Multidimensional Arrays and the *nArray* Package

Ofri Sadowsky, Daniel Li, Anton Deguet, Peter Kazanzides

July 1, 2007

Department of Computer Science, Johns Hopkins University
email:ofri@cs.jhu.edu

Abstract

At the Johns Hopkins University's Engineering Research Center for Computer-Integrated Surgical Systems and Technology (ERC-CISST) laboratory, we have designed and developed a platform-independent C++ software package, called the *nArray* library, that provides a unified framework for efficiently working with multidimensional data sets. In this paper, we present and discuss the core elements of the library, including its intuitive and uniform API, efficient arithmetic engine algorithm, and efficient sub-volume algorithm. We then compare the performance of the *nArray* library with that of an existing multidimensional array toolkit, ITK. We conclude that the *nArray* library is more efficient than ITK in many situations, especially in operations on sub-arrays, and that the two packages have comparable performance in many other scenarios.

1 Introduction

Multidimensional data sets are becoming increasingly prevalent in today's world, especially in the fields of computer-integrated surgery and image processing. A canonical example in image processing is a collection of three-dimensional CT scans over time that defines a four-dimensional data set. A three-dimensional image whose pixels each have a vector quantity, such as color or direction, also defines a four-dimensional data set. Such multidimensional data sets grow in size extremely quickly, and so it is important to have a software package that efficiently handles them.

At the Johns Hopkins University's Engineering Research Center for Computer-Integrated Surgical Systems and Technology (ERC-CISST) laboratory, we have designed and developed a platform-independent C++ software package, called the *nArray* (pronounced "EN-array") library, that provides a unified framework for efficiently working with multidimensional data sets. In addition to standard features such as STL-compatible iterators to traverse the data sets, the *nArray* library also provides immutable classes for safe data handling, efficient referencing of sub-volumes of data, layout manipulations, and an extensible generic computational engine.

In this article, we explore the unique layout of the *nArray* library, focusing specifically on the computational engine and the efficient sub-volume algorithm. We then compare the efficiency of the *nArray* package with an existing toolkit with multidimensional array support, the National Library of Medicine's Insight Toolkit (ITK) [1]. Through this paper, we will see how the efficiency of the *nArray* package proves to be an effective tool for managing multidimensional data sets.

1.1 Motivation

The conventional C notation for a multidimensional array looks similar to the following:

```
unsigned int *** uintVolume;   /* this is a "three-dimensional" array */
```

Elaborate allocation and deallocation methods are required to properly manage such a memory layout. Ultimately, the actual data typically is allocated as a single, continuous memory block, i.e., a "flat" structure. Smaller arrays of pointers are then used to dereference portions, or "slices," of the block. It is clear from the notation that the higher the dimensionality of the dataset, the more complicated the "bookkeeping" of its layout becomes. A main goal in designing the *nArray* package was to simplify this process for the user by providing a templated uniform interface. For example, the same three-dimensional array created with an *nArray* container looks like:

```
/* here the array's type and dimension are template parameters */
vctDynamicNArray<unsigned int, 3> uintVolume;
```

nArray containers of all dimensions have a common interface, i.e., methods and operations. This makes the learning curve shorter, gives scalability, and helps with the debugging. To support this generic usage, we have developed and implemented a set of algorithms to address the issues involved with bookkeeping. Some of these algorithms will be outlined in this paper later.

1.2 Inspiration

APIs and software tools that deal with multidimensional data sets have been around for a while. MATLAB® has many advantages in "rapid prototyping" of algorithms, e.g. in signal processing and in the running of interactive processes, but its poor handling of data structures, flow control, and memory allocation control makes it unsuitable for real-time application development. Other interpreted languages, such as Python, offer a similar functionality for similar performance costs. ITK has gained popularity in recent years in the medical image processing community. It is designed as a *data flow* oriented toolkit, where components are linked in a processing pipeline and final outputs are produced at the end of the pipeline chain. Our feeling is that this programming model is less intuitive to many C/C++ programmers and that it does not provide enough low-level access for optimizing one's code.

The *nArray* package was built as an extension to our prior development of the `cisstVector` library of vectors and matrices written in C++ [2]. Some of the functionality of *nArray*s involves the use of vectors from this library. The `cisstVector` library was inspired by existing toolkits in C++, such as VNL [3] and the Standard Template Library, in many of its aspects. We have tried to implement the good concepts in these libraries and improve those that seemed to need improvement.

A discussion of generic programming cannot be complete without mentioning the work of Todd Veldhuizen, the "father" of template metaprogramming [4]. The `cisstVector` library relies on metaprogramming structures, though different from Veldhuizen's Blitz++ library [5].

Finally, the development of the *nArray* computational engines, presented briefly in this paper, was an extension of an idea contributed by Robert Jacques, currently at the Johns Hopkins University. Jacques suggested an improvement to our matrix computation engines, which we later extended to *nArray*s.

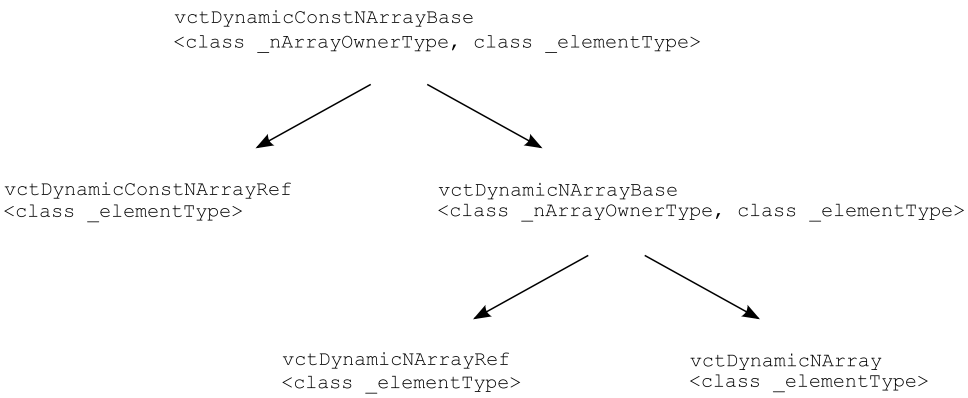

Figure 1: The *nArray* class inheritance hierarchy.

2 Library Elements

2.1 *nArray* class hierarchy

The *nArray* package interfaces with a given data set through one of three *nArray* containers. These three API-level containers are organized via the five-class inheritance hierarchy shown in Figure 1. Shared methods are defined in the base classes and are inherited by the three child classes, which correspond to the three API-level containers. This inheritance hierarchy maintains a uniform interface across the three *nArray* containers.

The two base classes `vctDynamicConstNArrayBase` and `vctDynamicNArrayBase` contain the bulk of the API methods. The first includes only immutable methods, while the second extends it with mutable methods. These two classes are only accessible for the end-user through the following specializations. The classes `vctDynamicConstNArrayRef` and `vctDynamicNArrayRef` are *overlay* objects, which can be used to access externally allocated memory layouts as if they were multidimensional arrays. For example, `vctDynamicConstNArrayRef` is an immutable overlay, providing read-only structured access to a memory block identified by a `const _elementType *` (i.e., an address). Finally, the class `vctDynamicNArray` is an *allocating* object, and it allocates and releases a storage memory block.

2.2 *nArray* API sampler

The `cisstVector` library, of which the *nArray* package is an extension, was designed with the Abstract Data Type programming paradigm in mind. This means that the objects in the library are data containers and operations between them are methods of their classes. The "ideal" ADT paradigm would use arithmetic operators to resemble a mathematical notation in the programming language wherever possible. To this end, the *nArray* containers use *named methods* for all of its operations, while in applicable cases, overloaded operators that use the named methods as subroutines are additionally defined.

Table 1 shows a few examples of the functional `cisstVector` API. When an equivalent C++ expression is available, it is listed in the third column. If `cisstVector` provides an overloaded operator, it is indicated in the fourth column. The fifth column indicates which operands are immutable, that is, not modifiable by the operation. Usually, if an operand is immutable, it appears in the method's signature as a template `vctDynamicConstNArrayBase`. This allows any of the *nArray* classes to be passed as the actual method

Operation Description	CISST Code	Equivalent C++ Notation	Overloaded Operator	Immutable Array Operands

Addition (two containers or a container and a scalar)	`c1.SumOf(c2, c3);`
	`c1.SumOf(c2, s);`
	`c1.Add(c2);`
	
	`c1.Add(s);`
		`c1 = c2 + c3;`
		`c1 = c2 + s;`
		`c1 += c2;`
		
		`c1 += s;`
			Yes
			Yes
			Yes
			
			Yes
				`c2,c3`
				
				`c2`

Elementwise multiplication	`c1.ElementwiseProductOf( c2, c3);`
	`c1.ElementwiseMultiply(c2);`
		`c1[i] = c2[i]*c3[i];`
		`c1[i] *= c2[i];`
			No
			
			No
				`c2,c3`
				
				`c2`

Division by scalar	`c1.RatioOf(c2, s);` `c.Divide(s);`	`c1 = c2 / s;` `c1 /= s;`	Yes Yes	`c2`
Sum of elements	`s = c.SumOfElements();`	N/A	No	`c`
Largest element	`s = c.MaxElement();`	N/A	No	`c`

Table 1: A sample of the operations in the *nArray* package.

parameter. In the table, `c` stands for a generic container, which in the `cisstVector` library can be a (fixed-size or dynamic) vector, a matrix, or an *nArray*; `s` stands for scalars of the same type as that of the array elements, e.g. `double`. Different operands, when they are involved, are distinguished by number. The table shows only a small selection of operations; the complete list is in the library's documentation.

