# What is the optimum design of a C++ numerical array library for large-scale scientific applications?

Robin J. Hogan[1,2]

[1] *European Centre for Medium Range Weather Forecasts, Reading, UK*

[2] *School of Mathematical, Physical and Computational Sciences, University of Reading, UK*

Version 1.0 (August 18, 2020)

**Abstract**

As a language for scientific computing, C++ is at a disadvantage compared to many other languages due to its lack of a well-designed standard for multi-dimensional arrays supporting efficient whole-array expressions, expressive array-subsetting syntax and linear algebra. To support the development of such a standard, in this paper I review the interface, capabilities and weaknesses of a number of free C++ array libraries (Adept, Armadillo, Blaze, Blitz++, Eigen, MTL4, ra-ra, uBLAS and Xtensor) as well as other languages supporting multi-dimensional arrays (particularly Fortran, Python, Matlab, IDL and Julia). These are contrasted with the verbose and limited whole-array capabilities in the C++20 Standard Template Library. To help ensure the standard meets the needs of large-scale scientific applications, I also present an analysis of array use in an Earth-system model for operational weather forecasting (2.2 million lines of code). I argue that an unlimited number of dimensions should be supported, not the limit of two imposed by many libraries focusing on linear algebra, and propose a solution to the lack of a matrix-multiplication operator in C++. A detailed investigation is presented of the problem that most C++ libraries cannot simply and efficiently pass a subset of an array to a function, and a solution is proposed. A total of 25 specific recommendations are made that will hopefully contribute to a discussion leading to the formulation of a standard.

## 1 Introduction

C++ is an incredibly versatile software language, and is the language of choice for most large-scale applications where speed is paramount. For scientific applications, the fundamental object is the *multi-dimensional array*, for which numerous languages have either excellent built-in support (e.g. Fortran, Matlab, Julia and R) or a dominant well-supported library (Python). These languages support whole-array operations, expressive array-subsetting syntax, a large number of array manipulation functions, and in the case of Fortran they are all optimized by the compiler. In principle, C++'s template capability should enable a multi-dimensional array class to be designed that has all these features and more. Indeed, it is 25 years since Veldhuizen (1995) demonstrated that the use of *expression templates* in C++ could enable the compiler to optimize whole-array expressions without allocating and deallocating temporary arrays, thereby achieving a speed comparable to Fortran. This he demonstrated in the pioneering and open-source *Blitz++* library, which supports up to 11 dimensions. In an ideal world this work would

Contact me at r.j.hogan@ecmwf.int if you have any comments. This document may be revised in future; the most recent version is available at http://www.met.reading.ac.uk/clouds/cpp_arrays/.

have led to a powerful multi-dimensional array library being introduced into the C++ Standard Template Library (STL), which would now be very widely used in scientific computing.

Frustratingly, this has not happened. Rather, Blitz++ development has stalled while many new C++ array libraries have emerged that are also built on expression templates. Some of the open-source ones are listed in Table 1, and their detailed properties are reviewed at various places through this paper. Numerous of these libraries specialize in matrix operations and linear algebra. For example, *Eigen* has demonstrated how a library's use of under-the-hood vector intrinsics can achieve excellent performance in array expressions, thereby moving the task of automatic vectorization from the compiler to the library. Iglberger et al. (2012) pointed out that expression templates should be used only for the parts of an expression that involve element-wise array operations, whereas matrix multiplication and other expensive linear-algebra operations are most efficiently implemented by consolidating the arguments to these operations into single matrices, creating temporaries if necessary, and delegating the operation to efficient kernels such as those provided by the BLAS and LAPACK libraries. This led to the development of the *Blaze* library.

However, Eigen, Blaze, uBLAS and MTL4 all support vectors and matrices, but not arrays with more than two dimensions. This, unfortunately, makes them quite unsuitable for many scientific applications that solve problems of fundamentally higher dimensionality (see section 3 for a detailed discussion). C++ libraries that support many more dimensions are listed in Table 1, but there are plenty of further issues to consider. Some libraries have cumbersome syntax, some lack features essential for certain applications, some are barely maintained, and surprisingly most have no way to efficiently pass array subsets to and from user-defined functions. Moreover, these libraries are all incompatible with one another, making it very messy to merge or interface one scientific code-base with another. In this situation, C++ puts itself at an unnecessary disadvantage compared to other languages for scientific applications. Why is there such a plethora of C++ array libraries, rather than a dominant player that attracts most of the development effort, like NumPy in the case of Python? One problem is that the code of expression-template libraries is always very difficult to understand and usually has the questionable decisions of the initial developer baked in at a low level. So if you have a great idea for a fundamental new capability or way to implement expression templates more efficiently, it is usually faster, and always more fun, to start a new library from scratch.[1]

Recognizing the need for a multi-dimensional array and/or linear algebra standard for C++, there have recently been proposals for future additions to the STL. Hoemmen et al. (2019) provided an excellent summary of the historical lessons to learn in considering the standardization of such a library. Davidson and Steagall (2019) proposed the addition of linear algebra support and focused on the design of the vector and matrix containers, but dismissed the notion that these classes should be specializations of a general $N$-dimensional array class. Hollman et al. (2019) proposed the addition of an `mdspan` class to the STL (slated for inclusion in C++23), a multi-dimensional array "view" of an existing span of memory, with a corresponding `mdarray` class that owns its data being proposed a little later. While these various proposals are rich in implementation details, what I haven't managed to find is:

1. A comprehensive discussion of the overarching *requirements* of such a library, informed by an analysis of array use in existing large- and small-scale scientific applications (including in other languages);

2. A detailed *comparison* of the provisions of other languages and C++ libraries;

[1]This was the story with my *Adept* library: I had developed a library using expression templates to do fast automatic differentiation on scalars (Hogan, 2014), and turning this into a library that could differentiate array expressions (Hogan, 2017) was far easier than hacking every class of an existing array library.

3. A focus on the best design for a concise and intuitive *user interface*.

There is therefore a danger that capabilities will be added piecemeal to the STL without a comprehensive vision of the overall solution, leading to the "valarray trap": a well-intentioned capability that lacks crucial features (more than one dimension in the case of the `valarray` class) so ends up being rarely used by its target audience.

This paper is intended to contribute to the optimal design of a multi-dimensional array capability for C++ by tackling the three points above, comparing the best and worst interfaces amongst the existing C++ array libraries as well as reviewing what can be learned from other languages. The existing C++ array libraries already tend to be modelled on their developer's favourite language or application: uBLAS was built as a wrapper for BLAS and LAPACK, Eigen and Armadillo aspire to reproduce the features of Matlab, while Xtensor takes its inspiration from Python/NumPy. It is important to point out that Matlab and Python are not suited for *large-scale* scientific applications, i.e. those run on supercomputers, where Fortran is still dominant (e.g. Guillen and Bader, 2017). While Fortran is a dirty word amongst many C++ programmers, modern Fortran (by which I mean Fortran 2003 and later) has shed almost all of the failings of Fortran 77, and as the only other genuinely compiled language in Table 1, it is crucially important that we learn from its clean and powerful treatment of multi-dimensional arrays (see also Decyk et al., 2007; Arabas et al., 2014). As a source of evidence to help inform the paper, the appendix presents an analysis of a piece of large-scale scientific software from my own field: a global Earth-system model used for operational weather forecasting. While it is not fully representative of all scientific applications, it is large enough to be a useful test case: if one was to convert the 2.2 million lines of Fortran 2003 into C++ using a newly designed array library, it should be at least as concise, elegant and efficient as the existing code.

The challenges in defining such a library are tackled in turn. Section 2 provides some overarching design principles based on accepted good C++ practice and the STL, as well as highlighting some examples of where the STL has an inferior interface to many other languages and libraries. Section 3 examines the question of how many array dimensions should be supported, and how the fundamental design of libraries and languages is affected by whether they see themselves as matrix libraries for linear-algebra, or general multi-dimensional numerical array libraries, and how this can prove limiting when they are inevitably used for both. Section 4 shows how the syntax for subsetting arrays in C++ libraries is much less clear than in other languages, and proposes a solution. Section 5 discusses the various array manipulation functions that should be supported. Section 6 examines the surprisingly challenging question of how to efficiently pass arrays, array subsets and array expressions as arguments to functions. Section 7 describes some of the features of existing libraries that should be avoided. Section 8 discusses interoperability between different implementations of the library, and with Fortran, proposing a data structure for array types. Section 9 discusses support for linear algebra functions and special array types. Section 10 describes the run-time checking capabilities that should be available for debugging. Numbered recommendations are presented in bold at various places in the text, and then summarized in section 11.

## 2 Overarching design principles

Naturally, if a standard is to be accepted it needs to conform to good C++ practices, but there are numerous examples in existing C++ array libraries where this is not the case. Three examples are as follows. (1) The copy constructor of an ordinary array object should perform a deep copy, rather than a shallow copy in which data are shared between two arrays which is not what most users would expect. (2) The library

Table 1: Some of the properties of numerical arrays in various languages and libraries, where the dimension order is either row-major (R) or column-major (C), the operator style is either array (A) or matrix (M) (see section 3.2), and the final two columns indicate the operator or function used to invoke whole-array element-wise and matrix multiplication (where N/A indicates that this feature is not available or applicable). C++ libraries are only listed if they are open source, accelerate whole-array operations using expression templates and contain an array type that can be resized at run-time.

| Language | Library or version | Max. dims. | Dim. order | Operator style | Element-wise multiplication | Matrix multiplication |
|---|---|---|---|---|---|---|
| C++ | STL `valarray` | 1 | N/A | A | `*` | N/A |
| | uBLAS | 2 | R‡ | M | `element_prod()` | `prod()` |
| | MTL4 | 2 | R‡ | M | `ele_prod()` | `*` |
| | Blaze | 2 | R‡ | M | `%` | `*` |
| | Eigen (`Matrix`) | 2 | C‡ | M | `.array()*` | `*` |
| | Eigen (`Array`) | 2 | C‡ | A | `*` | `.matrix()*` |
| | Armadillo | 3 | C | M | `%` | `*` |
| | Adept | 7 | R‡ | A | `*` | `**` |
| | Blitz++ | 11 | R‡ | A | `*` | N/A |
| | ra-ra | $\infty$ | R‡ | A | `*` | N/A |
| | Xtensor | $\infty$ | R‡ | A | `*` | `linalg::dot()` |
| Fortran | until Fortran 2003 | 7 | C | A | `*` | `matmul()` |
| | from Fortran 2008 | 15 | C | A | `*` | `matmul()` |
| Python | NumPy (`matrix`) | 2 | R | M | `.multiply()` | `*` |
| | NumPy (`ndarray`) | 32 | R | A | `*` | `@` |
| IDL, GDL | (built in) | 8 | C | A | `*` | `##` |
| PV-Wave | (built in) | 8 | C | A | `*` | `#` |
| Matlab | until version 4 | 2 | C | M | `.*` | `*` |
| | from version 5 | $\infty$ | C | M | `.*` | `*` |
| Octave | (built in) | $\infty$ | C | M | `.*` | `*` |
| Julia | (built in) | $\infty$ | C | M | `.*` | `*` |
| R, S-Plus | (built in) | $\infty$ | C | A | `*` | `%*%` |

‡The default dimension indexing order is column-major or row-major as shown, but the opposite order is available as an option.

should not allow a user to accidentally modify a `const` object. (3) Read-only array subsetting operations should be thread-safe, i.e. different threads ought to be able to safely read subsets of the same array at the same time. As explained in section 6.3, these three principles are all violated by the Adept (Hogan, 2014, 2017) and Blitz++ libraries in order that they can pass array subsets as arguments to functions, but sections 6.5 and 7.1 present a solution to this problem.

Non-member non-friend functions, rather than member functions, should be used for mathematical functions that do not modify an array (see item 23 of Meyers, 2005). Put another way, arrays should not be "God" objects that contain within them all possible mathematical functionality, but containers that provide an interface to enable external functions to operate on them. As pointed out by Hoemmen et al. (2019), this principle was violated by earlier versions of Eigen, which implemented mathematical func-

tions only as class members [e.g. M.sin() instead of sin(M)], although in more recent versions both patterns are accepted. Similarly, the maximum value in an Armadillo array is returned with M.max(), when this would better be implemented as a non-member function. These functions are also applicable to array expressions, but one would never expect to write (M+1.0).max() so clearly max should not be a member function. When are member functions acceptable? Obviously for functions that modify the array or access its internal data, but some libraries also use member functions for array subsetting operations that return lvalues: an operation like M.diag() would return the diagonal of matrix M as an lvalue vector that can be modified (e.g. M.diag() = 1.0), whereas diag(M+1.0) would return an rvalue.

**Recommendation 1: Follow widely-accepted guidance on good C++ practice, e.g. copy constructors of ordinary arrays perform a deep copy, constness is respected, read-only array operations are thread-safe, and non-member non-friend functions are used for mathematical operations that do not modify an array.**

In the hope that one day it could form the basis of a multi-dimensional numerical array library for the Standard Template Library (STL), arrays classes should look as much as possible like existing STL containers, leading to the following recommendations on style:

- Use lower-case names for classes, member functions, member types and so on. Around half of existing C++ array libraries do this.
- Provide member types and member functions matching those of std::vector, where possible, e.g. value_type, size_type, data(), empty(), size(), clear() and so on.
- Avoid using the same name as an existing STL function if the behaviour is different, e.g. end (see section 4.1) and min (see section 5.1).
- One-dimensional arrays should be valid STL containers, providing STL-compatible iterators (as in Armadillo). However, further discussion is needed before providing iterators for multi-dimensional arrays since it is not clear whether such an iterator should iterate over elements or array subsets.
- Allow arrays to be constructed and assigned from C++11 initializer lists, which may be nested:

  ```
  dmatrix<double> M = {{1.0, 2.0}, {3.0, 4.0}}; // Initialize size & data
  ```

  This form is supported by most existing libraries, but not by Blitz++ and Eigen that only support the Blitz++ style:

  ```
  darray<double> M(2,2);
  M << 1.0, 2.0, 3.0, 4.0;
  ```

  This is a smart C++98 work-around, but is not essential since the introduction of initializer lists. There may be some value in retaining it if it also allows an array to be filled by concatenating smaller arrays.

