Blitz++ User's Guide

A C++ class library for scientific computing
for version 0.8, 15 October 2004

Todd Veldhuizen

Table of Contents

Short Contents

1 Introduction

1.1 About this document

To use the Blitz++ library, you will need a compiler with near-ISO/ANSI C++ syntax support (see the following section for possible compilers). Information on what platforms are supported is available from `http://oonumerics.org/blitz/platforms/`. To download Blitz++, please go to the download page at `http://oonumerics.org/blitz/download/`.

If you need to do something that Blitz++ doesn't support, see a possible improvement, or notice an error in the documentation, please send a note to one of the Blitz++ mailing lists (described later).

1.2 Platform notes

For up-to-date information on supported platforms, please consult the Blitz++ web page:

`http://oonumerics.org/blitz/platforms/`

The information in this document may be out of date.

1.2.1 KAI C++

Blitz++ was developed and tested using KAI C++ under AIX. It should (in theory) port to other KAI C++ platforms (Cray, SGI, HP, Sun, Linux, Compaq) without difficulty. Since KAI C++ uses an EDG front end, other EDG front-ended compilers (e.g. Comeau) should be able to compile Blitz++.

Recommended compile flags are:

```
+K3 -O2 --restrict --abstract_pointer --abstract_float -tused
```

Note that you cannot compile with `-tall` (this will generate lots of errors).

Under Linux, you may need the flag `-D__signed__=`. You should omit `-tused` since this template instantiation model is not supported by `gcc`, which is used as the back-end compiler.

1.2.2 Intel C++

Blitz++ compiles under fairly recent versions of the Intel C++ compiler (version 7.1 or 8.x) for Linux platforms, as well as the comparable plug-in compiler for Windows that can be used within the Microsoft Visual Studio IDE.

More information:

`http://www.intel.com/software/products/compilers/clin`

`http://www.intel.com/software/products/compilers/cwin`

1.2.3 Microsoft VS.NET 2003

Blitz++ has been ported to the C++ compiler within the Microsoft VS.NET 2003 compiler and IDE package. A zip archive containing an appropriate configuration header file and project files for building the Blitz library and all of the testsuite codes. Previous versions of the Microsoft C++ compiler within Visual Studio do not have the required C++ features needed by Blitz++ and are not supported. Blitz can be compiled under Visual Studio by using the Intel plug-in C++ compiler for Windows.

1.2.4 gcc

GCC (`g++`) is a free GNU C++ compiler. It compiles Blitz++ reliably; in fact, most Blitz++ development work is done with `g++`.

gcc may be downloaded from `http://www.gnu.org/software/gcc/gcc.html`.

If you are using gcc under Solaris, SunOS, or OSF/1, please see the 'README.binutils' file included in the distribution.

1.2.5 Metrowerks

Metrowerks is sort-of supported; see the platforms web page and the mailing lists for more information. Support for Metrowerks is no longer being actively maintained.

1.2.6 Compaq cxx

The Compaq C++ compiler version 6.x is supported.

1.2.7 Cray T3E/Cray T90/Cray C90/Cray J90

As of Version 0.2-alpha-02 of Blitz++, Version 3.0.0.0 of the Cray C++ compiler is supported (well, tolerated anyway). It seems to be based on an older version of the EDG front end, so some kludges are required. It doesn't support partial ordering of member templates, so slicing arrays requires the workaround described in Section Section 2.4.4 [Slicing combo], page 15. Portions of the standard library are missing, such as `<limits>`, `<complex>`, and `<set>`. This means you won't be able to use complex numbers (well, not the ISO/ANSI C++ versions anyway), numeric inquiry functions, or fast traversal orders.

These compilation flags are recommended:

```
-h instantiate=used
```

For optimization, you'll want:

```
-O3 -h aggress
```

The ability of the Cray C++ compiler to optimize away temporary objects is disappointing. It's not able to optimize away expression templates overhead or comma-delimited array initializers. Please note that support for compiling Blitz++ under the Cray C++ compiler is no longer being actively maintained.

1.3 How to download Blitz++

To download the Blitz++ library, go to the Blitz++ download page, at http://oonumerics.org/blitz/download/

But please read the section on supported platforms and compilers first.

1.4 Installation and porting

1.4.1 Installation

Blitz++ uses GNU Autoconf, which handles rewriting Makefiles for various platforms and compilers. It has greatly simplified installation and porting. Many thanks for John W. Eaton and Brendan Kehoe for their help with this.

To install blitz, unpack the 'blitz-VERSION.tar.gz' file (it will install into a subdirectory 'blitz-VERSION'). For example:

```
[tveldhui@n2001:~] 32: ls -l blitz*.gz
-rw-r--r--   1 tveldhui users      480953 Jun 23 15:20 blitz-0.5.tar.gz
[tveldhui@n2001:~] 33: gunzip blitz-0.5.tar.gz
[tveldhui@n2001:~] 34: tar xvf blitz-0.5.tar
blitz-0.5/CHANGELOG
blitz-0.5/COPYING
blitz-0.5/INSTALL
blitz-0.5/Makefile.in
blitz-0.5/README
```

```
blitz-0.5/THANKS
   .
   .
```

Then go into the main blitz directory, and type:

```
./configure CXX=[compiler]
```

where [compiler] is one of `xlc++`, `icpc`, `xlC`, `cxx`, `aCC`, `CC`, `g++`, `KCC`, `pgCC` or `FCC`.

You can also specify special command-line options for your compiler, using this syntax:

```
./configure CXX=g++ CXXFLAGS="-ftemplate-depth-50"
```

If you are interested in benchmarking, you may want to use the option `--with-blas=...` to specify the path where the blas library is found. Run the configure script with the option `--help` to see all the available options.

Once the configure script is done, you can do any of these things:

`make lib` Check the compiler and create '`libblitz.a`'.

`make check-testsuite`
Make the blitz library plus build and run the testsuite.

`make check-examples`
Make the blitz library plus build and run the examples.

`make check-benchmarks`
Make the blitz library plus build and run the benchmarks.

`make all` Do all of the above. This may take a **long** time.

`make install`
Build the blitz library and documentation and install, along with the blitz header files, in prefix directory.

Building the benchmark programs requires both a Fortran 77 and Fortran 90 compiler.

1.4.2 The Blitz++ directory tree

The main Blitz++ directory contains these subdirectories:

`blitz` Blitz++ headers and source code files

`random` Random number generators

`src` Source code for '`libblitz.a`'

`lib` Location of '`libblitz.a`'

`doc` Documentation in HTML and PostScript

`testsuite`
Testsuite programs

`examples` Example programs

`benchmarks`
Benchmark programs

1.4.3 Porting Blitz++

If you want to try porting Blitz++ to a new compiler or platform, I suggest the following approach:

- First check the Blitz++ web page to make sure you have the latest snapshot, and that someone hasn't already ported blitz to your platform.
- Install autoconf (from e.g. `ftp://prep.ai.mit.edu/pub/gnu`) if you don't have it already.

- Run the configure script with CXX=[compiler]. This will exercise your compiler to see what language features it supports. If it doesn't have member templates and enum computations, just give up. You may need to set CXXFLAGS to use compiler options that enable some language features.
- Once you know what compiler options are needed, you can make these the default settings for your C++ compiler. Make a backup of '`m4/ac_cxx_flags_preset.m4`', and then edit the file to add an appropriate case for your compiler. Invoke autoconf to regenerate the configure script. Then try configure with your new preset flags.

1.5 Compiling with Blitz++

1.5.1 Header files

Blitz++ follows an X-windows style convention for header files. All headers are referred to with a prefix of '`blitz`'. For example, to use the `Array<T,N>` class, one needs to include `<blitz/array.h>` instead of just `<array.h>`. To make this work, the main Blitz++ directory must be in your include path. For example, if Blitz++ was installed in '`/software/Blitz++`', you will need to compile with `-I /software/Blitz++`.

If you have root privileges, you may want to put in a symbolic link from the standard include path (e.g. '`/usr/include/blitz/`') to the `blitz` directory of the distribution. This will allow you to omit the `-I ...` option when compiling.

1.5.2 Linking to the Blitz++ library

The Blitz++ library file '`libblitz.a`' contains a few pieces of global data. You should ensure that the '`lib`' subdirectory of the Blitz++ distribution is in your library path (e.g. `-L/usr/local/blitz-0.5/lib`) and include `-lblitz` on your command line. If you use math functions, you should also compile with `-lm`.

1.5.3 An example Makefile

Here is a typical skeletal Makefile for compiling with Blitz++ under gcc:

```
# Path where Blitz++ is installed
BZDIR = /usr/local/blitz

CXX = g++

# Flags for optimized executables
# CXXFLAGS = -O2 -I$(BZDIR) -ftemplate-depth-30

# Flags for debugging
CXXFLAGS = -ftemplate-depth-30 -g -DBZ_DEBUG -I$(BZDIR)

LDFLAGS =
LIBS = -L$(BZDIR)/lib -lblitz -lm

TARGETS = myprogram1 myprogram2

.SUFFIXES: .o .cpp

.cpp.o:
    $(CXX) $(CXXFLAGS) -c $*.cpp

$(TARGETS):
    $(CXX) $(LDFLAGS) $@.o -o $@ $(LIBS)

all:
    $(TARGETS)
```

```
myprogram1:      myprogram1.o
myprogram2:      myprogram2.o

clean:
    -rm -f *.o $(TARGETS)
```

There are more example makefiles in the examples, testsuite, and benchmarks directories of the distribution.

1.5.4 Explicit instantiation

It is not possible to do explicit instantiation of Blitz++ arrays. If you aren't familiar with explicit instantiation of templates, then this fact will never bother you.

The reason is that explicit instantiation results in all members of a class template being instantiated. This is **not** the case for implicit instantiation, in which only required members are instantiated. The `Array<T,N>` class contains members which are not valid for all types `T`: for example, the binary AND operation `&=` is nonsensical if `T=float`. If you attempt to explicitly instantiate an array class, e.g.

```
template class Array<float,3>;
```

then you will be rewarded with many compile errors, due to methods such as `&=` which are nonsensical for `float`.

As some consolation, explicit instantiation would not be much help with Blitz++ arrays. The typical use for explicit instantiation is to instantiate all the templates you need in one compilation unit, and turn off implicit instantiation in the others – to avoid duplicate instantiations and reduce compile times. This is only possible if you can predict ahead of time what needs instantiation. Easy for simple templates, but impossible for classes like `Array`. Almost every line of code you write using `Array` will cause a different set of things to be implicitly instantiated.

1.6 Licensing terms

The Blitz++ library is licensed under both the GPL and the more permissive "Blitz++ Artistic License". Take your pick. They are detailed in GPL and LICENSE, respectively. The artistic license is more appropriate for commercial use, since it lacks the "viral" properties of the GPL.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

1.7 Mailing lists and support

1.7.1 How to get help

The starting point for all bug reports, feature requests and support questions is the Blitz++ support page, at `http://oonumerics.org/blitz/support/`

Please note the search engine box on this web page which lets you search the mailing list archives. This will often turn up answers to your question if it has been asked before.

1.7.2 How to subscribe to a mailing list

To subscribe to a Blitz++ mailing list, send a message containing one (or more) of these lines to `majordomo@oonumerics.org`:

```
subscribe blitz-support
subscribe blitz-bugs
subscribe blitz-dev
```

1.7.3 blitz-bugs

You can report bugs (or things you suspect might be bugs) to `blitz-bugs@oonumerics.org`.

It's not a very interesting list to be subscribed to. There are archives available from the Blitz++ web site.

1.7.4 blitz-dev

Blitz++ is in open development: anyone can contribute features and code to the library. If you are interested in helping out with coding or porting, you should start by subscribing to the `blitz-dev` mailing list.

This list is also the appropriate place to send suggestions for features; just send email to `blitz-dev@oonumerics.org`. We can't implement it if you don't suggest it.

Archives of this list are available from the Blitz++ web site.

1.7.5 blitz-support

This mailing list is for posting and answering questions about using the Blitz++ library. Anyone can post questions; anyone can answer.

2 Arrays

2.1 Getting started

Currently, Blitz++ provides a single array class, called `Array<T_numtype,N_rank>`. This array class provides a dynamically allocated N-dimensional array, with reference counting, arbitrary storage ordering, subarrays and slicing, flexible expression handling, and many other useful features.

2.1.1 Template parameters

The `Array` class takes two template parameters:

- `T_numtype` is the numeric type to be stored in the array. `T_numtype` can be an integral type (`bool`, `char`, `unsigned char`, `short int`, `short unsigned int`, `int`, `unsigned int`, `long`, `unsigned long`), floating point type (`float`, `double`, `long double`), complex type (`complex<float>`, `complex<double>`, `complex<long double>`) or any user-defined type with appropriate numeric semantics.
- `N_rank` is the **rank** (or dimensionality) of the array. This should be a positive integer.

To use the `Array` class, include the header `<blitz/array.h>` and use the namespace `blitz`:

```
#include <blitz/array.h>

using namespace blitz;

Array<int,1>    x;    // A one-dimensional array of int
Array<double,2> y;    // A two-dimensional array of double
.
.
Array<complex<float>, 12> z; // A twelve-dimensional array of complex<float>
```

When no constructor arguments are provided, the array is empty, and no memory is allocated. To create an array which contains some data, provide the size of the array as constructor arguments:

```
Array<double,2> y(4,4);   // A 4x4 array of double
```

The contents of a newly-created array are garbage. To initialize the array, you can write:

```
y = 0;
```

and all the elements of the array will be set to zero. If the contents of the array are known, you can initialize it using a comma-delimited list of values. For example, this code excerpt sets `y` equal to a 4x4 identity matrix:

```
y = 1, 0, 0, 0,
    0, 1, 0, 0,
    0, 0, 1, 0,
    0, 0, 0, 1;
```

2.1.2 Array types

The `Array<T,N>` class supports a variety of arrays:

- Arrays of scalar types, such as `Array<int,1>` and `Array<float,3>`
- Complex arrays, such as `Array<complex<float>,2>`
- Arrays of user-defined types. If you have a class called `Polynomial`, then `Array<Polynomial,2>` is an array of `Polynomial` objects.

- Nested homogeneous arrays using `TinyVector` and `TinyMatrix`, in which each element is a fixed-size vector or array. For example, `Array<TinyVector<float,3>,3>` is a three-dimensional vector field.
- Nested heterogeneous arrays, such as `Array<Array<int,1>,1>`, in which each element is a variable-length array.

2.1.3 A simple example

Here's an example program which creates two 3x3 arrays, initializes them, and adds them:

```
#include <blitz/array.h>

using namespace blitz;

int main()
{
    Array<float,2> A(3,3), B(3,3), C(3,3);

    A = 1, 0, 0,
        2, 2, 2,
        1, 0, 0;

    B = 0, 0, 7,
        0, 8, 0,
        9, 9, 9;

    C = A + B;

    cout << "A = " << A << endl
         << "B = " << B << endl
         << "C = " << C << endl;

    return 0;
}
```

and the output:

```
A = 3 x 3
[         1         0         0
          2         2         2
          1         0         0 ]

B = 3 x 3
[         0         0         7
          0         8         0
          9         9         9 ]

C = 3 x 3
[         1         0         7
          2        10         2
         10         9         9 ]
```

2.1.4 Storage orders

Blitz++ is very flexible about the way arrays are stored in memory.

The default storage format is row-major, C-style arrays whose indices start at zero.

Fortran-style arrays can also be created. Fortran arrays are stored in column-major order, and have indices which start at one. To create a Fortran-style array, use this syntax: `Array<int,2> A(3, 3, fortranArray);` The last parameter, `fortranArray`, tells the `Array` constructor to use a fortran-style array format.

`fortranArray` is a global object which has an automatic conversion to type `GeneralArrayStorage<N>`. `GeneralArrayStorage<N>` encapsulates information about how an array is laid out in memory. By altering the contents of a `GeneralArrayStorage<N>` object, you can lay out your arrays any way you want: the dimensions can be ordered arbitrarily and stored in ascending or descending order, and the starting indices can be arbitrary.

Creating custom array storage formats is described in a later section (Section 2.9 [Array storage], page 26).

2.2 Public types

The `Array` class declares these public types:

- `T_numtype` is the element type stored in the array. For example, the type `Array<double,2>::T_numtype` would be `double`.
- `T_index` is a vector index into the array. The class `TinyVector` is used for this purpose.
- `T_array` is the array type itself (`Array<T_numtype,N_rank>`)
- `T_iterator` is an iterator type. NB: this iterator is not yet fully implemented, and is NOT STL compatible at the present time.

2.3 Constructors

2.3.1 Default constructor

```
Array();
Array(GeneralArrayStorage<N_rank> storage)
```

The default constructor creates a C-style array of zero size. Any attempt to access data in the array may result in a run-time error, because there isn't any data to access!

An optional argument specifies a storage order for the array.

Arrays created using the default constructor can subsequently be given data by the `resize()`, `resizeAndPreserve()`, or `reference()` member functions.

2.3.2 Creating an array from an expression

```
Array(expression...)
```

You may create an array from an array expression. For example,

```
Array<float,2> A(4,3), B(4,3);   // ...
Array<float,2> C(A*2.0+B);
```

This is an explicit constructor (it will not be used to perform implicit type conversions). The newly constructed array will have the same storage format as the arrays in the expression. If arrays with different storage formats appear in the expression, an error will result. (In this case, you must first construct the array, then assign the expression to it).

2.3.3 Constructors which take extent parameters

```
Array(int extent1);
Array(int extent1, int extent2);
Array(int extent1, int extent2, int extent3);
...
Array(int extent1, int extent2, int extent3, ..., int extent11)
```

These constructors take arguments which specify the size of the array to be constructed. You should provide as many arguments as there are dimensions in the array.[1]

An optional last parameter specifies a storage format:

[1] If you provide fewer than `N_rank` arguments, the missing arguments will be filled in using the last provided argument. However, for code clarity, it makes sense to provide all `N_rank` parameters.

```
Array(int extent1, GeneralArrayStorage<N_rank> storage);
Array(int extent1, int extent2, GeneralArrayStorage<N_rank> storage);
...
```

For high-rank arrays, it may be convenient to use this constructor:

```
Array(const TinyVector<int, N_rank>& extent);
Array(const TinyVector<int, N_rank>& extent,
      GeneralArrayStorage<N_rank> storage);
```

The argument `extent` is a vector containing the extent (length) of the array in each dimension. The optional second parameter indicates a storage format. Note that you can construct `TinyVector<int,N>` objects on the fly with the `shape(i1,i2,...)` global function. For example, `Array<int,2> A(shape(3,5))` will create a 3x5 array.

A similar constructor lets you provide both a vector of base index values (lbounds) and extents:

```
Array(const TinyVector<int, N_rank>& lbound,
      const TinyVector<int, N_rank>& extent);
Array(const TinyVector<int, N_rank>& lbound,
      const TinyVector<int, N_rank>& extent,
      GeneralArrayStorage<N_rank> storage);
```

The argument `lbound` is a vector containing the base index value (or lbound) of the array in each dimension. The argument `extent` is a vector containing the extent (length) of the array in each dimension. The optional third parameter indicates a storage format. As with the above constructor, you can use the `shape(i1,i2,...)` global function to create the `lbound` and `extent` parameters.

2.3.4 Constructors with Range arguments

These constructors allow arbitrary bases (starting indices) to be set:

```
Array(Range r1);
Array(Range r1, Range r2);
Array(Range r1, Range r2, Range r3);
...
Array(Range r1, Range r2, Range r3, ..., Range r11);
```

For example, this code:

```
Array<int,2> A(Range(10,20), Range(20,30));
```

will create an 11x11 array whose indices are 10..20 and 20..30. An optional last parameter provides a storage order:

```
Array(Range r1, GeneralArrayStorage<N_rank> storage);
Array(Range r1, Range r2, GeneralArrayStorage<N_rank> storage);
...
```

2.3.5 Referencing another array

This constructor makes a shared view of another array's data:

```
Array(Array<T_numtype, N_rank>& array);
```

After this constructor is used, both `Array` objects refer to the *same data*. Any changes made to one array will appear in the other array. If you want to make a duplicate copy of an array, use the `copy()` member function.