2.3 Layout manipulation

A central feature of the *nArray* containers is their ability to reference other *nArray*s using different *layouts*, or subsections of an existing *nArray* . This is achieved by configuring an overlay *nArray* to span the desired region of an existing *nArray* container. Overlaying allows the user to operate on array elements in-place, without copying them out and in.

A new layout may have the same dimension as its parent container, focusing on a smaller region; or it may have a lower dimension. We call the region focusing a *window* and the dimensionality reduction a *slice*. Combining windows and slices allows the user to specify any subsection of any dimension of an existing *nArray* container. Another useful configuration is changing the order in which *nArray* elements are accessed; this is called *permuting* the order of element access, and is similar to MATLAB's `permute` function, without the memory allocation and copy overhead.[1]

Let us demonstrate these ideas through an example. Suppose we want to select a small region of interest in a sagittal cross-section of a CT volume. Since a CT volume is by default created by stacking transverse cross-sections, we first perform a *permutation* on the volume to orient the order of access of elements correctly. Next, we perform a *slice* operation on the permuted volume to obtain a two-dimensional container that holds the desired sagittal slice. Finally, to focus on a particular region of the sagittal slice, we perform a *window* operation on the slice. This is illustrated in Figure 2 and in the code listing in Table 2.

Breaking up the overlay concept into these three distinct operations has two direct benefits: it creates easier-

[1]The *nArray* library API uses the term `Subarray` for what we define in this paper as the *window* operation. Throughout the text of this paper, we will stick to the *window* terminology, although the code samples we include continue to refer to it as a `Subarray`.

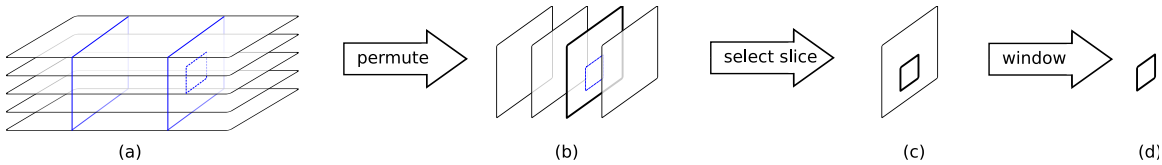

Figure 2: An illustration of *nArray*'s layout operations: permute, slice, window. (a) *Initial stack of transverse images.* Sagittal cross-sections are shown with blue lines; the region of interest is shown with blue dashes. (b) *Sagittal cross sections.* The slice of interest is shown with thick lines; the region of interest is shown with blue dashes. (c) *Slice of interest.* The region of interest is shown with thick lines. (d) *Region of interest.*

to-debug code, and it gives one flexibility in how one creates new layouts.

2.4 Strides

The windowed overlay and dimension permutation are achieved through a careful definition of *strides*, which indicate the increment in memory address between adjacent array elements in each stride's corresponding dimension; each stride can be either a positive or negative integer. Every *nArray* object includes a vector of stride values whose length is equal to the number of dimensions. In a simple, flat layout, the fastest changing dimension has a stride of 1, and the stride of any higher dimension is equal to the number of elements in all the lower-dimension slices. In a window overlay, however, a stride in one dimension can be larger than the number of elements in a lower dimension, which means that more than one memory cell is "skipped" in order to move from one slice to the next. When permutations are applied, the order of the strides changes arbitrarily. An example of overlay strides is shown in Figure 3.

3 Algorithms

3.1 *nArray* engines

The *nArray* engines are a set of classes and functions optimized to efficiently traverse the elements of *nArray* containers. The engines operate on both memory-allocating and overlaying containers, and the interface to call the engines on either type of container is identical, so the differences between operating on the two types of containers are transparent to the user. Also, the engines handle both contiguous and non-contiguous memory blocks.

Almost always the containers used with the engines play the part of operands of an operation (such as the addition of two arrays, with the sum stored into a third). Each engine is designed to support a specific combination of operands, such as unary, binary, or store-back operations (e.g. the `Abs`, `+` and `+=` operators), as well as various operand types (e.g. arrays, scalars, etc.).

Expression structure in the engines is abstracted and encapsulated. Any operation defined in the *nArray* API only requires the user to select the appropriate expression structure and provide the relevant operators. For example, computing the maximum element, the sum of elements, and the sum of squares of an *nArray* all make use of the same engine because they all have the same expression structure: compute a scalar function from all the elements of one *nArray* . The three operations differ only in the operations to be performed on or between the elements. For example, in *sum of elements* and *maximum*, there is no operation on individual

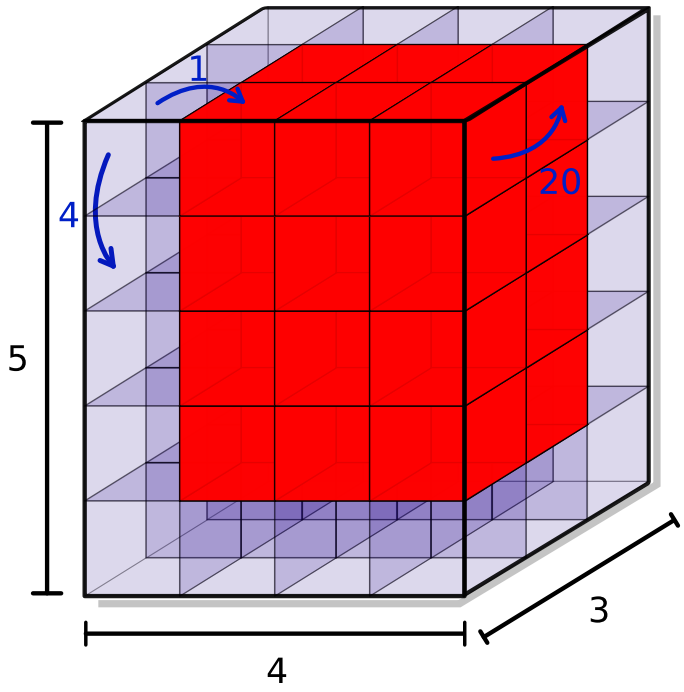

Figure 3: An example of strides in a three-dimensional array. The purple block of sizes of $3 \times 5 \times 4$ cells (the order of dimensions is Z,Y,X) is the "parent container". The red block, with $2 \times 4 \times 3$ cells, is a window overlay of the parent container. In both blocks, the memory strides between adjacent elements are (20, 4, 1) in corresponding dimension order.

```
// Create original CT volume.  The sizes are specified
// in the Z-Y-X order.  nsize_type is a "fixed-size" vector.
NArrayType::nsize_type originalSize(240, 512, 512);
NArrayType originalCTVolume(originalSize);

// load volume with CT data
/* ... */

// Create a permuted overlay of the original CT.
// Dimension 0 of the permuted array corresponds to dimension
// 2 of the original volume, that is, the X dimension, or sagittal
// cross sections.
NArrayType::nsize_type orderOfDimensions(2, 0, 1);
NArrayType::PermutationRefType sagittalOrientation;
sagittalOrientation.PermutationOf(originalCTVolume, orderOfDimensions);

// Select slice of interest in dimension 0.
NArrayType::size_type dimension = 0;
NArrayType::size_type index = 172;
NArrayType::SliceRefType sliceOfInterest;
sliceOfInterest.SliceOf(sagittalOrientation, dimension, index);

// Select region of interest.
NArrayType::SliceRefType::nsize_type regionSize(192, 192);
NArrayType::SliceRefType::nsize_type regionStart(10, 50);
NArrayType::SliceRefType::SubarrayRefType regionOfInterest;
regionOfInterest.SubarrayOf(sliceOfInterest, regionStart, regionSize);
```

Table 2: Code example of defining layout manipulation overlays.

elements, but in *sum of squares*, the square of each element is computed; in *sum of elements* and *sum of squares*, there is also an addition operation between elements, and in *maximum* there is a *max* operation between elements. Beyond these differences, the structure of all three operations is identical, and this similarity is expressed in the engines. In practice, the operations (*add*, *max* ...) are passed to the engines as template parameters whereas the signature of the engine function is identical for all operations. In this way, the engines make the containers very straightforward to use and encourage the creation of easy-to-debug code.

The engines have several features that contribute to the overall efficiency and scalability of the package while hiding the details of the traversal algorithm from the caller function. They can be viewed as an abstraction of a multi-level nested loop.