  Note that in the example above and through the rest of this document I use the following class to represent dynamically-sized (but statically dimensioned) arrays:

  ```
  // Any other template arguments have default arguments so can be hidden
  template <typename Type, unsigned int Rank>  class darray;
  // Aliases such that dvector<float> and dmatrix<float> represent a vector
  // and a matrix of floats
  template <typename Type>  using dvector = darray<Type,1>;
  template <typename Type>  using dmatrix = darray<Type,2>;
  ```

- Use a single namespace for the entire library. Namespaces are to avoid name collisions and the STL uses the single namespace `std` for everything (with a few exceptions such as the C++20 namespace `std::ranges` to avoid name collisions with `std`). Xtensor is an example of a library that arguably over-uses namespaces since its nested namespaces map to the names of packages within NumPy, which have a different function to C++ namespaces. A possible exception is for the concise symbols used to index arrays as discussed in section 4. Suppose our library was in the namespace `std`: rather than extracting every other column of a matrix with `M(std::__,std::_(1,std::last,2))`, we could put these symbols in namespace `std::symbols`, allowing the user to specify "`using namespace std::symbols`" followed by the much more readable `M(__,_(1,last,2))`.

More recent versions of the STL have provided limited array manipulation capabilities; obviously they are only applicable to 1D containers, but could they form a starting point for a full array capability? In the following examples, the syntax of array operations in Adept, Fortran and Matlab (generally very similar to the other languages and libraries in Table 1) are compared to the equivalent in the STL, as well as the slightly less verbose *ranges* capability in the C++20 STL:

```
// Sum the elements in a vector of floats, excluding the first and last
sum_v = sum(v(1,end-1));                              // Adept
sum_v = accumulate(v.cbegin()+1, v.cend()-1, 0.0); // STL

// Compute the product of all the elements in v
prod_v = product(v);                                  // Fortran, Adept
prod_v = accumulate(v.cbegin(), v.cend(), 1.0, std::multiplies<float>());// STL

// Are all the elements of v equal to zero?
is_all_zero = all(v == 0);                            // Fortran, Matlab, Adept...
is_all_zero = all_of(v.cbegin(), v.cend(), [](float f){return f==0;}); // STL
is_all_zero = std::ranges::all_of(v, [](float f){return f==0;}); // STL C++20

// How many elements are greater than five?
count_v = count(v > 5.0);                             // Fortran, Adept
count_v = count_if(v.cbegin(), v.cend(), [](float f){return f>5.0;}); // STL
count_v = std::ranges::count_if(v, [](float f){return f>5.0;}); // STL C++20

// What is the maximum value in v?
max_v = maxval(v);                                    // Fortran, Adept
max_v = *max_element(v.cbegin(), v.cend());           // STL
max_v = *std::ranges::max_element(v);                 // STL C++20

// Set all elements to a scalar
v = 5.0;                                              // Fortran, Adept...
fill(v.begin(), v.end(), 5.0);                        // STL
std::ranges::fill(v, 5.0);                            // STL C++20

// Copy the contents of one vector to another
w = v;                                                // Fortran, Matlab, Adept...
copy(v.cbegin(), v.cend(), w.begin());                // STL
std::ranges::copy(v, w.begin());                      // STL C++20

// Conditional assignment
w = v(find(v > 5.0));                                 // Matlab, Adept
copy_if(v.cbegin(), v.cend(), w.begin(), [](float f){return f>5.0;}); // STL
std::ranges::copy_if(v, w.begin(), [](float f){return f>5.0;}); // STL C++20
```

```
// Add one to a vector
w = v + 1.0;                                              // Fortran, Matlab, Adept...
transform(v.cbegin(), v.cend(), w.begin(), [](float f){return f+1.0;}); // STL
std::ranges::transform(v, w.begin(), [](float f){return f+1.0;}); // STL C++20

// Dot product
x = dot_product(v,w);                                     // Fortran, Adept
x = inner_product(v.cbegin(), v.cend(), w.begin(), 0.0); // STL
```

Note that the `std::` prefix has been omitted from any function where it can be deduced by argument-dependent lookup. It is striking that in every case above the STL syntax is much less readable and concise than any of the other libraries and languages listed in Table 1. The main problem is the STL's philosophy of iterators, which are unnecessary for sequential containers (like `std::vector` and the arrays considered in this paper) and unsuitable for arrays of more than one dimension. Would a multi-dimensional array class in the STL be forced to use the same style? If scientific programs are to be easily readable and writable then it is crucial that the interface is as simple as possible, so at some point the decision might need to be taken that the STL is not the best place for a multi-dimensional array standard that will meet the needs of scientists, and another location (e.g. Boost) is preferred.

**Recommendation 2: Follow the style of the Standard Template Library *where appropriate*, e.g. ensuring that 1D arrays can behave as a fully STL-compatible container, lower-case names and using a single flat namespace.**

**Recommendation 3: Keep the interface as simple as possible! Due to its use of iterators, the limited array manipulation functionality available in more recent versions of the STL is typically much less readable and concise than other languages used in scientific computing; this style should *not* be emulated.**

# 3 Dimensions, vectors and matrices

## 3.1 Dimensionality

A fundamental question in designing a general-purpose numerical array library is how many array dimensions should be supported. Table 1 shows that this varies immensely between existing libraries and languages. The STL contains the `valarray` class but it is little used as it is only 1D. Some C++ libraries, such as Eigen and Blaze, are designed primarily for linear-algebra applications and support only 1D and 2D arrays, i.e. vectors and matrices. Armadillo (Sanderson and Curtin, 2016) is similar but with a bolt-on `Cube` class for 3D arrays.

Is a maximum of two or three dimensions sufficient for large-scale scientific applications? In the case of the Earth-system model analyzed in the appendix and written in Fortran 2003, around 23,800 arrays (14% of the total) have three or more dimensions and 3,800 (2.2% of the total) have four or more dimensions. It is therefore inconceivable that this application could be coded in C++ using Eigen, Blaze or Armadillo. Even for smaller-scale applications, often more than two or three dimensions are required. Until version 4, Matlab only supported two dimensions but it was recognized that this was very limiting and version 5 (introduced in 1996) provided support for an essentially unlimited number of dimensions.

Adept supports 7 dimensions, which is the maximum in Fortran 2003. One would expect this to be sufficient for almost all scientific applications, although Blitz++, Fortran 2008, Python, IDL, Julia and R exceed this. What prevents a C++ library out-doing its competitor languages? Once we decide to support four or more dimensions then it makes sense to define a single array class whose number of

dimensions is defined by an integer template parameter, raising the maximum to more than anyone would ever need. Until C++11, the practical limit was the number of arguments accepted by `resize` and any other functions that require one argument per dimension. Since C++11, such functions can be defined recursively using variadic templates and therefore full support can be provided for an arbitrary number of dimensions (Aragón, 2014), as demonstrated by the Xtensor and ra-ra libraries in Table 1. The C++11 `initializer_list` class enables multi-dimensional arrays to be initialized by C-style nested sets of curly braces, as illustrated in section 2.

**Recommendation 4: Use the variadic-template and initializer-list features of C++11 to provide full support for any number of array dimensions.**

### 3.2 Matrix-style or array-style library?

As shown in Table 1, the interfaces to current array libraries tend to fall into two categories: *matrix-style libraries,* predominantly for applications involving linear algebra, and general-purpose *array-style libraries*, needed especially for large-scale applications such as the one in the appendix. This is related to the question of dimensionality discussed in the previous section, since matrix-style libraries are often limited to at most two dimensions. The three main differences are as follows:

- Matrix-style libraries use the `*` operator to invoke matrix multiplication, while array-style libraries use this operator to invoke element-wise multiplication (also known as the Hadamard or Schur product). Outside C++, equivalent differences arise as to whether the power operator for scalars (e.g. `**` in Fortran and Python and `^` in Matlab and IDL) invokes the matrix power when applied to matrices, or the element-by-element power operation. Also, Matlab invokes matrix right-divide with the `/` operator.

- In matrix-style libraries, vectors have an intrinsic orientation (i.e. they are column or row vectors), and behave as such in matrix multiplication. In array-style libraries, vectors usually have no intrinsic orientation, but (at least in Fortran, IDL, PV-Wave, Python, Adept and Xtensor) will adopt a particular orientation according to the context. Taking Python as an example, if `v` is a 1D array and `M` is a 2D array, then in the matrix multiplication operation `v @ M`, `v` will be treated as a row vector whereas in `M @ v`, `v` will be treated as a column vector.

- Matrix-style libraries tend to have a richer range of linear-algebra functions than array-style libraries, although this is not an intrinsic difference and there is no reason why the same range of functions could not be provided with an array-style library.

Whichever style of library we choose, we have the problem that there are two types of multiplication but only one `*` operator. Table 1 shows the variety of solutions adopted by different languages. The matrix-multiplication operators used by array-style Python, IDL and PV-Wave are concise, as is the element-wise operator used by Matlab, but none are available as overloadable operators in C++. In C++, the matrix-style Armadillo and Blaze libraries redefine the modulus operator `%` for element-wise multiplication, which is concise and has the same precedence as the `*` operator, but leads to an ambiguity: what should it do when applied to integer arrays, an element-wise modulus operation or element-wise multiplication? The `%` operator might be a reasonable matrix-multiplication operator in an array-style library, invoking matrix multiplication on floating-point arrays and the modulus operation on integer arrays. A rather cumbersome solution is adopted by Eigen: matrix multiplication can only be performed on `Matrix` objects and element-wise multiplication only on `Array` objects, and we must convert between them if

both types of multiplication are required. While the conversion is cheap, it can lead to very verbose code; compare Eigen to other libraries for an example containing both types of multiplication:

```
D = A.array().exp().matrix() * (B.array() * C.array()).matrix(); // Eigen
D = dot(exp(A), B*C); // Xtensor
D = exp(A) * (B%C);   // Armadillo, Blaze
D = exp(A) ** (B*C);  // Adept
```

As can be seen, Adept has an elegant solution: it creates a pseudo-operator `**` for matrix-multiplication by overloading the unary dereference operator (`*`) of the array class to create a wrapper type that invokes matrix multiplication when it appears as the right-hand argument of the binary multiply operator (`*`). The wrapper class is optimized away by the compiler so that `**` behaves as an operator with the same precedence as an ordinary multiplication, and does not conflict with any other operators. This leaves the `*` and `%` operators available to join all the other operators in performing the equivalent element-wise array operations to their scalar counterparts. To Fortran and Python users, `**` looks oddly like a power operator, whereas for others it is quite natural to associate with matrix multiplication since it can imply "multiplication but more so".

This review has established that with some combination of the `*`, `%` and `**` operators, we can have a C++ matrix-style library that can still invoke element-wise multiplication concisely, and an array-style library with an elegant and concise way to invoke matrix multiplication. So now to the key question: which style is best for a library that can serve the widest range of scientific applications? The first observation is that many scientific applications make extensive use of arrays but have little or no use for matrix multiplication, but by contrast applications solving a problem in terms of linear algebra and matrix multiplication invariably also make extensive use of element-wise multiplication. Moreover, in section 3.1 it was argued that an unlimited number of dimensions should be supported, and ideally the operator behavior should be consistent for arrays of any number dimensions. Since matrix multiplication does not make sense for arrays of more than two dimensions, it is preferable for the `*` operator to invoke element-wise multiplication of arrays of all dimensions.

What can be learned from experience with other languages? Having used Matlab myself for a wide range of applications over the last 25 years (including optimization problems solved using linear algebra), the matrix-style behaviour is a source of continued annoyance. First, the `.*` operator must be used far more often than the `*` operator, which one frequently forgets. Second, the fact that vectors always have an orientation is annoying in the majority of cases where orientation is irrelevant, as it frequently leads to bugs where element-wise operations on two vectors fail because they do not have the same orientation. In the case of Python, the original array capability provided by NumPy was the matrix-style `matrix` class, but it was recognized that this was not well suited to the majority of applications, and now all users are recommended to use the `ndarray`, which provides matrix multiplication via the `@` operator (Smith, 2014). These arguments lead to the following recommendations:

**Recommendation 5: The library should use "array-style" behaviour in which vectors have no intrinsic orientation but (as in many other languages and libraries) behave as row vectors in when used in vector-matrix multiplication and column vectors when used in matrix-vector multiplication.**

**Recommendation 6: The `*` operator should be used to express element-wise multiplication and the `**` pseudo-operator should be used to express matrix multiplication.**

Section 9 provides further discussion on how matrix multiplication should be implemented flexibly and efficiently, and the need for additional classes to represent matrices with special properties such as symmetric, banded and sparse matrices.

One final question: with no intrinsic vector orientation, should the matrix product of two vectors be the inner (dot) product or the outer product? Python performs the inner product, while IDL (via the `##` operator) and PV-Wave (via the `#` operator) perform the outer product. Due to the ambiguity, it is not valid to perform matrix multiplication on two vectors in either Fortran or Adept, with the user required to use the `dot_product` function or (available in Adept only) the `outer_product` function. As shown in section 2, the STL provides a `std::inner_product` function but it takes as input iterators to generic containers.