2.3.6 Constructing an array from an expression

Arrays may be constructed from expressions, which are described in Section 3.1 [Array expressions], page 33. The syntax is:

```
Array(...array expression...);
```

For example, this code creates an array B which contains the square roots of the elements in A:

```
Array<float,2> A(N,N);   // ...
Array<float,2> B(sqrt(A));
```

2.3.7 Creating an array from pre-existing data

When creating an array using a pointer to already existing data, you have three choices for how Blitz++ will handle the data. These choices are enumerated by the enum type `preexistingMemoryPolicy`:

```
enum preexistingMemoryPolicy {
  duplicateData,
  deleteDataWhenDone,
  neverDeleteData
};
```

If you choose `duplicateData`, Blitz++ will create an array object using a copy of the data you provide. If you choose `deleteDataWhenDone`, Blitz++ will not create a copy of the data; and when no array objects refer to the data anymore, it will deallocate the data using `delete []`. Note that to use `deleteDataWhenDone`, your array data must have been allocated using the C++ `new` operator – for example, you cannot allocate array data using Fortran or `malloc`, then create a Blitz++ array from it using the `deleteDataWhenDone` flag. The third option is `neverDeleteData`, which means that Blitz++ will not never deallocate the array data. This means it is your responsibility to determine when the array data is no longer needed, and deallocate it. You should use this option for memory which has not been allocated using the C++ `new` operator.

These constructors create array objects from pre-existing data:

```
Array(T_numtype* dataFirst, TinyVector<int, N_rank> shape,
      preexistingMemoryPolicy deletePolicy);
Array(T_numtype* dataFirst, TinyVector<int, N_rank> shape,
      preexistingMemoryPolicy deletePolicy,
      GeneralArrayStorage<N_rank> storage);
```

The first argument is a pointer to the array data. It should point to the element of the array which is stored first in memory. The second argument indicates the shape of the array. You can create this argument using the `shape()` function. For example:

```
double data[] = { 1, 2, 3, 4 };
Array<double,2> A(data, shape(2,2), neverDeleteData);   // Make a 2x2 array
```

The `shape()` function takes N integer arguments and returns a `TinyVector<int,N>`.

By default, Blitz++ arrays are row-major. If you want to work with data which is stored in column-major order (e.g. a Fortran array), use the second version of the constructor:

```
Array<double,2> B(data, shape(2,2), neverDeleteData,
                  FortranArray<2>());
```

This is a tad awkward, so Blitz++ provides the global object `fortranArray` which will convert to an instance of `GeneralArrayStorage<N_rank>`:

```
Array<double,2> B(data, shape(2,2), neverDeleteData, fortranArray);
```

Another version of this constructor allows you to pass an arbitrary vector of strides:

```
Array(T_numtype* _bz_restrict dataFirst, TinyVector<int, N_rank> shape,
      TinyVector<int, N_rank> stride,
      preexistingMemoryPolicy deletePolicy,
      GeneralArrayStorage<N_rank> storage = GeneralArrayStorage<N_rank>())
```

2.3.8 Interlacing arrays

For some platforms, it can be advantageous to store a set of arrays interlaced together in memory. Blitz++ provides support for this through the routines `interlaceArrays()` and `allocateArrays()`. An example:

```
Array<int,2> A, B;
interlaceArrays(shape(10,10), A, B);
```

The first parameter of `interlaceArrays()` is the shape for the arrays (10x10). The subsequent arguments are the set of arrays to be interlaced together. Up to 11 arrays may be interlaced. All arrays must store the same data type and be of the same rank. In the above example, storage is allocated so that `A(0,0)` is followed immediately by `B(0,0)` in memory, which is folloed by `A(0,1)` and `B(0,1)`, and so on.

A related routine is `allocateArrays()`, which has identical syntax:

```
Array<int,2> A, B;
allocateArrays(shape(10,10), A, B);
```

Unlike `interlaceArrays()`, which always interlaces the arrays, the routine `allocateArrays()` may or may not interlace them, depending on whether interlacing is considered advantageous for your platform. If the tuning flag `BZ_INTERLACE_ARRAYS` is defined in `<blitz/tuning.h>`, then the arrays are interlaced.

Note that the performance effects of interlacing are unpredictable: in some situations it can be a benefit, and in most others it can slow your code down substantially. You should only use `interlaceArrays()` after running some benchmarks to determine whether interlacing is beneficial for your particular algorithm and architecture.

2.3.9 A note about reference counting

Blitz++ arrays use reference counting. When you create a new array, a memory block is allocated. The `Array` object acts like a handle for this memory block. A memory block can be shared among multiple `Array` objects – for example, when you take subarrays and slices. The memory block keeps track of how many `Array` objects are referring to it. When a memory block is orphaned – when no `Array` objects are referring to it – it automatically deletes itself and frees the allocated memory.

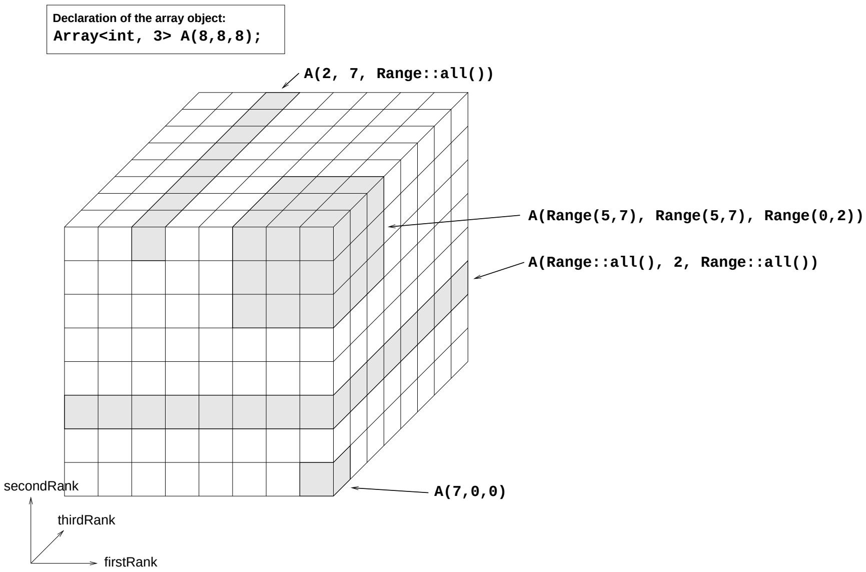
2.4 Indexing, subarrays, and slicing

This section describes how to access the elements of an array. There are three main ways:

- **Indexing** obtains a single element
- Creating a **subarray** which refers to a smaller portion of an array
- **Slicing** to produce a smaller-dimensional view of a portion of an array

Indexing, subarrays and slicing all use the overloaded parenthesis `operator()`.

As a running example, we'll consider the three dimensional array pictured below, which has index ranges (0..7, 0..7, 0..7). Shaded portions of the array show regions which have been obtained by indexing, creating a subarray, and slicing.



Examples of array indexing, subarrays, and slicing.

2.4.1 Indexing

There are two ways to get a single element from an array. The simplest is to provide a set of integer operands to `operator()`:

```
A(7,0,0) = 5;
cout << "A(7,0,0) = " << A(7,0,0) << endl;
```

This version of indexing is available for arrays of rank one through eleven. If the array object isn't `const`, the return type of `operator()` is a reference; if the array object is `const`, the return type is a value.

You can also get an element by providing an operand of type `TinyVector<int,N_rank>` where `N_rank` is the rank of the array object:

```
TinyVector<int,3> index;
index = 7, 0, 0;
A(index) = 5;
cout << "A(7,0,0) = " << A(index) << endl;
```

This version of `operator()` is also available in a const-overloaded version.

It's possible to use fewer than `N_rank` indices. However, missing indices are **assumed to be zero**, which will cause bounds errors if the valid index range does not include zero (e.g. Fortran arrays). For this reason, and for code clarity, it's a bad idea to omit indices.

2.4.2 Subarrays

You can obtain a subarray by providing `Range` operands to `operator()`. A `Range` object represents a set of regularly spaced index values. For example,

```
Array<int,3> B = A(Range(5,7), Range(5,7), Range(0,2));
```

The object B now refers to elements (5..7,5..7,0..2) of the array A.

The returned subarray is of type `Array<T_numtype,N_rank>`. This means that subarrays can be used wherever arrays can be: in expressions, as lvalues, etc. Some examples:

```
// A three-dimensional stencil (used in solving PDEs)
Range I(1,6), J(1,6), K(1,6);
B = (A(I,J,K) + A(I+1,J,K) + A(I-1,J,K) + A(I,J+1,K)
 + A(I,J-1,K) + A(I,J+1,K) + A(I,J,K+1) + A(I,J,K-1)) / 7.0;

// Set a subarray of A to zero
A(Range(5,7), Range(5,7), Range(5,7)) = 0.;
```

The bases of the subarray are equal to the bases of the original array:

```
Array<int,2> D(Range(1,5), Range(1,5));     // 1..5, 1..5
Array<int,2> E = D(Range(2,3), Range(2,3)); // 1..2, 1..2
```

An array can be used on both sides of an expression only if the subarrays don't overlap. If the arrays overlap, the result may depend on the order in which the array is traversed.

2.4.3 RectDomain and StridedDomain

The classes `RectDomain` and `StridedDomain`, defined in `blitz/domain.h`, offer a dimension-independent notation for subarrays.

`RectDomain` and `StridedDomain` can be thought of as a `TinyVector<Range,N>`. Both have a vector of lower- and upper-bounds; `StridedDomain` has a stride vector. For example, the subarray:

```
Array<int,2> B = A(Range(4,7), Range(8,11));  // 4..7, 8..11
```

could be obtained using `RectDomain` this way:

```
TinyVector<int,2> lowerBounds(4, 8);
TinyVector<int,2> upperBounds(7, 11);
RectDomain<2> subdomain(lowerBounds, upperBounds);

Array<int,2> B = A(subdomain);
```

Here are the prototypes of `RectDomain` and `StridedDomain`.

```
template<int N_rank>
class RectDomain {

public:
    RectDomain(const TinyVector<int,N_rank>& lbound,
        const TinyVector<int,N_rank>& ubound);

    const TinyVector<int,N_rank>& lbound() const;
    int lbound(int i) const;
    const TinyVector<int,N_rank>& ubound() const;
    int ubound(int i) const;
    Range operator[](int rank) const;
    void shrink(int amount);
    void shrink(int dim, int amount);
    void expand(int amount);
    void expand(int dim, int amount);
};

template<int N_rank>
class StridedDomain {

public:
    StridedDomain(const TinyVector<int,N_rank>& lbound,
        const TinyVector<int,N_rank>& ubound,
        const TinyVector<int,N_rank>& stride);

    const TinyVector<int,N_rank>& lbound() const;
    int lbound(int i) const;
    const TinyVector<int,N_rank>& ubound() const;
    int ubound(int i) const;
    const TinyVector<int,N_rank>& stride() const;
```

```
        int stride(int i) const;
        Range operator[](int rank) const;
        void shrink(int amount);
        void shrink(int dim, int amount);
        void expand(int amount);
        void expand(int dim, int amount);
    };
```

2.4.4 Slicing

A combination of integer and Range operands produces a **slice**. Each integer operand reduces the rank of the array by one. For example:

```
Array<int,2> F = A(Range::all(), 2, Range::all());
Array<int,1> G = A(2,            7, Range::all());
```

Range and integer operands can be used in any combination, for arrays up to rank 11.

Note: Using a combination of integer and Range operands requires a newer language feature (partial ordering of member templates) which not all compilers support. If your compiler does provide this feature, `BZ_PARTIAL_ORDERING` will be defined in `<blitz/config.h>`. If not, you can use this workaround:

```
Array<int,3> F = A(Range::all(), Range(2,2), Range::all());
Array<int,3> G = A(Range(2,2),   Range(7,7), Range::all());
```

2.4.5 More about Range objects

A `Range` object represents an ordered set of uniformly spaced integers. Here are some examples of using Range objects to obtain subarrays:

```
#include <blitz/array.h>

using namespace blitz;

int main()
{
    Array<int,1> A(7);
    A = 0, 1, 2, 3, 4, 5, 6;

    cout << A(Range::all())  << endl          // [ 0 1 2 3 4 5 6 ]
         << A(Range(3,5))    << endl          // [ 3 4 5 ]
         << A(Range(3,toEnd)) << endl         // [ 3 4 5 6 ]
         << A(Range(fromStart,3)) << endl     // [ 0 1 2 3 ]
         << A(Range(1,5,2)) << endl           // [ 1 3 5 ]
         << A(Range(5,1,-2)) << endl          // [ 5 3 1 ]
         << A(Range(fromStart,toEnd,2)) << endl;    // [ 0 2 4 6 ]

    return 0;
}
```

The optional third constructor argument specifies a stride. For example, `Range(1,5,2)` refers to elements [1 3 5]. Strides can also be negative: `Range(5,1,-2)` refers to elements [5 3 1].

Note that if you use the same Range frequently, you can just construct one object and use it multiple times. For example:

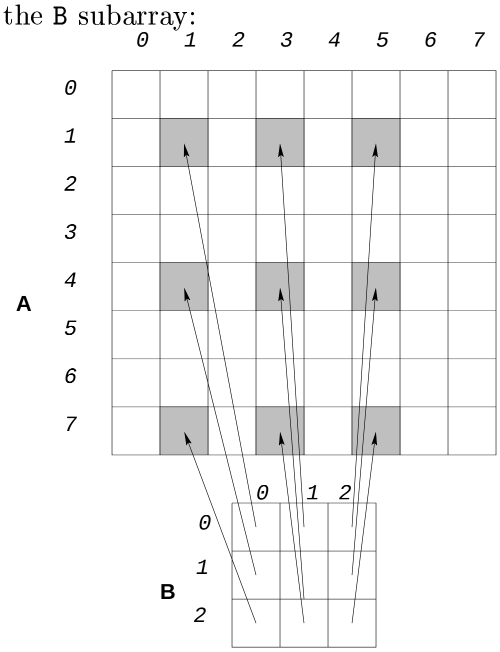
```
Range all = Range::all();
A(0,all,all) = A(N-1,all,all);
A(all,0,all) = A(all,N-1,all);
A(all,all,0) = A(all,all,N-1);
```

Here's an example of using strides with a two-dimensional array:

```
#include <blitz/array.h>
```

```
using namespace blitz;

int main()
{
    Array<int,2> A(8,8);
    A = 0;

    Array<int,2> B = A(Range(1,7,3), Range(1,5,2));
    B = 1;

    cout << "A = " << A << endl;
    return 0;
}
```

Here's an illustration of the B subarray:



Using strides to create non-contiguous subarrays.

And the program output:

```
A = 8 x 8
[         0         0         0         0         0         0         0
          0
          0         1         0         1         0         1         0
          0
          0         0         0         0         0         0         0
          0
          0         0         0         0         0         0         0
          0
          0         1         0         1         0         1         0
          0
          0         0         0         0         0         0         0
          0
          0         0         0         0         0         0         0
          0
          0         1         0         1         0         1         0
          0 ]
```

2.4.6 A note about assignment

The assignment operator (=) always results in the expression on the right-hand side (rhs) being *copied* to the lhs (i.e. the data on the lhs is overwritten with the result from the rhs). This is different from some array packages in which the assignment operator makes the lhs a reference (or alias) to the rhs. To further confuse the issue, the copy constructor for arrays *does* have reference semantics. Here's an example which should clarify things:

```
Array<int,1> A(5), B(10);
A = B(Range(0,4));                  // Statement 1
Array<int,1> C = B(Range(0,4));  // Statement 2
```

Statement 1 results in a portion of `B`'s data being copied into `A`. After Statement 1, both `A` and `B` have their own (nonoverlapping) blocks of data. Contrast this behaviour with that of Statement 2, which is **not** an assignment (it uses the copy constructor). After Statement 2 is executed, the array `C` is a reference (or alias) to `B`'s data.

So to summarize: If you want to copy the rhs, use an assignment operator. If you want to reference (or alias) the rhs, use the copy constructor (or alternately, the `reference()` member function in Section 2.6 [Array members], page 18).

Very important: whenever you have an assignment operator (`=`, `+=`, `-=`, etc.) the lhs **must** have the same shape as the **rhs**. If you want the array on the left hand side to be resized to the proper shape, you must do so by calling the `resize` method, for example:

```
A.resize(B.shape());    // Make A the same size as B
A = B;
```

2.4.7 An example

```
#include <blitz/array.h>

using namespace blitz;

int main()
{
    Array<int,2> A(6,6), B(3,3);

    // Set the upper left quadrant of A to 5
    A(Range(0,2), Range(0,2)) = 5;

    // Set the upper right quadrant of A to an identity matrix
    B = 1, 0, 0,
        0, 1, 0,
        0, 0, 1;
    A(Range(0,2), Range(3,5)) = B;

    // Set the fourth row to 1
    A(3, Range::all()) = 1;

    // Set the last two rows to 0
    A(Range(4, Range::toEnd), Range::all()) = 0;

    // Set the bottom right element to 8
    A(5,5) = 8;

    cout << "A = " << A << endl;

    return 0;
}
```

The output:

```
A = 6 x 6
[         5         5         5         1         0         0
          5         5         5         0         1         0
          5         5         5         0         0         1
          1         1         1         1         1         1
          0         0         0         0         0         0
          0         0         0         0         0         8 ]
```

2.5 Debug mode

The Blitz++ library has a debugging mode which is enabled by defining the preprocessor symbol `BZ_DEBUG`. For most compilers, the command line argument `-DBZ_DEBUG` should work.

In debugging mode, your programs will run *very slowly.* This is because Blitz++ is doing lots of precondition checking and bounds checking. When it detects something fishy, it will likely halt your program and display an error message.

For example, this program attempts to access an element of a 4x4 array which doesn't exist:

```
#include <blitz/array.h>

using namespace blitz;

int main()
{
    Array<complex<float>, 2> Z(4,4);

    Z = complex<float>(0.0, 1.0);

    Z(4,4) = complex<float>(1.0, 0.0);

    return 0;
}
```

When compiled with `-DBZ_DEBUG`, the out of bounds indices are detected and an error message results:

```
[Blitz++] Precondition failure: Module ../../blitz/array-impl.h line 1295
Array index out of range: (4, 4)
Lower bounds: 2 [          0          0 ]
Length:       2 [          4          4 ]

debug: ../../blitz/array-impl.h:1295: bool blitz::Array<T,
N>::assertInRange(int, int) const [with P_numtype = std::complex<float>, int
N_rank = 2]: Assertion '0' failed.
```

Precondition failures send their error messages to the standard error stream (`cerr`). After displaying the error message, `assert(0)` is invoked.

2.6 Member functions

2.6.1 A note about dimension parameters

Several of the member functions take a *dimension parameter* which is an integer in the range 0 .. `N_rank`-1. For example, the method `extent(int n)` returns the extent (or length) of the array in dimension `n`.

These parameters are problematic:

- They make the code cryptic. Someone unfamiliar with the `reverse()` member function won't stand a chance of understanding what `A.reverse(2)` does.
- Some users are used to dimensions being 1 .. `N_rank`, rather than 0 .. `N_rank`-1. This makes dimension numbers inherently error-prone. Even though I'm a experienced C/C++ programmer, I *still* want to think of the first dimension as 1 – it doesn't make sense to talk about the "zeroth" dimension.

As a solution to this problem, Blitz++ provides a series of symbolic constants which you can use to refer to dimensions:

```
const int firstDim    = 0;
const int secondDim   = 1;
const int thirdDim    = 2;
```

```
.
.
const int eleventhDim = 10;
```

These symbols should be used in place of the numerals 0, 1, ... `N_rank`-1. For example:

```
A.reverse(thirdDim);
```

This code is clearer: you can see that the parameter refers to a dimension, and it isn't much of a leap to realize that it's reversing the element ordering in the third dimension.

