NumCpp Documentation

Release 0.1alpha0

Tobias Knopp

January 16, 2017

NumCpp is a numerical C++ library, which allows to write numerical algorithms in a similar way as in Numpy or Matlab. It uses features of the new C++11 standard simplifying both the implementation of NumCpp and the external API. Here is a simple example of its usage:

```
#include <numcpp>
using namspace numcpp;

int main()
{
  Vector<double> x = ones(16);
  auto y = fft(x);
  auto z = reshape(y,4,4);

  print(z);
}
```

More complex examples can be found in the Tutorial.

CHAPTER 1

Features

The features of NumCpp are

- Simple API similar to Numpy/Matlab
- Truely multidimensional array object
- Clean implementation
- High performance using expression templates
- Mixing of datatypes in expressions

CHAPTER 2

Content

2.1 Concept

2.1.1 Simple Functional Interface / Clean Implementation

The aim of the NumCpp project is to develop a C++ library having a simple interface similar to the Numpy/Matlab API.

2.1.2 Multidimensional Array

Some of the excellent available C++ numerical libraries (e.g. Eigen) have the limitation that only 1D and 2D arrays (i.e. vectors and matrices) are supported. In this regards they differ significantly from Matlab and Numpy which both provide arrays of any dimension. Example applications where multidimensional array data is usually required is for instance in 3D magnetic resonance imaging (MRI) reconstruction. To reconstruct Cartesian sampled 3D MRI data, one basically has to perform a 3D Fourier transform. With NumCpp, this can be done like:

```
Array<cdouble,3> rawData(Nx,Ny,Nz);

// fill rawData ...

auto imageData = fft(rawData);
```

Unlike in Numpy, the dimension has to be known at compile time in NumCpp. This is essential to ensure high performance and simplifies the implementation of the library in several areas. For instance, one can use the dimension to overload functions and in turn allow for return types depending on the dimension. For instance in NumCpp, the dot function returns a scalar when applying two vectors and returns a vector when applying a matrix and a vector (matrix vector multiplication). If the dimension would not be a template parameter, this would not be possible in Cpp. A further advantage is that one does not have to do runtime checks on the dimension, which is usually necessary in Numpy and Matlab due to the dynamic nature of the underlying programming language.

Note, that NumCpp provides type aliases for 1D and 2D vectors such that a matrix vector multiplication can be written like:

```
Matrix<double> A = randn(N,N);
Vector<double> x = randn(N);

auto y = dot(A,x)
```

2.1.3 Mixing Datatypes

In Numpy/Matlab it is completly natural to mix datatype in expressions. In many C++ libraries this is not possible. For instance, the std::complex datatype does not support the following:

```
double x = 3.0;
std::complex<float>(2.0, 1.0);
auto z = x + y;
```

By including the numcpp header `#include <numcpp/core.h>` the expression gets legal as the library includes the missing mixed-type operators for the std::complex datatype. For convenience, the library has type aliases cdouble and cfloat for std::complex<double> and std::complex<float>. In a similar way to complex scalars, it is possible to mix the datatypes of arrays in expressions:

```
Matrix<cdouble> A = randn(N,N);
Vector<double> x = linspace(0,1,N);
Vector<int> y = ones(N);

auto z = dot(A,x) + y*4.0f;
```

2.1.4 Expression Templates

NumCpp uses so called Expression Templates to ensure that array expressions can be compiled into the maximum efficient code.

2.1.5 High Performance

2.2 Performance

2.3 Tutorial

2.3.1 Introduction

2.3.2 Image and Data Export

NumCpp provides functions for saving and loading arrays to files. Different file formats are supported. Besides a simple binary export, which serializes the binary data into a file, there is an ascii export, which saves a textual representation to file.

For more advanced file handling there is support for the HDF5 file format, which will require to link against the hdf5 library (-lhdf5):

```
// create Shepp Logan phantom (type Matrix<double>)
auto x = phantom(256);

// calculate the absolute value of the FFT of x
// the eval call is necessary as abs is an expression template
auto y = eval(abs(fftshift(fft(x))));

// export x and y to an pdf file using different colormaps
export_pdf( x, "shepp_logan_phantom.pdf", colormaps::autumn);
export_pdf( y, "shepp_logan_phantom_fft.pdf");
```

```
// export x and y into the same hdf5 file
hdf5write( x, "shepp_logan_phantom.h5", "/shepp_logan" );
hdf5write( y, "shepp_logan_phantom.h5", "/shepp_logan_fft" );

// export x into a simple binary file
tofile( x, "shepp_logan_phantom.dat");
```