### 3.3 Row-major, column-major or both?

There are two conventions in how array indices map to memory addresses. If the first element of an $n$-by-$m$ matrix `M` is at memory address `M.data`, then a *row-major* contiguous storage strategy with zero-based indexing stores element `M(i,j)` at `M.data[i*n+j]`, while *column-major* would use `M.data[i+j*m]`, and equivalently for higher-dimensional arrays. Table 1 shows that both conventions are widespread in numerical array languages and libraries. They are equally efficient provided that applications access memory sequentially. With no mathematical or performance reason to choose between them, which should be preferred? I argue for row-major, because it is the convention of C and C++ built-in multi-dimensional arrays: why should a C++ library ape the convention of another language instead? One minor advantage is that row-major grouping of data is clearly implied by the way that braces are nested when using C-style array initialization, as in the example in section 2. An awkward property of the column-major library Armadillo is that if it initialized a matrix in the same way then it would actually store the numbers in memory in the order 1, 3, 2, 4.

The discussion above concerns the appropriate default behaviour, but it is clearly useful to be able to represent both orientations, as supported by all existing libraries except Armadillo. Note that for the sake of limiting complexity, we would expect that the evaluation of a multi-dimensional array expression would always always traverse the last dimension in the tightest loop, and therefore would only be expected to vectorize an expression if all arguments were in row-major order and contiguous in their final dimension.

**Recommendation 7: As a library for the C++ language, it should by default follow expected C++ behaviour such as row-major array ordering and zero-based indexing, but supporting column-major order as an option for some arrays (possibly with reduced efficiency).**

## 4 Creating array subsets

### 4.1 Towards clear and concise syntax

This section concerns the creation of an array *subset* (also known as a *view*, a *slice* and a *projection*) that can be used as an lvalue or an rvalue in an array expression, with neither making a deep copy. Scientific codes use array subsets within complex mathematical expressions, so it is crucial that subsetting syntax is concise and intuitive in order that the mathematical intent of the code is clear and not obscured by syntactic noise. We established in section 3.1 that the ideal array library should support any number of dimensions, but the basic subsetting functionality for some of the matrix-style libraries in Table 1 is typically in terms of rows of columns (e.g. the `tail_cols` and `topLeftCorner` member functions in Armadillo and Eigen, respectively) so does not scale up to more than two dimensions. Rather, we should start with the very concise syntax of Fortran, Python, Matlab and Julia. Consider how one would extract every other column from a matrix, excluding the first, in these languages:

```
M(2::2,:);       // Fortran  (1-based col-major indexing, begin:end:stride)
M[:,1:-1:2];     // Python   (0-based row-major indexing, begin:end:stride)
M(2:2:end,:);    // Matlab   (1-based col-major indexing, begin:stride:end)
M[:,2:2:end];    // Julia    (1-based row-major indexing, begin:stride:end)
```

Note that Python uses negative indices to count back from the end of the array. Alas there is no overloadable colon operator in C++, and there is a big difference in how concisely C++ array libraries manage to reproduce its functionality:

```
M(seq(2,last,2),all);                       // Eigen   (begin,end,stride)
M(__,stride(1,2,end));                      // Adept   (begin,stride,end)
M(all,Range(2,toEnd,2));                    // Blitz++ (begin,end,stride)
M(all,iota(M.size(2)/2,1,2));               // ra-ra   (count,begin,stride)
strided_view(M,{all(),range(1,_,2)});       // Xtensor (begin,end,stride)
project(M,slice(0,1,M.size1()),slice(1,2,M.size2()/2));
                                            // uBLAS   (begin,stride,count)
```

It is striking how much better Fortran, Python, Matlab and Julia are than any of the C++ examples: the former use letters only in the array name and in "`end`", whereas the latter use letters for indexing functions, which reduces readability. The first four C++ examples are quite similar to each other in terms of conciseness, but the Xtensor and uBLAS syntax is far less concise or clear, not even starting with the name of the array; this could quickly make complex mathematical expressions unintelligible. The cumbersome notation of Xtensor was also pointed out by Hoemmen et al. (2019). The differences between the indexing styles of the C++ libraries is largely superficial: they use `seq`, `stride`, `range`, `iota`, `slice` (and `span` in Armadillo) to mean basically the same thing. With a standard, and a small amount of effort, the compatibility between the libraries could be significantly improved.

It can be seen that there are some useful ideas to express all elements along a dimension [`__`, `all` and `all()`] and to express the final element along a dimension (`end`, `toEnd` and `_`). Note that like Matlab and Julia, Adept allows positions relative the final element to be expressed by, for example, `end-1` and `end/2`. Unfortunately, `all` is better used as a reduction operation (see section 5.2), and `end` clashes badly with `std::end` where it means "one past the end" of a container. Alternatives are to use "`last`" or Python-style negative indexing.

In order to achieve the clarity of the other languages above, we seek a C++ subsetting syntax that uses the minimum possible number of letters. Given the considerations above, I propose these two alternatives:

```
M(__,_(1,-1,2));   // Python style negative indexing
M(__,_(1,last,2)); // Matlab/Julia style but with "last" instead of "end"
```

Here, the name of the range function is simply a single underscore, where the first two arguments are the begin and end indices, with an optional third argument for the stride (as in Fortran and Python). There is little to choose between the two options: the first uses no letters whatsoever in its subsetting syntax, whereas the latter is slightly more flexible as we could select the top half of a matrix with `M(_(0,last/2),__)`. It should be stressed that the "`_`" function returns an object representing a regular array of numbers, not the numbers themselves, in order that it is known at compile time that the subset will be regularly laid out in memory.

**Recommendation 8: Where possible, use indexing arguments named purely with underscores to achieve the same clarity and conciseness as Fortran, Python, Matlab and Julia. Do not support member functions specific to matrices that do not make sense in arrays of higher dimension (e.g. `topLeftCorner`).**

C-style indexing of a matrix is `M[i][j]` to extract a single element and `M[i]` to represent row $i$. To enable this functionality, `M[i]` should be a synonym for `M(i,__)`, returning an lvalue slice with one fewer dimensions than the original array, and if the array is already 1D then the `[]` operator returns a single element as an lvalue. The same behaviour is supported by Boost's `multi_array` class (which lacks whole-array expressions) and Python. Naturally, the recommended way to extract a single element is `M(i,j)` as it would be faster. Xtensor takes an approach similar to Matlab: if a multi-dimensional array is indexed either using `M[i]` or `M(i)` then it will be treated as if stretched into a vector. In Matlab this is useful for conditional operations using the `find` function, but since there are more efficient ways of enacting conditional operations (see section 5.1), this behaviour seems unnecessary.

There are a number of further operations that return an lvalue to part or all of the original array and should be supported. Taking Eigen as an example, the diagonal of a matrix can be returned as a vector lvalue with `diagonal()` and the transpose of a matrix with `transpose()`. Adept provides the member function `permute(i,j,...)` as a generalization of the transpose but for arrays of higher rank; since the new dimension ordering required is almost always known at compile time, it might be better to use template arguments, e.g. `permute<i,j,...>()`, enabling compile-time checks that the dimension indices each appear only once. Another useful function, variants of which are provided by Xtensor and Adept, is `reshape(n,m,...)`, which reinterprets the data in an array as having a different dimension lengths and possibly a different rank. The validity of such an expression is subject to the data in the original array being contiguous in memory and the total number of elements being conserved. Fortran also provides the functions `pack` and `unpack` (see Table 5) but their behaviour is closely related to `reshape`.

**Recommendation 9: Support C-style array indexing, and support other useful member functions that return lvalue subsets and views of arrays, such as `diag`/`diagonal`, `permute` and `reshape`.**

## 4.2 Return values

We now turn to the question of what kind of objects are returned by the subsetting operations in section 4.1. Eigen and Armadillo return different objects depending on what operation has been performed; for example, the Eigen `diagonal()` member function of the matrix class returns a `Diagonal` object that behaves as a vector but contains a reference to the original matrix, so that when element $i$ is requested, element $(i,i)$ of the original matrix is returned.

Blitz++ and Adept take a simpler approach: all the subsetting operations discussed in section 4.1 return one basic array object that shares its data with the appropriate part of the original array. This is possible because their array type is defined with data members similar to the following:

```
template <typename Type, int Rank> // Other template args have default values
class Array {
 protected:
  Type* data_;            // Pointer to first element
  int dimension_[Rank];   // Size of each dimension
  int stride_[Rank];      // Memory offset for each dimension
  // ...other data members concerning storage here...
}
```

Thus, if an element of matrix `M` is requested using `M(i,j)`, the memory address `data_+i*stride_[0]+j*stride_[1]` will be returned (and similarly for arrays of higher rank). Normally the final element of `stride_` is one, but by allowing it to take other values the result of all regular slicing operations can be represented. When such an object does not "own" its data but points to owned by another object, we refer to it as an *array reference*, which is the same concept as `mdspan`

proposed by Hollman et al. (2019). Consider the `diagonal()` member function applied to a contiguous 2D array dimensioned $N \times N$: we simply return a 1D array reference of length $N$ with the same value in `data_`, but its single `stride_` value set to $N + 1$. Other operations are trivial to implement; for example, the transpose of a matrix returns an array reference with the same value for `data_` but `dimension_` and `stride_` each have their two elements reversed. This approach to subsetting has several advantages over Eigen and Armadillo:

- The library is smaller: we do not need a new class for every new subsetting operation.
- It is likely to be more efficient because it removes the indirection associated with holding a reference to the original array object.
- An array subset, which may be non-contiguous, can be passed as writable array argument to a function without a deep copy being made, as in Fortran (see section 6).
- The basic array type can represent both row-major or column-major ordering without an additional template parameter.

One disadvantage of this approach is that it doesn't guarantee the final dimension of an array to be contiguous in memory, so a run-time check is needed for whether vectorization is possible. This can be mitigated in some contexts by the availability of template parameters to specify whether the final dimension of an array, or the entire array, is guaranteed to be contiguous (see section 8). Another idea, possible if ordinary arrays and array references are different classes (see section 6.5), is to guarantee that ordinary arrays are contiguous in memory but not array references. In any case, a run-time check is used by Fortran's "assumed-shape" array type and Fortran does not have a problem with speed.

In the subsetting example `M(__,_(1,last,2))` presented in section 4.1, both dimensions are indexed by a regularly-spaced range of values. But what happens if any arguments are not in this category? First, for every dimension that is indexed by a single integer, the rank of the returned array should be reduced by one. This is the behaviour of Fortran, Python and Julia, but not Matlab which leaves singleton dimensions in the returned array. Second, if any of the dimensions is indexed by a vector of arbitrary integers, then the result is an irregular subset that cannot be represented by a simple array reference, and a wrapper class is needed of the type discussed above for Eigen. Section 6 discusses how to pass such an object as an argument to functions.

**Recommendation 10: The basic array type should support arbitrary strides for all dimensions, enabling all regular subsetting operations to return an array reference, i.e. an array that is configured to point to data owned by another object. Irregular subsetting operations return an slower wrapper class.**

# 5 Array manipulation functions

In this section we discuss some of the array manipulation functions that should be supported. These functions make use of expression templates so can take either arrays or array expressions as arguments.

## 5.1 Conditional operations

Table 5 lists Fortran array functions in order of how often they are called in the large Earth-system model discussed in the appendix. The two most popular are `min` and `max`, which can be thought of as

the simplest conditional operations. They take the form `max(A,B)`, where `A` and `B` are array expressions of the same rank and size (or one may be a scalar), and return an array expression containing the element-by-element maximum or minimum between `A` and `B`. These functions are useful for putting upper and/or lower bounds on a quantity, including as a way to prevent division by zero, and have the advantage that they map to machine instructions so can be vectorized with typically the same cost as the addition operator. Unfortunately, the names `min` and `max` are unsuitable for this functionality in C++ because the behaviour described above is different from the behaviour of these functions in the STL (see Recommendation 2). Consider the following:

```
std::vector<float> v = std::min(std::vector<float>{1.0,2.0},
                                std::vector<float>{2.0,1.0});
```

The result is that `v` contains {`1.0,2.0`}, not {`1.0,1.0`} as we require, because `std::min` performs a *lexicographical* comparison of the two vectors, and deems the first vector to be "less than" the second. A better name for our element-wise functions applied to floating-point arrays is `fmin` and `fmax`, because the STL versions of these functions are designed to map to machine instructions. Even this is not perfect because we would like these functions to work the same on integer arrays (vectorized machine instructions exist for integer maximum and minimum, of course).

**Recommendation 11: Provide functions that perform the element-wise minimum and maximum of two array expressions, but do not call them `min` and `max` because the STL versions of these functions perform a lexicographical rather than element-wise comparison when applied to STL containers, returning a different result. The names `fmin` and `fmax` should be used for floating-point arrays, but more general names may be preferred for arrays of other types.**

Next we consider the array equivalent of the ternary operator: selecting elements from one of two array expressions depending on a third boolean array expression. This can be achieved in various ways:

```
where (B > 0); A = B; elsewhere; A = C; endwhere // Fortran "where"
A = merge(B, C, B > 0);                          // Fortran "merge"
A = np.where(B > 0, B, C);                       // Python
A = select(B > 0, B, C);                         // Blaze
A = where(B > 0, B, C);                          // Xtensor
A = (B > 0).select(B, C);                        // Eigen
A.where(B > 0) = either_or(B, C);                // Adept
```

None of these approaches generate temporary arrays. The clearest approach is a simple function call, as in Fortran "merge", Python, Blaze and Xtensor, although there seems to be little to decide whether the function should be called `where`, `merge` or `select`.