If you find `firstDim`, `secondDim`, ... aesthetically unpleasing, there are equivalent symbols `firstRank`, `secondRank`, `thirdRank`, ..., `eleventhRank`.

Why stop at eleven?

The symbols had to stop somewhere, and eleven seemed an appropriate place to stop. Besides, if you're working in more than eleven dimensions your code is going to be confusing no matter what help Blitz++ provides.

2.6.2 Member function descriptions

```
const TinyVector<int, N_rank>&    base() const;
int                               base(int dimension) const;
```

The *base* of a dimension is the first valid index value. A typical C-style array will have base of zero; a Fortran-style array will have base of one. The base can be different for each dimension, but only if you deliberately use a Range-argument constructor or design a custom storage ordering.

The first version returns a reference to the vector of base values. The second version returns the base for just one dimension; it's equivalent to the `lbound()` member function. See the note on dimension parameters such as `firstDim` above.

```
Array<T,N>::iterator              begin();
Array<T,N>::const_iterator        begin() const;
```

These functions return STL-style forward and input iterators, respectively, positioned at the first element of the array. Note that the array data is traversed in memory order (i.e. by rows for C-style arrays, and by columns for Fortran-style arrays). The `Array<T,N>::const_iterator` has these methods:

```
const_iterator(const Array<T,N>&);
T operator*() const;
const T* [restrict] operator->() const;
const_iterator& operator++();
void operator++(int);
bool operator==(const const_iterator<T,N>&) const;
bool operator!=(const const_iterator<T,N>&) const;
const TinyVector<int,N>& position() const;
```

Note that postfix `++` returns void (this is not STL-compliant, but is done for efficiency). The method `position()` returns a vector containing current index positions of the iterator. The `Array<T,N>::iterator` has the same methods as `const_iterator`, with these exceptions: `iterator& operator++(); T& operator*(); T* [restrict] operator->();` The `iterator` type may be used to modify array elements. To obtain iterator positioned at the end of the array, use the `end()` methods.

```
int                               cols() const;
int                               columns() const;
```

Both of these functions return the extent of the array in the second dimension. Equivalent to `extent(secondDim)`. See also `rows()` and `depth()`.

```
Array<T_numtype, N_rank>          copy() const;
```

This method creates a copy of the array's data, using the same storage ordering as the current array. The returned array is guaranteed to be stored contiguously in memory, and to be the only object referring to its memory block (i.e. the data isn't shared with any other array object).

```
const T_numtype* [restrict]       data() const;
      T_numtype* [restrict]       data();
const T_numtype* [restrict]       dataZero() const;
      T_numtype* [restrict]       dataZero();
const T_numtype* [restrict]       dataFirst() const;
      T_numtype* [restrict]       dataFirst();
```

These member functions all return pointers to the array data. The NCEG `restrict` qualifier is used only if your compiler supports it. If you're working with the default storage order (C-style arrays with base zero), you'll only need to use `data()`. Otherwise, things get complicated:

`data()` returns a pointer to the element whose indices are equal to the array base. With a C-style array, this means the element (0,0,...,0); with a Fortran-style array, this means the element (1,1,...,1). If `A` is an array object, `A.data()` is equivalent to (&A(A.base(firstDim), A.base(secondDim), ...)). If any of the dimensions are stored in reverse order, `data()` will not refer to the element which comes first in memory.

`dataZero()` returns a pointer to the element (0,0,...,0), even if such an element does not exist in the array. What's the point of having such a pointer? Say you want to access the element (i,j,k). If you add to the pointer the dot product of (i,j,k) with the stride vector (`A.stride()`), you get a pointer to the element (i,j,k).

`dataFirst()` returns a pointer to the element of the array which comes first in memory. Note however, that under some circumstances (e.g. subarrays), the data will not be stored contiguously in memory. You have to be very careful when meddling directly with an array's data.

Other relevant functions are: `isStorageContiguous()` and `zeroOffset()`.

```
int                               depth() const;
```

Returns the extent of the array in the third dimension. This function is equivalent to `extent(thirdDim)`. See also `rows()` and `columns()`.

```
int                               dimensions() const;
```

Returns the number of dimensions (rank) of the array. The return value is the second template parameter (`N_rank`) of the `Array` object. Same as `rank()`.

```
RectDomain<N_rank>                domain() const;
```

Returns the domain of the array. The domain consists of a vector of lower bounds and a vector of upper bounds for the indices. NEEDS_WORK– need a section to explain methods of `RectDomain<N>`.

```
Array<T,N>::iterator              end();
Array<T,N>::const_iterator        end() const;
```

Returns STL-style forward and input iterators (respectively) for the array, positioned at the end of the array.

```
int                               extent(int dimension) const;
```

The first version the extent (length) of the array in the specified dimension. See the note about dimension parameters such as `firstDim` in the previous section.

```
Array<T_numtype2,N_rank>          extractComponent(T_numtype2,
                                  int componentNumber, int numComponents);
```

This method returns an array view of a single component of a multicomponent array. In a multicomponent array, each element is a tuple of fixed size. The components are numbered 0, 1, ..., `numComponents-1`. Example:

```
Array<TinyVector<int,3>,2> A(128,128);  // A 128x128 array of int[3]

Array<int,2> B = A.extractComponent(int(), 1, 3);
```

Now the B array refers to the 2nd component of every element in A. Note: for complex arrays, special global functions `real(A)` and `imag(A)` are provided to obtain real and imaginary components of an array. See the **Global Functions** section.

```
void                              free();
```

This method resizes an array to zero size. If the array data is not being shared with another array object, then it is freed.

```
bool                              isMajorRank(int dimension) const;
```

Returns true if the dimension has the largest stride. For C-style arrays (the default), the first dimension always has the largest stride. For Fortran-style arrays, the last dimension has the largest stride. See also `isMinorRank()` below and the note about dimension parameters such as `firstDim` in the previous section.

```
bool                              isMinorRank(int dimension) const;
```

Returns true if the dimension *does not* have the largest stride. See also `isMajorRank()`.

```
bool                              isRankStoredAscending(int dimension) const;
```

Returns true if the dimension is stored in ascending order in memory. This is the default. It will only return false if you have reversed a dimension using `reverse()` or have created a custom storage order with a descending dimension.

```
bool                              isStorageContiguous() const;
```

Returns true if the array data is stored contiguously in memory. If you slice the array or work on subarrays, there can be skips – the array data is interspersed with other data not part of the array. See also the various `data..()` functions. If you need to ensure that the storage is contiguous, try `reference(copy())`.

```
int                               lbound(int dimension) const;
TinyVector<int,N_rank>            lbound() const;
```

The first version returns the lower bound of the valid index range for a dimension. The second version returns a vector of lower bounds for all dimensions. The lower bound is the first valid index value. If you're using a C-style array (the default), the lbound will be zero; Fortran-style arrays have lbound equal to one. The lbound can be different for each dimension, but only if you deliberately set them that way using a Range constructor or a custom storage ordering. This function is equivalent to `base(dimension)`. See the note about dimension parameters such as `firstDim` in the previous section.

```
void                              makeUnique();
```

If the array's data is being shared with another Blitz++ array object, this member function creates a copy so the array object has a unique view of the data.

```
int                               numElements() const;
```

Returns the total number of elements in the array, calculated by taking the product of the extent in each dimension. Same as `size()`.

```
const TinyVector<int, N_rank>&    ordering() const;
int                               ordering(int storageRankIndex) const;
```

These member functions return information about how the data is ordered in memory. The first version returns the complete ordering vector; the second version returns a single element from the ordering vector. The argument for the second version must be in the range 0 .. `N_rank`-1. The ordering vector is a list of dimensions in increasing order of stride; `ordering(0)` will return the dimension number with the smallest stride, and `ordering(N_rank-1)` will return the dimension number with largest stride. For a C-style array, the ordering vector contains the elements (`N_rank`-1, `N_rank`-2, ..., 0). For a Fortran-style array, the ordering vector is (0, 1, ..., `N_rank`-1). See also the description of custom storage orders in section Section 2.9 [Array storage], page 26.

```
int                              rank() const;
```

Returns the rank (number of dimensions) of the array. The return value is equal to `N_rank`. Equivalent to `dimensions()`.

```
void                             reference(Array<T_numtype,N_rank>& A);
```

This causes the array to adopt another array's data as its own. After this member function is used, the array object and the array `A` are indistinguishable – they have identical sizes, index ranges, and data. The data is shared between the two arrays.

```
void                             reindexSelf(const TinyVector<int,N_rank>&);
Array<T,N>                       reindex(const TinyVector<int,N_rank>&);
```

These methods reindex an array to use a new base vector. The first version reindexes the array, and the second just returns a reindexed view of the array, leaving the original array unmodified.

```
void                             resize(int extent1, ...);
void                             resize(const TinyVector<int,N_rank>&);
```

These functions resize an array to the specified size. If the array is already the size specified, then no memory is allocated. After resizing, the contents of the array are garbage. See also `resizeAndPreserve()`.

```
void                             resizeAndPreserve(int extent1, ...);
void                             resizeAndPreserve(const TinyVector<int,N_rank>&);
```

These functions resize an array to the specified size. If the array is already the size specified, then no change occurs (the array is not reallocated and copied). The contents of the array are preserved whenever possible; if the new array size is smaller, then some data will be lost. Any new elements created by resizing the array are left uninitialized.

```
Array<T,N>                       reverse(int dimension);
void                             reverseSelf(int dimension);
```

This method reverses the array in the specified dimension. For example, if `reverse(firstDim)` is invoked on a 2-dimensional array, then the ordering of rows in the array will be reversed; `reverse(secondDim)` would reverse the order of the columns. Note that this is implemented by twiddling the strides of the array, and doesn't cause any data copying. The first version returns a reversed "view" of the array data; the second version applies the reversal to the array itself.

```
int                              rows() const;
```

Returns the extent (length) of the array in the first dimension. This function is equivalent to `extent(firstDim)`. See also `columns()`, and `depth()`.

```
int                              size() const;
```

Returns the total number of elements in the array, calculated by taking the product of the extent in each dimension. Same as `numElements()`.

```
const TinyVector<int, N_rank>&   shape() const;
```

Returns the vector of extents (lengths) of the array.

```
const TinyVector<int, N_rank>&   stride() const;
int                              stride(int dimension) const;
```

The first version returns the stride vector; the second version returns the stride associated with a dimension. A stride is the distance between pointers to two array elements which are adjacent in a dimension. For example, `A.stride(firstDim)` is equal to `&A(1,0,0) -`

&A(0,0,0). The stride for the second dimension, A.stride(secondDim), is equal to &A(0,1,0) - &A(0,0,0), and so on. For more information about strides, see the description of custom storage formats in Section Section 2.9 [Array storage], page 26. See also the description of parameters like firstDim and secondDim in the previous section.

```
Array<T,N>                        transpose(int dimension1,
                                            int dimension2, ...);
void                              transposeSelf(int dimension1,
                                                int dimension2, ...);
```

These methods permute the dimensions of the array. The dimensions of the array are reordered so that the first dimension is dimension1, the second is dimension2, and so on. The arguments should be a permutation of the symbols firstDim, secondDim, ... Note that this is implemented by twiddling the strides of the array, and doesn't cause any data copying. The first version returns a transposed "view" of the array data; the second version transposes the array itself.

```
int                               ubound(int dimension) const;
TinyVector<int,N_rank>            ubound() const;
```

The first version returns the upper bound of the valid index range for a dimension. The second version returns a vector of upper bounds for all dimensions. The upper bound is the last valid index value. If you're using a C-style array (the default), the ubound will be equal to the extent(dimension)-1. Fortran-style arrays will have ubound equal to extent(dimension). The ubound can be different for each dimension. The return value of ubound(dimension) will always be equal to lbound(dimension)+extent(dimension)-1. See the note about dimension parameters such as firstDim in the previous section.

```
int                               zeroOffset() const;
```

This function has to do with the storage of arrays in memory. You may want to refer to the description of the data..() member functions and of custom storage orders in Section Section 2.9 [Array storage], page 26 for clarification. The return value of zeroOffset() is the distance from the first element in the array to the (possibly nonexistant) element (0,0,...,0). In this context, "first element" returns to the element (base(firstDim),base(secondDim),...).

2.7 Global functions

```
void                              allocateArrays(TinyVector<int,N>& shape,
                                                 Array<T,N>& A,
                                                 Array<T,N>& B, ...);
```

This function will allocate interlaced arrays, but only if interlacing is desirable for your architecture. This is controlled by the BZ_INTERLACE_ARRAYS flag in 'blitz/tuning.h'. You can provide up to 11 arrays as parameters. Any views currently associated with the array objects are lost. Here is a typical use:

```
Array<int,2> A, B, C;
allocateArrays(shape(64,64),A,B,C);
```

If array interlacing is enabled, then the arrays are stored in memory like this: A(0,0), B(0,0), C(0,0), A(0,1), B(0,1), ... If interlacing is disabled, then the arrays are allocated in the normal fashion: each array has its own block of memory. Once interlaced arrays are allocated, they can be used just like regular arrays.

```
#include <blitz/array/convolve.h>
Array<T,1>                        convolve(const Array<T,1>& B,
                                           const Array<T,1>& C);
```

This function computes the 1-D convolution of the arrays B and C:

$$A[i] = \sum_j B[j]C[i-j]$$

If the array B has domain $b_l \ldots b_h$, and array C has domain $c_l \ldots c_h$, then the resulting array has domain $a_l \ldots a_h$, with $l = b_l + c_l$ and $a_h = b_h + c_h$.

A new array is allocated to contain the result. To avoid copying the result array, you should use it as a constructor argument. For example: `Array<float,1> A = convolve(B,C);` The convolution is computed in the spatial domain. Frequency-domain transforms are not used. If you are convolving two large arrays, then this will be slower than using a Fourier transform.

Note that if you need a cross-correlation, you can use the convolve function with one of the arrays reversed. For example:

```
Array<float,1> A = convolve(B,C.reverse());
```

Autocorrelation can be performed using the same approach.

```
void                        cycleArrays(Array<T,N>& A, Array<T,N>& B);
void                        cycleArrays(Array<T,N>& A, Array<T,N>& B,
                                        Array<T,N>& C);
void                        cycleArrays(Array<T,N>& A, Array<T,N>& B,
                                        Array<T,N>& C, Array<T,N>& D);
void                        cycleArrays(Array<T,N>& A, Array<T,N>& B,
                                        Array<T,N>& C, Array<T,N>& D,
                                        Array<T,N>& E);
```

These routines are useful for time-stepping PDEs. They take a set of arrays such as [`A,B,C,D`] and cyclically rotate them to [`B,C,D,A`]; i.e. the `A` array then refers to what was `B`'s data, the `B` array refers to what was `C`'s data, and the `D` array refers to what was `A`'s data. These functions operate in constant time, since only the handles change (i.e. no data is copied; only pointers change).

```
Array<T,N>                  imag(Array<complex<T>,N>&);
```

This method returns a view of the imaginary portion of the array.

```
void                        interlaceArrays(TinyVector<int,N>& shape,
                                            Array<T,N>& A,
                                            Array<T,N>& B, ...);
```

This function is similar to `allocateArrays()` above, except that the arrays are **always** interlaced, regardless of the setting of the `BZ_INTERLACE_ARRAYS` flag.

```
Array<T,N>                  real(Array<complex<T>,N>&);
```

This method returns a view of the real portion of the array.

```
TinyVector<int,1>           shape(int L);
TinyVector<int,2>           shape(int L, int M);
TinyVector<int,3>           shape(int L, int M, int N);
TinyVector<int,4>           shape(int L, int M, int N, int O);
... [up to 11 dimensions]
```

These functions may be used to create shape parameters. They package the set of integer arguments as a `TinyVector` of appropriate length. For an example use, see `allocateArrays()` above.

2.8 Inputting and Outputting Arrays

2.8.1 Output formatting

The current version of Blitz++ includes rudimentary output formatting for arrays. Here's an example:

```
#include <blitz/array.h>

using namespace blitz;
```

```
int main()
{
    Array<int,2> A(4,5,FortranArray<2>());
    firstIndex i;
    secondIndex j;
    A = 10*i + j;

    cout << "A = " << A << endl;

    Array<float,1> B(20);
    B = exp(-i/100.);

    cout << "B = " << endl << B << endl;

    return 0;
}
```

And the output:

```
A = 4 x 5
[        11        12        13        14        15
         21        22        23        24        25
         31        32        33        34        35
         41        42        43        44        45 ]

B =
20
 [         1   0.99005  0.980199  0.970446  0.960789  0.951229  0.941765
   0.932394  0.923116  0.913931  0.904837  0.895834   0.88692  0.878095
   0.869358  0.860708  0.852144  0.843665   0.83527  0.826959  ]
```

2.8.2 Inputting arrays

Arrays may be restored from an istream using the `>>` operator. **Note:** you must know the dimensionality of the array being restored from the stream. The `>>` operator expects an array in the same input format as generated by the `<<` operator, namely:

- The size of the array, for example "32" for a 1-dimensional array of 32 elements, "12 x 64 x 128" for a 3-dimensional array of size 12x64x128.
- The symbol `'['` indicating the start of the array data
- The array elements, listed in memory storage order
- The symbol `']'` indicating the end of the array data

The operator prototype is:

```
template<class T, int N>
istream& operator>>(istream&, Array<T,N>&);
```

Here is an example of saving and restoring arrays from files. You can find this example in the Blitz++ distribution as '`examples/io.cpp`'.

```
#include <blitz/array.h>
#ifdef BZ_HAVE_STD
#include <fstream>
#else
#include <fstream.h>
#endif

BZ_USING_NAMESPACE(blitz)

const char* filename = "io.data";

void write_arrays()
{
```

```
    ofstream ofs(filename);
    if (ofs.bad())
    {
        cerr << "Unable to write to file: " << filename << endl;
        exit(1);
    }

    Array<float,3> A(3,4,5);
    A = 111 + tensor::i + 10 * tensor::j + 100 * tensor::k;
    ofs << A << endl;

    Array<float,2> B(3,4);
    B = 11 + tensor::i + 10 * tensor::j;
    ofs << B << endl;

    Array<float,1> C(4);
    C = 1 + tensor::i;
    ofs << C << endl;
}

int main()
{
    write_arrays();

    ifstream ifs(filename);
    if (ifs.bad())
    {
        cerr << "Unable to open file: " << filename << endl;
        exit(1);
    }

    Array<float,3> A;
    Array<float,2> B;
    Array<float,1> C;

    ifs >> A >> B >> C;

    cout << "Arrays restored from file: " << A << B << C << endl;

    return 0;
}
```

Note: The storage order and starting indices are not restored from the input stream. If you are restoring (for example) a Fortran-style array, you must create a Fortran-style array, and then restore it. For example, this code restores a Fortran-style array from the standard input stream:

```
Array<float,2> B(fortranArray);
cin >> B;
```

2.9 Array storage orders

Blitz++ is very flexible about the way arrays are stored in memory. Starting indices can be 0, 1, or arbitrary numbers; arrays can be stored in row major, column major or an order based on any permutation of the dimensions; each dimension can be stored in either ascending or descending order. An N dimensional array can be stored in $N!2^N$ possible ways.

Before getting into the messy details, a review of array storage formats is useful. If you're already familiar with strides and bases, you might want to skip on to the next section.

2.9.1 Fortran and C-style arrays

Suppose we want to store this two-dimensional array in memory:

```
[ 1 2 3 ]
[ 4 5 6 ]
[ 7 8 9 ]
```

Row major vs. column major

To lay the array out in memory, it's necessary to map the indices (i,j) into a one-dimensional block. Here are two ways the array might appear in memory:

```
[ 1 2 3 4 5 6 7 8 9 ]
[ 1 4 7 2 5 8 3 6 9 ]
```

The first order corresponds to a C or C++ style array, and is called *row-major ordering*: the data is stored first by row, and then by column. The second order corresponds to a Fortran style array, and is called *column-major ordering*: the data is stored first by column, and then by row.