3.2 Engine algorithm

The *nArray* engine uses a pointer, which we call the "current" pointer, to traverse an *nArray* container, much like an STL iterator, which traverses a container from beginning to end in sequential order. However, since the engine must operate on both memory-allocating and overlay *nArray* containers, the algorithm is not as simple as having the pointer run through the container's memory block from start to finish. Instead, when the pointer reaches the end of a dimension of the container, the engine must know by how many address spaces to increment the pointer in order to reach the next element in that dimension.

In designing the engine algorithm, we had to address three main issues. First, we could not use a nesting structure to loop through a container because there is an arbitrary number of dimensions in that container. We resolved this by using a vector of the same size as the number of dimensions of the container to store "target" pointers, which mark the end of each dimension. When the current pointer reaches a target pointer, the engine "wraps" the current pointer around this dimension by incrementing the pointer by the appropriate number of address spaces. The wrap-around test is performed in a recursive fashion, traversing down the list of targets until the appropriate one is found. This replaces the conventional nested loop code structure.

The second issue was how to efficiently wrap around the current pointer when it reaches a target pointer. This is resolved by having the engine precalculate the dimension offsets (the "stride to the next dimension", or STND, values) using the stride values.

Finally, the engines also have to update the target pointers accordingly when the wrap-around occurs. This is shown in the `IncrementPointers` method below. `IncrementPointers` is the equivalent of the "loop header", and it is called to move the current pointer to the next element. Notice that `IncrementPointers` returns the number of dimensions that have been exhausted and wrapped-around. We take advantage of this result to wrap-around the "current pointer" for other operand containers that may be involved. This topic, however, is not covered in this paper.

The result is an efficient and versatile engine algorithm that is capable of running on both memory-allocating and reference *nArray* containers. The source code for the algorithm is provided below. It is written in C-style notation.

```
// The PreProcess function computes the wrap-around strides (STND)
// and the target pointers before the engine loop is begun
void PreProcess(const unsigned int numDimensions,
  const unsigned int arraySizes[], const int arrayStrides[],
  const _elementPointer basePtr, int arraySTND[],
  _elementPointer arrayTargets[])
{
  unsigned int i;

  for (i = 0; i < numDimensions; ++i)
  {
    unsigned int span = arraySizes[i] * arrayStrides[i];
    /* calculate initial placement of target pointers */
    arrayTargets[i] = basePtr + span;
    /* calculate STND */
    arraySTND[i] = (i != 0) ? arrayStrides[i-1] - span : 0;
   }
}

// The function IncrementPointers checks which dimensions are
// exhausted, performs a wrap-around of the current pointer,
// and recomputes new targets if they need to be updated.  It
// returns the number of dimensions that were exhausted.
unsigned int IncrementPointers(const unsigned int numDimensions,
  _elementPointer targets[],  _elementPointer  & currentPointer,
  const int strides[],  const int stnd[])
  {
    unsigned int i = numDimensions - 1;
    unsigned int wrapCounter = 0;
    currentPointer += strides[numDimensions - 1];
```

```
    while (currentPointer == targets[i]) {  // the i-th dimension is exhausted
      currentPointer += stnd[i];
      ++wrapCounter;
      --i;
      if (wrapCounter == numDimensions) // exhausted all elements
        return wrapCounter;
    }

    // if no dimensions were exhausted, we can return immediately
    if (wrapCounter == 0)
      return wrapCounter;

    // now, update the targets forward from the current pointer
    ++i;
    do {
      targets[i] = currentPointer + (strides[i-1] - stnd[i]);
      ++i;
    } while (i < numDimensions);

    return wrapCounter;
  }
```

As an example, consider the *nArray* container in Figure 3. The red overlaying window container has size $2 \times 4 \times 3$ and stride values (20, 4, 1). The values the engine would calculate for this container are:

```
Sizes                2   4   3
Strides             20   4   1
STND                 0   4   1
Target offsets      40  16   3
```

3.3 Iterators

Iterators are a widely-used method of traversing all the elements of a container, utilizing an object to keep tabs on the location. We created the *nArray* iterators to conform with the Standard Template Library's specification for a random access iterator.

The *nArray* iterators are designed to handle complicated layouts such as dimension permutations and non-contiguous memory blocks. While a mechanism similar to the *nArray* engines could be encapsulated as an object, we believe this might be overkill for iterators. Instead, an *nArray* iterator keeps a "meta-index" internally, which indicates the sequential position of an element in the container. The iterator converts the meta-index to a "position index", which is a tuple of zero-based coordinates, similar to our intuitive notion of a multidimensional index. The position index then is converted again to an address offset from the array's base pointer by computing its dot product with the strides of the *nArray* .

To explain this mechanism, let us first write down a recursive function, presented below, which takes as input the array of sizes (i.e., numbers of elements in each dimension) of the *nArray* , the array of strides in the respective dimensions, and a meta-index, and returns an offset from the base pointer and a small array of indices. For simplicity of presentation, the function is written in C-style notation.

```
(0) unsigned int MetaIndexToOffsetAndMultiIndex(const unsigned int numDimensions,
       const unsigned int arraySizes[], const int arrayStrides[],
```

```
        const unsigned int metaIndex, unsigned int multiIndex[])
    {
(1)     if (numDimensions == 0)
(2)         return 0;
(3)     const unsigned int d = numDimensions-1;
(4)     multiIndex[d] = metaIndex % arraySizes[d];
(5)     const unsigned int dContribution = strides[d] * multiIndex[d];
(6)     return dContribution +
            MetaIndexToOffsetAndMultiIndex(numDimensions-1,
            arraySizes, arrayStrides, metaIndex / arraySizes[d], multiIndex);
    }
```

As an example, let us follow the operation of this function on a three-dimensional flat container of size $3 \times 5 \times 4$ (see Figure 3) with corresponding strides of (20, 4, 1). We start with a meta-index `metaIndex=23`. Unknown values are written as question marks. The number preceding each line refers to the corresponding line number in the code above. For simplicity, we do not replicate the array parameters through the nesting of the recursion.

```
(0) numDimensions = 3, arraySizes = [3, 5, 4], arrayStrides = [20, 4, 1],
       metaIndex = 23, multiIndex = [?, ?, ?]
(3) d = 2
(4) multiIndex = [?, ?, 23%5] = [?, ?, 3]
(5) dContribution = arrayStrides[d] * multiIndex[d] = 1 * 3 = 3
      (0) numDimensions = 2, metaIndex = 23/5 = 4
      (3) d = 1
      (4) multiIndex = [?, 4%4, 3] = [?, 0, 3]
      (5) dContribution = arrayStrides[d] * multiIndex[d] = 4 * 0 = 0
            (0) numDimensions = 1, metaIndex = 4/4 = 1
            (3) d = 0
            (4) multiIndex = [1%3, 0, 3] = [1, 0, 3]
            (5) dContribution = arrayStrides[d] * multiIndex[d] = 20 * 1 = 20
                  (0) numDimensions = 0, metaIndex = 1 / 3 = 0
                  (2) return 0
            (6) return dContribution + 0 = 20
      (6) return dContribution + 20 = 20
(6) return dContribution + 20 = 23
```

Since the layout in this example is flat, the final offset is equal to 23, which is the original meta-index. However, applying this mechanism to a configuration with different strides still computes a correct offset, which may be different from the meta-index. The final values in `multiIndex` are (1, 0, 3), which are the zero-based "coordinates" of the element whose meta-index is 23 in an ordinary indexing mode.

It is worth noting that a memory vs. runtime tradeoff is expected when comparing the engines and the iterators, with the engines being the more runtime-efficient. In principle, either mechanism could replace the other, but in practice, the engines are better optimized for runtime than a replication of iterators is, even if a similar wrap-around mechanism were implemented for the iterators; this is because the wrap-around decision inside an engine needs only to be made once, while individual iterators must decide on the wrap-around independently. Therefore, most of the operations that need to be performed on an *nArray* should be done via the engines and not the iterators.

4 Performance Benchmarks

4.1 Performance discussion

As a general rule, more regular data layouts can be processed with faster algorithms. With regards to multidimensional arrays, a flat layout can be processed faster than another layout, such as a non-contiguous block or a layout manipulation overlay, for the same data size. This is because the overhead of keeping tabs on the pointer position is easier in the flat case.

When we compare the performance of *nArray* operations via expression engines with other software packages, we have to take care to compare features on level terms. If, for example, a certain library supports only flat layouts and has a fast algorithm for evaluating expressions on them, then it should be compared with the performance of the CISST package on flat containers, i.e., vectors. On the other hand, if a library supports certain layout manipulators, such as regions of interest, then we can compare them with the engine methods in the *nArray* package. In addition, we can compare the *nArray* engines with the expression engines for lower-dimension containers in the CISST package, namely, dynamic vectors. This comparison should provide an estimate of the bookkeeping overhead involved with traversing the *nArray* .

Considering this overhead, the efficiency of using overlay arrays can depend on the frequency of their use. Evaluating a single expression involving a layout manipulation can be faster using the overlay structure than if the manipulated data layout needs first to be copied to a second container before the expression is evaluated. However, if many expressions involving the same immutable dataset are considered, it is usually more efficient to copy the elements once into a flat container and evaluate all the expressions using the new block. An important advantage of the *nArray* overlays is that they provide flexibility to the user in choosing the preferred processing method.

