WaveSimC

0.8

Generated by Doxygen 1.9.6

Chapter 1

README

1.1 COMSW4995 Final Project: WaveSimC

This is the repository for our final project for the discpline COMSW4995: Design in C++ at Columbia University during the Fall of 2022.

This project aims to implement in modern C++ a wave equation solver for geophysical application.

In addition, a custom implementation of numpy in modern C++ is also included as a header library. That library aims to make c++ more pythonic and easier to use for scientific computing. Instead of numpy n-dimensional arrays the library use boost::multi_array and contains many utilities to expand the functionality of the library.

1.1.1 Detailed documentation

1.1.2 Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

1.1.3 Acknowledgments

We would like to thank Professor Bjarne Stroustrup for his guidance and support during the development of this project.

1.2 Theory

1.2.1 Wave simulation

When waves travel in an inhomogeneous medium, they may be delayed, reflected, and refracted, and the wave data encodes information about the medium—this is what makes geophysical imaging possible. The propagation of waves in a medium is described by a partial differential equation known as the wave equation. In two dimension, the wave equation is given by:

```
\begin{align*}
\frac{1}{v^2}\frac{\partial ^2 u}{\partial t^2} - \bigg(\frac{\partial ^2 u}{\partial x^2} + \frac{\partial
       ^2 u}{\partial y^2} \bigg) = f &\qquad \text{in }\mathbb{R}^2 \times (0,T)\\
u|_{t=0} = \frac{\partial u}{\partial t}\bigg|_{t = 0} = 0 & \qquad \text{in }\mathbb{R}^2.
\end{align*}
```

In our simulation, the numerical scheme we use is the finite difference method with the perfectly matched layers [1]:

```
\begin{equation}
    \begin{aligned}
        u^{n+1}
        &= \bigg[ \left(\frac{\Delta t}{2}(\sigma_1+\sigma_2) - 1\right) u^{n-1}  + \left(2 - (\Delta t)^2
       \sigma_1\sigma_2  \right)u^n \\
        &\qquad+ (\Delta t)^2v^2\left(\Delta u^{n} - \nabla\cdot ({\boldsymbol \sigma} \odot {\boldsymbol
       q}^n) + \sigma_2\frac{\partial  q_1^n}{\partial x} +\sigma_1\frac{\partial  q_2^n}{\partial y} +f^n
       \right) \bigg]\bigg/\left(\frac{\Delta t}{2}(\sigma_1+\sigma_2) + 1\right) \\
        q_1^{n+1} & = \left[ \Delta t \frac{\partial }{\partial x}\left( \frac{u^n+u^{n+1}}{2} \right) +
       \left( 1-\frac{\Delta t}{2} \sigma_1\right) q_1^{n}\right] \bigg/ \left( 1+\frac{\Delta t}{2}
       \sigma_1\right) \\
        q_2^{n+1} &= \left[ \Delta t \frac{\partial }{\partial y}\left( \frac{u^n+u^{n+1}}{2} \right) +
       \left( 1-\frac{\Delta t}{2} \sigma_2\right) q_2^{n} \right] \bigg / \left( 1+\frac{\Delta t}{2}
       \sigma_2\right) .
    \end{aligned}
\end{equation}
```

1.2.2 References

[1] Johnson, Steven G. (2021). Notes on perfectly matched layers (PMLs). arXiv preprint arXiv:2108.05348.

1.2.3 Design Philosophy

1.2.3.1 Numpy implementation

We have noticed that many users are very familiar with python and use it extensively with libraries such as numpy and scipy. However their code is often slow and not very low-level friendly. Even with numpy and scipy's low-level optimizations, there could still be margin for improvement by converting everything to C++, which would allow users to unleash even more optimizations and exert more control over how their code runs. This could also allow the code to run on less powerful devices that often don't support python.

With that in mind we decided to find a way to make transferring that numpy, scipy, etc code to C++ in an easy way, while keeping all of the high level luxuries of python. We decided to implement a numpy-like library in C++ that would allow users to write code in a similar way to python, but with the performance of C++.

We started with the implementation of the functions used in the python version of the wave solver and plan to expand the library to include more functions and features in the future.

The library is contained in a header library format for easy of use.

