isomorphism between the mathematical formulation of matrices and the operations matrix multiplication and the taking of the transpose as a tensor product of dual vectors and equivalently tensor product of vectors, and the C++11 implementation of a class template `Mat`. I point out that by doing so, we've discovered a more read-memory efficient and more optimized way of doing matrix multiplication, taking the transpose, and accessing the values of the entries of the matrix, than a "naive" implementation. I was also able to utilize and showcase a number of novel, new features from C++11; `std::vector` as a container and `<numerics>`.

The developer using this code or the framework for it should be clearly able to add new methods and features to the class template, as long as he or she understands the underlying mathematics. In fact, we should agree upon the mathematics first and then try as much as we could to have a 1-to-1 correspondence between the math and the code, in object-oriented programming (OOP) design. I argue this because it should make the code maintainable and clear because we should all agree on the same thing with the math.

Also, with these more efficient memory-accessing, we have made matrix multiplication and the taking of the transpose for matrices not only faster, but more scalable.