Fewer languages and libraries provide the functionality to perform an operation on an array only for elements where some condition is met, e.g. conditional incrementing of an array:

```
where (B > 0); A = A + 1.0; endwhere  // Fortran
A.where(B > 0) += 1.0;                // Adept
```

although the other libraries can achieve the same result by including a redundant copy-to-self if the condition is not met, e.g.

```
A = where(B > 0, A+1.0, A);           // Xtensor
```

All these approaches are superior to the Matlab approach of

```
A(find(B > 0)) = A(find(B > 0)) + 1.0;
```

because the `find` function returns a temporary array of integers (twice in this case) allocated on the heap.

**Recommendation 12: Provide a function representing the array equivalent of the ternary operator, following `where` in Xtensor and Python, `select` in Blaze and (with arguments reordered) `merge` in Fortran.**

## 5.2 Reduction functions

"Reduction" functions are those that operate on a single array expression and return an array (or scalar) with a lower rank. Functions labeled with "R" in Table 5 list those available in Fortran 2003 and how often they are used in the Earth-system model considered in the appendix. Most reduction functions return an array or scalar of the same type as the array elements, e.g. `sum`, `product`/`prod`, `min`/`minval`, `max`/`maxval`, `mean` and `norm2`. Interpreted languages such as Matlab and Python also provide functions for the variance and standard deviation, but since these are simply derived from two types of `mean` it seems reasonable to omit them. A popular function in Table 5 is `count` (called `count_nonzero` in Python and Xtensor) which returns the number of `true` values in a boolean expression. Two other important functions provided by Fortran, Matlab, Python and Julia are `all` and `any`, which return a boolean array or scalar indicating whether all or any of the boolean elements of the array argument are true.

Clearly it would be desirable for all reduction functions to provide the same interface. Most languages and libraries (e.g. Fortran, Python, Julia, IDL, Blaze, Armadillo, Adept, Blitz++ and Xtensor) offer two basic modes of operation: `sum(A)` sums over the entire array and returns a scalar, while `sum(A,dim)` sums over dimension `dim` and returns an array with a rank one less than that of `A`. Two variants of the second form are worth commenting on. Python and Xtensor allow the operation to be performed on more than one dimension, e.g. `sum(A,{dim1,dim2})`, in which case the rank of the returned array is reduced by one for every dimension present. Recognizing that the dimension of reduction is almost always known by the programmer, Blaze provides it as a template parameter, e.g. `sum<dim>(A)`, which in principle could be used for more than one dimension via `sum<dim1,dim2>(A)`. Knowing the dimension(s) at compile time should lead to slightly more efficient code and a slightly smaller executable. A further mode of operation is available in Fortran and Python, whereby a boolean array can be provided as an additional argument specifying the elements that should be considered in the reduction. An example application would be to reproduce the behaviour of Matlab's `nansum`, where a summation is performed only over the non-NaN elements in an array, i.e. `sum(A,!isnan(A))`.

Two further Fortran functions `minloc` and `maxloc` provide the location of the minimum or maximum value in the array, and have slightly different behaviour when called without a dimension argument. Rather than returning a scalar they return an integer vector with the same number of elements as the rank of the array, which contains the coordinates of the minimum or maximum value in the whole array. In principle, the STL function `max_element` (see section 2) combines the functionality of Fortran's `maxval` and `maxloc`: it returns an iterator that enables both the location and the value of the maximum element to be extracted. However, the significant downside to this design is that it cannot be vectorized, even if only the value is required.

**Recommendation 13: A comprehensive set of reduction functions should be provided that present the same interface, allowing the reduction to be performed either on the entire array or along one or more dimensions (possibly with the dimensions being specified as template arguments), and allowing a boolean mask array to be provided specifying which elements to consider when performing the reduction.**

### 5.3 Linking array references to other arrays

The third most used Fortran array function in Table 5 is the => operator, which associates an array with the "pointer" attribute to another array or array subset so that they share the data they point to. It is used widely in the Earth-system model in the appendix to help manage the large number of array-containing data structures. Adept reproduces this feature by overloading the >>= operator, e.g.

```
A >>= B(__,stride(1,2,end));
```

This is valid if array A is unallocated, and associates A with the specified subset of array B such that subsequent operations on one array are seen as changes in the other.

**Recommendation 14: Include the capability to explicitly associate an array reference with another array or array subset, such that subsequent operations on one array are seen by the other. This could be done with an operator (e.g. >>=) or a more traditional member function.**

## 6 Passing array subsets efficiently to functions

A crucial requirement of a C++ array library is that we can write functions with array arguments such that the data are passed *by reference;* that is, the receiving function reads or writes the data at its original location rather than a deep copy being made. This is the behaviour of Fortran and indeed is one of the reasons why Fortran's array handling is so efficient. Section 6.1 explains some of the ways that arrays can be passed to Fortran functions and subroutines, and how even array subsets are passed with only a shallow copy being made. Surely there is nothing to stop this being done in C++: just make the function take C++ references as arguments (e.g. "`dmatrix<float>&`" or "`const dmatrix<float>&`")? This is fine if we only want to pass whole arrays to a function, but as explained in section 6.2, in most existing C++ array libraries if we try to pass a *subset of an array* then it will make a deep copy when passed to a `const dmatrix<float>&` argument and fail to compile if passed to a `dmatrix<float>&` argument. Two partially satisfactory solutions are available: that of Blitz++ and Adept described in section 6.3, and that of Eigen in 6.4. In section 6.5 I build on these two approaches to propose a better solution to this problem.

### 6.1 Array arguments in Fortran

Consider the following Fortran subroutine (equivalent to a function returning `void` in C++):

```
subroutine algorithm(A, B, C)
  real, intent(inout), allocatable :: A(:,:) ! Like dmatrix<float>&
  real, intent(inout)              :: B(:,:) ! A bit like dmatrix<float>&
  real, intent(in)                 :: C(:,:) ! A bit like const dmatrix<float>&
  ! ...subroutine content here...
end subroutine
```

It takes three arguments consisting of 2D `float` arrays: A and B are read-write and C is read-only. Only whole arrays can be passed into A: not only can their data elements be read and written, but the whole array may be allocated or deallocated with the result being visible in the calling routine. In the large codebase considered in the appendix, there are a total of 82,565 integer or floating-point array arguments to functions or subroutines; of these, 0.14% are of type A, 57% are of type B and just under 53% are of type C. Thus it is very rare for a function to be able to resize one of its array arguments.

In the case of B, the first five of the following are all valid arguments (where D is a 2D array of `float`s in the calling routine), while C will take *all* of the following:

Table 2: Summary of how different types of array arguments are treated when passed to a function accepting constant and non-constant arrays, where **S** indicates that an efficient shallow copy is performed (i.e. little more than a pointer to the data and the size and stride of each dimension are passed); **S*** indicates that a shallow copy is performed but that if a subset of a *constant* array is passed then it may be changed inside the function, ignoring `const`ness; D indicates that an inefficient deep copy is passed (the data in the object being passed is copied to a new array), and "–" indicates an unavailable feature that will fail to compile.

| Passed into function | Fortran | Fortran `contiguous` | C++ references | Blitz++ and Adept | Eigen `Ref` |
|---|---|---|---|---|---|
| *Received as non-constant array* | | | | | |
| 1. Contiguous array | **S** | **S** | **S** | **S** | **S** |
| 2. Contiguous subset | **S** | **S** | – | **S*** | **S** |
| 3. Partially contiguous subset | **S** | D | – | **S*** | **S** |
| 4. Strided subset | **S** | D | – | **S*** | – |
| 5. Irregular subset | D | D | – | – | – |
| *Received as constant array* | | | | | |
| 1. Contiguous array | **S** | **S** | **S** | **S** | **S** |
| 2. Contiguous subset | **S** | **S** | D | **S** | **S** |
| 3. Partially contiguous subset | **S** | D | D | **S** | **S** |
| 4. Strided subset | **S** | D | D | **S** | D |
| 5. Irregular subset | D | D | D | D | D |
| 6. Array expression | D | D | D | D | D |

```
D                 ! 1. Whole array, which will be contiguous if just allocated
D(:,3:7)          ! 2. Fully contiguous array subset
D(3:7,:)          ! 3. Partially contiguous array subset
D(3:7:2,3:7:2)    ! 4. Strided array subset
D(ilist,jlist)    ! 5. Irregular array subset
D + exp(D)        ! 6. Array expression, only passable to an intent(in) argument
```

Fortran uses column-major dimension ordering so if `D` is stored contiguously in memory, `D(:,3:7)` also corresponds to a contiguous section of memory. Example 3 is partially contiguous in the sense that it consists of rows of 5 contiguous elements from the original array, but adjacent rows are not contiguous. In example 4, neither dimension is contiguous but adjacent elements have a regular stride in memory. In example 5, `ilist` and `jlist` are 1D arrays of integers indicating which rows/columns to select.

As summarized in Table 2, the first four examples invoke an efficient shallow copy, thereby allowing the subroutine to read and optionally write directly to the memory where `D` is stored. This is possible because these Fortran-90 "assumed-shape" arrays store not only the starting address of the data and the array dimensions, but also the stride between adjacent elements in each dimension. Thus, the regular array subsets in examples 2–4 can actually be implemented as array objects themselves, but ones that happen to be sharing data with a larger array (see also section 4.2).

In the case of an irregular array subset (example 5 above), the indices may not be regularly spaced so it cannot be represented as an ordinary array object. Fortran still allows it to be passed as a read-write array argument, by performing a deep copy into a temporary array when the subroutine is called, and a further deep copy back to the indexed array just after the subroutine exits. While less efficient than

passing regular array subsets, it is very useful. In example 6 above, an array expression may be passed to the subroutine; since an expression is an rvalue it can only be received by `intent(in)` (equivalent to `const`) arguments. The expression is evaluated into a temporary array which is seen inside the routine.

Array operations can only be vectorized by the compiler if the data are contiguous in memory, but this is not known for array arguments at compile time, so a Fortran compiler typically generates the vectorized and non-vectorized code for an array operation and chooses between them at run-time by checking for contiguity. This is a modest overhead except for small arrays. Since Fortran 2008, subroutines and functions can optionally declare their array arguments as `contiguous`, removing the need for this run-time check. However, if a non-contiguous array or array subset is passed into a function receiving contiguous arguments, Fortran deep-copies the argument into a contiguous local array, and if it is read-write, deep-copies it back after the function exits. The second Fortran column in Table 2 indicates when shallow and deep copies are made. To aid optimization, many Fortran compilers can be configured to issue a run-time warning if a deep copy is performed in this instance.

## 6.2 How can we do this in C++?

The most obvious way to try to implement the Fortran subroutine above in C++ would be as follows:

```
void algorithm(dmatrix<float>& A,
               dmatrix<float>& B,
               const dmatrix<float>& C) { ... }
```

Argument `A`, which may be resized within the function, works fine since it is only intended to accept whole arrays. However, as shown by the "C++ references" column of Table 2, using references for `B` and `C` is much less successful than Fortran. This is because in most C++ array libraries, an array subset is implemented as a different class than than a standard array. When a "`const dmatrix<float>&`" argument receives an object representing a subset of an array, its copy constructor is called, which typically performs a deep copy. Even worse, when a `dmatrix<float>&` argument receives an object representing a subset of an array, it will fail to compile. This is because even if the array subset is convertible to a `dmatrix<float>`, the conversion will create a temporary object that by the rules of C++ cannot be matched by a non-constant reference. The user would have to manually create a temporary object, for example (using the indexing style proposed in section 4):

```
dmatrix<float> Etmp = E(__,_(2,6));
algorithm(D, Etmp, F);
E(__,_(2,6)) = Etmp;
```

It is ironic that C++ array libraries take immense pride in their use of expression templates to eliminate temporaries from array expressions, and yet cannot pass a subset of an array to a function without needing to create a temporary.

Most C++ array libraries have exactly this problem. One workaround proposed in the Eigen documentation is to write your functions using templated arguments, so that they can accept either whole arrays or array subsets. This is a poor general solution as it is completely infeasible to inline every function in a large-scale application, and any libraries it calls. Moreover, it requires the user to be familiar with complex and ugly template notation when they just want to write clear scientific code.

## 6.3 Array arguments in Blitz++ and Adept

Blitz++ and Adept achieve a shallow copy in all of the cases that Fortran does by two approaches. Firstly, they use a flexible Fortran-like array type where each dimension may be strided, as shown in section 4.2.