1.2.4 Multi Arrays and how math is done on them

Representing arrays with more than one dimensions is a difficult task in any programming language, specially in a language like C++ that implements strict type checking. To implement that in a flexible and typesafe way, we chose to build our code around the boost::multi_array. This library provides a container that can be used to represent arrays with any number of dimensions. The library is very flexible and allows the user to define the type of the array and the number of dimensions at compile time. The library is sadly not very well documented but the documentation can be found here: https://www.boost.org/doc/libs/1_75_0/libs/multi_↩ array/doc/index.html

We decided to build the math functions in a pythonic way, so we implemented numpy functions into our C++ library in a way that they would accept n-dimensions through a template parameters and act accordingly while enforcing dimensional conistency at compile time. We also used concepts and other modern C++ concepts to make sure that, for example, a python call such as np.max(my_n_dimensional_array) would be translated to np::max(my_n_↩ dimensional_array) in C++.

To perform operations on an n-dimensional array we choose to iterate over it and convert the pointers to indexes using a simple arithmetic operation with one division. This is somewhat time consuming since we don't have O(1) time access to any point in the array, instead having O(n) where n is the amount of elements in the multi array. This is the tradeoff necessary to have n-dimensions represented in memory, hopefully in modern cpus this overhead won't be too high. Better solutions could be investigated further.

We also implemented simple arithmetic operators with multi arrays to make them more arithmetic friendly such as they are in python.

Only one small subset of numpy functions were implemented, but the library is easily extensible and more functions can be added in the future.

1.3 Building

1.3.1 Install the boost library

It is important to install the boost library before building the project. The boost library is used for data structures and algorithms. The boost library can be installed using the following command on ubuntu:

```
sudo apt-get install libboost-all-dev
```

For Mac:

```
brew install boost
```

1.3.2 Build the project

```
mkdir build
cd build
cmake ..
make Main
```

1.3.3 Running

```
./Main
```

1.3.4 Building the documentation

Docs building script:

```
./compileDocs.sh
```

Manually:

```
doxygen dconfig
cd documentation/latex
pdflatex refman.tex
cp refman.pdf ../WaveSimC-0.8-doc.pdf
```

Chapter 2

Module Index

2.1 Modules

Here is a list of all modules:

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Module Documentation

5.1 Np

Namespaces

- namespace np

  Custom implementation of numpy in C++.

Functions

- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator* (boost::multi_array< T, ND > const &lhs, boost::multi_array< T, ND > const &rhs)

  Multiplication operator between two multi arrays, element-wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator* (T const &lhs, boost::multi_array< T, ND > const &rhs)

  Multiplication operator between a multi array and a scalar.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator* (boost::multi_array< T, ND > const &lhs, T const &rhs)

  Multiplication operator between a multi array and a scalar.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator+ (boost::multi_array< T, ND > const &lhs, boost::multi_array< T, ND > const &rhs)

  Addition operator between two multi arrays, element wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator+ (T const &lhs, boost::multi_array< T, ND > const &rhs)

  Addition operator between a multi array and a scalar.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator+ (boost::multi_array< T, ND > const &lhs, T const &rhs)

  Addition operator between a scalar and a multi array.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator- (boost::multi_array< T, ND > const &lhs, boost::multi_array< T, ND > const &rhs)

  Minus operator between two multi arrays, element-wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator- (T const &lhs, boost::multi_array< T, ND > const &rhs)

Minus operator between a scalar and a multi array, element-wise.

- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator- (boost::multi_array< T, ND > const &lhs, T const &rhs)

  Minus operator between a multi array and a scalar, element-wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator/ (boost::multi_array< T, ND > const &lhs, boost::multi_array< T, ND > const &rhs)

  Division between two multi arrays, element wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator/ (T const &lhs, boost::multi_array< T, ND > const &rhs)

  Division between a scalar and a multi array, element wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > operator/ (boost::multi_array< T, ND > const &lhs, T const &rhs)

  Division between a multi array and a scalar, element wise.

5.1.1 Detailed Description

5.1.2 Function Documentation

5.1.2.1 operator∗() [1/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator* (
            boost::multi_array< T, ND > const & lhs,
            boost::multi_array< T, ND > const & rhs )  [inline]
```

Multiplication operator between two multi arrays, element-wise.

Definition at line 460 of file np.hpp.