The simplest way of mapping the indices (i,j) into one-dimensional memory is to take a linear combination.[2] Here's the appropriate linear combination for row major ordering:

```
memory offset = 3*i + 1*j
```

And for column major ordering:

```
memory offset = 1*i + 3*j
```

The coefficients of the (i,j) indices are called *strides*. For a row major storage of this array, the *row stride* is 3 – you have to skip three memory locations to move down a row. The *column stride* is 1 – you move one memory location to move to the next column. This is also known as *unit stride*. For column major ordering, the row and column strides are 1 and 3, respectively.

Bases

To throw another complication into this scheme, C-style arrays have indices which start at zero, and Fortran-style arrays have indices which start at one. The first valid index value is called the *base*. To account for a non-zero base, it's necessary to include an offset term in addition to the linear combination. Here's the mapping for a C-style array with i=0..3 and j=0..3:

```
memory offset =  0 + 3*i + 1*j
```

No offset is necessary since the indices start at zero for C-style arrays. For a Fortran-style array with i=1..4 and j=1..4, the mapping would be:

```
memory offset = -4 + 3*i + 1*j
```

By default, Blitz++ creates arrays in the C-style storage format (base zero, row major ordering). To create a Fortran-style array, you can use this syntax:

```
Array<int,2> A(3, 3, FortranArray<2>());
```

The third parameter, `FortranArray<2>()`, tells the `Array` constructor to use a storage format appropriate for two-dimensional Fortran arrays (base one, column major ordering).

A similar object, `ColumnMajor<N>`, tells the `Array` constructor to use column major ordering, with base zero:

```
Array<int,2> B(3, 3, ColumnMajor<2>());
```

This creates a 3x3 array with indices i=0..2 and j=0..2.

In addition to supporting the 0 and 1 conventions for C and Fortran-style arrays, Blitz++ allows you to choose arbitrary bases, possibly different for each dimension. For example, this declaration creates an array whose indices have ranges i=5..8 and j=2..5:

```
Array<int,2> A(Range(5,8), Range(2,5));
```

[2] Taking a linear combination is sufficient for dense, asymmetric arrays, such as are provided by the Blitz++ `Array` class.

2.9.2 Creating custom storage orders

All `Array` constructors take an optional parameter of type `GeneralArrayStorage<N_rank>`. This parameter encapsulates a complete description of the storage format. If you want a storage format other than C or Fortran-style, you have two choices:

- You can create an object of type `GeneralArrayStorage<N_rank>`, customize the storage format, and use the object as a argument for the `Array` constructor.
- You can create your own storage format object which inherits from `GeneralArrayStorage<N_rank>`. This is useful if you will be using the storage format many times. This approach (inheriting from `GeneralArrayStorage<N_rank>`) was used to create the `FortranArray<N_rank>` objects. If you want to take this approach, you can use the declaration of `FortranArray<N_rank>` in `<blitz/array.h>` as a guide.

The next sections describe how to modify a `GeneralArrayStorage<N_rank>` object to suit your needs.

In higher dimensions

In more than two dimensions, the choice of storage order becomes more complicated. Suppose we had a 3x3x3 array. To map the indices (i,j,k) into memory, we might choose one of these mappings:

```
memory offset = 9*i + 3*j + 1*k
memory offset = 1*i + 3*j + 9*k
```

The first corresponds to a C-style array, and the second to a Fortran-style array. But there are other choices; we can permute the strides (1,3,9) any which way:

```
memory offset = 1*i + 9*j + 3*k
memory offset = 3*i + 1*j + 9*k
memory offset = 3*i + 9*j + 1*k
memory offset = 9*i + 1*j + 3*k
```

For an N dimensional array, there are N! such permutations. Blitz++ allows you to select any permutation of the dimensions as a storage order. First you need to create an object of type `GeneralArrayStorage<N_rank>`:

```
GeneralArrayStorage<3> storage;
```

`GeneralArrayStorage<N_rank>` contains a vector called `ordering` which controls the order in which dimensions are stored in memory. The `ordering` vector will contain a permutation of the numbers 0, 1, ..., N_rank-1. Since some people are used to the first dimension being 1 rather than 0, a set of symbols (firstDim, secondDim, ..., eleventhDim) are provided which make the code more legible.

The `ordering` vector lists the dimensions in increasing order of stride. You can access this vector using the member function `ordering()`. A C-style array, the default, would have:

```
storage.ordering() = thirdDim, secondDim, firstDim;
```

meaning that the third index (k) is associated with the smallest stride, and the first index (i) is associated with the largest stride. A Fortran-style array would have:

```
storage.ordering() = firstDim, secondDim, thirdDim;
```

Reversed dimensions

To add yet another wrinkle, there are some applications where the rows or columns need to be stored in reverse order.[3]

[3] For example, certain bitmap formats store image rows from bottom to top rather than top to bottom.

Blitz++ allows you to store each dimension in either ascending or descending order. By default, arrays are always stored in ascending order. The `GeneralArrayStorage<N_rank>` object contains a vector called `ascendingFlag` which indicates whether each dimension is stored ascending (`true`) or descending (`false`). To alter the contents of this vector, use the `ascendingFlag()` method:

```
// Store the third dimension in descending order
storage.ascendingFlag() = true, true, false;

// Store all the dimensions in descending order
storage.ascendingFlag() = false, false, false;
```

Setting the base vector

`GeneralArrayStorage<N_rank>` also has a `base` vector which contains the base index value for each dimension. By default, the base vector is set to zero. `FortranArray<N_rank>` sets the base vector to one.

To set your own set of bases, you have two choices:

- You can modify the `base` vector inside the `GeneralArrayStorage<N_rank>` object. The method `base()` returns a mutable reference to the `base` vector which you can use to set the bases.
- You can provide a set of `Range` arguments to the `Array` constructor.

Here are some examples of the first approach:

```
// Set all bases equal to 5
storage.base() = 5;

// Set the bases to [ 1 0 1 ]
storage.base() = 1, 0, 1;
```

And of the second approach:

```
// Have bases of 5, but otherwise C-style storage
Array<int,3> A(Range(5,7), Range(5,7), Range(5,7));

// Have bases of [ 1 0 1 ] and use a custom storage
Array<int,3> B(Range(1,4), Range(0,3), Range(1,4), storage);
```

Working simultaneously with different storage orders

Once you have created an array object, you will probably never have to worry about its storage order. Blitz++ should handle arrays of different storage orders transparently. It's possible to mix arrays of different storage orders in one expression, and still get the correct result.

Note however, that mixing different storage orders in an expression may incur a performance penalty, since Blitz++ will have to pay more attention to differences in indexing than it normally would.

You may not mix arrays with different domains in the same expression. For example, adding a base zero to a base one array is a no-no. The reason for this restriction is that certain expressions become ambiguous, for example:

```
Array<int,1> A(Range(0,5)), B(Range(1,6));
A=0;
B=0;
using namespace blitz::tensor;
int result = sum(A+B+i);
```

Should the index `i` take its domain from array `A` or array `B`? To avoid such ambiguities, users are forbidden from mixing arrays with different domains in an expression.

Debug dumps of storage order information

In debug mode (`-DBZ_DEBUG`), class `Array` provides a member function `dumpStructureInformation()` which displays information about the array storage:

```
Array<float,4> A(3,7,8,2,FortranArray<4>());
A.dumpStructureInformation(cerr);
```

The optional argument is an `ostream` to dump information to. It defaults to `cout`. Here's the output:

```
Dump of Array<f, 4>:
ordering_      = 4 [         0          1          2          3 ]
ascendingFlag_ = 4 [         1          1          1          1 ]
base_          = 4 [         1          1          1          1 ]
length_        = 4 [         3          7          8          2 ]
stride_        = 4 [         1          3         21        168 ]
zeroOffset_    = -193
numElements()  = 336
isStorageContiguous() = 1
```

A note about storage orders and initialization

When initializing arrays with comma delimited lists, note that the array is filled in storage order: from the first memory location to the last memory location. This won't cause any problems if you stick with C-style arrays, but it can be confusing for Fortran-style arrays:

```
Array<int,2> A(3, 3, FortranArray<2>());
A = 1, 2, 3,
    4, 5, 6,
    7, 8, 9;
cout << A << endl;
```

The output from this code excerpt will be:

```
A = 3 x 3
         1         4         7
         2         5         8
         3         6         9
```

This is because Fortran-style arrays are stored in column major order.

2.9.3 Storage orders example

```
#include <blitz/array.h>

BZ_USING_NAMESPACE(blitz)

int main()
{
    // 3x3 C-style row major storage, base zero
    Array<int,2> A(3, 3);

    // 3x3 column major storage, base zero
    Array<int,2> B(3, 3, ColumnMajorArray<2>());

    // A custom storage format:
    // Indices have range 0..3, 0..3
    // Column major ordering
    // Rows are stored ascending, columns stored descending
    GeneralArrayStorage<2> storage;
    storage.ordering() = firstRank, secondRank;
    storage.base() = 0, 0;
    storage.ascendingFlag() = true, false;

    Array<int,2> C(3, 3, storage);
```

```
    // Set each array equal to
    // [ 1 2 3 ]
    // [ 4 5 6 ]
    // [ 7 8 9 ]

    A = 1, 2, 3,
        4, 5, 6,
        7, 8, 9;

    cout << "A = " << A << endl;

    // Comma-delimited lists initialize in memory-storage order only.
    // Hence we list the values in column-major order to initialize B:

    B = 1, 4, 7, 2, 5, 8, 3, 6, 9;

    cout << "B = " << B << endl;

    // Array C is stored in column major, plus the columns are stored
    // in descending order!

    C = 3, 6, 9, 2, 5, 8, 1, 4, 7;

    cout << "C = " << C << endl;

    Array<int,2> D(3,3);
    D = A + B + C;

#ifdef BZ_DEBUG
    A.dumpStructureInformation();
    B.dumpStructureInformation();
    C.dumpStructureInformation();
    D.dumpStructureInformation();
#endif

    cout << "D = " << D << endl;

    return 0;
}
```

And the output:

```
A = 3 x 3
[         1         2         3
          4         5         6
          7         8         9 ]

B = 3 x 3
[         1         2         3
          4         5         6
          7         8         9 ]

C = 3 x 3
[         1         2         3
          4         5         6
          7         8         9 ]

Dump of Array<i, 2>:
ordering_      = 2 [          1         0 ]
ascendingFlag_ = 2 [          1         1 ]
base_          = 2 [          0         0 ]
length_        = 2 [          3         3 ]
stride_        = 2 [          3         1 ]
zeroOffset_    = 0
```

```
numElements()  = 9
isStorageContiguous() = 1
Dump of Array<i, 2>:
ordering_      = 2 [          0         1 ]
ascendingFlag_ = 2 [          1         1 ]
base_          = 2 [          0         0 ]
length_        = 2 [          3         3 ]
stride_        = 2 [          1         3 ]
zeroOffset_    = 0
numElements()  = 9
isStorageContiguous() = 1
Dump of Array<i, 2>:
ordering_      = 2 [          0         1 ]
ascendingFlag_ = 2 [          1         0 ]
base_          = 2 [          0         0 ]
length_        = 2 [          3         3 ]
stride_        = 2 [          1        -3 ]
zeroOffset_    = 6
numElements()  = 9
isStorageContiguous() = 1
Dump of Array<i, 2>:
ordering_      = 2 [          1         0 ]
ascendingFlag_ = 2 [          1         1 ]
base_          = 2 [          0         0 ]
length_        = 2 [          3         3 ]
stride_        = 2 [          3         1 ]
zeroOffset_    = 0
numElements()  = 9
isStorageContiguous() = 1
D = 3 x 3
[         3         6         9
         12        15        18
         21        24        27 ]
```

3 Array Expressions

Array expressions in Blitz++ are implemented using the *expression templates* technique. Unless otherwise noted, expression evaluation will never generate temporaries or multiple loops; an expression such as

```
Array<int,1> A, B, C, D;    // ...

A = B + C + D;
```

will result in code similar to

```
for (int i=A.lbound(firstDim); i <= A.ubound(firstDim); ++i)
    A[i] = B[i] + C[i] + D[i];
```

3.1 Expression evaluation order

A commonly asked question about Blitz++ is what order it uses to evaluate array expressions. For example, in code such as

```
A(Range(2,10)) = A(Range(1,9))
```

does the expression get evaluated at indices 1, 2, ..., 9 or at 9, 8, ..., 1? This makes a big difference to the result: in one case, the array will be shifted to the right by one element; in the other case, most of the array elements will be set to the value in `A(1)`.

Blitz always selects the traversal order it thinks will be fastest. For 1D arrays, this means it will go from beginning to the end of the array in memory (see notes below). For multidimensional arrays, it will do one of two things:

- try to go through the destination array in the order it is laid out in memory (i.e. row-major for row-major arrays, column-major for column-major arrays).
- if the expression is a stencil, Blitz will do tiling to improve cache use. Under some circumstances blitz will even use a traversal based on a hilbert curve (a fractal) for 3D arrays.

Because the traversal order is not always predictable, it is safest to put the result in a new array if you are doing a stencil-style expression. Blitz guarantees this will always work correctly. If you try to put the result in one of the operands, you have to guess correctly which traversal order blitz will choose. This is easy for the 1D case, but hard for the multidimensional case.

Some special notes about 1D array traversals:

- if your array is stored in reverse order, i.e. because of a A.reverse(firstDim) or funny storage order, blitz will go through the array from end to beginning in array coordinates, but from beginning to end in memory locations.
- many compilers/architecture combinations are equally fast at reverse order. But blitz has a specialized version for stride = +1, and it would be wasteful to also specialize for the case stride = -1. So 1D arrays are traversed from beginning to end (in memory storage order).

3.2 Expression operands

An expression can contain any mix of these operands:

- An array of any type, so long as it is of the same rank. Expressions which contain a mixture of array types are handled through the type promotion mechanism described below.
- Scalars of type `int`, `float`, `double`, `long double`, or `complex<T>`
- Index placeholders, described below
- Other expressions (e.g. `A+(B+C)`)

3.3 Array operands

Using subarrays in an expression

Subarrays may be used in an expression. For example, this code example performs a 5-point average on a two-dimensional array:

```
Array<float,2> A(64,64), B(64,64);   // ...
Range I(1,62), J(1,62);

A(I,J) = (B(I,J) + B(I+1,J) + B(I-1,J)
                 + B(I,J+1) + B(I,J-1)) / 5;
```

Mixing arrays with different storage formats

Arrays with different storage formats (for example, C-style and Fortran-style) can be mixed in the same expression. Blitz++ will handle the different storage formats automatically. However:

- Evaluation may be slower, since a different traversal order may be used.
- If you are using index placeholders (see below) or reductions in the expression, you may **not** mix array objects with different starting bases.

3.4 Expression operators

These binary operators are supported:

```
+ - * / % > < >= <= == != && || ^ & |
```

Note: operator `<<` and `>>` are reserved for use in input/output. If you need a bit-shift operation on arrays, you may define one yourself; see Section 3.10 [User et], page 42.

These unary operators are supported:

```
- ~ !
```

The operators `> < >= <= == != && || !` result in a bool-valued expression.

All operators are applied *elementwise*.

You can only use operators which are well-defined for the number type stored in the arrays. For example, bitwise XOR (`^`) is meaningful for integers, so this code is all right:

```
Array<int,3> A, B, C;   // ...
A = B ^ C;
```

Bitwise XOR is *not* meaningful on floating point types, so this code will generate a compiler error:

```
Array<float,1> A, B, C;   // ...
C = B ^ C;
```

Here's the compiler error generated by KAI C++ for the above code:

```
"../../blitz/ops.h", line 85: error: expression must have integral or enum type
  BZ_DEFINE_OP(BitwiseXor,^);
  ^

          detected during:
            instantiation of "blitz::BitwiseXor<float, float>::T_numtype
                      blitz::BitwiseXor<float, float>::apply(float, float)" at
                      line 210 of "../../blitz/arrayexpr.h"
            instantiation of ...
                        .
                        .
```

If you are creating arrays using a type you have created yourself, you will need to overload whatever operators you want to use on arrays. For example, if I create a class `Polynomial`, and want to write code such as:

```
Array<Polynomial,2> A, B, C;   // ...
C = A * B;
```

I would have to provide `operator*` for `Polynomial` by implementing

```
Polynomial Polynomial::operator*(Polynomial);)
```

or

```
Polynomial operator*(Polynomial, Polynomial);)
```

3.5 Assignment operators

These assignment operators are supported:

```
= += -= *= /= %= ^= &= |= >>= <<=
```

An array object should appear on the left side of the operator. The right side can be:

- A constant (or literal) of type `T_numtype`
- An array of appropriate rank, possibly of a different numeric type
- An array expression, with appropriate rank and shape

3.6 Index placeholders

Blitz++ provides objects called *index placeholders* which represent array indices. They can be used directly in expressions.

There is a distinct index placeholder type associated with each dimension of an array. The types are called `firstIndex`, `secondIndex`, `thirdIndex`, ..., `tenthIndex`, `eleventhIndex`. Here's an example of using an index placeholder:

```
Array<float,1> A(10);
firstIndex i;
A = i;
```

This generates code which is similar to:

```
for (int i=0; i < A.length(); ++i)
    A(i) = i;
```

Here's an example which fills an array with a sampled sine wave:

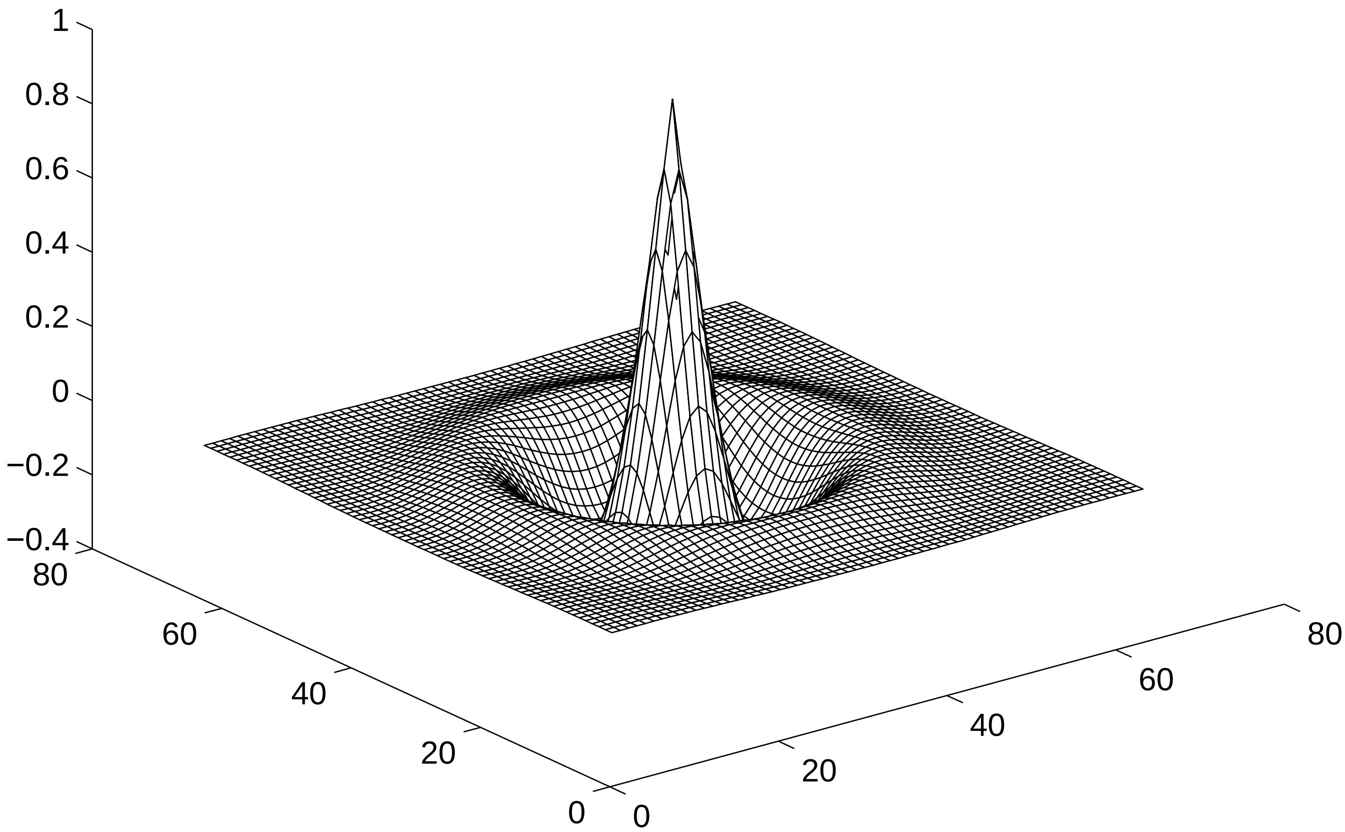
```
Array<float,1> A(16);
firstIndex i;

A = sin(2 * M_PI * i / 16.);
```

If your destination array has rank greater than 1, you may use multiple index placeholders:

```
// Fill a two-dimensional array with a radially
// symmetric, decaying sinusoid

// Create the array
int N = 64;
Array<float,2> F(N,N);

// Some parameters
float midpoint = (N-1)/2.;
int cycles = 3;
float omega = 2.0 * M_PI * cycles / double(N);
```

```
float tau = - 10.0 / N;

// Index placeholders
firstIndex i;
secondIndex j;

// Fill the array
F = cos(omega * sqrt(pow2(i-midpoint) + pow2(j-midpoint)))
    * exp(tau * sqrt(pow2(i-midpoint) + pow2(j-midpoint)));
```

Here's a plot of the resulting array:



Array filled using an index placeholder expression.