This approach enables regular array subsetting operations (such as examples 2–4 above) to return an `Array` object that shares its data with the original array. When passed to a "`const Array&`" argument, the temporary `Array` enters the function directly and no copying of data takes place. Secondly, and less satisfactorily, the copy constructor for their `Array` classes performs a shallow rather than a deep copy such that the copying array shares its data with the copied array. This enables array data to be modified within a function, provided that the matrix argument `B` in the previous function declaration is rendered *without* the reference qualifier (`&`):

```
typedef Array<float,2> Matrixf; // Shortcut to a Blitz++-style Array type
void algorithm(Matrixf& A, Matrixf B, const Matrixf& C) { ... }
```

When an array or an array subset is passed to argument `B`, a shallow copy is performed so that operations on `B` within the function will be seen by the calling routine. Table 2 shows that this results in arrays being passed as efficiently as Fortran in almost all cases. There are several disadvantages to this approach, the most obvious being that users do not expect copies to be shallow, so bugs are likely to arise from the following:

```
Vectorf u = {1.0, 2.0}; // Create a new vector
Vectorf v(u);           // Invoke copy constructor: shallow copy
v = {3.0, 4.0};         // Unexpected behaviour: u is also modified!
Vectorf w = u;          // Another way to invoke copy constructor
w = {5.0, 6.0};         // Unexpected behaviour: u is also modified!
```

The Blitz++ and Adept documentation simply recommend users to copy arrays using copy-assignment instead, as follows, although it is easy to forget:

```
Vectorf x; x = u;       // Deep rather than shallow copy
```

Another problem is that `const`ness is not always respected. We have already seen that an array subsetting operation in Adept and Blitz++, e.g. `D(__,range(2,6))`, returns an `Array` object that shares data with `D`. But what happens if we are within a function where `D` is a constant reference to an array (e.g. `const Matrixf&`)? Unfortunately, there is no way to specify in the constructor of an ordinary array object that the object constructed must be constant. Therefore, in order for it to be even possible to subset a constant array in Adept and Blitz++, both libraries throw away the `const`ness of the array, which means that the following (wrongly) compiles and runs:

```
void evil_function(const Matrixf& M) {
  M(__,range(2,6)) = 5.0; // This changes the contents of a constant object!
}
```

### 6.4 Array arguments in Eigen

Eigen has a working solution to the array-passing problem via its `Ref` class:

```
void algorithm(MatrixXf& A, Ref<MatrixXf> B, Ref<const MatrixXf> C) { ... }
```

where `MatrixXf` is the shortcut for Eigen's dense dynamically sized matrix. A `Ref<X>` object behaves similarly to `X`, except that its copy constructor performs a shallow copy if possible, and a deep copy only if `X` is constant. This leads to the behaviour shown in the final column of Table 2. Unlike Adept and Blitz++, it doesn't need to throw away `const`ness.

It is slightly less flexible than Fortran, Adept and Blitz++ in that if the fastest varying dimension is not known to be contiguous in memory at compile time, it will fail to compile if `X` is non-constant and perform a deep copy if `X` is constant. This is simply because of a design choice that to aid vectorization, Eigen's vectors (e.g. `VectorXd`) and the fastest varying dimension of its matrices should by default be

contiguous in memory. Fortran, Adept and Blitz++ instead implement vectorized and non-vectorized code, and switch between them according to a run-time check for contiguity. In fact, Eigen can accept non-contiguous arguments in both the non-constant and constant cases, but it is ugly:

```
void algorithm(MatrixXf& A, Ref<MatrixXf,0,Stride<Dynamic,Dynamic> > B,
               Ref<const MatrixXf,0,Stride<Dynamic,Dynamic> > C) { ... }
```

### 6.5 A proposal for array arguments in C++

As shown in the appendix, a substantial fraction of arrays in a large-scale scientific application come into existence as function arguments, and are actually a shallow copy of another array. Therefore, it seems sensible to express them as a perfectly normal type of array (as in Fortran, Adept and Blitz++), rather than something unusual as implied by the Eigen approach.

This can be achieved by adding a template argument to the array type to state whether it is a "reference" to another array (although not exactly a C++ reference), and then to define shortcuts, e.g.:

```
template <typename Type, unsigned int Rank, bool IsRef = false>  class darray;
template <typename Type>  using dmatrix      = darray<Type,2>;
template <typename Type>  using dmatrix_ref  = darray<Type,2,true>;
template <typename Type>  using dmatrix_cref = darray<const Type,2,true>;
```

The result is that functions can be declared cleanly:

```
void algorithm(dmatrix<float>& A,
               dmatrix_ref<float> B,
               dmatrix_cref<float> C) { ... }
```

The body of the `darray` class would need a few differences depending on the value of `IsRef` and the `const`ness of `Type`, mainly in the copy constructor: considering the shortcuts above, obviously the copy constructor of a `dmatrix` will always perform a deep copy. For a `dmatrix_cref`, a shallow copy will be performed if possible, but if the array is constructed from an array expression (e.g. if an array expression were passed as an argument to `C` above) then a deep copy will be performed. Finally, the copy constructor of `dmatrix_ref` should only compile if a shallow copy can be made, as it should not be possible to pass an array expression to `B` above. There may be other differences in the behaviour of an `darray` depending on the value of `IsRef`, such as preventing an array reference from being resized.

This solution also solves the problem of `const`ness being ignored in section 6.3: a subsetting operation on a constant `dmatrix<float>` returns a `dmatrix_cref<float>` object, i.e. its `const`ness is conveyed through the scalar type being `const float`. It is worth noting that the need for `dmatrix_ref`- and `dmatrix_cref`-like objects as the return type of a subsetting operation was recognized early in the development of ideas for C++ array libraries by Barton and Nackman (1994).

**Recommendation 15: To enable arrays and regular array subsets to be passed into functions with only a shallow copy being performed, three variants of the main array class should be used (differentiated by template arguments). An ordinary array is responsible for the destruction of its own data, and its copy constructor performs a deep copy. An *array reference*, which may be constant or non-constant, usually points to external data and its copy constructor performs a shallow copy. Constant array references should be capable of owning their own data when initialized from an array expression, in order that an array expression can be passed as an argument to a function.**

The final challenge in this section, apparent in Table 2, is that only Fortran allows an *irregular* array subset to be passed to a function such that it can be modified within the function, while none of the C++ libraries do. This operation is not particularly efficient, since a deep copy must be performed both when the function is entered and exited, but it is certainly convenient. Can we allow the `dmatrix_ref` object

to be constructed from an type representing an irregular array subset, but in this instance to somehow keep a reference or pointer to the indexed array so that when its destructor is called, it copies the data back? This is challenging because the irregular array subset is likely to be a complex templated type depending on how each dimension has been indexed, and the basic `Array` type has no way to store this type, which it would need to copy back correctly. A possible solution is for the user to have to wrap the indexed argument as follows:

```
algorithm(D, copy_back(E(ilist,jlist)), F);
```

The templated `copy_back(X)` function returns a `CopyBack` object containing a reference to `X` and a `dmatrix_ref` object that is either a deep or shallow copy of `X`. When `algorithm` is called, the `dmatrix_ref` constructor of its second argument makes a shallow copy of the `dmatrix_ref` inside `CopyBack`. Finally, when `algorithm` exits, the `CopyBack` object is destructed at which point it makes a deep copy back to `X`. An additional advantage of this approach is that by having to use `copy_back`, the user has to be explicit that they want to do something potentially inefficient.

**Recommendation 16: Enable an irregular subset of an array to be passed as an lvalue to a function by the use of a `copy_back` function that manages the conversion to and from a contiguous array.**

# 7 Features and designs to avoid

## 7.1 Reference counting

The previous section described the problem of how to pass modifiable arrays as arguments to a function, but we also need to give some thought to the treatment of functions whose return value is an array. Since the returned array will be destructed as soon as it is copied, we cannot perform a shallow copy, but a deep copy is inefficient. The introduction of Move Semantics in C++11 provides a straightforward solution to this problem, as we can write a copy constructor and copy assignment operator specifically for temporary objects that "steal" the data about to be destructed rather than performing a deep copy.

Blitz++ was written using the C++98 standard, before Move Semantics were introduced, and for the reasons explained in section 6.3, the copy constructor of Blitz++ arrays always performs a shallow copy. Therefore Blitz++ faced the problem of how to use arrays as return values from functions. The solution was to use *reference counting*: the raw data of an array are stored in a separate `Storage` object that keeps a count of how many array objects are using it. Each array subsetting operation creates a new array object pointing to the same data, and the reference count is incremented; when an array goes out of scope the reference count is decremented and if it falls to zero the data are deallocated. Adept is written to be backwards compatible with the C++98 standard, and also uses reference counting.

The problem with this approach comes to light in a multi-threaded environment: a common pattern is to spawn threads that work simultaneously on different parts of a large array. Each time a thread subsets the array, the reference count of its `Storage` object is modified, even if only read access is required. This soon leads to the reference counter becoming corrupted. Both Blitz++ and Adept may be compiled with a preprocessor option to specify that reference counters are to be protected by a mutex, although this slightly slows down slicing operations on all arrays, not just those accessed by multiple threads. Adept provides a more targeted solution: `M.soft_link()` returns an array pointing to the data in `M` but without touching its reference counter. This array is valid until `M` is resized or destructed. While effective, this solution does lead to cluttered code and is not practical when large sections of code are parallelized.

With the availability of Move Semantics in C++11, and the use of array references outlined in

section 6.5, it does appear that reference counting is not needed: even if slicing operations mean that many arrays point to the same data, only one of them has the responsibility to deallocate memory when it is destructed. However, this puts more responsibility on the programmer to ensure that the lifetime of an array reference does not exceed the lifetime of the array whose data it points to, especially with the capability to explicitly link array references to other arrays (see section 5.3). Consider the following:

```
// Case 1
dvector_ref<float> Aref;
{
  dvector<float> A = {1.0, 2.0};     // A owns its data
  Aref >>= A;                        // Shallow copy: Aref points to A's data
}
// Aref is now invalid because A has gone out of scope!

// Case 2
dvector_ref<float> Bref;
dvector<float> B = random_vec(10);  // Move constructor: ownership moved to B
Bref >>= B;                          // Shallow copy: Bref points to B's data
B = random_vec(10);                  // Move assignment: what should happen?
// Is Bref valid here?
```

In case 1, any use of `Aref` after the final brace is clearly a mistake by the programmer because the data it points to was deallocated when `A` went out of scope. Case 2 is more subtle. The first call of `random_vec` returns a vector about to be destructed, triggering `B`'s move constructor which can safely steal ownership of the vector without performing a deep copy. The vector returned from the second call to `random_vec` is received by the move assignment operator. Most efficient would be to throw away `B`'s existing data and steal ownership of the new vector, but this invalidates `Bref`. Adept simply uses the reference count to check whether any other objects are referencing `B` and if they are then a safe but more expensive deep copy is performed. Without reference counting, the safest behaviour is for the move assignment operator to assume that any data it currently owns is being shared, and perform a deep copy.

**Recommendation 17: To avoid problems of thread-safety, do not use reference counting for array data. With the introduction of Move Semantics in C++11, and the use of array references described in section 6.5, the need for reference counting has gone away, although for safety an array's move assignment operator should assume that any existing data might be shared with another array.**

## 7.2 Extreme flexibility in array types and features

The plethora of C++ array libraries has meant that many interesting features and design possibilities have been explored, but in designing a common standard, we should strive to keep it simple where possible. The following are examples of array types I do not believe it necessary to include:

- *Arrays with dynamical rank.* For maximum compatibility with Python/NumPy, Xtensor supports an array type whose number of dimensions can change at run-time. As discussed in the Xtensor documentation, this flexibility comes at considerable computational cost compared to arrays that know their dimensionality at compile time. There are very few situations when a programmer does not know the rank of the arrays they are working with, so this does not seem to be an important feature.

- *Arrays with mixtures of static and dynamic dimensions.* Most existing C++ libraries offer two basic array types, one with dynamic dimension sizes whose data are stored on the heap, and the other

with fixed dimension sizes (specified by template parameters) that are stored on the stack. Eigen matrices can have one dimension with a static size and the other with a dynamic size, which in principle aids vectorization if compiler knows the size of the fastest-varying dimension. However, it would be cumbersome to add this flexibility to arrays with an arbitrary number of dimensions, and the demand for this feature is likely small enough that it need not be supported. Blaze offers a hybrid matrix type that is stored on the stack and its maximum dimension sizes are provided as template arguments, but which can be resized to a smaller size. Again, the demand for such a feature will be small.

**Recommendation 18: Exclude array types that significantly increase complexity and yet are likely to be used only rarely, such as arrays with dynamical rank and arrays with mixtures of static and dynamic dimensions.**

Normally each term in an array expression must match in rank and in the size of each dimension; agreement in rank is checked at compile time and agreement in dimension is optionally checked at run time. The exception is for scalars which are implicitly treated as if they match the rank and size of any array they interact with. Xtensor goes a step further: if the rank of two arrays in an operation does not match it will try to *broadcast* one of the dimensions of the smaller array to match the rank of the larger. Furthermore, if a dimension is found not to match between the two arrays at run time then if one of them is of length 1, it will be broadcast to match the dimension of the other array. This behaviour matches that of Python/NumPy, where it increases efficiency because the looping is performed in a fast C kernel rather than in the slower interpreted language, although these benefits are unlikely to be realized in a compiled language like C++. However, the most common cause of disagreement between array rank or size is programmer error, and it is important for debugging that these disagreements are reported at compile-time or run-time. It is my opinion that if the programmer wants an array to be broadcast, they should be explicit about it. For example, Julia introduces an explicit broadcast function: `broadcast(+, A, B)` applies the addition operator to `A` and `B`, with broadcasting of singleton dimensions to ensure the dimensions match. In Adept the function `spread<d>(M,n)` replicates array `M` along new dimension `d` a total of `n` times, but by using expression templates no temporary array is created. This is similar to the Fortran function of the same name. Extra flexibility could be added by allowing multiple dimensions to be specified as template arguments, and optionally allowing the `n` argument to be omitted, in which case the size of the new dimension(s) would automatically match that of the array it was combined with.

**Recommendation 19: Do not support implicit broadcasting, but allow users to explicitly broadcast an array along a new array dimension with a `broadcast` or `spread` function that takes one or more template arguments specifying the new dimensions.**

# 8 Interoperability of array libraries

## 8.1 Data layout of array classes

It is envisaged that there could be a number of implementations of a standards-conforming array library: in addition to a reference implementation and (eventually) vendor optimized ones, it may be possible for existing libraries to add standards-conforming array types derived from their existing expression template framework, which sit alongside their existing arrays. It may be difficult to convert a library from supporting only two dimensions to supporting an infinite number, but a partial implementation of the interface would still be valuable.