```
00461 {
00462     std::function<T(T, T)> func = std::multiplies<T>();
00463     return np::element_wise_duo_apply(lhs, rhs, func);
00464 }
```

5.1.2.2 operator∗() [2/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator* (
            boost::multi_array< T, ND > const & lhs,
            T const & rhs )  [inline]
```

Multiplication operator between a multi array and a scalar.

Definition at line 476 of file np.hpp.

```
00477 {
00478     return rhs * lhs;
00479 }
```

5.1.2.3 operator∗() [3/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator* (
            T const & lhs,
            boost::multi_array< T, ND > const & rhs )  [inline]
```

Multiplication operator between a multi array and a scalar.

Definition at line 468 of file np.hpp.

```
00469 {
00470     std::function<T(T)> func = [lhs](T item)
00471     { return lhs * item; };
00472     return np::element_wise_apply(rhs, func);
00473 }
```

5.1.2.4 operator+() [1/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator+ (
            boost::multi_array< T, ND > const & lhs,
            boost::multi_array< T, ND > const & rhs )
```

Addition operator between two multi arrays, element wise.

Definition at line 484 of file np.hpp.

```
00485 {
00486     std::function<T(T, T)> func = std::plus<T>();
00487     return np::element_wise_duo_apply(lhs, rhs, func);
00488 }
```

5.1.2.5 operator+() [2/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator+ (
            boost::multi_array< T, ND > const & lhs,
            T const & rhs )  [inline]
```

Addition operator between a scalar and a multi array.

Definition at line 501 of file np.hpp.

```
00502 {
00503     return rhs + lhs;
00504 }
```

5.1.2.6 operator+() [3/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator+ (
            T const & lhs,
            boost::multi_array< T, ND > const & rhs )  [inline]
```

Addition operator between a multi array and a scalar.

Definition at line 492 of file np.hpp.

```
00493 {
00494     std::function<T(T)> func = [lhs](T item)
00495     { return lhs + item; };
00496     return np::element_wise_apply(rhs, func);
00497 }
```

5.1.2.7 operator-() [1/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator- (
            boost::multi_array< T, ND > const & lhs,
            boost::multi_array< T, ND > const & rhs )
```

Minus operator between two multi arrays, element-wise.

Definition at line 509 of file np.hpp.

```
00510 {
00511     std::function<T(T, T)> func = std::minus<T>();
00512     return np::element_wise_duo_apply(lhs, rhs, func);
00513 }
```

5.1.2.8 operator-() [2/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator- (
            boost::multi_array< T, ND > const & lhs,
            T const & rhs )  [inline]
```

Minus operator between a multi array and a scalar, element-wise.

Definition at line 526 of file np.hpp.

```
00527 {
00528     return rhs - lhs;
00529 }
```

5.1.2.9 operator-() [3/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator- (
            T const & lhs,
            boost::multi_array< T, ND > const & rhs )  [inline]
```

Minus operator between a scalar and a multi array, element-wise.

Definition at line 517 of file np.hpp.

```
00518 {
00519     std::function<T(T)> func = [lhs](T item)
00520     { return lhs - item; };
00521     return np::element_wise_apply(rhs, func);
00522 }
```

5.1.2.10 operator/() [1/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator/ (
            boost::multi_array< T, ND > const & lhs,
            boost::multi_array< T, ND > const & rhs )
```

Division between two multi arrays, element wise.

Definition at line 534 of file np.hpp.

```
00535 {
00536     std::function<T(T, T)> func = std::divides<T>();
00537     return np::element_wise_duo_apply(lhs, rhs, func);
00538 }
```

5.1.2.11 operator/() [2/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator/ (
            boost::multi_array< T, ND > const & lhs,
            T const & rhs )  [inline]
```

Division between a multi array and a scalar, element wise.

Definition at line 551 of file np.hpp.

```
00552 {
00553     return rhs / lhs;
00554 }
```

5.1.2.12 operator/() [3/3]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > operator/ (
            T const & lhs,
            boost::multi_array< T, ND > const & rhs )  [inline]
```

Division between a scalar and a multi array, element wise.

Definition at line 542 of file np.hpp.

```
00543 {
00544     std::function<T(T)> func = [lhs](T item)
00545     { return lhs / item; };
00546     return np::element_wise_apply(rhs, func);
00547 }
```

Chapter 6

Namespace Documentation

6.1 np Namespace Reference

Custom implementation of numpy in C++.