Likewise, if we want to compare the performance of overlays with a library that only supports copy-and-evaluate implementations, the timing of one expression involving an overlay should be compared with the timing of copying and evaluating the expression, combined, and not just with the time of evaluating. If new memory allocation is needed before the copy, then the timing for memory allocation must be counted as well.

4.2 Benchmark specifications

We performed two kinds of benchmark runs. The first consisted of extracting a manipulated layout from a parent container and copying its elements into another container. Considering the different software architectures, this simple operation was chosen to highlight the performance of the array traversal algorithm in different configurations. Notice that only two data containers were involved: the parent and the destination, and no arithmetic operations were performed on the data elements.

We compared the CISST *nArray* package with ITK's Image class. Using both libraries, we created the following test cases: a four-dimensional array; a smaller region of interest ("window") which is also four-dimensional; a three-dimensional slice of the larger array; and an axis permutation, having the same number of elements as the parent container but in a different access order. In the *nArray* package, each layout was created as an overlay on the parent container. We called the `Assign()` method to read elements from an overlay array and write them to a memory-owning array. In ITK, each operation was represented as a "filter" object, which stores a copy of the outcome. The evaluation of the expression was triggered by calling the method `Update()` on the filter.

The second kind of benchmark involved container arithmetic, namely, adding two four-dimensional arrays

into a third four-dimensional array. Here, the two input operands were "windows" or subregions of two larger parent containers. The output container was allocated independently. The purpose of this benchmark was to compare the *nArray*'s overlay approach with the more traditional copy-out approach in ITK. There are two ways to compute the sum of two *nArray*s, as can be seen in Table 1: (a) apply the method `SumOf` to a third *nArray* object (the result operand); or (b) use an overloaded `operator +`. We compared both methods to demonstrate the performance cost of using an overloaded operator. In ITK, we created an `AddImageFilter` object to compute and store the sum; its inputs were the outputs of two `RegionOfInterestImageFilter` objects, which in turn copied the contents of the regions to an internal storage.

The source files for the benchmarking programs are as follows.

Benchmark	***nArray***	**ITK**
Layout manipulations	`Subarray_nArray_Benchmark.cpp`	`Subarray_ITK_Benchmark.cpp`
Array sum	`ImageAdd_nArray_Benchmark.cpp`	`ImageAdd_ITK_Benchmark.cpp`

4.3 Performance evaluation

In the C++ programs listed above, the operations of interest were surrounded by calls to a "stopwatch" object with `Start()` and `Stop()` methods to measure the evaluation time. The stopwatch uses the high-frequency timer in either Windows or Linux. The output was rounded to a millisecond precision. Both the CISST -based and the ITK-based programs were compiled and run on the following systems.

- Windows XP workstation: Two dual-core Intel Xeon CPU, 3.06 GHz, 2.00 GB RAM. Compiler: Microsoft Visual Studio 7.1
- Linux server: Two dual-core Intel Xeon CPU 64 bit, 2.0 GHz, 6.0 GB RAM, 4 MB cache per CPU. Ubuntu Linux distribution. Compiler: gcc 4.1.2

The compilation on both system was in "Release" mode, using compiler optimizations for speed. The accumulated times for 30 repetitions of the test are summarized in Table 3; all times are in milliseconds. The data sizes are as given in the source files listed above.

For the *slice* operation, the CISST implementation consistently performed 46% to 53% faster than ITK. In the other tests, however, it was more difficult to obtain a consistent comparison. For the *window* operation,

Operation	**CISST time, Windows**	**ITK time, Windows**	**CISST time, Linux**	**ITK time, Linux**
Window	8,814	14,860	4,093	3,538
Slice	250	465	122	261
Permute (3, 1, 2, 0)	202,342	183,173	73,125	99,123
Permute (3, 0, 2, 1)	193,742	95,403	58,738	18,767
Add 4D arrays	12,203	(a) 28,674 (b) 17,501 (c) 46,262	3,757	(a) 6,875 (b) 3,950 (c) 10,848
Operator +	19,044	N/A	8,790	N/A

Table 3: Benchmark times for the CISST *nArray* and ITK operations on Windows and Linux systems. The times are in milliseconds for 30 repetitions of the operation.

CISST was about 41% faster on the Windows build but about 16% slower on Linux. The performance of the *permute* operation was also inconsistent, depending strongly on the specific reordering of the elements (possibly due to cache coherence or access-order optimizations); we present in the table two different permutations to demonstrate this. For example, in the tests that we performed, ITK showed a ratio of about 5.3 in the computation time for different permutation access orders on the Linux system. The CISST implementation also yielded significant time variations, making comparison between CISST and ITK difficult. Similarly inconclusive results occurred in a few other tests of the *permute* operation, which are not shown here.

Note, on the other hand, that the CISST *nArray* arithmetic engine was consistently faster than a similar computation in ITK. If we account for the filter extraction time, CISST was 65% to 74% faster in computing the addition of two arrays; if we do not account for the extraction time, CISST was consistently 5% to 30% faster.

To obtain the time ITK took to add two arrays, we had to isolate the time the `AddImageFilter` operation took to execute from the time the entire pipeline took to execute. In ITK, the timing for `AddImageFilter` should normally include the time it takes to update the two `RegionOfInterestImageFilter` objects, which were its input sources in our tests. However, we needed to isolate these in order to consider the time it takes to add the outputs only. Therefore, we measured three different times with the `AddImageFilter` using the following computations: (a) `Update()` the two region of interest filters (extraction); (b) add the regions of interest once they were extracted; and (c) `Update()` the final sum image after the two input sets were `Modified()`, triggering an implicit `Update()` of the region of interest filters. The times (a) and (b) do not necessarily add up to the time (c), since all three times were measured as "atomic" operations. Note that the parent container and region of interest sizes for the arithmetic operation benchmark are slightly different from the ones we used for the overlay benchmarks.

The results show that for the isolated layout manipulations, it is hard to determine in advance what the computation time will be. There are dependencies on the sizes of the containers (the results from these tests are not included here) and on the order of element access. Nevertheless, CISST is not consistently outperformed by ITK. For more complex operations, such as array arithmetic, the CISST overlay structures can perform much faster than ITK's filters, even when narrowed down to the actual evaluation of the result. In addition, the overlays use memory more efficiently because they do not require storage space for the overlays.

As we also show in the last example of Table 3, the CISST package provides overloaded arithmetic operators, which to many users are more intuitive than methods or filters. The overloaded operators are generally less efficient than named methods, however, because they require the creation of a temporary object to hold the computational outcome which is assigned to the final container object after evaluation, while the named method directly stores its output to the final container. This is a general weakness of the C++ language which can be overcome using expression analysis structures (some examples are in the Blitz++ library [5]). Even so, evaluating an overloaded operator on an *nArray* container is more efficient than the full cycle of evaluation in ITK.

Through benchmarking these two toolkits, we have found that the algorithms implemented in CISST *nArray* are comparable to, and in a wide variety of cases outperform, those of ITK. It is important to note, though, that this is not an exhaustive benchmark comparison of the two toolkits. Nevertheless, the overlay concept and the engine algorithms are powerful tools for improving the efficiency of managing multidimensional data sets.

5 Final Words

Our goal in designing the CISST *nArray* package was to provide a cross-platform software library for multidimensional arrays that is computationally efficient, easy to learn, and easy to extend. The performance of the traversal algorithms implemented in CISST is comparable to or better than another popular library used in medical image processing, ITK. The use of overlay arrays reduces the computational overhead incurred by ITK when subarray extraction is combined with other operations, such as array arithmetics.

Researchers at the ERC-CISST laboratory are already using the *nArray* library in medical image processing and statistical analysis of multidimensional data.

Future development plans of the CISST *nArray* library include: a redesign of the expression engines to optimize calculations involving containers with flat or partially-flat layouts; improving the interoperability of *nArray* and lower-dimension containers, i.e., 1-D vectors and 2-D matrices; and implementing the techniques developed for *nArray*s in those lower-dimension containers.

6 Acknowledgments

This work is supported by NSF ERC Grant 9731478.

References

[1] Ibanez, Schroeder, Ng, Cates: The ITK Software Guide. Kitware, Inc. ISBN 1-930934-15-7. 1

[2] The CISST Software Package. On the web: http://www.cisst.org/cisst. 1.2

[3] The VxL Libraries. On the web: http://vxl.sourceforge.net. 1.2

[4] Veldhuizen, T.: Using C++ template metaprograms. C++ Report 7 (1995)3643 Reprinted in C++ Gems, ed. Stanley Lippman. 1.2

[5] The Blitz++ library. On the web: http://www.oonumerics.org/blitz/. 1.2, 4.3

Multidimensional Arrays and the *nArray* Package

Ofri Sadowsky, Daniel Li, Anton Deguet, Peter Kazanzides

July 2, 2007

Department of Computer Science, Johns Hopkins University
email:ofri@cs.jhu.edu

Abstract

At the Johns Hopkins University's Engineering Research Center for Computer-Integrated Surgical Systems and Technology (ERC-CISST) laboratory, we have designed and developed a platform-independent C++ software package, called the *nArray* library, that provides a unified framework for efficiently working with multidimensional data sets. In this paper, we present and discuss the core elements of the library, including its intuitive and uniform API, efficient arithmetic engine algorithm, and efficient sub-volume algorithm. We then compare the performance of the *nArray* library with that of an existing multidimensional array toolkit, ITK. We conclude that the *nArray* library is more efficient than ITK in many situations, especially in operations on sub-arrays, and that the two packages have comparable performance in many other scenarios. The underlying algorithms, if incorporated in ITK, can help improve its performance.

