Document number: P1166R0
Date: 2019-01-12
Reply-to: Guy Davidson, guy@hatcat.com
Reply-to: Bob Steagall, bob.steagall.cpp@gmail.com
Audience: Game dev and low latency (SG14), Numerics (SG6), Library Evolution Incubator (LEWGI), Library Evolution (LEWG)

What do we need from a linear algebra library?

Abstract

There has been plenty of chatter recently about finally teaching linear algebra to C++. It is indeed slightly strange that there is no standard way of describing the parts of linear algebra: the matrix, the vector, and the operations that can be performed on these entities. Linear algebra is fundamental to most aspects of mathematics, with direct application to many problems addressed in computing, for example, linear regression as used in machine learning, geometry as used in graphics presentation, and even the simple solution of simultaneous equations.

Many libraries have been created with varying design criteria. A brief inspection of the literature will reveal libraries as different as BLAS, Blaze, Eigen and many others. The purpose of the committee is to standardise existing practice, not design new libraries, by describing an API, not an implementation: so how do we describe existing practice in such a way that captures these criteria?

Disambiguation

It is important to note that there are overloaded identifiers that need to be disambiguated. The first is matrix. A matrix is an m x n array of numbers, where m is the number of rows and n is the number of columns. Such a matrix has size m x n.

Within the domain of linear algebra, it is a representation of the coefficients of a system of linear equations with well-defined operations. However, the term matrix is frequently used in computer science to mean an array of numbers of arbitrary size and two dimensions.

Consider dimension: in the linear algebra domain, a vector space is said to be of *finite dimension n* or to be *n-dimensional* if there exist n linearly independent vectors which span V. In computer science, dimension refers to the number of indices used to access an element in an array.

The next term is rank. The rank of a matrix is the dimension of the vector space spanned by its columns, which corresponds to the maximal number of linearly independent columns of that matrix[1].

However, rank also has a meaning in computer science where it is a synonym for number of dimensions[2]. In the C++ standard, at [meta.unary.prop.query] rank is described as the number of dimensions of T if T names an array, otherwise it is zero.

During the development of this paper, these ambiguities caused considerable confusion. It is important to understand that this is a “linear algebra” proposal, not an “array” or “matrix” proposal. This author looks forward to further development and refinement of the array concept: it will be of great value to improving the implementation of a linear algebra library.

Design aspects

Consider the following aspects that affect the design of a linear algebra library:

1. How elements are stored in memory
2. Whether they are stored on the heap or the stack
3. Whether matrix or vector dimensions are fixed at compile-time or resizable at run-time
4. Whether or not the matrix or vector owns the data
5. Whether and how the matrix and vector operations are optimised for concurrency, parallelism or neither

Element storage is significant because data may be sparse or dense. Some applications, for example geometry, use vectors to model positions in space, and matrices to model transformations in space. Every element is significant. Others, for example linear regression of dynamically collected data, use matrices many of whose elements may be zero.

Heap/stack storage comes into play when a matrix may be resized as the operation of the program continues. We want the ability to use a matrix of size RxC that is stored in a data structure having a capacity of PxQ, where R<=P and C<=Q.

The determination of the size of vectors or matrices is significant because the application may model well-defined entities. In the previous example, geometric entities will always be of fixed size, while dynamically collected data will perforce grow monotonically.

Data ownership can again be demonstrated with the same pair of examples. Geometric entities will most likely own their data since the matrix or vector will be a part of the object. For applications using dynamically collected data, the operations may work on a view of the data rather than transforming the data in situ.

---

[1] https://en.wikipedia.org/wiki/Rank_(linear_algebra)
[2] https://en.wikipedia.org/wiki/Rank_(computer_programming)

Finally, the efficiency of the various operations are sensitive to many factors. Plenty of research has been conducted into alternative, faster methods for calculating matrix inversion and multiplication, given particular constraints. The execution context also affects the choice of algorithm.

Orthogonality of concerns

Significantly, these are orthogonal concerns. However, it is possible to optimise according to combinations of these concerns. For example, if you know that your matrices consist of four rows and four columns, stored contiguously, with a 32-bit floating point scalar type, then there are relevant SIMD instructions and intrinsics you can use in your implementation.

Additionally, it is possible to trade run-time cost for compile-time cost by using expression templates as an optimisation. This is particularly valuable when adding more than two matrices together in a single operation. Also, intrinsic functions will be required to support optimisation, which may prevent the use of constexpr decoration of the API.

Any API must therefore not preclude the opportunity to take advantage of these optimisations. This implies that there must be facility to specialise any of these orthogonal concerns independently. Such an API must offer a way of enabling the creation of a rich set of overloads, so that the criteria that are favoured by the libraries created to date can be met.

More fundamentally, the types required by linear algebra should not be affected by their scalar type: a matrix of float should be operable with a matrix of double. Nor should mixed storage strategies limit operations. This means that there should be a mechanism for constructing new and appropriate types from combinations of other types.

Identifiers

As mentioned above in the disambiguation section, one problem with such a proposal is the set of identifiers, both semantic and syntactic. Consider the concepts of size, dimension and rank. These have different meanings for arrays, which may be used to implement matrices, and matrices themselves. For example, the rank of a matrix depends on its content, while the rank of an array depends on the layout. There is already a symbol, std::rank, a type trait which supports the array concept of rank, not the matrix concept. std::vector is of course already taken, and can be viewed as similar to the linear algebra concept. std::inner_product does the job of inner product sufficiently, and std::modulus is an operation of modulo arithmetic; however these already have constraints which are not suitable for a linear algebra proposal.

Additionally there is the notion of product. The usual linear algebra product is the matrix product, M×M (the `×` symbol is unicode code point U+00D7), as also used in geometry, but also popular is the Hadamard product, M○M (the `○` symbol is unicode code point U+25CB), a component-wise multiplication of matrices which is constrained similarly to matrix addition. There is also the cross product, V×V, an operation on pairs of three-element vectors which

can be geometrically understood as the normal vector to the plane described by those two vectors.

While the symbols issue remains thorny, it is clear that a linear algebra proposal must offer a matrix product; any other product is going to be domain specific, which means that specialisation must be possible at the correct level of abstraction. These specialisations should be presented as Succession Proposals, such as for Euclidean geometry or linear regression.

Abstractions

The set of types and functions should be the minimal set required to perform functions in finite dimensional spaces. This includes:

- Matrix, row vector and column vector class templates
- Binary operations for addition, subtraction and multiplication of matrices and vectors
- Binary operations for scalar multiplication and division of matrices and vectors

Syntactic sugar should also feature element access, as well as element storage and promotion helpers. Type traits for the abstractions such as is_column_vector, is_matrix and so on would be very helpful.

Consideration should be given to overloading the function operators for submatrix and subvector views with spans as parameters, as well as element access using an index pair. Additionally, the square operator should be used for individual element access.

If the language syntax could be modified to allow element access through the square operator, for example through comma operator overload, that would be a great help to teachability.

Non-member functions should include at least transpose, modulus, unit and identity. Calculating the inverse and determinant of matrices is not trivial, and should be delegated to those domains which use them.

Wish list

It seems therefore that to support the orthogonal concerns, a linear algebra library should offer:

1. Customisation via template parameter of scalar type, size, element layout, storage and growth
2. Promotion helpers for operations between matrices of different scalar types
3. Customisation of operations

Additionally, it should expose:

1. Identifiers in a sub-namespace of std
2. Overloaded operators for views and element selection
3. Type traits describing the nature of each abstraction
4. Non-member functions for typical, trivial linear algebra operations