You can use index placeholder expressions in up to 11 dimensions. Here's a three dimensional example:

```
// Fill a three-dimensional array with a Gaussian function
Array<float,3> A(16,16,16);
firstIndex i;
secondIndex j;
thirdIndex k;
float midpoint = 15/2.;
float c = - 1/3.0;
A = exp(c * (sqr(i-midpoint) + sqr(j-midpoint)
    + sqr(k-midpoint)));
```

You can mix array operands and index placeholders:

```
Array<int,1> A(5), B(5);
firstIndex i;

A = 0, 1, 1, 0, 2;
B = i * A;          // Results in [ 0, 1, 2, 0, 8 ]
```

For your convenience, there is a namespace within blitz called tensor which declares all the index placeholders:

```
namespace blitz {
  namespace tensor {
    firstIndex i;
    secondIndex j;
    thirdIndex k;
     ...
    eleventhIndex t;
  }
}
```

So instead of declaring your own index placeholder objects, you can just say

```
namespace blitz::tensor;
```

when you would like to use them. Alternately, you can just preface all the index placeholders with tensor::, for example:

```
A = sin(2 * M_PI * tensor::i / 16.);
```

This will make your code more readable, since it is immediately clear that i is an index placeholder, rather than a scalar value.

3.7 Type promotion

When operands of different numeric types are used in an expression, the result gets promoted according to the usual C-style type promotion. For example, the result of adding an Array<int> to an Arrray<float> will be promoted to float. Generally, the result is promoted to whichever type has greater precision.

Type promotion for user-defined types

The rules for type promotion of user-defined types (or types from another library) are a bit complicated. Here's how a pair of operand types are promoted:

- If both types are intrinsic (e.g. bool, int, float) then type promotion follows the standard C rules. This generally means that the result will be promoted to whichever type has greater precision. In Blitz++, these rules have been extended to incorporate complex<float>, complex<double>, and complex<long double>.
- If one of the types is intrinsic (or complex), and the other is a user-defined type, then the result is promoted to the user-defined type.
- If both types are user-defined, then the result is promoted to whichever type requires more storage space (as determined by sizeof()). The rationale is that more storage space probably indicates more precision.

If you wish to alter the default type promotion rules above, you have two choices:

- If the type promotion behaviour isn't dependent on the type of operation performed, then you can provide appropriate specializations for the class promote_trait<A,B> which is declared in <blitz/promote.h>.

- If type promotion does depend on the type of operation, then you will need to specialize the appropriate function objects in `<blitz/ops.h>`.

Note that you can do these specializations in your own header files (you don't have to edit 'promote.h' or 'ops.h').

Manual casts

There are some inconvenient aspects of C-style type promotion. For example, when you divide two integers in C, the result gets truncated. The same problem occurs when dividing two integer arrays in Blitz++:

```
Array<int,1> A(4), B(4);
Array<float,1> C(4);

A = 1, 2, 3, 5;
B = 2, 2, 2, 7;

C = A / B;      // Result:  [ 0  1  1  0 ]
```

The usual solution to this problem is to cast one of the operands to a floating type. For this purpose, Blitz++ provides a function `cast(expr,type)` which will cast the result of *expr* as *type*:

```
C = A / cast(B, float());   // Result: [ 0.5  1  1.5  0.714 ]
```

The first argument to `cast()` is an array or expression. The second argument is a dummy object of the type to which you want to cast. Once compilers support templates more thoroughly, it will be possible to use this cast syntax:

```
C = A / cast<float>(B);
```

But this is not yet supported.

3.8 Single-argument math functions

All of the functions described in this section are *element-wise*. For example, this code–

```
Array<float,2> A, B;   //
A = sin(B);
```

results in `A(i,j) = sin(B(i,j))` for all (i,j).

ANSI C++ math functions

These math functions are available on all platforms:

`abs()` Absolute value

`acos()` Inverse cosine. For real arguments, the return value is in the range $[0, \pi]$.

`arg()` Argument of a complex number (`atan2(Im,Re)`).

`asin()` Inverse sine. For real arguments, the return value is in the range $[-\pi/2, \pi/2]$.

`atan()` Inverse tangent. For real arguments, the return value is in the range $[-\pi/2, \pi/2]$. See also `atan2()` in section Section 3.9 [Math functions 2], page 41.

`ceil()` Ceiling function: smallest floating-point integer value not less than the argument.

`cexp()` Complex exponential; same as `exp()`.

`conj()` Conjugate of a complex number.

`cos()` Cosine. Works for `complex<T>`.

cosh() Hyperbolic cosine. Works for complex<T>.

csqrt() Complex square root; same as sqrt().

exp() Exponential. Works for complex<T>.

fabs() Same as abs().

floor() Floor function: largest floating-point integer value not greater than the argument.

log() Natural logarithm. Works for complex<T>.

log10() Base 10 logarithm. Works for complex<T>.

pow2(), pow3(), pow4(), pow5(), pow6(), pow7(), pow8()
These functions compute an integer power. They expand to a series of multiplications, so they can be used on any type for which multiplication is well-defined.

sin() Sine. Works for complex<T>.

sinh() Hyperbolic sine. Works for complex<T>.

sqr() Same as pow2(). Computes x*x. Works for complex<T>.

sqrt() Square root. Works for complex<T>.

tan() Tangent. Works for complex<T>.

tanh() Hyperbolic tangent. Works for complex<T>.

IEEE/System V math functions

These functions are only available on platforms which provide the IEEE Math library (libm.a) and/or System V Math Library (libmsaa.a). Apparently not all platforms provide all of these functions, so what you can use on your platform may be a subset of these. If you choose to use one of these functions, be aware that you may be limiting the portability of your code.

On some platforms, the preprocessor symbols _XOPEN_SOURCE and/or _XOPEN_SOURCE_EXTENDED need to be defined to use these functions. These symbols can be enabled by compiling with -DBZ_ENABLE_XOPEN_SOURCE. (In previous version of Blitz++, _XOPEN_SOURCE and _XOPEN_SOURCE_EXTENDED were declared by default. This was found to cause too many problems, so users must manually enable them with -DBZ_ENABLE_XOPEN_SOURCE.).

In the current version, Blitz++ divides these functions into two groups: IEEE and System V. This distinction is probably artificial. If one of the functions in a group is missing, Blitz++ won't allow you to use any of them. You can see the division of these functions in the files 'Blitz++/compiler/ieeemath.cpp' and 'Blitz++/compiler/sysvmath.cpp'. This arrangement is unsatisfactory and will probably change in a future version.

You may have to link with -lm and/or -lmsaa to use these functions.

None of these functions are available for complex<T>.

acosh() Inverse hyperbolic cosine

asinh() Inverse hyperbolic sine

atanh() Inverse hyperbolic tangent

_class() Classification of floating point values. The return type is integer and will be one of:

FP_PLUS_NORM
Positive normalized, nonzero

FP_MINUS_NORM
Negative normalized, nonzero

`FP_PLUS_DENORM`
: Positive denormalized, nonzero

`FP_MINUS_DENORM`
: Negative denormalized, nonzero

`FP_PLUS_ZERO`
: +0.0

`FP_MINUS_ZERO`
: -0.0

`FP_PLUS_INF`
: Positive infinity

`FP_MINUS_INF`
: Negative infinity

`FP_NANS` Signalling Not a Number (NaNS)

`FP_NANQ` Quiet Not a Number (NaNQ)

`cbrt()` Cubic root

`expm1()` Computes exp(x)-1

`erf()` Computes the error function:

$$\mathrm{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

Note that for large values of the parameter, calculating can result in extreme loss of accuracy. Instead, use `erfc()`.

`erfc()` Computes the complementary error function $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

`finite()` Returns a nonzero integer if the parameter is a finite number (i.e. not +INF, -INF, NaNQ or NaNS).

`ilogb()` Returns an integer which is equal to the unbiased exponent of the parameter.

`blitz_isnan()`
: Returns a nonzero integer if the parameter is NaNQ or NaNS (quiet or signalling Not a Number).

`itrunc()` Round a floating-point number to a signed integer. Returns the nearest signed integer to the parameter in the direction of 0.

`j0()` Bessel function of the first kind, order 0.

`j1()` Bessel function of the first kind, order 1.

`lgamma()` Natural logarithm of the gamma function. The gamma function is defined as:

$$\mathrm{Gamma}(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

`logb()` Returns a floating-point double that is equal to the unbiased exponent of the parameter.

`log1p()` Calculates log(1+x), where x is the parameter.

`nearest()`
: Returns the nearest floating-point integer value to the parameter. If the parameter is exactly halfway between two integer values, an even value is returned.

`rint()` Rounds the parameter and returns a floating-point integer value. Whether `rint()` rounds up or down or to the nearest integer depends on the current floating-point rounding mode. If you haven't altered the rounding mode, `rint()` should be equivalent to `nearest()`. If rounding mode is set to round towards +INF, `rint()` is equivalent to `ceil()`. If the mode is round toward -INF, `rint()` is equivalent to `floor()`. If the mode is round toward zero, `rint()` is equivalent to `trunc()`.

`rsqrt()` Reciprocal square root.

`uitrunc()`
Returns the nearest unsigned integer to the parameter in the direction of zero.

`y0()` Bessel function of the second kind, order 0.

`y1()` Bessel function of the second kind, order 1.

There may be better descriptions of these functions in your system man pages.

3.9 Two-argument math functions

The math functions described in this section take two arguments. Most combinations of these types may be used as arguments:

- An Array object
- An Array expression
- An index placeholder
- A scalar of type `float`, `double`, `long double`, or `complex<T>`

ANSI C++ math functions

These math functions are available on all platforms, and work for complex numbers.

`atan2(x,y)`
Inverse tangent of (y/x). The signs of both parameters are used to determine the quadrant of the return value, which is in the range $[-\pi, \pi]$. Works for `complex<T>`.

`blitz::polar(r,t)`
Computes ; i.e. converts polar-form to Cartesian form complex numbers. The `blitz::` scope qualifier is needed to disambiguate the ANSI C++ function template `polar(T,T)`. This qualifier will hopefully disappear in a future version.

`pow(x,y)` Computes x to the exponent y. Works for `complex<T>`.

IEEE/System V math functions

See the notes about IEEE/System V math functions in the previous section. None of these functions work for complex numbers. They will all cast their arguments to double precision.

`copysign(x,y)`
Returns the x parameter with the same sign as the y parameter.

`drem(x,y)`
Computes a floating point remainder. The return value r is equal to r = x - n * y, where n is equal to `nearest(x/y)` (the nearest integer to x/y). The return value will lie in the range [-y/2, +y/2]. If y is zero or x is +INF or -INF, NaNQ is returned.

`fmod(x,y)`
Computes a floating point modulo remainder. The return value r is equal to r = x - n * y, where n is selected so that r has the same sign as x and magnitude less than abs(y). In order words, if x > 0, r is in the range [0, |y|], and if x < 0, r is in the range [-|y|, 0].

`hypot(x,y)`
: Computes so that underflow does not occur and overflow occurs only if the final result warrants it.

`nextafter(x,y)`
: Returns the next representable number after x in the direction of y.

`remainder(x,y)`
: Equivalent to drem(x,y).

`scalb(x,y)`
: Calculates.

`unordered(x,y)`
: Returns a nonzero value if a floating-point comparison between x and y would be unordered. Otherwise, it returns zero.

3.10 Declaring your own math functions on arrays

There are four macros which make it easy to turn your own scalar functions into functions defined on arrays. They are:

```
BZ_DECLARE_FUNCTION(f)                   // 1
BZ_DECLARE_FUNCTION_RET(f,return_type)   // 2
BZ_DECLARE_FUNCTION2(f)                  // 3
BZ_DECLARE_FUNCTION2_RET(f,return_type)  // 4
```

Use version 1 when you have a function which takes one argument and returns a result of the same type. For example:

```
#include <blitz/array.h>

using namespace blitz;

double myFunction(double x)
{
    return 1.0 / (1 + x);
}

BZ_DECLARE_FUNCTION(myFunction)

int main()
{
    Array<double,2> A(4,4), B(4,4);  // ...
    B = myFunction(A);
}
```

Use version 2 when you have a one argument function whose return type is different than the argument type, such as

```
int g(double x);
```

Use version 3 for a function which takes two arguments and returns a result of the same type, such as:

```
double g(double x, double y);
```

Use version 4 for a function of two arguments which returns a different type, such as:

```
int g(double x, double y);
```

3.11 Tensor notation

Blitz++ arrays support a tensor-like notation. Here's an example of real-world tensor notation:

$$A^{ijk} = B^{ij} C^k$$

A is a rank 3 tensor (a three dimensional array), B is a rank 2 tensor (a two dimensional array), and C is a rank 1 tensor (a one dimensional array). The above expression sets `A(i,j,k) = B(i,j) * C(k)`.

To implement this product using Blitz++, we'll need the arrays and some index placeholders:

```
Array<float,3> A(4,4,4);
Array<float,2> B(4,4);
Array<float,1> C(4);

firstIndex i;    // Alternately, could just say
secondIndex j;   // using namespace blitz::tensor;
thirdIndex k;
```

Here's the Blitz++ code which is equivalent to the tensor expression:

```
A = B(i,j) * C(k);
```

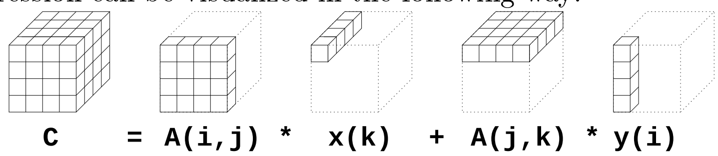
The index placeholder arguments tell an array how to map its dimensions onto the dimensions of the destination array. For example, here's some real-world tensor notation:

$$C^{ijk} = A^{ij} x^k - A^{jk} y^i$$

In Blitz++, this would be coded as:

```
using namespace blitz::tensor;

C = A(i,j) * x(k) - A(j,k) * y(i);
```

This tensor expression can be visualized in the following way:



Examples of array indexing, subarrays, and slicing.

Here's an example which computes an outer product of two one-dimensional arrays:

```
#include <blitz/array.h>

using namespace blitz;

int main()
{
    Array<float,1> x(4), y(4);
    Array<float,2> A(4,4);

    x = 1, 2, 3, 4;
    y = 1, 0, 0, 1;

    firstIndex i;
    secondIndex j;

    A = x(i) * y(j);

    cout << A << endl;
```

```
    return 0;
}
```

And the output:

```
4 x 4
[         1         0         0         1
          2         0         0         2
          3         0         0         3
          4         0         0         4 ]
```

Index placeholders can *not* be used on the left-hand side of an expression. If you need to reorder the indices, you must do this on the right-hand side.

In real-world tensor notation, repeated indices imply a contraction (or summation). For example, this tensor expression computes a matrix-matrix product:

$$C^{ij} = A^{ik} B^{kj}$$

The repeated k index is interpreted as meaning

$$c_{ij} = \sum_k a_{ik} b_{kj}$$

In Blitz++, repeated indices do *not* imply contraction. If you want to contract (sum along) an index, you must use the `sum()` function:

```
Array<float,2> A, B, C;   // ...
firstIndex i;
secondIndex j;
thirdIndex k;

C = sum(A(i,k) * B(k,j), k);
```

The `sum()` function is an example of an *array reduction*, described in the next section.

Index placeholders can be used in any order in an expression. This example computes a kronecker product of a pair of two-dimensional arrays, and permutes the indices along the way:

```
Array<float,2> A, B;   // ...
Array<float,4> C;      // ...
fourthIndex l;

C = A(l,j) * B(k,i);
```

This is equivalent to the tensor notation

$$C^{ijkl} = A^{lj} B^{ki}$$

Tensor-like notation can be mixed with other array notations:

```
Array<float,2> A, B;  // ...
Array<double,4> C;    // ...

C = cos(A(l,j)) * sin(B(k,i)) + 1./(i+j+k+l);
```

An important efficiency note about tensor-like notation: the right-hand side of an expression is *completely evaluated* for *every* element in the destination array. For example, in this code:

```
Array<float,1> x(4), y(4);
Array<float,2> A(4,4):

A = cos(x(i)) * sin(y(j));
```

The resulting implementation will look something like this:

```
for (int n=0; n < 4; ++n)
  for (int m=0; m < 4; ++m)
    A(n,m) = cos(x(n)) * sin(y(m));
```

The functions `cos` and `sin` will be invoked sixteen times each. It's possible that a good optimizing compiler could hoist the `cos` evaluation out of the inner loop, but don't hold your breath – there's a lot of complicated machinery behind the scenes to handle tensor notation, and most optimizing compilers are easily confused. In a situation like the above, you are probably best off manually creating temporaries for `cos(x)` and `sin(y)` first.

3.12 Array reductions

Currently, Blitz++ arrays support two forms of reduction:

- Reductions which transform an array into a scalar (for example, summing the elements). These are referred to as **complete reductions**.
- Reducing an N dimensional array (or array expression) to an N-1 dimensional array expression. These are called **partial reductions**.

3.13 Complete reductions

Complete reductions transform an array (or array expression) into a scalar. Here are some examples:

```
Array<float,2> A(3,3);
A = 0, 1, 2,
    3, 4, 5,
    6, 7, 8;
cout << sum(A) << endl          // 36
     << min(A) << endl          // 0
     << count(A >= 4) << endl;  // 5
```

Here are the available complete reductions:

`sum()` Summation (may be promoted to a higher-precision type)

`product()`
Product

`mean()` Arithmetic mean (promoted to floating-point type if necessary)

`min()` Minimum value

`max()` Maximum value

`minIndex()`
Index of the minimum value (`TinyVector<int,N_rank>`)

`maxIndex()`
Index of the maximum value (`TinyVector<int,N_rank>`)

`count()` Counts the number of times the expression is logical true (`int`)

`any()` True if the expression is true anywhere (`bool`)

`all()` True if the expression is true everywhere (`bool`)

Note: `minIndex()` and `maxIndex()` return TinyVectors, even when the rank of the array (or array expression) is 1.

Reductions can be combined with `where` expressions (Section 3.15 [Where expr], page 48) to reduce over some part of an array. For example, `sum(where(A > 0, A, 0))` sums only the positive elements in an array.