1 Introduction

Multidimensional data sets are becoming increasingly prevalent in today's world, especially in the fields of computer-integrated surgery and image processing. A canonical example in image processing is a collection of three-dimensional CT scans over time that defines a four-dimensional data set. A three-dimensional image whose pixels each have a vector quantity, such as color or direction, also defines a four-dimensional data set. Such multidimensional data sets grow in size extremely quickly, and so it is important to have a software package that efficiently handles them.

At the Johns Hopkins University's Engineering Research Center for Computer-Integrated Surgical Systems and Technology (ERC-CISST) laboratory, we have designed and developed a platform-independent C++ software package, called the *nArray* (pronounced "EN-array") library, that provides a unified framework for efficiently working with multidimensional data sets. In addition to standard features such as STL-compatible iterators to traverse the data sets, the *nArray* library also provides immutable classes for safe data handling, efficient referencing of sub-volumes of data, layout manipulations, and an extensible generic computational engine.

In this article, we explore the unique layout of the *nArray* library, focusing specifically on the computational engine and the efficient sub-volume algorithm. We then compare the efficiency of the *nArray* package with an existing toolkit with multidimensional array support, the National Library of Medicine's Insight Toolkit

(ITK) [1]. Through this paper, we will see how the efficiency of the *nArray* package proves to be an effective tool for managing multidimensional data sets.

1.1 Motivation

The conventional C notation for a multidimensional array looks similar to the following:

```
unsigned int *** uintVolume;   /* this is a "three-dimensional" array */
```

Elaborate allocation and deallocation methods are required to properly manage such a memory layout. Ultimately, the actual data typically is allocated as a single, continuous memory block, i.e., a "flat" structure. Smaller arrays of pointers are then used to dereference portions, or "slices," of the block. It is clear from the notation that the higher the dimensionality of the dataset, the more complicated the "bookkeeping" of its layout becomes. A main goal in designing the *nArray* package was to simplify this process for the user by providing a templated uniform interface. For example, the same three-dimensional array created with an *nArray* container looks like:

```
/* here the array's type and dimension are template parameters */
vctDynamicNArray<unsigned int, 3> uintVolume;
```

nArray containers of all dimensions have a common interface, i.e., methods and operations. This makes the learning curve shorter, gives scalability, and helps with the debugging. To support this generic usage, we have developed and implemented a set of algorithms to address the issues involved with bookkeeping. Some of these algorithms will be outlined in this paper later.

1.2 Inspiration

APIs and software tools that deal with multidimensional data sets have been around for a while. MATLAB® has many advantages in "rapid prototyping" of algorithms, e.g. in signal processing and in the running of interactive processes, but its poor handling of data structures, flow control, and memory allocation control makes it unsuitable for real-time application development. Other interpreted languages, such as Python, offer a similar functionality for similar performance costs. ITK has gained popularity in recent years in the medical image processing community. It is designed as a *data flow* oriented toolkit, where components are linked in a processing pipeline and final outputs are produced at the end of the pipeline chain. Our feeling is that this programming model is less intuitive to many C/C++ programmers and that it does not provide enough low-level access for optimizing one's code.

The *nArray* package was built as an extension to our prior development of the `cisstVector` library of vectors and matrices written in C++ [2, 3]. Some of the functionality of *nArray*s involves the use of vectors from this library. The `cisstVector` library was inspired by existing toolkits in C++, such as VNL [4] and the Standard Template Library, in many of its aspects. We have tried to implement the good concepts in these libraries and improve those that seemed to need improvement.

A discussion of generic programming cannot be complete without mentioning the work of Todd Veldhuizen, the "father" of template metaprogramming [5]. The `cisstVector` library relies on metaprogramming structures, though different from Veldhuizen's Blitz++ library [6].

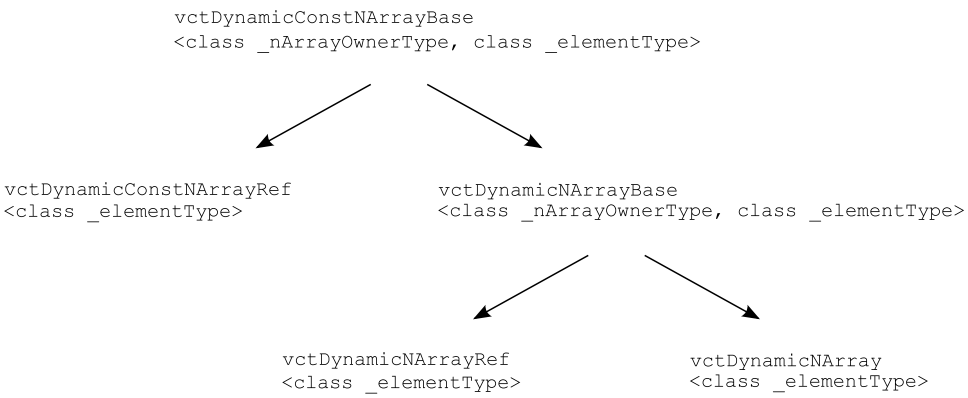

Figure 1: The *nArray* class inheritance hierarchy.

Finally, the development of the *nArray* computational engines, presented briefly in this paper, was an extension of an idea contributed by Robert Jacques, currently at the Johns Hopkins University. Jacques suggested an improvement to our matrix computation engines, which we later extended to *nArray*s.

2 Library Elements

2.1 *nArray* class hierarchy

The *nArray* package interfaces with a given data set through one of three *nArray* containers. These three API-level containers are organized via the five-class inheritance hierarchy shown in Figure 1. Shared methods are defined in the base classes and are inherited by the three child classes, which correspond to the three API-level containers. This inheritance hierarchy maintains a uniform interface across the three *nArray* containers.

The two base classes `vctDynamicConstNArrayBase` and `vctDynamicNArrayBase` contain the bulk of the API methods. The first includes only immutable methods, while the second extends it with mutable methods. These two classes are only accessible for the end-user through the following specializations. The classes `vctDynamicConstNArrayRef` and `vctDynamicNArrayRef` are *overlay* objects, which can be used to access externally allocated memory layouts as if they were multidimensional arrays. For example, `vctDynamicConstNArrayRef` is an immutable overlay, providing read-only structured access to a memory block identified by a `const _elementType *` (i.e., an address). Finally, the class `vctDynamicNArray` is an *allocating* object, and it allocates and releases a storage memory block.

2.2 *nArray* API sampler

The `cisstVector` library, of which the *nArray* package is an extension, was designed with the Abstract Data Type programming paradigm in mind. This means that the objects in the library are data containers and operations between them are methods of their classes. The "ideal" ADT paradigm would use arithmetic operators to resemble a mathematical notation in the programming language wherever possible. To this end, the *nArray* containers use *named methods* for all of its operations, while in applicable cases, overloaded operators that use the named methods as subroutines are additionally defined.

Operation Description	CISST Code	Equivalent C++ Notation	Overloaded Operator	Immutable Array Operands

Addition (two containers or a container and a scalar)	`c1.SumOf(c2, c3);`
	`c1.SumOf(c2, s);`
	`c1.Add(c2);`
	
	`c1.Add(s);`
		`c1 = c2 + c3;`
		`c1 = c2 + s;`
		`c1 += c2;`
		
		`c1 += s;`
			Yes
			Yes
			Yes
			
			Yes
				`c2,c3`
				
				`c2`

Elementwise multiplication	`c1.ElementwiseProductOf( c2, c3);`
	`c1.ElementwiseMultiply(c2);`
		`c1[i] = c2[i]*c3[i];`
		`c1[i] *= c2[i];`
			No
			
			No
				`c2,c3`
				
				`c2`

Division by scalar	`c1.RatioOf(c2, s);` `c.Divide(s);`	`c1 = c2 / s;` `c1 /= s;`	Yes Yes	`c2`
Sum of elements	`s = c.SumOfElements();`	N/A	No	`c`
Largest element	`s = c.MaxElement();`	N/A	No	`c`

Table 1: A sample of the operations in the *nArray* package.