Typedefs

- typedef double ndArrayValue

Enumerations

- enum **indexing** { **xy** , **ij** }

Functions

- template<std::size_t ND>
  boost::multi_array< ndArrayValue, ND >::index getIndex (const boost::multi_array< ndArrayValue, ND > &m, const ndArrayValue *requestedElement, const unsigned short int direction)

  Gets the index of one element in a multi_array in one axis.
- template<std::size_t ND>
  boost::array< typename boost::multi_array< ndArrayValue, ND >::index, ND > getIndexArray (const boost::multi_array< ndArrayValue, ND > &m, const ndArrayValue *requestedElement)

  Gets the index of one element in a multi_array.
- template<typename Array , typename Element , typename Functor >
  void for_each (const boost::type< Element > &type_dispatch, Array A, Functor &xform)
- template<typename Element , typename Functor >
  void for_each (const boost::type< Element > &, Element &Val, Functor &xform)

  Function to apply a function to all elements of a multi_array.
- template<typename Element , typename Iterator , typename Functor >
  void for_each (const boost::type< Element > &type_dispatch, Iterator begin, Iterator end, Functor &xform)

  Function to apply a function to all elements of a multi_array.
- template<typename Array , typename Functor >
  void for_each (Array &A, Functor xform)

- template<long unsigned int ND>
  constexpr std::vector< boost::multi_array< double, ND > > gradient (boost::multi_array< double, ND > inArray, std::initializer_list< double > args)
- boost::multi_array< double, 1 > linspace (double start, double stop, long unsigned int num)

  Implements the numpy linspace function.
- template<long unsigned int ND>
  std::vector< boost::multi_array< double, ND > > meshgrid (const boost::multi_array< double, 1 >(&cinput)[ND], bool sparsing=false, indexing indexing_type=xy)
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > element_wise_apply (const boost::multi_array< T, ND > &input_array, std::function< T(T)> func)

  Creates a new array and fills it with the values of the result of the function called on the input array element-wise.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > sqrt (const boost::multi_array< T, ND > &input_array)

  Implements the numpy sqrt function on multi arrays.
- template<class T >
  T sqrt (const T input)

  Implements the numpy sqrt function on scalars.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > exp (const boost::multi_array< T, ND > &input_array)

  Implements the numpy exp function on multi arrays.
- template<class T >
  T exp (const T input)

  Implements the numpy exp function on scalars.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > log (const boost::multi_array< T, ND > &input_array)

  Implements the numpy log function on multi arrays.
- template<class T >
  T log (const T input)

  Implements the numpy log function on scalars.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > pow (const boost::multi_array< T, ND > &input_array, const T exponent)

  Implements the numpy pow function on multi arrays.
- template<class T >
  T pow (const T input, const T exponent)

  Implements the numpy pow function on scalars.
- template<class T , long unsigned int ND>
  boost::multi_array< T, ND > element_wise_duo_apply (boost::multi_array< T, ND > const &lhs, boost::multi_array< T, ND > const &rhs, std::function< T(T, T)> func)

6.1.1 Detailed Description

Custom implementation of numpy in C++.

6.1.2 Typedef Documentation

6.1.2.1 ndArrayValue

```
typedef double np::ndArrayValue
```

Definition at line 22 of file np.hpp.

6.1.3 Enumeration Type Documentation

6.1.3.1 indexing

```
enum np::indexing
```

Definition at line 172 of file np.hpp.

```
00173     {
00174         xy,
00175         ij
00176     };
```

6.1.4 Function Documentation

6.1.4.1 element_wise_apply()

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::element_wise_apply (
            const boost::multi_array< T, ND > & input_array,
            std::function< T(T)> func )  [inline]
```

Creates a new array and fills it with the values of the result of the function called on the input array element-wise.

Definition at line 243 of file np.hpp.

