

C++ in scientific computing

Part XV - Blitz++

Max Moorkamp

Dublin Institute for Advanced Studies

Dublin, 6 July 2006

Outline

- 1 Introduction to Blitz++
 - Ranges
 - Stencils

What is Blitz++

- High level numerical library
- Focused on tensor/matrix/vector operations
- Aims at providing high performance
- At the same time operations should be easy to write
- Unfortunately documentation does not cover everything
- No safeguards to prevent wrong memory access etc.

An overview of some types provided

- `Array<type,dim>` the general Vector/Array/Tensor class with elements of `type` and dimension `dim`
- `TinyVector<type,size>` A vector with known size `size` at compile time that holds elements of type `type`
- `TinyMatrix<type,xsize,ysize>` A matrix with known size `xsize * ysize`
- Several helper classes that provide operations and views on the data

A first example

A simple vector/matrix program

```
#include <blitz/array.h>
#include <iostream>
using namespace blitz;
int main() {
    TinyMatrix<float,3,3> M;
    M= 2,0,0,0,3,0,0,0,4;
    TinyVector<float,3> v;
    v = 1,1,1;
    TinyVector<float,3> r;
    r = product(M,v);
    std::cout << r << std::endl;
}
```

The same thing slightly differently

Using tensor style notation

```

Array<float,2> M(4,3);
M= 2,0,0,0,3,0,0,0,4,0,0,5;
Array<float,1> v(3);
v = 1,1,1;
Array<float,1> r(4);
r = sum(M(tensor::i, tensor::j)
        *v(tensor::j), tensor::j);
std::cout << r << std::endl;

```

This corresponds to the Einstein tensor notation

$$r_i = M_{ij}v_j$$

But we have to perform the summation explicitly

More on tensor notation

Assigning computed values

```
const int N = 128;  
Array<float, 2> M(N,N);  
float midpoint = (N-1)/2.;  
float omega = 2.0 * M_PI * 3 / double(N);  
M = cos(omega * sqrt(pow2(tensor::i - midpoint)  
                    + pow2(tensor::j - midpoint)))  
    * exp(-10.0/N * sqrt(pow2(tensor::i - midpoint)  
                        + pow2(tensor::j - midpoint)));
```

The resulting array will look like this

$$M_{ij} = \cos(\omega \sqrt{(i-p)^2 + (j-p)^2}) \exp\left(\frac{-10}{N} \sqrt{(i-p)^2 + (j-p)^2}\right)$$

Ranges

Ranges provide simple access to a subset of the Array

Using a range

```
const int N = 12;  
Array<float,2> A(N,N);  
Range R(1,10);  
Array<float,2> B = A(R,R);
```

Note that B is sharing the data with A . It is just an alternative view. If we want to copy the data we have to write

Creating a copy

```
Array<float,2> B = A(R,R).copy(); //or in two lines  
Array<float,2> B(2,2);  
B = A(R,R);
```

Slicing

We can use a combination of indices and ranges to produce a *slice*

Using a range

```
const int N = 12;  
Array<float,3> A(N,N,N);  
Array<float,2> B = A(Range::all(),3,Range::all());
```

What are stencils

- Stencils provide a powerful way for operations on the whole Matrix
- They are marked as experimental, so implementation and syntax can change
- A lot of build-in stencils provided
- Can declare new stencils with the help of provided macros

Using a stencil

Calculating the normalized central difference

$$y_i = \frac{x_{i+1} - x_{i-1}}{2}$$

In 1D

```
const int length = 100;
Array<float,1> A(length),B(length);
A = cos(tensor::i /10.0);
Array<float,1> der = A(Range(1,length-1));
B = central12n(der,firstDim);
```

A more complicated stencil

In 1D

```
const int size = 100;
Array<float,2> A(size, size), B(size, size);
A = cos(tensor::i / 10.0) * exp(-10.0 / size * tensor::j);
Array<float,2> der = A(Range(2, size - 2), Range(2, size - 2));
B = Laplacian2D(der);
```