Table 1 shows a few examples of the functional `cisstVector` API. When an equivalent C++ expression is available, it is listed in the third column. If `cisstVector` provides an overloaded operator, it is indicated in the fourth column. The fifth column indicates which operands are immutable, that is, not modifiable by the operation. Usually, if an operand is immutable, it appears in the method's signature as a template `vctDynamicConstNArrayBase`. This allows any of the *nArray* classes to be passed as the actual method parameter. In the table, `c` stands for a generic container, which in the `cisstVector` library can be a (fixed-size or dynamic) vector, a matrix, or an *nArray*; `s` stands for scalars of the same type as that of the array elements, e.g. `double`. Different operands, when they are involved, are distinguished by number. The table shows only a small selection of operations; the complete list is in the library's documentation.

2.3 Layout manipulation

A central feature of the *nArray* containers is their ability to reference other *nArray*s using different *layouts*, or subsections of an existing *nArray* . This is achieved by configuring an overlay *nArray* to span the desired region of an existing *nArray* container. Overlaying allows the user to operate on array elements in-place, without copying them out and in.

A new layout may have the same dimension as its parent container, focusing on a smaller region; or it may have a lower dimension. We call the region focusing a *window* and the dimensionality reduction a *slice*. Combining windows and slices allows the user to specify any subsection of any dimension of an existing *nArray* container. Another useful configuration is changing the order in which *nArray* elements are accessed; this is called *permuting* the order of element access, and is similar to MATLAB's `permute` function, without the memory allocation and copy overhead.[1]

Let us demonstrate these ideas through an example. Suppose we want to select a small region of interest in a sagittal cross-section of a CT volume. Since a CT volume is by default created by stacking transverse cross-

[1]The *nArray* library API uses the term `Subarray` for what we define in this paper as the *window* operation. Throughout the text of this paper, we will stick to the *window* terminology, although the code samples we include continue to refer to it as a `Subarray`.

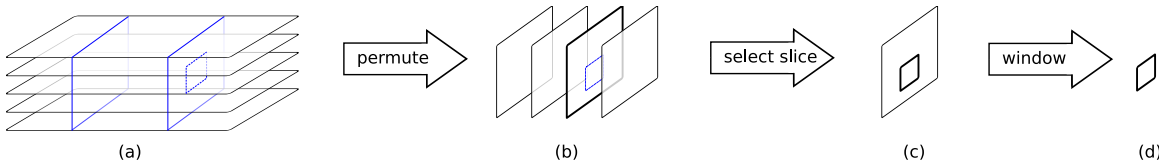

Figure 2: An illustration of *nArray*'s layout operations: permute, slice, window. (a) *Initial stack of transverse images.* Sagittal cross-sections are shown with blue lines; the region of interest is shown with blue dashes. (b) *Sagittal cross sections.* The slice of interest is shown with thick lines; the region of interest is shown with blue dashes. (c) *Slice of interest.* The region of interest is shown with thick lines. (d) *Region of interest.*

sections, we first perform a *permutation* on the volume to orient the order of access of elements correctly. Next, we perform a *slice* operation on the permuted volume to obtain a two-dimensional container that holds the desired sagittal slice. Finally, to focus on a particular region of the sagittal slice, we perform a *window* operation on the slice. This is illustrated in Figure 2 and in the code listing in Table 2.

Breaking up the overlay concept into these three distinct operations has two direct benefits: it creates easier-to-debug code, and it gives one flexibility in how one creates new layouts.

2.4 Strides

The windowed overlay and dimension permutation are achieved through a careful definition of *strides*, which indicate the increment in memory address between adjacent array elements in each stride's corresponding dimension; each stride can be either a positive or negative integer. Every *nArray* object includes a vector of stride values whose length is equal to the number of dimensions. In a simple, flat layout, the fastest changing dimension has a stride of 1, and the stride of any higher dimension is equal to the number of elements in all the lower-dimension slices. In a window overlay, however, a stride in one dimension can be larger than the number of elements in a lower dimension, which means that more than one memory cell is "skipped" in order to move from one slice to the next. When permutations are applied, the order of the strides changes arbitrarily. An example of overlay strides is shown in Figure 3.

3 Algorithms

3.1 *nArray* engines

The *nArray* engines are a set of classes and functions optimized to efficiently traverse the elements of *nArray* containers. The engines operate on both memory-allocating and overlaying containers, and the interface to call the engines on either type of container is identical, so the differences between operating on the two types of containers are transparent to the user. Also, the engines handle both contiguous and non-contiguous memory blocks.

Almost always the containers used with the engines play the part of operands of an operation (such as the addition of two arrays, with the sum stored into a third). Each engine is designed to support a specific combination of operands, such as unary, binary, or store-back operations (e.g. the `Abs`, `+` and `+=` operators), as well as various operand types (e.g. arrays, scalars, etc.).

Expression structure in the engines is abstracted and encapsulated. Any operation defined in the *nArray* API

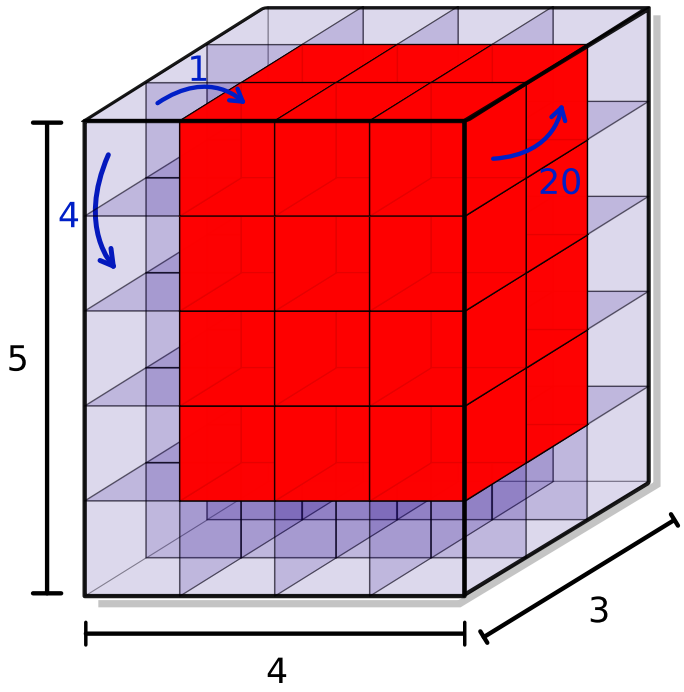

Figure 3: An example of strides in a three-dimensional array. The purple block of sizes of $3 \times 5 \times 4$ cells (the order of dimensions is Z,Y,X) is the "parent container". The red block, with $2 \times 4 \times 3$ cells, is a window overlay of the parent container. In both blocks, the memory strides between adjacent elements are (20, 4, 1) in corresponding dimension order.

```
// Create original CT volume.  The sizes are specified
// in the Z-Y-X order.  nsize_type is a "fixed-size" vector.
NArrayType::nsize_type originalSize(240, 512, 512);
NArrayType originalCTVolume(originalSize);

// load volume with CT data
/* ... */

// Create a permuted overlay of the original CT.
// Dimension 0 of the permuted array corresponds to dimension
// 2 of the original volume, that is, the X dimension, or sagittal
// cross sections.
NArrayType::nsize_type orderOfDimensions(2, 0, 1);
NArrayType::PermutationRefType sagittalOrientation;
sagittalOrientation.PermutationOf(originalCTVolume, orderOfDimensions);

// Select slice of interest in dimension 0.
NArrayType::size_type dimension = 0;
NArrayType::size_type index = 172;
NArrayType::SliceRefType sliceOfInterest;
sliceOfInterest.SliceOf(sagittalOrientation, dimension, index);

// Select region of interest.
NArrayType::SliceRefType::nsize_type regionSize(192, 192);
NArrayType::SliceRefType::nsize_type regionStart(10, 50);
NArrayType::SliceRefType::SubarrayRefType regionOfInterest;
regionOfInterest.SubarrayOf(sliceOfInterest, regionStart, regionSize);
```

Table 2: Code example of defining layout manipulation overlays.

only requires the user to select the appropriate expression structure and provide the relevant operators. For example, computing the maximum element, the sum of elements, and the sum of squares of an *nArray* all make use of the same engine because they all have the same expression structure: compute a scalar function from all the elements of one *nArray* . The three operations differ only in the operations to be performed on or between the elements. For example, in *sum of elements* and *maximum*, there is no operation on individual elements, but in *sum of squares*, the square of each element is computed; in *sum of elements* and *sum of squares*, there is also an addition operation between elements, and in *maximum* there is a *max* operation between elements. Beyond these differences, the structure of all three operations is identical, and this similarity is expressed in the engines. In practice, the operations (*add*, *max* ...) are passed to the engines as template parameters whereas the signature of the engine function is identical for all operations. In this way, the engines make the containers very straightforward to use and encourage the creation of easy-to-debug code.

The engines have several features that contribute to the overall efficiency and scalability of the package while hiding the details of the traversal algorithm from the caller function. They can be viewed as an abstraction of a multi-level nested loop.

3.2 Engine algorithm

The *nArray* engine uses a pointer, which we call the "current" pointer, to traverse an *nArray* container, much like an STL iterator, which traverses a container from beginning to end in sequential order. However, since the engine must operate on both memory-allocating and overlay *nArray* containers, the algorithm is not as simple as having the pointer run through the container's memory block from start to finish. Instead, when the pointer reaches the end of a dimension of the container, the engine must know by how many address spaces to increment the pointer in order to reach the next element in that dimension.