3.14 Partial Reductions

Here's an example which computes the sum of each row of a two-dimensional array:

```
Array<float,2> A;    // ...
Array<float,1> rs;   // ...
firstIndex i;
secondIndex j;

rs = sum(A, j);
```

The reduction `sum()` takes two arguments:

- The first argument is an array or array expression.
- The second argument is an index placeholder indicating the dimension over which the reduction is to occur.

Reductions have an **important restriction**: It is currently only possible to reduce over the *last* dimension of an array or array expression. Reducing a dimension other than the last would require Blitz++ to reorder the dimensions to fill the hole left behind. For example, in order for this reduction to work:

```
Array<float,3> A;   // ...
Array<float,2> B;   // ...
secondIndex j;

// Reduce over dimension 2 of a 3-D array?
B = sum(A, j);
```

Blitz++ would have to remap the dimensions so that the third dimension became the second. It's not currently smart enough to do this.

However, there is a simple workaround which solves some of the problems created by this limitation: you can do the reordering manually, prior to the reduction:

```
B = sum(A(i,k,j), k);
```

Writing `A(i,k,j)` interchanges the second and third dimensions, permitting you to reduce over the second dimension. Here's a list of the reduction operations currently supported:

`sum()` Summation

`product()`
Product

`mean()` Arithmetic mean (promoted to floating-point type if necessary)

`min()` Minimum value

`max()` Maximum value

`minIndex()`
Index of the minimum value (int)

`maxIndex()`
Index of the maximum value (int)

`count()` Counts the number of times the expression is logical true (int)

`any()` True if the expression is true anywhere (bool)

`all()` True if the expression is true everywhere (bool)

`first()` First index at which the expression is logical true (int); if the expression is logical true nowhere, then `tiny(int())` (INT_MIN) is returned.

`last()` Last index at which the expression is logical true (int); if the expression is logical true nowhere, then `huge(int())` (INT_MAX) is returned.

The reductions `any()`, `all()`, and `first()` have short-circuit semantics: the reduction will halt as soon as the answer is known. For example, if you use `any()`, scanning of the expression will stop as soon as the first true value is encountered.

To illustrate, here's an example:

```
Array<int, 2> A(4,4);

A =  3,   8,   0,   1,
     1,  -1,   9,   3,
     2,  -5,  -1,   1,
     4,   3,   4,   2;

Array<float, 1> z;
firstIndex i;
secondIndex j;

z = sum(A(j,i), j);
```

The array `z` now contains the sum of `A` along each column:

```
[ 10    5     12    7 ]
```

This table shows what the result stored in `z` would be if `sum()` were replaced with other reductions:

```
sum                       [         10         5         12          7 ]
mean                      [        2.5      1.25          3       1.75 ]
min                       [          1        -5         -1          1 ]
minIndex                  [          1         2          2          0 ]
max                       [          4         8          9          3 ]
maxIndex                  [          3         0          1          1 ]
first((A < 0), j)         [ -2147483648        1          2 -2147483648 ]
product                   [         24       120          0          6 ]
count((A(j,i) > 0), j)    [          4         2          2          4 ]
any(abs(A(j,i)) > 4, j)   [          0         1          1          0 ]
all(A(j,i) > 0, j)        [          1         0          0          1 ]
```

Note: the odd numbers for first() are `tiny(int())` i.e. the smallest number representable by an int. The exact value is machine-dependent.

The result of a reduction is an array expression, so reductions can be used as operands in an array expression:

```
Array<int,3> A;
Array<int,2> B;
Array<int,1> C;   // ...

secondIndex j;
thirdIndex k;

B = sqrt(sum(sqr(A), k));

// Do two reductions in a row
C = sum(sum(A, k), j);
```

Note that this is not allowed:

```
Array<int,2> A;
firstIndex i;
secondIndex j;

// Completely sum the array?
int result = sum(sum(A, j), i);
```

You cannot reduce an array to zero dimensions! Instead, use one of the global functions described in the previous section.

3.15 where statements

Blitz++ provides the `where` function as an array expression version of the `( ? : )` operator. The syntax is:

```
where(array-expr1, array-expr2, array-expr3)
```

Wherever `array-expr1` is true, `array-expr2` is returned. Where `array-expr1` is false, `array-expr3` is returned. For example, suppose we wanted to sum the squares of only the positive elements of an array. This can be implemented using a where function:

```
double posSquareSum = sum(where(A > 0, pow2(A), 0));
```

4 Stencils

Blitz++ provides an implementation of stencil objects which is currently **experimental**. This means that the exact details of how they are declared and used may change in future releases. Use at your own risk.

4.1 Motivation: a nicer notation for stencils

Suppose we wanted to implement the 3-D acoustic wave equation using finite differencing. Here is how a single iteration would look using subarray syntax:

```
Range I(1,N-2), J(1,N-2), K(1,N-2);

P3(I,J,K) = (2-6*c(I,J,K)) * P2(I,J,K)
            + c(I,J,K)*(P2(I-1,J,K) + P2(I+1,J,K) + P2(I,J-1,K) + P2(I,J+1,K)
            + P2(I,J,K-1) + P2(I,J,K+1)) - P1(I,J,K);
```

This syntax is a bit klunky. With stencil objects, the implementation becomes:

```
BZ_DECLARE_STENCIL4(acoustic3D_stencil,P1,P2,P3,c)
  P3 = 2 * P2 + c * Laplacian3D(P2) - P1;
BZ_END_STENCIL

  .
  .

applyStencil(acoustic3D_stencil(), P1, P2, P3, c);
```

4.2 Declaring stencil objects

A stencil declaration may not be inside a function. It can appear inside a class declaration (in which case the stencil object is a nested type).

Stencil objects are declared using the macros BZ_DECLARE_STENCIL1, BZ_DECLARE_STENCIL2, etc. The number suffix is how many arrays are involved in the stencil (in the above example, 4 arrays– P1, P2, P3, c – are used, so the macro BZ_DECLARE_STENCIL4 is invoked).

The first argument is a name for the stencil object. Subsequent arguments are names for the arrays on which the stencil operates.

After the stencil declaration, the macro BZ_END_STENCIL must appear (or the macro BZ_END_STENCIL_WITH_SHAPE, described in the next section).

In between the two macros, you can have multiple assignment statements, if/else/elseif constructs, function calls, loops, etc.

Here are some simple examples:

```
BZ_DECLARE_STENCIL2(smooth2D,A,B)
  A = (B(0,0) + B(0,1) + B(0,-1) + B(1,0) + B(-1,0)) / 5.0;
BZ_END_STENCIL

BZ_DECLARE_STENCIL4(acoustic2D,P1,P2,P3,c)
  A = 2 * P2 + c * (-4 * P2(0,0) + P2(0,1) + P2(0,-1) + P2(1,0) + P2(-1,0))
      - P1;
BZ_END_STENCIL

BZ_DECLARE_STENCIL8(prop2D,E1,E2,E3,M1,M2,M3,cE,cM)
  E3 = 2 * E2 + cE * Laplacian2D(E2) - E1;
  M3 = 2 * M2 + cM * Laplacian2D(M2) - M1;
```

```
BZ_END_STENCIL

BZ_DECLARE_STENCIL3(smooth2Db,A,B,c)
  if ((c > 0.0) && (c < 1.0))
    A = c * (B(0,0) + B(0,1) + B(0,-1) + B(1,0) + B(-1,0)) / 5.0
      + (1-c)*B;
  else
    A = 0;
BZ_END_STENCIL
```

Currently, a stencil can take up to 11 array parameters.

You can use the notation `A(i,j,k)` to read the element at an offset `(i,j,k)` from the current element. If you omit the parentheses (i.e. as in "`A`" then the current element is read.

You can invoke *stencil operators* which calculate finite differences and laplacians.

4.3 Automatic determination of stencil extent

In stencil declarations such as

```
BZ_DECLARE_STENCIL2(smooth2D,A,B)
  A = (B(0,0) + B(0,1) + B(0,-1) + B(1,0) + B(-1,0)) / 5.0;
BZ_END_STENCIL
```

Blitz++ will try to automatically determine the spatial extent of the stencil. This will usually work for stencils defined on integer or float arrays. However, the mechanism does not work well for complex-valued arrays, or arrays of user-defined types. If you get a peculiar error when you try to use a stencil, you probably need to tell Blitz++ the special extent of the stencil manually.

You do this by ending a stencil declaration with `BZ_END_STENCIL_WITH_SHAPE`:

```
BZ_DECLARE_STENCIL2(smooth2D,A,B)
  A = (B(0,0) + B(0,1) + B(0,-1) + B(1,0) + B(-1,0)) / 5.0;
BZ_END_STENCIL_WITH_SHAPE(shape(-1,-1),shape(+1,+1))
```

The parameters of this macro are: a `TinyVector` (constructed by the `shape()` function) containing the lower bounds of the stencil offsets, and a `TinyVector` containing the upper bounds. You can determine this by looking at the the terms in the stencil and finding the minimum and maximum value of each index:

```
      A = (B(0,  0)
         + B(0, +1)
         + B(0, -1)
         + B(+1, 0)
         + B(-1, 0)) / 5.0;
           --------
min indices  -1, -1
max indices  +1, +1
```

4.4 Stencil operators

This section lists all the stencil operators provided by Blitz++. They assume that an array represents evenly spaced data points separated by a distance of `h`. A 2nd-order accurate operator has error term $O(h^2)$; a 4th-order accurate operator has error term $O(h^4)$.

All of the stencils have factors associated with them. For example, the `central12` operator is a discrete first derivative which is 2nd-order accurate. Its factor is 2h; this means that to get the first derivative of an array A, you need to use `central12(A,firstDim)`$/(2h)$. Typically when designing stencils, one factors out all of the h terms for efficiency.

The factor terms always consist of an integer multiplier (often 1) and a power of h. For ease of use, all of the operators listed below are provided in a second "normalized" version in which the integer multiplier is 1. The normalized versions have an `n` appended to the name, for example `central12n` is the normalized version of `central12`, and has factor h instead of $2h$.

These operators are defined in `blitz/array/stencilops.h` if you wish to see the implementation.

4.4.1 Central differences

`central12(A,dimension)`

1st derivative, 2nd order accurate. Factor: $2h$

-1	0	1
-1		1

`central22(A,dimension)`

2nd derivative, 2nd order accurate. Factor: h^2

-1	0	1
1	**-2**	1

`central32(A,dimension)`

3rd derivative, 2nd order accurate. Factor: $2h^3$

-2	-1	0	1	2
-1	2		-2	1

`central42(A,dimension)`

4th derivative, 2nd order accurate. Factor: h^4

-2	-1	0	1	2
1	-4	**6**	-4	1

`central14(A,dimension)`

1st derivative, 4th order accurate. Factor: $12h$

-2	-1	0	1	2
1	-8		8	-1

`central24(A,dimension)`

2nd derivative, 4th order accurate. Factor: $12h^2$

-2	-1	0	1	2
-1	16	**-30**	16	-1

`central34(A,dimension)`

3rd derivative, 4th order accurate. Factor: $8h^3$

-2	-1	0	1	2
-8	13		-13	8

`central44(A,dimension)`

4th derivative, 4th order accurate. Factor: $6h^4$

-2	-1	0	1	2
12	-39	**56**	-39	12

Note that the above are available in normalized versions `central12n`, `central22n`, ..., `central44n` which have factors of h, h^2, h^3, or h^4 as appropriate.

These are available in multicomponent versions: for example, `central12(A,component,dimension)` gives the central12 operator for the specified component (Components are numbered 0, 1, ... N-1).

4.4.2 Forward differences

`forward11(A,dimension)`
1st derivative, 1st order accurate. Factor: h

0	1
-1	1

`forward21(A,dimension)`
2nd derivative, 1st order accurate. Factor: h^2

0	1	2
1	-2	1

`forward31(A,dimension)`
3rd derivative, 1st order accurate. Factor: h^3

0	1	2	3
-1	3	-3	1

`forward41(A,dimension)`
4th derivative, 1st order accurate. Factor: h^4

0	1	2	3	4
1	-4	6	-4	1

`forward12(A,dimension)`
1st derivative, 2nd order accurate. Factor: $2h$

0	1	2
-3	4	-1

`forward22(A,dimension)`
2nd derivative, 2nd order accurate. Factor: h^2

0	1	2	3
2	-5	4	-1

`forward32(A,dimension)`
3rd derivative, 2nd order accurate. Factor: $2h^3$

0	1	2	3	4
-5	18	-24	14	-3

`forward42(A,dimension)`
4th derivative, 2nd order accurate. Factor: h^4

0	1	2	3	4	5
3	-14	26	-24	11	-2

Note that the above are available in normalized versions `forward11n`, `forward21n`, ..., `forward42n` which have factors of h, h^2, h^3, or h^4 as appropriate.

These are available in multicomponent versions: for example, `forward11(A,component,dimension)` gives the forward11 operator for the specified component (Components are numbered 0, 1, ... N-1).

4.4.3 Backward differences

`backward11(A,dimension)`

1st derivative, 1st order accurate. Factor: h

-1	0
-1	**1**

`backward21(A,dimension)`

2nd derivative, 1st order accurate. Factor: h^2

-2	-1	0
1	-2	**1**

`backward31(A,dimension)`

3rd derivative, 1st order accurate. Factor: h^3

-3	-2	-1	0
-1	3	-3	**1**

`backward41(A,dimension)`

4th derivative, 1st order accurate. Factor: h^4

-4	-3	-2	-1	0
1	-4	6	-4	**1**

`backward12(A,dimension)`

1st derivative, 2nd order accurate. Factor: $2h$

-2	-1	0
1	-4	**3**

`backward22(A,dimension)`

2nd derivative, 2nd order accurate. Factor: h^2

-3	-2	-1	0
-1	4	-5	**2**

`backward32(A,dimension)`

3rd derivative, 2nd order accurate. Factor: $2h^3$

-4	-3	-2	-1	0
3	-14	24	-18	**5**

`backward42(A,dimension)`

4th derivative, 2nd order accurate. Factor: h^4

-5	-4	-3	-2	-1	0
-2	11	-24	26	-14	**3**

Note that the above are available in normalized versions `backward11n`, `backward21n`, ..., `backward42n` which have factors of h, h^2, h^3, or h^4 as appropriate.

These are available in multicomponent versions: for example, `backward42(A,component,dimension)` gives the backward42 operator for the specified component (Components are numbered 0, 1, ... N-1).

4.4.4 Laplacian (∇^2) operators

`Laplacian2D(A)`

2nd order accurate, 2-dimensional laplacian. Factor: h^2

	-1	0	1
-1		1	
0	1	**-4**	1
1		1	

`Laplacian3D(A)`
: 2nd order accurate, 3-dimensional laplacian. Factor: h^2

`Laplacian2D4(A)`
: 4th order accurate, 2-dimensional laplacian. Factor: $12h^2$

	-2	-1	0	1	2
-2			-1		
-1			16		
0	-1	16	**-60**	16	-1
1			16		
2			-1		

`Laplacian3D4(A)`
: 4th order accurate, 3-dimensional laplacian. Factor: $12h^2$

Note that the above are available in normalized versions `Laplacian2D4n`, `Laplacian3D4n` which have factors h^2.

4.4.5 Gradient (∇) operators

These return `TinyVectors` of the appropriate numeric type and length:

`grad2D(A)`
: 2nd order, 2-dimensional gradient (vector of first derivatives), generated using the central12 operator. Factor: $2h$

`grad2D4(A)`
: 4th order, 2-dimensional gradient, using central14 operator. Factor: $12h$

`grad3D(A)`
: 2nd order, 3-dimensional gradient, using central12 operator. Factor: $2h$

`grad3D4(A)`
: 4th order, 3-dimensional gradient, using central14 operator. Factor: $12h$

These are available in normalized versions `grad2Dn`, `grad2D4n`, `grad3Dn` and `grad3D4n` which have factors h.

4.4.6 Jacobian operators

The Jacobian operators are defined over 3D vector fields only (e.g. `Array<TinyVector<double,3>,3>`). They return a `TinyMatrix<T,3,3>` where T is the numeric type of the vector field.

`Jacobian3D(A)`
: 2nd order, 3-dimensional Jacobian using the central12 operator. Factor: $2h$.

`Jacobian3D4(A)`
: 4th order, 3-dimensional Jacobian using the central14 operator. Factor: $12h$.

These are also available in normalized versions `Jacobian3Dn` and `Jacobain3D4n` which have factors h.

4.4.7 Grad-squared operators

There are also grad-squared operators, which return `TinyVectors` of second derivatives:

`gradSqr2D(A)`
: 2nd order, 2-dimensional grad-squared (vector of second derivatives), generated using the central22 operator. Factor: h^2

`gradSqr2D4(A)`
: 4th order, 2-dimensional grad-squared, using central24 operator. Factor: $12h^2$

`gradSqr3D(A)`
: 2nd order, 3-dimensional grad-squared, using the central22 operator. Factor: h^2

`gradSqr3D4(A)`
: 4th order, 3-dimensional grad-squared, using central24 operator. Factor: $12h^2$

Note that the above are available in normalized versions `gradSqr2Dn`, `gradSqr2D4n`, `gradSqr3Dn`, `gradSqr3D4n` which have factors h^2.

4.4.8 Curl ($\nabla\times$) operators

These curl operators return scalar values:

`curl(Vx,Vy)`
: 2nd order curl operator using the central12 operator. Factor: $2h$

`curl4(Vx,Vy)`
: 4th order curl operator using the central14 operator. Factor: $12h$

`curl2D(V)`
: 2nd order curl operator on a 2D vector field (e.g. `Array<TinyVector<float,2>,2>`), using the central12 operator. Factor: $2h$

`curl2D4(V)`
: 4th order curl operator on a 2D vector field, using the central12 operator. Factor: $12h$

Available in normalized forms `curln`, `curl4n`, `curl2Dn`, `curl2D4n`.

These curl operators return three-dimensional `TinyVector`s of the appropriate numeric type:

`curl(Vx,Vy,Vz)`
: 2nd order curl operator using the central12 operator. Factor: $2h$

`curl4(Vx,Vy,Vz)`
: 4th order curl operator using the central14 operator. Factor: $12h$

`curl(V)`
: 2nd order curl operator on a 3D vector field (e.g. `Array<TinyVector<double,3>,3>`, using the central12 operator. Factor: $2h$

`curl4(V)`
: 4th order curl operator on a 3D vector field, using the central14 operator. Factor: $12h$

Note that the above are available in normalized versions `curln` and `curl4n`, which have factors of `h`.

4.4.9 Divergence ($\nabla\cdot$) operators

The divergence operators return a scalar value.

`div(Vx,Vy)`
: 2nd order div operator using the central12 operator. Factor: $2h$

`div4(Vx,Vy)`
: 4th order div operator using the central14 operator. Factor: $12h$

`div(Vx,Vy,Vz)`
: 2nd order div operator using the central12 operator. Factor: $2h$

`div4(Vx,Vy,Vz)`
: 4th order div operator using the central14 operator. Factor: $12h$

`div2D(V)` 2nd order div operator on a 2D vector field, using the central12 operator. Factor: $2h$

`div2D4(V)`
2nd order div operator on a 2D vector field, using the central14 operator. Factor: $12h$

`div3D(V)` 2nd order div operator on a 3D vector field, using the central12 operator. Factor: $2h$

`div3D4(V)`
2nd order div operator on a 3D vector field using the central14 operator. Factor: $12h$

These are available in normalized versions `divn`, `div4n`, `div2Dn`, `div2D4n`, `div3Dn`, and `div3D4n` which have factors of h.

4.4.10 Mixed partial derivatives

`mixed22(A,dim1,dim2)`
2nd order accurate, 2nd mixed partial derivative. Factor: $4h^2$

`mixed24(A,dim1,dim2)`
4th order accurate, 2nd mixed partial derivative. Factor: $144h^2$

There are also normalized versions of the above, `mixed22n` and `mixed24n` which have factors h^2.