So far this paper has focused on the interface to the library needed for API compatibility. However,

it would be desirable to achieve ABI compatibility, i.e. to be able to link together pre-compiled and sometimes proprietary packages that pass array between them, even if they had been compiled with different implementations of the array library (although using the same compiler). This requires the data layout and template arguments of array types to be specified in the standard. The following is an attempt to define a "straw man" for the layout of a dynamically sized array; the actual names of data members and the class itself are simply placeholders:

```
template <typename Type, unsigned int Rank, unsigned int Options = 0,
          class Allocator = std::allocator>
class darray {
 public:
  // Unpack the booleans encoded in Options as follows:
  static const bool            is_ref            = (Options & (1 << 0));
  static const bool            is_row_contiguous  = (Options & (1 << 1));
  static const bool            is_all_contiguous  = (Options & (1 << 2));
  // ...
 protected:
  Type*                        data_;      // Pointer to first element
  std::size_t                  capacity_;  // Number of elements allocated
  std::array<std::size_t,Rank> dimension_; // Length of each dimension
  std::array<std::size_t,Rank> stride_;    // Memory stride of each dimension
  Allocator                    allocator_; // Object to allocate memory
};
```

Remarks:

- Arrays in existing libraries often use boolean template arguments to specify additional properties, such as whether the storage is to be row-major or column-major (Blaze) or whether an array is to be automatically differentiated (Adept). Here, we have followed the approach of Eigen and used a bitfield `Options` template argument, which keeps the template argument list short (making compiler errors less cryptic) while allowing for future extensibility without breaking binary compatibility for the basic array configurations. In this example, the first bit specifies whether the array is a "reference" as described in section 6.5, which controls aspects of the array behaviour such as whether the copy constructor performs a deep or shallow copy. The next two bits describe whether individual rows, or the entire array, is guaranteed to be contiguous in memory, which means that run-time checks on these properties can be avoided when deciding whether an array operation can be vectorized. The other bits are available for other properties.

- The array dimensions and the stride in memory of each dimension are held as `std::array` objects, which are convenient alternatives to plain C arrays of fixed length. As explained in section 4.2, this flexible design enables an array to use either row-major or column-major storage and to act as a strided subset of another array.

- For maximum compatibility with the STL, we follow the practice of containers such as `std::vector` and use the final template argument to specify a class that will be used to allocate and deallocate data, and hold an instance of this class within an array object. Holding an instance is needed in case the allocator contains state information, although the default `std::allocator` class is stateless. Unfortunately it means that if the array class is to be ABI compatible, the allocator class must be too. Note that `std::allocator` does not guarantee to align data to aid vectorization, so might not be the appropriate allocator to use by default. One requirement of a conforming allocator class is that the function call to deallocate data also needs to be provided with

the number of elements to free, and hence we need to store `capacity_` within the array. For an array reference that does not own its data, this would be zero, providing a simple run-time check on whether an array is responsible for the deallocation of its data.

My proposed structure for a fixed-size array is as follows:

```
// Note that Options cannot have a default argument because the dimensions
// must come last
template <typename Type, unsigned int Options, std::size_t... Dim>
class farray_base {
 public:
  // The rank of the array is the total number of template arguments in Dim,
  // and the total size is the product of the array dimensions (computed at
  // compile time using some meta-programming)
  static const unsigned int  rank        = sizeof...(Dim);
  static const std::size_t   total_size  = internal::product<Dim...>::value;
  // ...
 protected:
  std::array<Type, total_size> data_; // Data stored contiguously on the stack

};
// The most basic fixed-size array has all the bits in Options set to false
template <typename Type, std::size_t... Dim>
using farray = farray_base<Type, 0, Dim...>
```

For maximum flexibility, the base class also contains an `Options` template argument which could be used to specify row-major or column-major orientation. As with the dynamic array, it could be used by an automatic differentiation library such as Adept to indicate whether an array is to be differentiated while allowing non-differentiated arrays to be ABI compatible.

## 8.2 Fortran interoperability

Modern Fortran passes its "assumed-shape" arrays (the most similar to the dynamically sized C++ arrays considered in this document) to and from functions in the form of a pointer to an array descriptor. The 2018 Fortran standard provides a means to pass assumed-shape arrays to and from C by requiring implementations to provide a C header file `ISO_Fortran_binding.h` containing a definition of the array descriptor in the form of a C structure "`CFI_cdesc_t`" (Reid, 2018). In gfortran version 9.3, the structure is as follows:

```
// Information held per dimension
struct CFI_dim_t {
  ptrdiff_t lower_bound; // Normally zero when passed into C
  ptrdiff_t extent;      // Dimension length
  ptrdiff_t sm;          // Stride in memory (in bytes)
};
// The full array descriptor
struct CFI_cdesc_t {
  void*      base_addr;  // Address of first element of array
  size_t     elem_len;   // Size in bytes of an array element
  int        version;    // Version number of the array descriptor format
  int8_t     rank;       // Number of array dimensions
  int8_t     attribute;  // Is the array allocatable, a pointer, or neither
  int16_t    type;       // Code indicating the element type
  CFI_dim_t dim[];       // Information about each dimension (see above)
};
```

We can see that the basic array structure information is the same as in the C++ class proposed in section 8.1, with a stride specified for each dimension. There should therefore be no difficulty in passing arrays to and from Fortran, although note that at the time of writing, support for the 2018 standard is patchy amongst Fortran compilers.

**Recommendation 20: Provide a simple interface for passing arrays to and from Fortran making use of the interoperability features of the Fortran 2018 standard.**

# 9 Linear algebra and special matrix types

## 9.1 Two libraries?

There is a potentially huge list of linear-algebra operations that one would like an array library to support; the most common include matrix inversion, solving systems of linear equations, eigendecomposition and singular value decomposition, but the LAPACK routines number in the thousands. And what about other algorithms such as the Fast Fourier Transform? Very soon we have a do-everything "God" library to rival Matlab. How much of the possible numerical array functionality should realistically be included in a standard, and should it be divided into separate libraries?

The previous sections of this paper have concerned the behaviour of the array classes themselves, or considered functions such as `sin` and `sum` that operate on array expressions and are implemented in terms of the expression templates that lie deep in the implementation of the library. Such functionality must therefore be part of the core array library. However, it was demonstrated by Iglberger et al. (2012) that expensive array operations such as dense matrix multiplication are best not implemented in terms of expression templates. Consider the following, where all arguments are large dense $N \times N$ matrices:

```
A = B ** (C*D); // * = element-wise multiplication, ** = matrix multiplication
```

In a classical 3-loop implementation of matrix multiplication, the computational cost is $O(N^3)$ and each element of the two input arguments to the matrix multiplication is read $N$ times. This is why it is inefficient to use expression templates in this case: each element of `C*D` would be computed $N$ times. It makes much more sense to evaluate `C*D` into a temporary matrix first, the overhead of allocating the memory likely being smaller than the $N-1$ unnecessary evaluations of `C*D`. The same argument applies to other expensive array operations such as solving linear systems of equations and the Fast Fourier Transform.

This means that linear algebra functions and indeed matrix multiplication can be implemented without knowing any details about the expression-template implementation. Consider a function implementing the inverse of a dense matrix of `doubles`. Its function declaration could be simply

```
dmatrix<double> inv(dmatrix_cref<double> input);
```

where as proposed in section 6.5, `dmatrix_cref` is an array reference with constant elements. The operation `A=inv(B)`, where `B` is an ordinary array, would lead to the `dmatrix_cref` copy constructor performing a cheap shallow copy such that `input` shares its data with `B` until the function returns. The operation `A=inv(B*C)` would lead to the `dmatrix_cref` copy constructor storing the result of `B*C` in `input` and deallocating it on return.

Consider next a function implementing the solution to a linear system of equations. This is slightly more involved because we want to cope with the case when the two arguments have different types:

```
// Implementations of solve for specific types
dmatrix<float>  solve_body(dmatrix_cref<float>  A, dvector_cref<float>  b);
dmatrix<double> solve_body(dmatrix_cref<double> A, dvector_cref<double> b);
```

```
// ... also include versions for complex arguments

// "solve" function that accepts either array or array-expression arguments,
// promoting the type if necessary, and evaluating array expressions into a
// dmatrix_cref object if necessary
template <typename LType, class L, typename RType, class R>
typename std::enable_if_t<L::rank==2 && R::rank==1,
                          dmatrix<typename promote_t<LType,RType> >
solve(const expression<LType,L>& left, const expression<RType,R>& right) {
  typedef typename promote_t<LType,RType> PType;
  return solve_body(dmatrix_cref<PType>(left.cast()),
                    dvector_cref<PType>(right.cast()));
}
```

In this example, the templated `solve` function accepts two arguments derived from the `expression` template (which includes arrays) provided that the first has a rank of 2 and the second a rank of 1. The `promote_t<LType,RType>` trait is required to return the larger of the two types, and to promote a real type to complex if the other is complex. The `solve` function constructs `dmatrix_cref` and `dvector_cref` objects from the expression arguments, where `cast()` casts the expression back to its original type (an example of the Curiously Recurring Template Pattern). If it is an array then a shallow copy is performed, otherwise the expression is evaluated and stored in the object. These objects are then passed to the relevant `solve_body` function.

In the first example above, the `inv` function needs absolutely no knowledge of how expression templates are implemented. In the second example, the `solve` function needs to know only that expression objects are called `expression`, they take two template arguments, and they implement `rank` and `cast()`. Thus we can clearly delineate the boundary between two libraries:

1. The *array* library defines multi-dimensional array containers and all the basic array operations that can be applied on them via expression templates. Such a library could be implemented using header files only. The standard for the interface to the array library would be tightly specified, but would still allow for innovation in the development of specific implementations, for example optimized for a particular hardware, or with specialist features such as automatic differentiation.

2. The *linear algebra* library contains more expensive operations that are interfaced with 1D and 2D arrays defined by the first library. It may link to externally compiled libraries, such as LAPACK. The potentially endless number of possible functions, and the fact that they are individually called much less frequently in user code than those of the array library, means that it may be sufficient to simply provide guidelines for the design of the interface.

An nice feature of this separation is that any implementation of the array library could be used with any implementation of the linear algebra library; thus one could choose the first with the fastest vectorization of array expressions, and the second with the best choice of algorithms. This separation is in the spirit of the STL where containers are separated from the generic algorithms that can be run on them.

**Recommendation 21: Two libraries should be considered: the array library covers the definition of multi-dimensional containers and the basic functionality implemented in terms of expression templates, and the linear algebra library covers generally much more expensive functions implemented in terms of array types defined in the first library.**

It is not clear to me to which of the two libraries matrix multiplication belongs, or indeed whether it deserves to be treated as a library separate from the other two. The interface recommended in section 3.2 is very simple, consisting of the `**` pseudo-operator and perhaps two functions `inner_product` and

Table 3: The special matrix types supported by six of the C++ libraries listed in Table 1 (the others do not support special matrix types). "**TD**" indicates that the matrix exists as an independent type with dense storage, while "**TC**" indicates that the it exists as an independent type with compressed storage (i.e. memory is not used to store zeros or elements known to be repeated elsewhere in the matrix). "F" indicates that linear-algebra functions exist that are specialized for these types of matrix, but that they do not exist as an independent matrix type; rather an ordinary dense matrix is used to store the data, but its special property is communicated to the linear-algebra function by another means.

| | Symmetric | Hermitian | Triangular | Diagonal | Banded | Sparse |
|---|---|---|---|---|---|---|
| uBLAS | **TC** | **TC** | **TC** | **TC** | **TC** | **TC** |
| MTL4 | **TD** | | **TD** | **TC** | **TC** | **TC** |
| Blaze | **TD** | **TD** | **TD** | **TD** | | |
| Eigen | F | F | F | **TC** | | **TC** |
| Armadillo | F | F | F | F | F | **TC** |
| Adept | **TD** | | **TD** | **TC** | **TC** | |

`outer_product`. But there is still considerable scope for innovation in implementation. For example, the efficiency of multiplying a chain of non-square matrices (e.g. `A = B ** C ** D ** E`) depends on the order in which it is performed, and the default left-to-right associativity is often not optimal. Armadillo implements a simple version of the matrix chain algorithm of Barthels et al. (2018) to try to choose the best ordering at run-time based on the matrix dimensions, but there is undoubtedly scope for improvement. The Blaze library adapts its behaviour to the size of the problem, multiplying small matrices inline and only calling BLAS functions for larger matrices.

## 9.2 Special array types

The detailed specification for a linear array library is beyond the scope of this paper, but a quick look at the different interfaces provided by the existing C++ libraries is instructive. Consider the problem of solving a linear system of equations $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an non-symmetric dense square matrix:

```
x = A.colPivHouseholderQr().solve(b);                       // Eigen
x = solve(A, b);                                            // Adept
x = solve(A, b, solve_opts::fast + solve_opts::no_approx); // Armadillo
```

At one extreme, Eigen requires the user to be very explicit about what sort of decomposition method they want to use, while at the other Adept offers the user no control over the method (always calling LAPACK's `gesv` routine). The best interface would appear to be Armadillo's: users who don't know or don't care would call it as in Adept, trusting that the default algorithm is a good compromise between speed and accuracy for general matrices. Users who want more control can add a third argument specifying the algorithm or other preferences. Naturally, it should also be possible to call the individual decompositions such as QR and LU as well.