In designing the engine algorithm, we had to address three main issues. First, we could not use a nesting structure to loop through a container because there is an arbitrary number of dimensions in that container. We resolved this by using a vector of the same size as the number of dimensions of the container to store "target" pointers, which mark the end of each dimension. When the current pointer reaches a target pointer, the engine "wraps" the current pointer around this dimension by incrementing the pointer by the appropriate number of address spaces. The wrap-around test is performed in a recursive fashion, traversing down the list of targets until the appropriate one is found. This replaces the conventional nested loop code structure.

The second issue was how to efficiently wrap around the current pointer when it reaches a target pointer. This is resolved by having the engine precalculate the dimension offsets (the "stride to the next dimension", or STND, values) using the stride values.

Finally, the engines also have to update the target pointers accordingly when the wrap-around occurs. This is shown in the `IncrementPointers` method below. `IncrementPointers` is the equivalent of the "loop header", and it is called to move the current pointer to the next element. Notice that `IncrementPointers` returns the number of dimensions that have been exhausted and wrapped-around. We take advantage of this result to wrap-around the "current pointer" for other operand containers that may be involved. This topic, however, is not covered in this paper.

The result is an efficient and versatile engine algorithm that is capable of running on both memory-allocating and reference *nArray* containers. The source code for the algorithm is provided below. It is written in C-style notation.

```
// The PreProcess function computes the wrap-around strides (STND)
// and the target pointers before the engine loop is begun
void PreProcess(const unsigned int numDimensions,
  const unsigned int arraySizes[], const int arrayStrides[],
  const _elementPointer basePtr, int arraySTND[],
  _elementPointer arrayTargets[])
{
  unsigned int i;

  for (i = 0; i < numDimensions; ++i)
  {
    unsigned int span = arraySizes[i] * arrayStrides[i];
    /* calculate initial placement of target pointers */
    arrayTargets[i] = basePtr + span;
    /* calculate STND */
    arraySTND[i] = (i != 0) ? arrayStrides[i-1] - span : 0;
   }
}

// The function IncrementPointers checks which dimensions are
```

```
// exhausted, performs a wrap-around of the current pointer,
// and recomputes new targets if they need to be updated.  It
// returns the number of dimensions that were exhausted.
unsigned int IncrementPointers(const unsigned int numDimensions,
  _elementPointer targets[],  _elementPointer  & currentPointer,
  const int strides[],  const int stnd[])
  {
    unsigned int i = numDimensions - 1;
    unsigned int wrapCounter = 0;
    currentPointer += strides[numDimensions - 1];
    while (currentPointer == targets[i]) {  // the i-th dimension is exhausted
      currentPointer += stnd[i];
      ++wrapCounter;
      --i;
      if (wrapCounter == numDimensions) // exhausted all elements
        return wrapCounter;
    }

    // if no dimensions were exhausted, we can return immediately
    if (wrapCounter == 0)
      return wrapCounter;

    // now, update the targets forward from the current pointer
    ++i;
    do {
      targets[i] = currentPointer + (strides[i-1] - stnd[i]);
      ++i;
    } while (i < numDimensions);

    return wrapCounter;
  }
```

As an example, consider the *nArray* container in Figure 3. The red overlaying window container has size $2 \times 4 \times 3$ and stride values (20, 4, 1). The values the engine would calculate for this container are:

```
Sizes                2   4   3
Strides             20   4   1
STND                 0   4   1
Target offsets      40  16   3
```

3.3 Iterators

Iterators are a widely-used method of traversing all the elements of a container, utilizing an object to keep tabs on the location. We created the *nArray* iterators to conform with the Standard Template Library's specification for a random access iterator.

The *nArray* iterators are designed to handle complicated layouts such as dimension permutations and non-contiguous memory blocks. While a mechanism similar to the *nArray* engines could be encapsulated as an object, we believe this might be overkill for iterators. Instead, an *nArray* iterator keeps a "meta-index" internally, which indicates the sequential position of an element in the container. The iterator converts the meta-index to a "position index", which is a tuple of zero-based coordinates, similar to our intuitive notion

of a multidimensional index. The position index then is converted again to an address offset from the array's base pointer by computing its dot product with the strides of the *nArray* .

To explain this mechanism, let us first write down a recursive function, presented below, which takes as input the array of sizes (i.e., numbers of elements in each dimension) of the *nArray* , the array of strides in the respective dimensions, and a meta-index, and returns an offset from the base pointer and a small array of indices. For simplicity of presentation, the function is written in C-style notation.

```
(0) unsigned int MetaIndexToOffsetAndMultiIndex(const unsigned int numDimensions,
       const unsigned int arraySizes[], const int arrayStrides[],
       const unsigned int metaIndex, unsigned int multiIndex[])
    {
(1)    if (numDimensions == 0)
(2)        return 0;
(3)    const unsigned int d = numDimensions-1;
(4)    multiIndex[d] = metaIndex % arraySizes[d];
(5)    const unsigned int dContribution = strides[d] * multiIndex[d];
(6)    return dContribution +
           MetaIndexToOffsetAndMultiIndex(numDimensions-1,
           arraySizes, arrayStrides, metaIndex / arraySizes[d], multiIndex);
    }
```

As an example, let us follow the operation of this function on a three-dimensional flat container of size $3 \times 5 \times 4$ (see Figure 3) with corresponding strides of (20, 4, 1). We start with a meta-index `metaIndex=23`. Unknown values are written as question marks. The number preceding each line refers to the corresponding line number in the code above. For simplicity, we do not replicate the array parameters through the nesting of the recursion.

```
(0) numDimensions = 3, arraySizes = [3, 5, 4], arrayStrides = [20, 4, 1],
       metaIndex = 23, multiIndex = [?, ?, ?]
(3) d = 2
(4) multiIndex = [?, ?, 23%5] = [?, ?, 3]
(5) dContribution = arrayStrides[d] * multiIndex[d] = 1 * 3 = 3
      (0) numDimensions = 2, metaIndex = 23/5 = 4
      (3) d = 1
      (4) multiIndex = [?, 4%4, 3] = [?, 0, 3]
      (5) dContribution = arrayStrides[d] * multiIndex[d] = 4 * 0 = 0
            (0) numDimensions = 1, metaIndex = 4/4 = 1
            (3) d = 0
            (4) multiIndex = [1%3, 0, 3] = [1, 0, 3]
            (5) dContribution = arrayStrides[d] * multiIndex[d] = 20 * 1 = 20
                  (0) numDimensions = 0, metaIndex = 1 / 3 = 0
                  (2) return 0
            (6) return dContribution + 0 = 20
      (6) return dContribution + 20 = 20
(6) return dContribution + 20 = 23
```

Since the layout in this example is flat, the final offset is equal to 23, which is the original meta-index. However, applying this mechanism to a configuration with different strides still computes a correct offset, which may be different from the meta-index. The final values in `multiIndex` are (1, 0, 3), which are the zero-based "coordinates" of the element whose meta-index is 23 in an ordinary indexing mode.

It is worth noting that a memory vs. runtime tradeoff is expected when comparing the engines and the iterators, with the engines being the more runtime-efficient. In principle, either mechanism could replace the other, but in practice, the engines are better optimized for runtime than a replication of iterators is, even if a similar wrap-around mechanism were implemented for the iterators; this is because the wrap-around decision inside an engine needs only to be made once, while individual iterators must decide on the wrap-around independently. Therefore, most of the operations that need to be performed on an *nArray* should be done via the engines and not the iterators.

4 Performance Benchmarks

4.1 Performance discussion

As a general rule, more regular data layouts can be processed with faster algorithms. With regards to multidimensional arrays, a flat layout can be processed faster than another layout, such as a non-contiguous block or a layout manipulation overlay, for the same data size. This is because the overhead of keeping tabs on the pointer position is easier in the flat case.

When we compare the performance of *nArray* operations via expression engines with other software packages, we have to take care to compare features on level terms. If, for example, a certain library supports only flat layouts and has a fast algorithm for evaluating expressions on them, then it should be compared with the performance of the CISST package on flat containers, i.e., vectors. On the other hand, if a library supports certain layout manipulators, such as regions of interest, then we can compare them with the engine methods in the *nArray* package. In addition, we can compare the *nArray* engines with the expression engines for lower-dimension containers in the CISST package, namely, dynamic vectors. This comparison should provide an estimate of the bookkeeping overhead involved with traversing the *nArray* .

Considering this overhead, the efficiency of using overlay arrays can depend on the frequency of their use. Evaluating a single expression involving a layout manipulation can be faster using the overlay structure than if the manipulated data layout needs first to be copied to a second container before the expression is evaluated. However, if many expressions involving the same immutable dataset are considered, it is usually more efficient to copy the elements once into a flat container and evaluate all the expressions using the new block. An important advantage of the *nArray* overlays is that they provide flexibility to the user in choosing the preferred processing method.