```
00244     {
00245
00246         // Create output array copying extents
00247         using arrayIndex = boost::multi_array<double, ND>::index;
00248         using ndIndexArray = boost::array<arrayIndex, ND>;
00249         boost::detail::multi_array::extent_gen<ND> output_extents;
00250         std::vector<size_t> shape_list;
00251         for (std::size_t i = 0; i < ND; i++)
00252         {
00253             shape_list.push_back(input_array.shape()[i]);
00254         }
00255         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00256         boost::multi_array<T, ND> output_array(output_extents);
00257
00258         // Looping through the elements of the output array
00259         const T *p = input_array.data();
00260         ndIndexArray index;
00261         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00262         {
00263             index = getIndexArray(input_array, p);
00264             output_array(index) = func(input_array(index));
00265             ++p;
00266         }
00267         return output_array;
00268     }
```

6.1.4.2 element_wise_duo_apply()

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::element_wise_duo_apply (
            boost::multi_array< T, ND > const & lhs,
            boost::multi_array< T, ND > const & rhs,
            std::function< T(T, T)> func )
```

Creates a new array in which the value at each index is the the result of the input function applied to an element of the left hand side array and one on the righ hand side array in the same index Outputs a copy of the result

Definition at line 337 of file np.hpp.

```
00338     {
00339         // Create output array copying extents
00340         using arrayIndex = boost::multi_array<double, ND>::index;
00341         using ndIndexArray = boost::array<arrayIndex, ND>;
00342         boost::detail::multi_array::extent_gen<ND> output_extents;
00343         std::vector<size_t> shape_list;
00344         for (std::size_t i = 0; i < ND; i++)
00345         {
00346             shape_list.push_back(lhs.shape()[i]);
00347         }
00348         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00349         boost::multi_array<T, ND> output_array(output_extents);
00350 
00351         // Looping through the elements of the output array
00352         const T *p = lhs.data();
00353         ndIndexArray index;
00354         for (std::size_t i = 0; i < lhs.num_elements(); i++)
00355         {
00356             index = getIndexArray(lhs, p);
00357             output_array(index) = func(lhs(index), rhs(index));
00358             ++p;
00359         }
00360         return output_array;
00361     }
```

6.1.4.3 exp() [1/2]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::exp (
            const boost::multi_array< T, ND > & input_array )  [inline]
```

Implements the numpy exp function on multi arrays.

Definition at line 289 of file np.hpp.

```
00290     {
00291         std::function<T(T)> func = (T(*)(T))std::exp;
00292         return element_wise_apply(input_array, func);
00293     }
```

6.1.4.4 exp() [2/2]

```
template<class T >
T np::exp (
            const T input )  [inline]
```

Implements the numpy exp function on scalars.

Definition at line 297 of file np.hpp.

```
00298     {
00299         return std::exp(input);
00300     }
```

6.1.4.5 for_each() [1/4]

```
template<typename Array , typename Functor >
void np::for_each (
             Array & A,
             Functor xform )  [inline]
```

Function to apply a function to all elements of a multi_array Simple overload

Definition at line 80 of file np.hpp.

```
00081     {
00082         // Dispatch to the proper function
00083         for_each(boost::type<typename Array::element>(), A.begin(), A.end(), xform);
00084     }
```

6.1.4.6 for_each() [2/4]

```
template<typename Element , typename Functor >
void np::for_each (
             const boost::type< Element > & ,
             Element & Val,
             Functor & xform )  [inline]
```

Function to apply a function to all elements of a multi_array.

Definition at line 59 of file np.hpp.

```
00060     {
00061         Val = xform(Val);
00062     }
```

6.1.4.7 for_each() [3/4]

```
template<typename Array , typename Element , typename Functor >
void np::for_each (
             const boost::type< Element > & type_dispatch,
             Array A,
             Functor & xform )  [inline]
```

Function to apply a function to all elements of a multi_array Simple overload

Definition at line 51 of file np.hpp.

```
00053     {
00054         for_each(type_dispatch, A.begin(), A.end(), xform);
00055     }
```

6.1.4.8 for_each() [4/4]

```
template<typename Element , typename Iterator , typename Functor >
void np::for_each (
            const boost::type< Element > & type_dispatch,
            Iterator begin,
            Iterator end,
            Functor & xform )  [inline]
```

Function to apply a function to all elements of a multi_array.

Definition at line 66 of file np.hpp.