4.5 Declaring your own stencil operators

You can declare your own stencil operators using the macro `BZ_DECLARE_STENCIL_OPERATOR1`. For example, here is the declaration of `Laplacian2D`:

```
BZ_DECLARE_STENCIL_OPERATOR1(Laplacian2D, A)
    return -4*A(0,0) + A(-1,0) + A(1,0) + A(0,-1) + A(0,1);
BZ_END_STENCIL_OPERATOR
```

To declare a stencil operator on 3 operands, use the macro `BZ_DECLARE_STENCIL_OPERATOR3`. Here is the declaration of `div`:

```
BZ_DECLARE_STENCIL_OPERATOR3(div,vx,vy,vz)
  return central12(vx,firstDim) + central12(vy,secondDim)
    + central12(vz,thirdDim);
BZ_END_STENCIL_OPERATOR
```

The macros aren't magical; they just declare an inline template function with the names and arguments you specify. For example, the declaration of `div` could also be written

```
template<class T>
inline typename T::T_numtype div(T& vx, T& vy, T& vz)
{
  return central12(vx,firstDim) + central12(vy,secondDim)
                                + central12(vz,thirdDim);
}
```

The template parameter `T` is an iterator type for arrays.

You are encouraged to use the macros when possible, because it is possible the implementation could be changed in the future.

To declare a difference operator, use this syntax:

```
BZ_DECLARE_DIFF(central12,A) {
  return A.shift(1,dim) - A.shift(-1,dim);
}
```

The method `shift(offset,dim)` retrieves the element at `offset` in dimension `dim`.

Stencil operator declarations cannot occur inside a function. If declared inside a class, they are scoped by the class.

4.6 Applying a stencil

The syntax for applying a stencil is:

```
applyStencil(stencilname(),A,B,C...,F);
```

Where `stencilname` is the name of the stencil, and `A,B,C,...,F` are the arrays on which the stencil operates.

For examples, see '`examples/stencil.cpp`' and '`examples/stencil2.cpp`'.

Blitz++ interrogates the stencil object to find out how large its footprint is. It only applies the stencil over the region of the arrays where it won't overrun the boundaries.

5 Multicomponent, complex, and user type arrays

5.1 Multicomponent and complex arrays

Multicomponent arrays have elements which are vectors. Examples of such arrays are vector fields, colour images (which contain, say, RGB tuples), and multispectral images. Complex-valued arrays can also be regarded as multicomponent arrays, since each element is a 2-tuple of real values.

Here are some examples of multicomponent arrays:

```
// A 3-dimensional array; each element is a length 3 vector of float
Array<TinyVector<float,3>,3> A;

// A complex 2-dimensional array
Array<complex<double>,2> B;

// A 2-dimensional image containing RGB tuples
struct RGB24 {
  unsigned char r, g, b;
};

Array<RGB24,2> C;
```

5.1.1 Extracting components

Blitz++ provides some special support for such arrays. The most important is the ability to extract a single component. For example:

```
Array<TinyVector<float,3>,2> A(128,128);
Array<float,2> B = A.extractComponent(float(), 1, 3);
B = 0;
```

The call to `extractComponent` returns an array of floats; this array is a view of the second component of each element of A. The arguments of `extractComponent` are: (1) the type of the component (in this example, float); (2) the component number to extract (numbered 0, 1, ... N-1); and (3) the number of components in the array.

This is a little bit messy, so Blitz++ provides a handy shortcut using `operator[]`:

```
Array<TinyVector<float,3>,2> A(128,128);
A[1] = 0;
```

The number inside the square brackets is the component number. However, for this operation to work, Blitz++ has to already know how many components there are, and what type they are. It knows this already for `TinyVector` and `complex<T>`. If you use your own type, though, you will have to tell Blitz++ this information using the macro `BZ_DECLARE_MULTICOMPONENT_TYPE()`. This macro has three arguments:

```
BZ_DECLARE_MULTICOMPONENT_TYPE(T_element, T_componentType, numComponents)
```

`T_element` is the element type of the array. `T_componentType` is the type of the components of that element. `numComponents` is the number of components in each element.

An example will clarify this. Suppose we wanted to make a colour image, stored in 24-bit HSV (hue-saturation-value) format. We can make a class `HSV24` which represents a single pixel:

```
#include <blitz/array.h>

using namespace blitz;
```

```
class HSV24 {
public:
    // These constants will makes the code below cleaner; we can
    // refer to the components by name, rather than number.

    static const int hue=0, saturation=1, value=2;

    HSV24() { }
    HSV24(int hue, int saturation, int value)
      : h_(hue), s_(saturation), v_(value)
    { }

    // Some other stuff here, obviously

private:
    unsigned char h_, s_, v_;
};
```

Right after the class declaration, we will invoke the macro `BZ_DECLARE_MULTICOMPONENT_TYPE` to tell Blitz++ about HSV24:

```
// HSV24 has 3 components of type unsigned char
BZ_DECLARE_MULTICOMPONENT_TYPE(HSV24, unsigned char, 3);
```

Now we can create HSV images and modify the individual components:

```
int main()
{
    Array<HSV24,2> A(128,128);   // A 128x128 HSV image
    ...

    // Extract a greyscale version of the image
    Array<unsigned char,2> A_greyscale = A[HSV24::value];

    // Bump up the saturation component to get a
    // pastel effect
    A[HSV24::saturation] *= 1.3;

    // Brighten up the middle of the image
    Range middle(32,96);
    A[HSV24::value](middle,middle) *= 1.2;
}
```

5.1.2 Special support for complex arrays

Since complex arrays are used frequently, Blitz++ provides two special methods for getting the real and imaginary components:

```
Array<complex<float>,2> A(32,32);

real(A) = 1.0;
imag(A) = 0.0;
```

The function `real(A)` returns an array view of the real component; `imag(A)` returns a view of the imaginary component.

Note: Blitz++ provides numerous math functions defined over complex-valued arrays, such as `conj`, `polar`, `arg`, `abs`, `cos`, `pow`, etc. See the section on math functions (Section 3.8 [Math functions 1], page 38) for details.

5.1.3 Zipping together expressions

Blitz++ provides a function `zip()` which lets you combine two or more expressions into a single component. For example, you can combine two real expressions into a complex expression, or three integer expressions into an HSV24 expression. The function has this syntax:

```
resultexpr zip(expr1, expr2, T_element)
resultexpr zip(expr1, expr2, expr3, T_element)         ** not available yet
resultexpr zip(expr1, expr2, expr3, expr4, T_element)  ** not available yet
```

The types `resultexpr`, `expr1` and `expr2` are array expressions. The third argument is the type you want to create. For example:

```
int N = 16;
Array<complex<float>,1> A(N);
Array<float,1> theta(N);

 ...

A = zip(cos(theta), sin(theta), complex<float>());
```

The above line is equivalent to:

```
for (int i=0; i < N; ++i)
   A[i] = complex<float>(cos(theta[i]), sin(theta[i]));
```

5.2 Creating arrays of a user type

You can use the `Array` class with types you have created yourself, or types from another library. If you want to do arithmetic on the array, whatever operators you use on the arrays have to be defined on the underlying type.

For example, here's a simple class for doing fixed point computations in the interval [0,1]:

```
#include <blitz/array.h>
#include <blitz/numinquire.h> // for huge()

using namespace blitz;

// A simple fixed point arithmetic class which represents a point
// in the interval [0,1].
class FixedPoint {

public:
  // The type to use for the mantissa
    typedef unsigned int T_mantissa;

    FixedPoint() { }

    FixedPoint(T_mantissa mantissa)
    {
        mantissa_ = mantissa;
    }

    FixedPoint(double value)
    {
        assert((value >= 0.0) && (value <= 1.0));
        mantissa_ = static_cast<T_mantissa>(value * huge(T_mantissa()));
    }
```

```
    FixedPoint operator+(FixedPoint x)
    { return FixedPoint(mantissa_ + x.mantissa_); }

    double value() const
    { return mantissa_ / double(huge(T_mantissa())); }

private:
    T_mantissa mantissa_;
};

ostream& operator<<(ostream& os, const FixedPoint& a)
{
    os << a.value();
    return os;
}
```

The function `huge(T)` returns the largest representable value for type T; in the example above, it's equal to `UINT_MAX`.

The `FixedPoint` class declares three useful operations: conversion from `double`, addition, and outputing to an `ostream`. We can use all of these operations on an `Array<FixedPoint>` object:

```
#include <fixed-point.h> // FixedPoint class

int main()
{
    // Create an array using the FixedPoint class:

    Array<FixedPoint, 2> A(4,4), B(4,4);

    A = 0.5, 0.3, 0.8, 0.2,
        0.1, 0.3, 0.2, 0.9,
        0.0, 1.0, 0.7, 0.4,
        0.2, 0.3, 0.8, 0.4;

    B = A + 0.05;

    cout << "B = " << B << endl;

    return 0;
}
```

Note that the array `A` is initialized using a comma-delimited list of `double`; this makes use of the constructor `FixedPoint(double)`. The assignment `B = A + 0.05` uses `FixedPoint::operator+(FixedPoint)`, with an implicit conversion from `double` to `FixedPoint`. Formatting the array `B` onto the standard output stream is done using the output operator defined for `FixedPoint`.

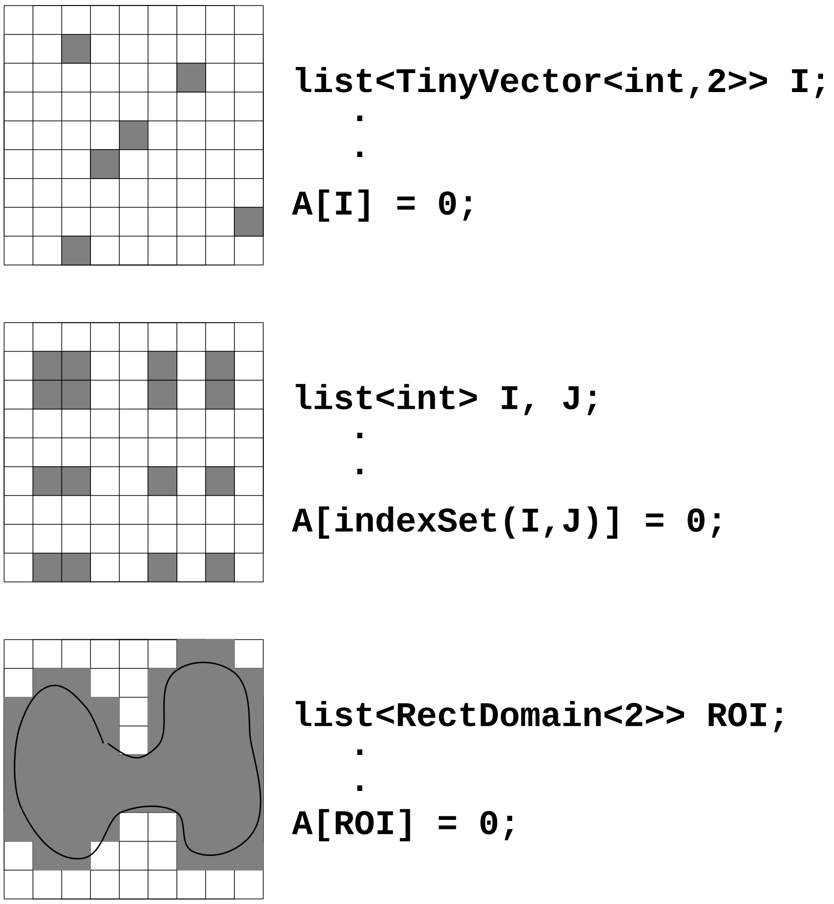
Here's the program output:

```
B = 4 x 4
[      0.55      0.35      0.85      0.25
       0.15      0.35      0.25      0.95
       0.05      0.05      0.75      0.45
       0.25      0.35      0.85      0.45 ]
```

6 Indirection

Indirection is the ability to modify or access an array at a set of selected index values. Blitz++ provides several forms of indirection:

- **Using a list of array positions**: this approach is useful if you need to modify an array at a set of scattered points.
- **Cartesian-product indirection**: as an example, for a two-dimensional array you might have a list `I` of rows and a list `J` of columns, and you want to modify the array at all (i,j) positions where i is in `I` and j is in `J`. This is a **cartesian product** of the index sets `I` and `J`.
- **Over a set of strips**: for efficiency, you can represent an arbitrarily-shaped subset of an array as a list of one-dimensional strips. This is a useful way of handling **Regions Of Interest** (ROIs).



Three styles of indirection.[1]

In all cases, Blitz++ expects a Standard Template Library container. Some useful STL containers are `list<>`, `vector<>`, `deque<>` and `set<>`. Documentation of these classes is often provided with your compiler, or see also the good documentation at `http://www.sgi.com/Technology/STL/`. STL containers are used because they are widely available and provide easier manipulation of "sets" than Blitz++ arrays. For example, you can easily expand and merge sets which are stored in STL containers; doing this is not so easy with Blitz++ arrays, which are designed for numerical work.

STL containers are generally included by writing

[1] From top to bottom: (1) using a list of array positions; (2) Cartesian-product indirection; (3) using a set of strips to represent an arbitrarily-shaped subset of an array.

```
#include <list>   // for list<>
#include <vector> // for vector<>
#include <deque>  // for deque<>
#include <set>    // for set<>
```

The [] operator is overloaded on arrays so that the syntax `array[container]` provides an indirect view of the array. So far, this indirect view may only be used as an lvalue (i.e. on the left-hand side of an assignment statement).

The examples in the next sections are available in the Blitz++ distribution in '`<examples/indirect.cpp>`'.

6.1 Indirection using lists of array positions

The simplest kind of indirection uses a list of points. For one-dimensional arrays, you can just use an STL container of integers. Example:

```
Array<int,1> A(5), B(5);
A = 0;
B = 1, 2, 3, 4, 5;

vector<int> I;
I.push_back(2);
I.push_back(4);
I.push_back(1);

A[I] = B;
```

After this code, the array A contains `[ 0 2 3 0 5 ]`.

Note that arrays on the right-hand-side of the assignment must have the same shape as the array on the left-hand-side (before indirection). In the statement `A[I] = B`, A and B must have the same shape, not I and B.

For multidimensional arrays, you can use an STL container of `TinyVector<int,N_rank>` objects. Example:

```
Array<int,2> A(4,4), B(4,4);
A = 0;
B = 10*tensor::i + tensor::j;

typedef TinyVector<int,2> coord;

list<coord> I;
I.push_back(coord(1,1));
I.push_back(coord(2,2));

A[I] = B;
```

After this code, the array A contains:

```
0   0   0   0
0  11   0   0
0   0  22   0
0   0   0   0
```

(The `tensor::i` notation is explained in the section on index placeholders Section 3.6 [Index placeholders], page 35).

6.2 Cartesian-product indirection

The Cartesian product of the sets I, J and K is the set of (i,j,k) tuples for which i is in I, j is in J, and k is in K.

Blitz++ implements cartesian-product indirection using an **adaptor** which takes a set of STL containers and iterates through their Cartesian product. Note that the cartesian product is never explicitly created. You create the Cartesian-product adaptor by calling the function:

```
template<class T_container>
indexSet(T_container& c1, T_container& c2, ...)
```

The returned adaptor can then be used in the [] operator of an array object.

Here is a two-dimensional example:

```
Array<int,2> A(6,6), B(6,6);
A = 0;
B = 10*tensor::i + tensor::j;

vector<int> I, J;
I.push_back(1);
I.push_back(2);
I.push_back(4);

J.push_back(0);
J.push_back(2);
J.push_back(5);

A[indexSet(I,J)] = B;
```

After this code, the A array contains:

```
 0   0   0   0   0   0
10   0  12   0   0  15
20   0  22   0   0  25
 0   0   0   0   0   0
40   0  42   0   0  45
 0   0   0   0   0   0
```

All the containers used in a cartesian product must be the same type (e.g. all `vector<int>` or all `set<TinyVector<int,2> >`), but they may be different sizes. Singleton containers (containers containing a single value) are fine.

6.3 Indirection with lists of strips

You can also do indirection with a container of one-dimensional **strips**. This is useful when you want to manipulate some arbitrarily-shaped, well-connected subdomain of an array. By representing the subdomain as a list of strips, you allow Blitz++ to operate on vectors, rather than scattered points; this is much more efficient.

Strips are represented by objects of type `RectDomain<N>`, where `N` is the dimensionality of the array. The `RectDomain<N>` class can be used to represent any rectangular subdomain, but for indirection it is only used to represent strips.

You create a strip by using this function:

```
RectDomain<N> strip(TinyVector<int,N> start,
                    int stripDimension, int ubound);
```

The `start` parameter is where the strip starts; `stripDimension` is the dimension in which the strip runs; `ubound` is the last index value for the strip. For example, to create a 2-dimensional strip from (2,5) to (2,9), one would write:

```
TinyVector<int,2> start(2,5);
RectDomain<2> myStrip = strip(start,secondDim,9);
```

Here is a more substantial example which creates a list of strips representing a circle subset of an array:

```
const int N = 7;
Array<int,2> A(N,N), B(N,N);
typedef TinyVector<int,2> coord;

A = 0;
B = 1;

double centre_i = (N-1)/2.0;
double centre_j = (N-1)/2.0;
double radius = 0.8 * N/2.0;

// circle will contain a list of strips which represent a circular
// subdomain.

list<RectDomain<2> > circle;
for (int i=0; i < N; ++i)
{
  double jdist2 = pow2(radius) - pow2(i-centre_i);
  if (jdist2 < 0.0)
    continue;

  int jdist = int(sqrt(jdist2));
  coord startPos(i, int(centre_j - jdist));
  circle.push_back(strip(startPos, secondDim, int(centre_j + jdist)));
}

// Set only those points in the circle subdomain to 1
A[circle] = B;
```

After this code, the A array contains:

```
0  0  0  0  0  0  0
0  0  1  1  1  0  0
0  1  1  1  1  1  0
0  1  1  1  1  1  0
0  1  1  1  1  1  0
0  0  1  1  1  0  0
0  0  0  0  0  0  0
```

7 TinyVector

The `TinyVector` class provides a small, lightweight vector object whose size is known at compile time. It is included via the header `<blitz/tinyvec.h>`.

Note that `TinyVector` lives in the `blitz` namespace, so you will need to refer to it as `blitz::TinyVector`, or use the directive `using namespace blitz;`.

The Blitz++ `Array` object uses `TinyVector` internally, so if you include `<blitz/array.h>`, the `TinyVector` header is automatically included. However, to use `TinyVector` expressions, you will need to include `<blitz/tinyvec-et.h>`.

7.1 Template parameters and types

The `TinyVector<T,N>` class has two template parameters:

`T` is the numeric type of the vector (float, double, int, `complex<float>`, etc.;

`N` is the number of elements in the vector.

Inside the `TinyVector` class, these types are declared:

`T_numtype`
is the numeric type stored in the vector (the template parameter `T`)

`T_vector` is the vector type `TinyVector<T,N>`.

`iterator` is an STL-style iterator.

`constIterator`
is an STL-style const iterator.

7.2 Constructors

```
TinyVector();
```

The elements of the vector are left uninitialized.

```
TinyVector(const TinyVector<T,N>& x);
```

The elements of vector `x` are copied.

```
TinyVector(T value);
```

All elements are initialized to `value`.

```
TinyVector(T value1, T value2, ...);
```

The vector is initialized with the list of values given. These constructors are provided for up to N=11.

7.3 Member functions

```
TinyVector<T,N>::iterator                 begin();
TinyVector<T,N>::const_iterator           begin() const;
```

Returns an STL-style iterator for the vector, positioned at the beginning of the data.

```
TinyVector<T,N>::iterator                 end();
TinyVector<T,N>::const_iterator           end() const;
```

Returns an STL-style iterator for the vector, positioned at the end of the data.

```
T_numtype* [restrict]                     data();
const T_numtype* [restrict]               data() const;
```

Returns a pointer to the first element in the vector.

```
int                                         length() const;
```

Returns the length of the vector (the template parameter `N`).

```
T_numtype                                   operator()(int i) const;
T_numtype&                                  operator()(int i);
T_numtype                                   operator[](int i) const;
T_numtype&                                  operator[](int i);
```

Returns the `i`th element of the vector. If the code is compiled with debugging enabled (`-DBZ_DEBUG`), bounds checking is performed.