2.4 NumCpp / Numpy / Matlab Command Table

2.4.1 Array Size Properties

Matlab	Numpy	Numcpp	Description
ndims(x)	ndim(a) or a.ndim	ndims(a) or a.ndims	number of dimensions (rank) of a
size(x)	shape(a) or a.shape	shape(a) or a.shape()	vector of
size(x,d)	shape(a,d) or a.shaped[d]	shape(a,d) or a.shape(d)	number of elements along axis d
length(a(:))	a.size	size(a) or a.size()	total number of elements of a

2.4.2 Matrix Manipulation

Matlab	Numpy	Numcpp	Description
fliplr(a)	fliplr(a)	fliplr(a) or fliplr_(a)	Flip left-right (use fliplr_ for inplace)
flipud(a)	flipud(a)	flipud(a) or flipud_(a)	Flip up-down (use flipud_ for inplace)
flipdim(a,d)	flipdim(a,d)	flipdim(a,d) or flipdim_(a,d)	Flip elements along axis d (use flipdim_ for inplace)
rot90(a)	rot90(a)	rotl90(a)	Rotate 90 degrees

2.5 API Documentation

NumCpp consists of several modules, which can be independently included to ensure that the build time can be kept low. Most important is the core module that provides the basic multidimensional array classes including the abstract base class.

2.5.1 Core Module

AbstractArray

The core module of NumCpp includes the basic datatypes of the library. The central datatype is a multidimensional array whose interface is described by the class AbstractArray. Basically, the **AbstractArray** is defined as follows:

```
template<class T, // datatype
         int D, // dimension
         class Derived>
class AbstractArray
{
public:
  size_t size() // total number of elements

  size_t shape(int d) // shape of the array along axis/direction d

  T& operator[](size_t index) // flat index operator

  MagicReturnType operator()(A...args) // multi index operator
};
```

For performance reasons, the member functions of the **AbstractArray** are not virtual so that do runtime polymorphism is used to implement the base interface. Instead, we use the Curiously Recurring Template Pattern (CRTP), which allows for implementing static polymorphism with no overhead. The basic idea of CRTP is to feed the type of the interfaces' implementation as template parameter *Derived* to the abstract base class. In this way, the base class can call the implementation of a function in the derived class while in modern compilers the function will be inlined.

Template parameters

Besides the Derived class, the **AbstractArray** has two template perameters. The parameter *T* is the datatype of the elements of the array and will usally be of type `double`, `float`, `complex<double>`, `complex<float>`, `int`, `long`, or `bool`. The second template parameter is the dimension / rank of the array. In this way the dimension of the array has to be known at compile time. While this can be an issue when building dynamic programs, it has two important advantages:

- Performance: If D is static, the compiler can for instance unroll the loop in the multidimensional index operator.
- Dimension Dispatch: Provide didicated implementations for dedicated dimensions

One example where it is usfull to dispatch on the dimension is the implementation of the `dot` function. In the one dimensional case the dot function should return a scalar and thus be declared as:

```
template<class T, class Derived> T dot(const AbstractArray<T,1,Derived>& x, const AbstractArray<T,1,I
```

while for a matrix vector multiplication, the declaration would be:

```
template<class T, class Derived> AbstractArray<T,1,Derived> dot(const AbstractArray<T,2,Derived>& x,
```

class **`AbstractArray`**<T, D, Derived>

Abstract D dimensional containing elements of type T. The template parameter Derived has the type of the concrete implementation (using the Curiously Recurring Template Pattern (CRTP)).

class **`Array`**<T, D>

Dense D dimensional array containing elements of type T. Implements the abstract interface AbstractArray.

2.5.2 Base Module

Overview

Trigonometric Functions

sdf	sf
sin(x)	Trigonometric sine, element-wise
cos(x)	Trigonometric cosine, element-wise

Reference

Trigonometric Functions

ExpressionTemplate **sin**(**const** *AbstractArray*<T, D, R> &*x*)

Vecotrized sinus function. Calculates sinus of each element. Note that sin is implemented via an expression template and evaluated lazyly.

ExpressionTemplate **cos**(**const** *AbstractArray*<T, D, R> &*x*)

Vecotrized cosine function. Calculates cosine of each element. Note that cosine is implemented via an expression template and evaluated lazyly.

CHAPTER 3

License

The core functionallity of NumCpp is licensed under the LGPL. However, some functionality requires to link against software that is licensed under the GPL GPL. For instance the fft function internally uses the FFTW library, which is licensed under the GPL. If you plan to use NumCpp in a closed source software, please contact us to obtain a version of NumCpp where all GPL modules are replaced by GPL-free modules.

CHAPTER 4

Contact

For questions and discussions you can use the google group numcpp.

If you are missing functionality in NumCpp you can either contribute to the project, or request features on a consulting basis (please contact us)

CHAPTER 5

Indices and tables

- genindex
- modindex
- search