```
00069     {
00070         while (begin != end)
00071         {
00072             for_each(type_dispatch, *begin, xform);
00073             ++begin;
00074         }
00075     }
```

6.1.4.9 getIndex()

```
template<std::size_t ND>
boost::multi_array< ndArrayValue, ND >::index np::getIndex (
            const boost::multi_array< ndArrayValue, ND > & m,
            const ndArrayValue * requestedElement,
            const unsigned short int direction )  [inline]
```

Gets the index of one element in a multi_array in one axis.

Definition at line 27 of file np.hpp.

```
00028     {
00029         int offset = requestedElement - m.origin();
00030         return (offset / m.strides()[direction] % m.shape()[direction] + m.index_bases()[direction]);
00031     }
```

6.1.4.10 getIndexArray()

```
template<std::size_t ND>
boost::array< typename boost::multi_array< ndArrayValue, ND >::index, ND > np::getIndexArray
(
            const boost::multi_array< ndArrayValue, ND > & m,
            const ndArrayValue * requestedElement )  [inline]
```

Gets the index of one element in a multi_array.

Definition at line 36 of file np.hpp.

```
00037     {
00038         using indexType = boost::multi_array<ndArrayValue, ND>::index;
00039         boost::array<indexType, ND> _index;
00040         for (unsigned int dir = 0; dir < ND; dir++)
00041         {
00042             _index[dir] = getIndex(m, requestedElement, dir);
00043         }
00044
00045         return _index;
00046     }
```

6.1.4.11 gradient()

```
template<long unsigned int ND>
constexpr std::vector< boost::multi_array< double, ND > > np::gradient (
            boost::multi_array< double, ND > inArray,
            std::initializer_list< double > args )  [inline], [constexpr]
```

Takes the gradient of a n-dimensional multi_array Todo: Actually implement the gradient calculation template <long unsigned int ND, typename... Args>

Definition at line 90 of file np.hpp.

```
00091     {
00092         // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions
      of the array");
00093         using arrayIndex = boost::multi_array<double, ND>::index;
00094 
00095         using ndIndexArray = boost::array<arrayIndex, ND>;
00096 
00097         // constexpr std::size_t n = sizeof...(Args);
00098         std::size_t n = args.size();
00099         // std::tuple<Args...> store(args...);
00100         std::vector<double> arg_vector = args;
00101         boost::multi_array<double, ND> my_array;
00102         std::vector<boost::multi_array<double, ND» output_arrays;
00103         for (std::size_t i = 0; i < n; i++)
00104         {
00105             boost::multi_array<double, ND> dfdh = inArray;
00106             output_arrays.push_back(dfdh);
00107         }
00108 
00109         ndArrayValue *p = inArray.data();
00110         ndIndexArray index;
00111         for (std::size_t i = 0; i < inArray.num_elements(); i++)
00112         {
00113             index = getIndexArray(inArray, p);
00114             /*
00115             std::cout « "Index: ";
00116             for (std::size_t j = 0; j < n; j++)
00117             {
00118                 std::cout « index[j] « " ";
00119             }
00120             std::cout « "\n";
00121             */
00122             // Calculating the gradient now
00123             // j is the axis/dimension
00124             for (std::size_t j = 0; j < n; j++)
00125             {
00126                 ndIndexArray index_high = index;
00127                 double dh_high;
00128                 if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
00129                 {
00130                     index_high[j] += 1;
00131                     dh_high = arg_vector[j];
00132                 }
00133                 else
00134                 {
00135                     dh_high = 0;
00136                 }
00137                 ndIndexArray index_low = index;
00138                 double dh_low;
00139                 if (index_low[j] > 0)
00140                 {
00141                     index_low[j] -= 1;
00142                     dh_low = arg_vector[j];
00143                 }
00144                 else
00145                 {
00146                     dh_low = 0;
00147                 }
00148 
00149                 double dh = dh_high + dh_low;
00150                 double gradient = (inArray(index_high) - inArray(index_low)) / dh;
00151                 // std::cout « gradient « "\n";
00152                 output_arrays[j](index) = gradient;
00153             }
00154             // std::cout « " value = " « inArray(index) « "  check = " « *p « std::endl;
00155             ++p;
00156         }
00157         return output_arrays;
00158     }
```

6.1.4.12 linspace()

```
boost::multi_array< double, 1 > np::linspace (
             double start,
             double stop,
             long unsigned int num )  [inline]
```

Implements the numpy linspace function.

Definition at line 161 of file np.hpp.