7.4 Assignment operators

The assignment operators =, +=, -=, *=, /=, %=, ^=, &=, |=, >>= and <<= are all provided. The right hand side of an assignment may be a scalar of type `T_numtype`, a `TinyVector` of any type but the same size, or a vector expression.

7.5 Expressions

Expressions involving tiny vectors may contain any combination of the operators

```
+ - * / % ^ & | >> <<
```

with operands of type TinyVector, scalar, or vector expressions. The usual math functions (see the Array documentation) are supported on TinyVector.

7.6 Global functions

```
dot(TinyVector, TinyVector);
dot(vector-expr, TinyVector);
dot(TinyVector, vector-expr);
dot(vector-expr, vector-expr);
```

These functions calculate a dot product between `TinyVectors` (or vector expressions). The result is a scalar; the type of the scalar follows the usual type promotion rules.

```
product(TinyVector);
```

Returns the product of all the elements in the vector.

```
sum(TinyVector);
```

Returns the sum of the elements in the vector.

```
TinyVector<T,3> cross(TinyVector<T,3> x, TinyVector<T,3> y);
```

Returns the cross product of `x` and `y`.

7.7 Arrays of `TinyVector`

7.8 Input/output

```
ostream& operator<<(ostream&, const TinyVector<T,N>& x);
```

This function outputs a `TinyVector` onto a stream. Here's an illustration of the format for a length 3 vector:

```
[         0.5        0.2        0.9 ]
```

8 Parallel Computing with Blitz++

While Blitz++ can be used for parallel computing, it was not designed primarily for this purpose. For this reason, you may want to investigate some other available libraries, such as POOMA, before choosing to implement a parallel code using Blitz++.

8.1 Blitz++ and thread safety

To enable thread-safety in Blitz++, you need to do one of these things:

- Compile with `gcc -pthread`, or `CC -mt` under Solaris. (These options define `_REENTRANT`, which tells Blitz++ to generate thread-safe code).
- Compile with `-DBZ_THREADSAFE`, or `#define BZ_THREADSAFE` before including any Blitz++ headers.

In threadsafe mode, Blitz++ array reference counts are safeguarded by a mutex. By default, pthread mutexes are used. If you would prefer a different mutex implementation, add the appropriate `BZ_MUTEX` macros to `<blitz/blitz.h>` and send them to `blitz-dev@oonumerics.org` for incorporation.

Blitz++ does not do locking for every array element access; this would result in terrible performance. It is the job of the library user to ensure that appropriate synchronization is used.

9 Random Number Generators

9.1 Overview

These are the basic random number generators (RNGs):

`Uniform` Uniform reals on [0,1)

`Normal` Normal with specified mean and variance

`Exponential`
Exponential with specified mean

`DiscreteUniform`
Integers uniformly distributed over a specified range.

`Beta` Beta distribution

`Gamma` Gamma distribution

`F` F distribution

To use these generators, you need to include some subset of these headers:

```
#include <random/uniform.h>
#include <random/normal.h>
#include <random/exponential.h>
#include <random/discrete-uniform.h>
#include <random/beta.h>
#include <random/gamma.h>
#include <random/chisquare.h>
#include <random/F.h>

using namespace ranlib;
```

All the generators are inside the namespace **ranlib**, so a **using namespace ranlib** directive is required (alternately, you can write e.g. `ranlib::Uniform<>`).

These generators are all class templates. The first template parameter is the number type you want to generate: float, double or long double for continuous distributions, and integer for discrete distributions. This parameter defaults to `float` for continuous distributions, and `unsigned int` for discrete distributions.

The constructors are:

```
Uniform();
Normal(T mean, T standardDeviation);
Exponential(T mean);
DiscreteUniform(T n);   // range is 0 .. n-1
Beta(T a, T b);
Gamma(T mean);
ChiSquare(T df);
F(T dfn, T dfd);
```

where `T` is the first template parameter (`float`, `double`, or `long double`). To obtain a random number, use the method `random()`. Here is an example of constructing and using a `Normal` generator:

```
#include <random/normal.h>

using namespace ranlib;
```

```
void foo()
{
    Normal<double> normalGen;
    double x = normalGen.random();    // x is a normal random number
}
```

9.2 Note: Parallel random number generators

The generators which Blitz++ provides are not suitable for parallel programs. If you need parallel RNGs, you may find `http://www.ncsa.uiuc.edu/Apps/SPRNG` useful.

9.3 Seeding a random number generator

You may seed a random number generator using the member function `seed(unsigned int)`. By default, all random number generators share the same underlying integer random number generator. So seeding one generator will seed them all. (Note: you can create generators with their own internal state; see the sections below). You should generally only seed a random number generator once, at the beginning of a program run.

Here is an example of seeding with the system clock:

```
#include <random/uniform.h>
#include <time.h>

using namespace ranlib;

int main()
{
    // At start of program, seed with the system time so we get
    // a different stream of random numbers each run.
    Uniform<float> x;
    x.seed((unsigned int)time(0));

    // Rest of program
    ...
}
```

Note: you may be tempted to seed the random number generator from a static initializer. **Don't do it!** Due to an oddity of C++, there is no guarantee on the order of static initialization when templates are involved. Hence, you may seed the RNG before its constructor is invoked, in which case your program will crash. If you don't know what a static initializer is, don't worry – you're safe!

9.4 Detailed description of RNGs

There are really two types of RNGs:

`Integer` RNGs provide uniformly distributed, unsigned 32 bit integers.

`RNGs` use Integer RNGs to provide other kinds of random numbers.

By default, the Integer RNG used is a faithful adaptation of the Mersenne Twister `MT19937` Nishimura (see *ACM Transactions on Modeling and Computer Simulation, Vol. 8, No. 1, January 1998, pp 3-30*, `http://www.math.keio.ac.jp/~matumoto/emt.html`, `http://www.acm.org/pubs/citations/journals/tomacs/1998-8-1/p3-matsumoto/`). This generator has a period of $2^{19937}-1$, passed several stringent statistical tests (including the `http://stat.fsu.edu/~geo/diehard.html` tests), and has speed comparable to other modern generators.

9.5 Template parameters

RNGs take three template parameters, all of which have default values. Using the `Uniform` RNG as an example, the template parameters of `Uniform<T, IRNG, stateTag>` are:

`T` is the type of random number to generate (one of `float`, `double`, or `long double` for continuous distributions; an integer type for discrete distributions). Note that generating double and long double RNGs takes longer, because filling the entire mantissa with random bits requires several random integers. The default parameter for most generators is `float`.

`IRNG` is the underlying Integer RNG to use. The default is MersenneTwister.

`stateTag` is either `sharedState` or `independentState`. If `sharedState`, the IRNG is shared with other generators. If `independentState`, the RNG contains its own IRNG. The default is sharedState.

9.6 Member functions

RNGs have these methods:

```
T random();
```

Returns a random number.

```
void seed(unsigned int);
```

Seeds the underlying IRNG. See above for an example of seeding with the system timer.

9.7 Detailed listing of RNGs

To save space in the below list, template parameters have been omitted and only constructors are listed. The notation [a,b] means an interval which includes the endpoints a and b; (a,b) is an interval which does not include the endpoints.

9.7.1 'random/uniform.h'

```
Uniform<>()
```

Continuous uniform distribution on [0,1).

```
UniformClosedOpen<>()
```

Continuous uniform distribution on [0,1). Same as `Uniform<>`.

```
UniformClosed<>()
```

Continuous uniform distribution on [0,1].

```
UniformOpen<>()
```

Continuous uniform distribution on (0,1).

```
UniformOpenClosed<>()
```

Continuous uniform distribution on (0,1].

9.7.2 'random/normal.h'

```
NormalUnit<>()
```

Continuous normal distribution with mean 0 and variance 1.

```
Normal<>(T mean, T standardDeviation)
```

Continuous normal distribution with specified mean and standard deviation.

9.7.3 'random/exponential.h'

`ExponentialUnit<>()`

Continuous exponential distribution with mean 1.

`Exponential<>(T mean)`

Continuous exponential distribution with specified mean.

9.7.4 'random/beta.h'

`Beta<>(T a, T b)`

Beta distribution with parameters a and b. The mean of the distribution is $a/(a+b)$ and its variance is $ab/((a+b)^2(a+b+1))$. Use the method `setParameters(T a, T b)` to change the parameters.

9.7.5 'random/chisquare.h'

`ChiSquare<>(T df)`

χ^2 distribution with `df` degrees of freedom. The parameter df must be positive. Use the method `setDF(T df)` to change the degrees of freedom.

9.7.6 'random/gamma.h'

`Gamma<>(T mean)`

Gamma distribution with specified mean. The mean must be positive. Use the method `setMean(T mean)` to change the mean.

9.7.7 'random/F.h'

`F<>(T numeratorDF, T denominatorDF)`

F distribution with numerator and denominator degrees of freedom specified. Both these parameters must be positive. Use `setDF(T dfn, T dfd)` to change the degrees of freedom.

9.7.8 'random/discrete-uniform.h'

`DiscreteUniform<>(T n)`

Discrete uniform distribution over $0, 1, \ldots, n-1$.

10 Numeric properties

10.1 Introduction

Blitz++ provides a set of functions to access numeric properties of intrinsic types. They are provided as an alternative to the somewhat klunky `numeric_limits<T>::yadda_yadda` syntax provided by the ISO/ANSI C++ standard. Where a similar Fortran 90 function exists, the same name has been used.

The argument in all cases is a dummy of the appropriate type.

All functions described in this section assume that `numeric_limits<T>` has been specialized for the appropriate case. If not, the results are not useful. The standard requires that `numeric_limits<T>` be specialized for all the intrinsic numeric types (float, double, int, bool, unsigned int, etc.).

To use these functions, you must first include the header `<blitz/numinquire.h>`. Also, note that these functions may be unavailable if your compiler is non-ANSI compliant. If the preprocessor symbol `BZ_HAVE_NUMERIC_LIMITS` is false, then these functions are unavailable.

10.2 Function descriptions

`T denorm_min(T) throw;`
: Minimum positive denormalized value. Available for floating-point types only.

`int digits(T);`
: The number of radix digits (read: bits) in the mantissa. Also works for integer types. The official definition is "number of radix digits that can be represented without change".

`int digits10(T);`
: The number of base-10 digits that can be represented without change.

`T epsilon(T);`
: The smallest amount which can be added to 1 to produce a result which is not 1. Floating-point types only.

`bool has_denorm(T);`
: True if the representation allows denormalized values (floating-point only).

`bool has_denorm_loss(T);`
: True if a loss of precision is detected as a denormalization loss, rather than as an inexact result (floating-point only).

`bool has_infinity(T);`
: True if there is a special representation for the value "infinity". If true, the representation can be obtained by calling `infinity(T)`.

`bool has_quiet_NaN(T);`
: True if there is a special representation for a quiet (non-signalling) Not A Number (`NaN`). If so, use the function `quiet_NaN(T)` to obtain it.

`bool has_signaling_NaN(T);`
: True if there is a special representation for a signalling Not A Number (`NaN`). If so, use the function `signalling_NaN(T)` to obtain it.

`bool has_signalling_NaN(T);`
: Same as `has_signaling_NaN()`.

`T huge(T) throw;`
: Returns the maximum finite representable value. Equivalent to `CHAR_MAX`, `SHRT_MAX`, `FLT_MAX`, etc. For floating types with denormalization, the maximum positive **normalized** value is returned.

`T infinity(T) throw;`
: Returns the representation of positive infinity, if available. Note that you should check availability with `has_infinity(T)` before calling this function.

`bool is_bounded(T);`
: True if the set of values represented by the type is finite. All built-in types are bounded. (This function was provided so that e.g. arbitrary precision types could be distinguished).

`bool is_exact(T);`
: True if the representation is exact. All integer types are exact; floating-point types generally aren't. A rational arithmetic type could be exact.

`bool is_iec559(T);`
: True if the type conforms to the IEC 559 standard. IEC is the International Electrotechnical Commission. Note that IEC 559 is the same as IEEE 754. Only relevant for floating types.

`bool is_integer(T);`
: True if the type is integer.

`bool is_modulo(T);`
: True if the type is modulo. Integer types are usually modulo: if you add two integers, they might wrap around and give you a small result. (Some special kinds of integers don't wrap around, but stop at an upper or lower bound; this is called saturating arithmetic). This is false for floating types.

`bool is_signed(T);`
: True if the type is signed (i.e. can handle both positive and negative values).

`int max_exponent(T);`
: The maximum exponent (`Max_exp`) is the maximum positive integer such that the radix (read: 2) raised to the power `Max_exp-1` is a representable, finite floating point number. Floating types only.

`int max_exponent10(T);`
: The maximum base-10 exponent (`Max_exp10`) is the maximum positive integer such that 10 raised to the power `Max_exp10` is a representable, finite floating point number. Floating types only.

`int min_exponent(T);`
: The minimum exponent (`Min_exp`) is the minimum negative integer such that the radix (read: 2) raised to the power `Min_exp-1` is a **normalized** floating point number. Floating types only.

`int min_exponent10(T);`
: The minimum base-10 exponent (`Min_exp10`) is the minimum negative integer such that 10 raised to the power `Min_exp10` is in the range of **normalized** floating point numbers.

`T neghuge(T);`
: This returns the maximally negative value for a type. For integers, this is the same as min(). For floating-point types, it is `-huge(T())`.

`T one(T);` Returns a representation for "1"

`int precision(T);`
: Same as `digits10()`.

`T quiet_NaN(T) throw;`
: Returns the representation for a quiet (non-signalling) Not A Number (`NaN`), if available. You should check availability using the `has_quiet_NaN(T)` function first.

`int radix(T);`
: For floating-point types, this returns the radix (base) of the exponent. For integers, it specifies the base of the representation.

`Range range(T);`
: Returns `Range(min_exponent10(T()), max_exponent10(T()))`, i.e. the range of representable base-10 exponents.

`T round_error(T) throw;`
: Returns a measure of the maximum rounding error for floating-point types. This will typically be `0.5`.

`std::float_round_style round_style(T);`
: Returns the current rounding style for floating-point arithmetic. The possibilities are: `round_indeterminate` (i.e. don't have a clue), `round_toward_zero`, `round_to_nearest` (round to nearest representable value), `round_toward_infinity` (round toward positive infinity), and `round_neg_infinity` (round toward negative infinity).

`T signaling_NaN(T) throw;`
: Returns the representation for a signalling Not A Number (`NaN`), if available. You should check availability by calling `has_signalling_NaN(T)` first.

`T signalling_NaN(T) throw;`
: Same as `signaling_NaN()`.

`T tiny(T);`
: For integer types, this returns the minimum finite value, which may be negative. For floating types, it returns the minimum positive value. For floating types with denormalization, the function returns the minimum positive **normalized** value.

`T tinyness_before(T);`
: True if tinyness is detected before rounding. Other than this description, I don't have a clue what this means; anyone have a copy of IEC 559/IEEE 754 floating around?

`T traps(T);`
: True if trapping is implemented for this type.

`T zero(T);`
: Returns a representation for zero.

11 Frequently Asked Questions

11.1 Questions about installation

- **I downloaded Blitz++, but when I try to gunzip it, I get "invalid compressed data–crc error"**

You forgot to set binary download mode in ftp. Do so with the "binary" command.

- **The compiler complains that there is no Array class, even though I've included `<blitz.h>`.**

You need to have the line:

```
using namespace blitz;
```

after including `<blitz.h>`.

- **I can't use gcc on my elderly PC because it requires 45–150 Mb to compile with Blitz++**

Unfortunately this is true. If this problem is ever fixed, it will be by the gcc developers, so my best suggestion is to post a bug report to the gcc-bugs list.

- **I am using gcc under Solaris, and I get errors about "relocation against external symbol"**

This problem can be fixed by installing the gnu linker and binutils. Peter Nordlund found that by using `gnu-binutils-2.9.1`, this problem disappeared. You can read a detailed discussion at `http://oonumerics.org/blitz/support/blitz-support/archive/0029.html`.

- **I am using gcc under Solaris, and the assembler gives me an error that a symbol is too long.**

This problem can also be fixed by installing the gnu linker and binutils. See the above question.

- **DECcxx reports problems about "templates with C linkage"**

This problem was caused by a problem in some versions of DECcxx's '`math.h`' header: XOPEN_SOURCE_EXTENDED was causing an `extern "C" { ... }` section to have no closing brace. There is a kludge which is included in recent versions of Blitz++.

- **On some platforms (especially SGI) the testsuite program `minsumpow` fails with the error: `Template instantiation resulted in an unexpected function type of...`**

This is a known bug in the older versions of the EDG front end, which many C++ compilers use. There is no known fix. Most of Blitz++ will work, but you won't be able to use some array reductions.

11.2 Questions about Blitz++ functionality

- **For my problem, I need SVD, FFTs, QMRES, PLU, QR, ...**

Blitz++ does not currently provide any of these. However, there are numerous C++ and C packages out there which do, and it is easy to move data back and forth between Blitz++ and other libraries. See these terms in the index: creating an array from pre-existing data, `data()`, `stride()`, `extent()`, `fortranArray`. For a list of other numerical C++ libraries, see the Object Oriented Numerics Page at `http://oonumerics.org/oon/`.

- **Can Blitz++ be interfaced with Python?**

Phil Austin has done so successfully. See a description of his setup in `http://oonumerics.org/blitz/support/blitz-support/archive/0053.html`.

Also see Harry Zuzan's Python/Blitz image processing example code at `http://www.stat.duke.edu/~hz/blitz_py/index.html`.

- **If I try to allocate an array which is too big, my program just crashes or goes into an infinite loop. Is there some way I can handle this more elegantly?**

Blitz++ uses `new` to allocate memory for arrays. In theory, your compiler should be throwing a `bad_alloc` exception when you run out of memory. If it does, you can use a `try/catch` block to handle the out of memory exception. If your compiler does not throw `bad_alloc`, you can install your own new handler to handle out of memory.

Here is an excerpt from the ISO/ANSI C++ standard which describes the behaviour of `new`:

- Executes a loop: Within the loop, the function first attempts to allocate the requested storage. Whether the attempt involves a call to the Standard C library function malloc is unspecified.
- Returns a pointer to the allocated storage if the attempt is successful. Otherwise, if the last argument to `set_new_handler()` was a null pointer, throw `bad_alloc`.
- Otherwise, the function calls the current `new_handler` (lib.new.handler). If the called function returns, the loop repeats.
- The loop terminates when an attempt to allocate the requested storage is successful or when a called new_handler function does not return.

You can use `set_new_handler` to create a new handler which will issue an error message or throw an exception. For example:

```
void my_new_handler()
{
cerr << "Out of memory" << endl;
cerr.flush();
abort();
}

...

// First line in main():
set_new_handler(my_new_handler);
```

• **When I pass arrays by value, the function which receives them can modify the array data. Why?**

It's a result of reference-counting. You have to think of array objects as being "handles" to underlying arrays. The function doesn't receive a copy of the array data, but rather a copy of the handle. The alternative would be to copy the array data when passing by value, which would be grossly inefficient.

• **Why can't I use e.g. `A >> 3` to do bitshifting on arrays?**

The operators `<<` and `>>` are used for input/ouput of arrays. It would cause problems with the expression templates implementation to also use them for bitshifting. However, it is easy enough to define your own bitshifting function – see Section 3.10 [User et], page 42.

• **When I write `TinyMatrix * TinyVector` I get an error.**

Try `product(d2,d1)`. This works for matrix-matrix and matrix-vector products.

Blitz Keyword Index

J

L

M

N

O

P

Q

R

S

T

U

X

Y

Z

Concept Index

I

J

K

L

M

N

O

P

R