Likewise, if we want to compare the performance of overlays with a library that only supports copy-and-evaluate implementations, the timing of one expression involving an overlay should be compared with the timing of copying and evaluating the expression, combined, and not just with the time of evaluating. If new memory allocation is needed before the copy, then the timing for memory allocation must be counted as well.

4.2 Benchmark specifications

We performed two kinds of benchmark runs. The first consisted of extracting a manipulated layout from a parent container and copying its elements into another container. Considering the different software architectures, this simple operation was chosen to highlight the performance of the array traversal algorithm in different configurations. Notice that only two data containers were involved: the parent and the destination, and no arithmetic operations were performed on the data elements.

We compared the CISST *nArray* package with ITK's Image class. Using both libraries, we created the following test cases: a four-dimensional array; a smaller region of interest ("window") which is also four-dimensional; a three-dimensional slice of the larger array; and an axis permutation, having the same number of elements as the parent container but in a different access order. In the *nArray* package, each layout was created as an overlay on the parent container. We called the `Assign()` method to read elements from an overlay array and write them to a memory-owning array. In ITK, each operation was represented as a "filter" object, which stores a copy of the outcome. The evaluation of the expression was triggered by calling the method `Update()` on the filter.

The second kind of benchmark involved container arithmetic, namely, adding two four-dimensional arrays into a third four-dimensional array. Here, the two input operands were "windows" or subregions of two larger parent containers. The output container was allocated independently. The purpose of this benchmark was to compare the *nArray*'s overlay approach with the more traditional copy-out approach in ITK. There are two ways to compute the sum of two *nArray*s, as can be seen in Table 1: (a) apply the method `SumOf` to a third *nArray* object (the result operand); or (b) use an overloaded `operator +`. We compared both methods to demonstrate the performance cost of using an overloaded operator. In ITK, we created an `AddImageFilter` object to compute and store the sum; its inputs were the outputs of two `RegionOfInterestImageFilter` objects, which in turn copied the contents of the regions to an internal storage.

The source files for the benchmarking programs are as follows.

Benchmark	***nArray***	**ITK**
Layout manipulations	`Subarray_nArray_Benchmark.cpp`	`Subarray_ITK_Benchmark.cpp`
Array sum	`ImageAdd_nArray_Benchmark.cpp`	`ImageAdd_ITK_Benchmark.cpp`

4.3 Performance evaluation

In the C++ programs listed above, the operations of interest were surrounded by calls to a "stopwatch" object with `Start()` and `Stop()` methods to measure the evaluation time. The stopwatch uses the high-frequency timer in either Windows or Linux. The output was rounded to a millisecond precision. Both the CISST -based and the ITK-based programs were compiled and run on the following systems.

- Windows XP workstation: Two dual-core Intel Xeon CPU, 3.06 GHz, 2.00 GB RAM. Compiler: Microsoft Visual Studio 7.1
- Linux server: Two dual-core Intel Xeon CPU 64 bit, 2.0 GHz, 6.0 GB RAM, 4 MB cache per CPU. Ubuntu Linux distribution. Compiler: gcc 4.1.2

The compilation on both systems was in "Release" mode, using compiler optimizations for speed. The accumulated times for 30 repetitions of the test are summarized in Table 3; all times are in milliseconds. The data sizes are as given in the source files listed above.

For the *slice* operation, the CISST implementation consistently performed 46% to 53% faster than ITK. In the other tests, however, it was more difficult to obtain a consistent comparison. For the *window* operation, CISST was about 41% faster on the Windows build but about 16% slower on Linux. The performance of the *permute* operation was also inconsistent, depending strongly on the specific reordering of the elements (possibly due to cache coherence or access-order optimizations); we present in the table two different permutations to demonstrate this. For example, in the tests that we performed, ITK showed a ratio of about 5.3 in the computation time for different permutation access orders on the Linux system. The CISST implementation also yielded significant time variations, making comparison between CISST and ITK difficult.

Similarly inconclusive results occurred in a few other tests of the *permute* operation, which are not shown here.

Note, on the other hand, that the CISST *nArray* arithmetic engine was consistently faster than a similar computation in ITK. If we account for the filter extraction time, CISST was 65% to 74% faster in computing the addition of two arrays; if we do not account for the extraction time, CISST was consistently 5% to 30% faster.

To obtain the time ITK took to add two arrays, we had to isolate the time the `AddImageFilter` operation took to execute from the time the entire pipeline took to execute. In ITK, the timing for `AddImageFilter` should normally include the time it takes to update the two `RegionOfInterestImageFilter` objects, which were its input sources in our tests. However, we needed to isolate these in order to consider the time it takes to add the outputs only. Therefore, we measured three different times with the `AddImageFilter` using the following computations: (a) `Update()` the two region of interest filters (extraction); (b) add the regions of interest once they were extracted; and (c) `Update()` the final sum image after the two input sets were `Modified()`, triggering an implicit `Update()` of the region of interest filters. The times (a) and (b) do not necessarily add up to the time (c), since all three times were measured as "atomic" operations. Note that the parent container and region of interest sizes for the arithmetic operation benchmark are slightly different from the ones we used for the overlay benchmarks.

The results show that for the isolated layout manipulations, it is hard to determine in advance what the computation time will be. There are dependencies on the sizes of the containers (the results from these tests are not included here) and on the order of element access. Nevertheless, CISST is not consistently outperformed by ITK. For more complex operations, such as array arithmetic, the CISST overlay structures can perform much faster than ITK's filters, even when narrowed down to the actual evaluation of the result. In addition, the overlays use memory more efficiently because they do not require storage space for the overlays.

As we also show in the last example of Table 3, the CISST package provides overloaded arithmetic operators, which to many users are more intuitive than methods or filters. The overloaded operators are generally less efficient than named methods, however, because they require the creation of a temporary object to hold the computational outcome which is assigned to the final container object after evaluation, while the named method directly stores its output to the final container. This is a general weakness of the C++ language which can be overcome using expression analysis structures (some examples are in the Blitz++ library [6]). Even so, evaluating an overloaded operator on an *nArray* container is more efficient than the full cycle of

Operation	**CISST time, Windows**	**ITK time, Windows**	**CISST time, Linux**	**ITK time, Linux**
Window	8,814	14,860	4,093	3,538
Slice	250	465	122	261
Permute (3, 1, 2, 0)	202,342	183,173	73,125	99,123
Permute (3, 0, 2, 1)	193,742	95,403	58,738	18,767
Add 4D arrays	12,203	(a) 28,674 (b) 17,501 (c) 46,262	3,757	(a) 6,875 (b) 3,950 (c) 10,848
Operator +	19,044	N/A	8,790	N/A

Table 3: Benchmark times for the CISST *nArray* and ITK operations on Windows and Linux systems. The times are in milliseconds for 30 repetitions of the operation.

evaluation in ITK.

Through benchmarking these two toolkits, we have found that the algorithms implemented in CISST *nArray* are comparable to, and in a wide variety of cases outperform, those of ITK. It is important to note, though, that this is not an exhaustive benchmark comparison of the two toolkits. Nevertheless, the overlay concept and the engine algorithms are powerful tools for improving the efficiency of managing multidimensional data sets, and their implementation may improve the performance of ITK in cases it sacrifices now.

5 Final Words

Our goal in designing the CISST *nArray* package was to provide a cross-platform software library for multidimensional arrays that is computationally efficient, easy to learn, and easy to extend. The performance of the traversal algorithms implemented in CISST is comparable to or better than another popular library used in medical image processing, ITK. The use of overlay arrays reduces the computational overhead incurred by ITK when subarray extraction is combined with other operations, such as array arithmetics.

Researchers at the ERC-CISST laboratory are already using the *nArray* library in medical image processing and statistical analysis of multidimensional data.

Future development plans of the CISST *nArray* library include: a redesign of the expression engines to optimize calculations involving containers with flat or partially-flat layouts; improving the interoperability of *nArray* and lower-dimension containers, i.e., 1-D vectors and 2-D matrices; and implementing the techniques developed for *nArray*s in those lower-dimension containers. Additionally, we are examining ways to create inter-opearbility between the CISST package and ITK.

6 Acknowledgments

This work is supported by NSF ERC Grant 9731748.

References

[1] Ibanez, Schroeder, Ng, Cates: The ITK Software Guide. Kitware, Inc. ISBN 1-930934-15-7. 1

[2] Kazanzides, P., Deguet, A., Kapoor, A., Sadowsky, O., LaMora, A., and Taylor, R.H.: Development of open source software for computer-assisted intervention systems. ISC/NAMIC/MICCAI Workshop on Open-Source Software (2005). 1.2

[3] The CISST Software Package. On the web: http://www.cisst.org/cisst. 1.2

[4] The VxL Libraries. On the web: http://vxl.sourceforge.net. 1.2

[5] Veldhuizen, T.: Using C++ template metaprograms. C++ Report 7 (1995)3643 Reprinted in C++ Gems, ed. Stanley Lippman. 1.2

[6] The Blitz++ library. On the web: http://www.oonumerics.org/blitz/. 1.2, 4.3