The other piece of information to help the library choose the best algorithm is whether the matrix has a particular structure that can be exploited. Six of the existing C++ libraries considered in this paper

do this by providing additional matrix types that enforce a particular structure, as summarized in Table 3. This is convenient, although element-wise access is slower due to the checks that need to be performed to ensure the structure is preserved. It also enables compressed storage to be used so that memory is not wasted on elements that will always be zero, and makes the interface easy to BLAS and LAPACK that provide functions optimized for diagonal and banded matrices with compressed storage. Eigen does not provide specialized types for symmetric, Hermitian or triangular matrices, but rather provides specialized "view" member functions for ordinary dense matrices; e.g. `M.triangularView<Lower>()` informs a function at compile time that it should treat `M` as a lower-triangular matrix. Armadillo has several mechanisms to inform functions of matrix structure, the least satisfactory being a run-time check of each element of a matrix to see if a particular structure is used.

The most powerful and memory efficient solution would seem to be to provide additional matrix types, which need to be in the array rather than the linear-algebra library because they would be implemented in terms of expression templates. Davidson and Steagall (2019) argued that square matrices should not be specializations of a general multi-dimensional array class (a tensor) on the basis that a square is not strictly Liskov substitutable for rectangle. It is certainly true that a matrices with a special structure will lead to different behaviour if substituted in the code for a rectangle, so should not derive from a general 2D array. For a general square matrix the point seems rather pedantic, and indeed all the C++ libraries in Table 1 that implement matrix inversion accept a general 2D array and perform a run-time check that it is square. It would not be difficult to add a square matrix class that enforces squareness alongside the others listed in 3, and use it as the argument type for functions that only accept square matrices. However, this would likely be invisible to the user since if they passed an array expression or a general 2D array to the function, it would need to be automatically converted to a square matrix (with a shallow copy if possible), at which time a run-time check on squareness would be performed.

**Recommendation 22: The array library should provide additional array types for square matrices with a specific structure (symmetric, Hermitian, triangular, diagonal and banded), as well as for arbitrary rank arrays with sparse storage. These can then be used by linear-algebra functions (as well as matrix multiplication) to select the most efficient algorithm. As with ordinary arrays, the equivalent constant and non-constant array reference types should be provided for use in function arguments.**

## 10 Debugging capabilities

To facilitate debugging we need to provide the user with fine-grained control over the checks performed at run-time; during development most of these would be turned on, but then most would be turned off for speed during operational use. If run-time errors are found then an exception should be thrown, reporting as much information about the error as possible. The possible run-time checks are:

1. *Array dimensions.* This checks that the dimensions match in array expressions (note that we can check that the *rank* of arrays matches at compile time). For expressions involving large arrays the cost is low since the check is performed once per array statement. In the gfortran compiler this check is turned on with `-fcheck=bounds`.

2. *Array bounds.* This checks that when individual elements of an array are extracted, or when a subsetting operation is performed, the requested element or subset lies within the array bounds. Bounds checking when individual elements are accessed can add a large overhead. In gfortran it is also turned on with `-fcheck=bounds`.

3. *Use of uninitialized values.* This is checked by instructing arrays to initialize themselves with signaling NaNs, and turning on trapping of the appropriate floating-point exceptions. In gfortran this is achieved with `-finit-real=snan -ffpe-trap=invalid,zero,overflow`, although in C and C++, trapping of floating-point exceptions is typically activated by a function call within the code, e.g. to `feenableexcept`.

We need to decide which checks to perform by default and how to turn them on or off. The STL provides little helpful guidance on this matter: its solution to the bounds-checking question is for the user to use `at()` if bounds checking is required and `operator[]` if it is not. This makes it impossible to easily switch bounds checking on or off for the whole code. On the basis of computational cost it seems appropriate that item 1 in the list above is on by default but the other two are off. Regarding how to turn these checks on or off, the easiest is via preprocessor variables that can be set on the compiler command line. Unfortunately this falls foul of the popular mantra that "the preprocessor is evil", and so seems incompatible with Recommendations 1 and 2 in section 2, but it is difficult to see an alternative that gives the user the necessary control.

**Recommendation 23: To facilitate debugging, allow the user to control run-time checking of array dimension agreement (on by default) and array bounds (off by default), as well as the ability to automatically initialize arrays to NaN (off by default). The easiest way to control this behaviour would be through preprocessor variables that affect all arrays.**

A further check that should be considered is aliasing, illustrated by the following:

```
v = w;                  // 1. No aliasing if v and w own their data
v = pow(v,1.5);         // 2. Aliasing, but safe
v = v(_(last,0,-1));  // 3. Aliasing, and the wrong result is likely
v(_(0,last-1,2)) = v(_(1,last,2)); // 4. Safe, but difficult to detect
```

If no alias checking is performed, case 3 will produce an unexpected result because the first half of `v` will be modified before it is read. Before the invention of expression templates this would not happen because the right-hand side would be evaluated into a temporary array first. Different languages and libraries take a different attitude to whose responsibility it is to detect aliasing.

- Fortran compilers do compile-time checks on whether the same symbol name (or one known to alias it) is present on both sides of an assignment, and if so will resolve the right-hand side into a temporary. But arguments passed into a function are assumed not to alias, and no run-time check is performed on their addresses. It is therefore the programmer's responsibility.
- uBlas and Xtensor take a premature pessimization approach, always evaluating the right-hand side into a temporary in case there is aliasing. This can be turned off by wrapping the left-hand side in the `noalias()` function.
- Adept checks whether the memory range accessed in the left-hand argument overlaps with the memory range of any of the right-hand arguments and if so uses a temporary. Unfortunately it is not smart enough to detect that case 2 is safe. The programmer can wrap the `noalias()` function around all or part of the right-hand side of a statement to turn off alias checking for that part. Note that Adept does not perform alias checking if the array on the left-hand size is of fixed-size, because such arrays are much less likely to be aliased, and the small size of the arrays means that such checks have a relatively higher computational overhead.
- Eigen performs run-time checks and is smart enough to detect case 2, but if aliasing is detected at run-time it throws an exception rather than fixing the problem. It is then the programmer's job to add an `eval()` call to all or part of the right-hand side, which explicitly creates a temporary array.

- Blaze can also detect case 2, but quietly fixes the problem by creating a temporary. This seems to be the best behaviour. It does still offer the `noalias()` function for turning alias checking off.

To my knowledge, none of the array libraries can detect that case 4 (assigning the even-indexed elements to the odd-indexed elements of an array) is safe, so would assume aliasing is taking place and create a temporary.

**Recommendation 24: Perform run-time alias checking of array expressions with a dynamic (but not fixed-size) array on the left-hand side, if necessary creating a temporary to avoid incorrect results. Functions `noalias()` and `eval()` should be available to give the user finer control, as well as the ability to compile the entire code with alias checking turned off. The rules of alias checking should be clearly defined so that the programmer can work out when `noalias()` and `eval()` are necessary.**

Relevant to the questions both of dimension and alias checking, what should happen in the following case, when `v` is not empty?

```
v = v(_(5,10));
```

Libraries such as MTL4, Blitz++ and Adept would throw an exception, as they detect that the left- and right-hand sides have different dimensions and do not allow implicit resizing of non-empty arrays. Eigen and Xtensor, however, follow the behaviour of Matlab and implicitly resize (i.e. free and reallocate) the left-hand side if dimensions do not match. This silently invalidates any array references pointing to the original data, which is a problem in the case above because the right-hand side then points to invalid data. A further problem with implicit resizing is that some bugs are not detected. Basically if a programmer wants an array to be resized they should be explicit about it.

**Recommendation 25: An exception should be thrown if a non-empty array is assigned to an array expression of a different size, rather than implicitly resizing the array.**

## 11 Summary and recommendations

To say that it is long overdue for the C++ Standard Template Library to support multi-dimensional arrays suitable for scientific computing would be an understatement. In the 25 years since the pioneering Blitz++ library was written, the core array capabilities of other languages have leaped forward while C++ users have had to roll their own, leading to a profusion of innovative but incompatible libraries. This is frustrating when the underlying power of the language means that a C++ array capability should be *better* than the offerings of other languages in terms of functionality, efficiency and elegance. The problem is that designing such a standard is hard: to support all the features users need requires an overarching vision of a large domain of functionality, rather than a focus on smaller components that has served the STL well in other domains.

In contributing to a discussion that will hopefully lead to the development of a standard, this paper has targeted areas that appear to have received less attention previously: understanding the *requirements* of an array library through analyzing patterns of array use in existing large-scale scientific software, a detailed *comparison* of the provisions of existing C++ array libraries and other languages, and a focus more on the *user interface* than the implemention. The recommendations made through the paper are now recapitulated. Some reflect my own personal sense of aesthetics in interface design, but even if you disagree with them you will hopefully find the paper useful in shining a light on the numerous issues that need to be considered when designing an array library.

1. Follow widely-accepted guidance on good C++ practice, e.g. copy constructors of ordinary arrays

perform a deep copy, `constness` is respected, read-only array operations are thread-safe, and non-member non-friend functions are used for mathematical operations that do not modify an array. Several instances were noted where existing libraries do not follow this guidance.

2. Follow the style of the Standard Template Library where appropriate, e.g. ensuring that 1D arrays behave as a fully STL-compatible container, lower-case names and using a single flat namespace.

3. Keep the interface as simple as possible. Due to its use of iterators, the limited array manipulation functionality available in more recent versions of the STL is typically much less readable and concise than other languages used in scientific computing; this style should *not* be emulated.

4. Use the variadic template feature of C++11 to provide full support for any number of array dimensions (demonstrated in Xtensor and ra-ra). A maximum of two or three dimensions, as in many existing libraries focusing on linear-algebra, is definitely not enough for large-scale scientific applications.

5. Adopt "array-style" behaviour in which vectors have no intrinsic orientation but (as in many other languages) behave as row vectors in when used in vector-matrix multiplication and column vectors when used in matrix-vector multiplication.

6. The `*` operator should be used to express element-wise multiplication and the `**` pseudo-operator to express matrix multiplication.

7. As a library for the C++ language, it should by default follow expected C++ behaviour such as row-major array ordering and zero-based indexing, but supporting column-major order as an option for some arrays (possibly with reduced efficiency).

8. Where possible, use indexing arguments named purely with underscores to achieve the same clarity and conciseness as other languages, so that for example `M(__,_(1,last,2))` extracts every other column of matrix `M`. This improves readability when array subsets are used in complex mathematical expressions. Do not support member functions specific to matrices that do not make sense in arrays of higher dimension (e.g. `topLeftCorner`).

9. Support C-style array indexing, such that `M[i]` returns row *i* of matrix `M` and `M[i][j]` returns a single element. Support other useful member functions that return lvalue subsets and views of arrays, such as `diagonal`, `permute` and `reshape`.

10. The basic array type should support arbitrary strides for all dimensions, enabling all regular subsetting operations to return an *array reference*, i.e. an array that is configured to point to data owned by another object. Irregular subsetting operations return an slower wrapper class.

11. Provide functions that perform the element-wise minimum and maximum of two array expressions, but do not call them `min` and `max` because the STL versions of these functions perform a lexicographical rather than element-wise comparison when applied to STL containers, returning a different result. The names `fmin` and `fmax` should be used for floating-point arrays, but more general names may be preferred for arrays of other types.

12. Provide a function representing the array equivalent of the ternary operator, following `where` in Xtensor and Python, `select` in Blaze and (with arguments reordered) `merge` in Fortran.

13. A comprehensive set of reduction functions should be provided that present the same interface, allowing the reduction to be performed either on the entire array or along one or more dimensions (possibly with the dimensions being specified as template arguments), and allowing a boolean mask array to be provided specifying which elements to consider when performing the reduction.

14. Include the capability to explicitly associate an array reference with another array or array subset, such that subsequent operations on one array are seen by the other. This could be done with an operator (e.g. `>>=`) or a more traditional member function.

15. To enable arrays and regular array subsets to be passed into functions with only a shallow copy being performed, three variants of the main array class should be used (differentiated by template arguments). An ordinary array is responsible for the destruction of its own data, and its copy constructor performs a deep copy. An array reference, which may be constant or non-constant, usually points to external data and its copy constructor performs a shallow copy. Constant array references should be capable of owning their own data when initialized from an array expression, in order that an array expression can be passed as an argument to a function.

16. Enable an irregular subset of an array to be passed as an lvalue to a function by the use of a `copy_back` function that manages the conversion to and from a contiguous array.

17. To avoid problems of thread-safety, do not use reference counting for array data (as in Blitz++ and Adept). With the introduction of Move Semantics in C++11, and the use of array references described in section 6.5, the need for reference counting has gone away, although for safety an array's move assignment operator should assume that any existing data might be shared with another array.

18. Exclude array types that significantly increase complexity and yet are likely to be used only rarely, such as arrays with dynamical rank and arrays with mixtures of static and dynamic dimensions.

19. Do not support implicit broadcasting, but allow users to explicitly broadcast an array along a new array dimension with a `broadcast` or `spread` function that takes one or more template arguments specifying the new dimensions.

20. Provide a simple interface for passing arrays to and from Fortran making use of the interoperability features of the Fortran 2018 standard.

21. Two libraries should be considered: the array library covers the definition of multi-dimensional containers and the basic functionality implemented in terms of expression templates, and the linear algebra library covers generally much more expensive functions implemented in terms of array types defined in the first library.

22. The array library should provide additional array types for square matrices with a specific structure (symmetric, Hermitian, triangular, diagonal and banded), as well as for arbitrary rank arrays with sparse storage. This can then be used by linear-algebra functions (as well as matrix multiplication) to select the most efficient algorithm. As with ordinary arrays, the equivalent constant and non-constant array reference types should be provided for use in function arguments.

23. To facilitate debugging, allow the user to control run-time checking of array dimension agreement (on by default) and array bounds (off by default), as well as the ability to automatically initialize arrays to NaN (off by default). The easiest way to control this behaviour would be through preprocessor variables that affect all arrays.

24. Perform run-time alias checking of array expressions with a dynamic (but not fixed-size) array on the left-hand side, if necessary creating a temporary to avoid incorrect results. Functions `noalias()` and `eval()` should be available to give the user finer control, as well as the ability to compile the entire code with alias checking turned off. The rules of alias checking should be clearly defined so that the programmer can work out when `noalias()` and `eval()` are necessary.

25. An exception should be thrown if a non-empty array is assigned to an array expression of a different size, rather than implicitly resizing the array.

## Appendix: Analysis of a large-scale scientific application in Fortran

As discussed in the introduction, one of the requirements of the C++ array library considered in this paper is be applicable to large-scale scientific applications. To illustrate the needs of such applications in terms of array handling, the source code for one particular application has been analysed: the Integrated Forecasting System (IFS) of the European Centre for Medium Range Weather Forecasts (ECMWF). It is used operationally to produce forecasts for timescales from one day to one year, and is commonly referred to as the "European Model" in the North American media.

The IFS numerically solves partial differential equations to evolve a three-dimensional description of the global atmosphere and ocean forward in time (see, for example, Bauer et al., 2015). In its highest operational resolution, the atmosphere is described by 0.9 billion individual grid cells and the program is parallelized across 352 nodes, each with 36 CPU cores, using a mixture of MPI and OpenMP parallelism. It is written in Fortran 2003 and consists of 2.2 million lines of code and 9285 source files. In terms of physical processes and mathematical techniques, it employs global spectral transforms in its dynamical core and a mixture of implicit and explicit schemes to treat cloud and precipitation processes, turbulence, ocean waves, surface exchange processes, and the transfer of solar and thermal radiation. A large fraction of the code solves the nonlinear optimization problem of using 40 million observations globally each day to create an accurate atmospheric state from which to make the forecast. In terms of dimensionality of arrays, only a small fraction of the routines "see" the full 3D atmosphere as many processes work only in vertical columns of the atmosphere, but array dimensions are frequently used for non-spatial dimensions such as the wavelength of radiation or the viewing angle of a satellite.

We first examine the use of arrays in the IFS by parsing all the array declarations within routines and class definitions. Figure 1 depicts the number of occurrences of different types of array as a function of the number of array dimensions. The types are defined in terms of their Fortran attributes in Table 4. The first family are "explicit-shape" arrays; these are stored contiguously in memory and when passed as an argument to a routine, Fortran passes only a pointer to the first memory address, so the size of each dimension must be declared at the top of the routine (illustrated by `n` in Table 4). The three types of explicit-shape array shown here distinguish arrays defined locally to a routine, and ones received as a constant or non-constant argument to a routine. Over 127,000 explicit-shape floating-point arrays are used in the IFS, of which 11% have 3 or more dimensions and 1.5% have 4 or more dimensions.

The second family of array types were introduced in Fortran 90 and under the hood consist of an array descriptor (sometimes called a dope vector) that stores the size and stride of each dimension, in addition to a pointer to the data (see section 8). This way non-contiguous array subsets may be passed into a routine without a deep copy being performed. This array type is therefore the most like the C++ arrays considered in this paper. The four versions of this type distinguish: resizable arrays declared locally to a routine or class definition, arrays passed as an argument to a routine with the routine permitted to resize them, and arrays (or subsets of arrays) passed as a constant or non-constant argument to a routine

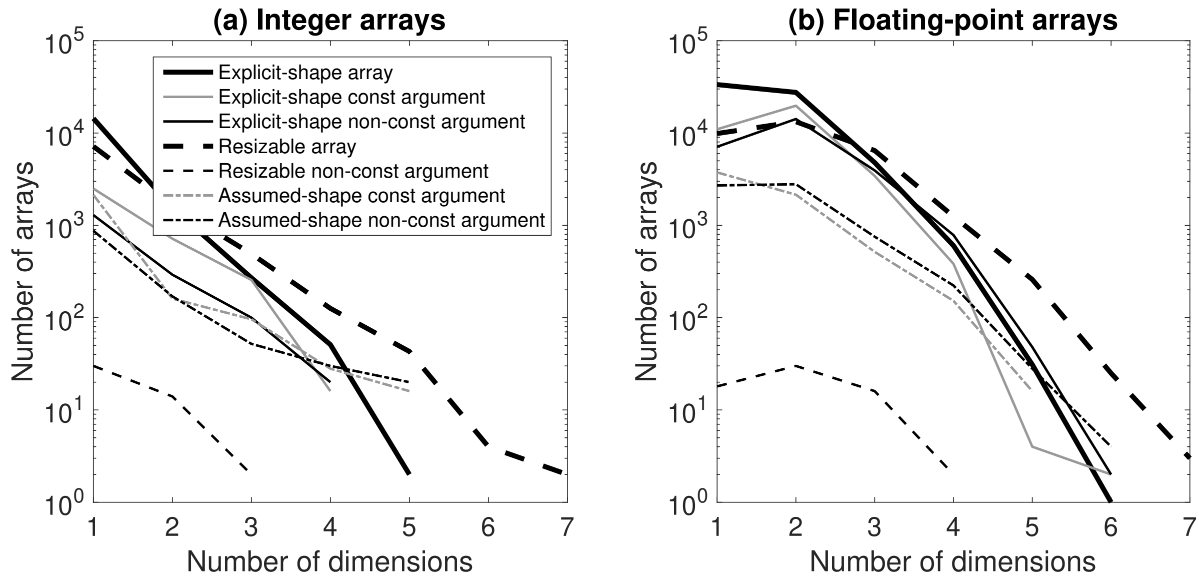


Figure 1: The number of occurrences of (a) integer and (b) floating-point arrays, as a function of the number of dimensions, in Cycle 44 of the Integrated Forecasting System. The array types are described in Table 4 and in the text. The thick lines show arrays allocated within a function, while the thin lines show arrays passed as arguments to a function.

Table 4: The Fortran array attributes corresponding to the array types plotted in Fig. 1. Note that "resizable arrays" may have either of the similar attributes `allocatable` or `pointer`.

| Array type | Typical Fortran array attributes (in case of one dimension) |
|---|---|
| 1. Explicit-shape array | `dimension(n)` |
| 2. Explicit-shape const argument | `dimension(n), intent(in)` |
| 3. Explicit-shape non-const argument | `dimension(n), intent(inout)` |
| 4. Resizable array | `dimension(:), allocatable` |
| 5. Resizable non-const argument | `dimension(:), intent(inout), allocatable` |
| 6. Assumed-shape const argument | `dimension(:), intent(in)` |
| 7. Assumed-shape non-const argument | `dimension(:), intent(inout)` |

where the routine does not have permission to resize the array. The latter two versions are referred to as "assumed-shape" arrays in Fortran. Over 44,000 floating-point arrays fall into this second family, of which 22% have 3 or more dimensions while 4.4% have 4 or more dimensions. It is clear from this analysis that it would be impossible to code the IFS efficiently and elegeantly in C++ without arrays of more than two dimensions, or the capability to pass arrays and subsets of arrays between routines.

Lastly we consider functions for efficiently manipulating arrays. Fortran provides a rich set of such functions, and to provide some information to help decide what functionality is most important to include in a C++ array library, Table 5 lists how often each of these functions are used in the IFS code, as well as their type. The most common functions are `min` and `max`, which are "elemental" functions in the sense that they are simply the array equivalent of the scalar functions of the same name: they can be applied to two arrays of the same shape, or to an array and a scalar. They are widely used in the IFS as a security to prevent division by zero and other floating-point exceptions, and can be vectorized

Table 5: List of Fortran array functions and the number of times they are used in the IFS, where "type" has the following meaning: E: Elemental, O: Operator, R: Reduce, K: Keyword, M: Matrix multiplication, A: Array.

| Command | Type | Calls | Meaning |
|---|---|---|---|
| `min` | E | 14296 | Minimum of two array expressions |
| `max` | E | 13233 | Maximum of two array expressions |
| `=>` | O | 12073 | Associate array pointer with target array |
| `count` | R | 3099 | Number of `true` values in boolean array expression |
| `sum` | R | 2972 | Sum all elements in an array |
| `where` | K | 2238 | Execute array expressions conditional on a boolean array |
| `maxval` | R | 1982 | Maximum value in an array |
| `all` | R | 1790 | True if all elements of an array are true |
| `any` | R | 1584 | True if any elements of an array are true |
| `minval` | R | 1441 | Minimum value in an array |
| `dot_product` | M | 779 | Inner product of two vectors |
| `product` | R | 476 | Product of all elements in an array |
| `reshape` | A | 431 | Reshape the data in an array optionally reordering dimensions |
| `pack` | A | 286 | Pack data from an array into a vector |
| `unpack` | A | 226 | Unpack data from a vector into an array |
| `minloc` | R | 172 | Location of minimum value in an array |
| `maxloc` | R | 153 | Location of maximum value in an array |
| `spread` | A | 127 | Replicate an array along an additional dimension |
| `matmul` | M | 65 | Built-in matrix multiplication |
| `transpose` | A | 51 | Transpose a matrix |
| `merge` | A | 43 | Select from two arrays according to a third boolean array |
| `dgemm` | M | 30 | Matrix-matrix multiplication from BLAS library |
| `eoshift` | A | 20 | "End-off" shift along specified dimension |
| `dgemv` | M | 8 | Matrix-vector multiplication from BLAS library |
| `cshift` | A | 6 | Circular shift along specified dimension |

by the compiler. The third most common "function" is actually implemented as an operator in Fortran: "=>" associates an array (that was declared with the `pointer` modifier) with another whole array or array subset, so that both arrays share the same data (see section 5.3) . Many of the other functions are discussed at other points in section 5. The least used are those that perform a shift along a particular dimension (`cshift` and `eoshift`) and can be safelty excluded from a C++ library.

## References


Arabas, S., D. Jarecka, A. Jaruga and M. Fijałkowski, 2014: Formula translation in Blitz++, NumPy and modern Fortran: A case study of the language choice tradeoffs. *Sci. Programming,* **22,** 201–222.

Aragón, A. M., 2014: A C++11 implementation of arbitrary-rank tensors for high-performance computing. *Comp. Phys. Comm.,* **185,** 1681–1696.

Barthels, H., M. Copik and P. Bientinesi, 2018: The generalized matrix chain algorithm. In *Proc. 2018 Int. Symp. on Code Gen. and Optimization (CGO 2018).* ACM, New York, 138–148. doi:10.1145/3168804.

Barton, J. J., and L. R. Nackman, 1994: *Scientific and engineering C++: an introduction with advanced techniques and examples.* Addison-Wesley, 671 pp.

Bauer, P., A. Thorpe and G. Brunet, 2015: The quiet revolution of numerical weather prediction. *Nature,* **525,** 47–55.

Davidson, G., and B. Steagall, 2019: A proposal to add linear algebra support to the C++ standard library. *C++ Standards Committee Paper P1385R3,* available at `http://wg21.link/P1385` (retrieved 1 November 2019).

Decyk, V. K., C. D. Norton and H. J. Gardner, 2007: Why Fortran? *Comp. in Sci. & Eng.* **9,** 68–71.

Guillen, C., and R. Bader, 2017: Selecting compilers for a supercomputer. *ADMIN Mag.,* `http://www.admin-magazine.com/HPC/Articles/Selecting-Compilers-for-a-Supercomputer` (retrieved 30 October 2019).

Hoemmen, M., J. Badwaik, M. Brucher, A. Iliopoulos and J. Michopoulos, 2019: Historical lessons for C++ linear algebra library standardization. *C++ Standards Committee Paper P1417R0,* available at `https://isocpp.org/files/papers/p1417r0.pdf` (retrieved 30 October 2019).

Hogan, R. J., 2014: Fast reverse-mode automatic differentiation using expression templates in C++. *ACM Trans. Math. Softw.,* **40,** 26:1-26:16.

Hogan, R. J., 2017: Adept 2.0: A combined automatic differentiation and array library for C++. doi:10.5281/zenodo.1004730.

Hollman, D. S., B. A. Lelbach, H. C. Edwards, M. Hoemmen, D. Sunderland and C. R. Trott, 2019: mdspan in C++: A case study in the integration of performance protable features into international language standards. *Proc. 2019 IEEE/ACM Int. Workshop on Performance, Portability & Productivity in HPC,* doi:10.1109/P3HPC49587.2019.00011.

Iglberger, K., G. Hager, J. Treibig and U. Rüde, 2012: Expression templates revisited: A performance analysis of current methodologies. *SIAM J. Sci. Comp.,* **34,** C42–C69.

Meyers, S., 2005: Effective C++: 55 specific ways to improve your programs and designs. Addison-Wesley Pro., 316 pp.

Reid, J., 2018: The new features of Fortran 2018. *ACM SIGPLAN Fortran Forum,* **37,** 5–43, available at

Sanderson, C., and R. Curtin, 2016: Armadillo: a template-based C++ library for linear algebra. *J. Open Source Softw.,* **1,** 26 pp.

Smith, N. J., 2014: A dedicated infix operator for matrix multiplication. *Python Enhancement Proposal 465,* available at `https://www.python.org/dev/peps/pep-0465/` (retrieved 7 August 2018).

Veldhuizen, T., 1995: Expression templates. *C++ Report,* **7,** 2631-.