```
00162     {
00163         double step = (stop - start) / (num - 1);
00164         boost::multi_array<double, 1> output(boost::extents[num]);
00165         for (std::size_t i = 0; i < num; i++)
00166         {
00167             output[i] = start + i * step;
00168         }
00169         return output;
00170     }
```

6.1.4.13 log() [1/2]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::log (
             const boost::multi_array< T, ND > & input_array )  [inline]
```

Implements the numpy log function on multi arrays.

Definition at line 304 of file np.hpp.

```
00305     {
00306         std::function<T(T)> func = std::log<T>();
00307         return element_wise_apply(input_array, func);
00308     }
```

6.1.4.14 log() [2/2]

```
template<class T >
T np::log (
             const T input )  [inline]
```

Implements the numpy log function on scalars.

Definition at line 312 of file np.hpp.

```
00313     {
00314         return std::log(input);
00315     }
```

6.1.4.15 meshgrid()

```
template<long unsigned int ND>
std::vector< boost::multi_array< double, ND > > np::meshgrid (
             const boost::multi_array< double, 1 >(&) cinput[ND],
             bool sparsing = false,
             indexing indexing_type = xy )  [inline]
```

Implementation of meshgrid TODO: Implement sparsing=true If the indexing type is xx, then reverse the order of the first two elements of ci if the number of dimensions is 2 or 3 In accordance with the numpy implementation

Definition at line 184 of file np.hpp.

```
00185     {
00186         using arrayIndex = boost::multi_array<double, ND>::index;
00187         using ndIndexArray = boost::array<arrayIndex, ND>;
00188         std::vector<boost::multi_array<double, ND» output_arrays;
00189         boost::multi_array<double, 1> ci[ND];
00190         // Copy elements of cinput to ci, do the proper inversions
00191         for (std::size_t i = 0; i < ND; i++)
00192         {
00193             std::size_t source = i;
00194             if (indexing_type == xy && (ND == 3 || ND == 2))
00195             {
00196                 switch (i)
00197                 {
00198                 case 0:
00199                     source = 1;
00200                     break;
00201                 case 1:
00202                     source = 0;
00203                     break;
00204                 default:
00205                     break;
00206                 }
00207             }
00208             ci[i] = boost::multi_array<double, 1>();
00209             ci[i].resize(boost::extents[cinput[source].num_elements()]);
00210             ci[i] = cinput[source];
00211         }
00212         // Deducing the extents of the N-Dimensional output
00213         boost::detail::multi_array::extent_gen<ND> output_extents;
00214         std::vector<size_t> shape_list;
00215         for (std::size_t i = 0; i < ND; i++)
00216         {
00217             shape_list.push_back(ci[i].shape()[0]);
00218         }
00219         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00220
00221         // Creating the output arrays
00222         for (std::size_t i = 0; i < ND; i++)
00223         {
00224             boost::multi_array<double, ND> output_array(output_extents);
00225             ndArrayValue *p = output_array.data();
00226             ndIndexArray index;
00227             // Looping through the elements of the output array
00228             for (std::size_t j = 0; j < output_array.num_elements(); j++)
00229             {
00230                 index = getIndexArray(output_array, p);
00231                 boost::multi_array<double, 1>::index index_1d;
00232                 index_1d = index[i];
00233                 output_array(index) = ci[i][index_1d];
00234                 ++p;
00235             }
00236             output_arrays.push_back(output_array);
00237         }
00238         return output_arrays;
00239     }
```

6.1.4.16 pow() [1/2]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::pow (
```

```
            const boost::multi_array< T, ND > & input_array,
            const T exponent )  [inline]
```

Implements the numpy pow function on multi arrays.

Definition at line 319 of file np.hpp.

```
00320     {
00321         std::function<T(T)> pow_func = [exponent](T input)
00322         { return std::pow(input, exponent); };
00323         return element_wise_apply(input_array, pow_func);
00324     }
```

6.1.4.17 pow() [2/2]

```
template<class T >
T np::pow (
            const T input,
            const T exponent )  [inline]
```

Implements the numpy pow function on scalars.

Definition at line 328 of file np.hpp.

```
00329     {
00330         return std::pow(input, exponent);
00331     }
```

6.1.4.18 sqrt() [1/2]

```
template<class T , long unsigned int ND>
boost::multi_array< T, ND > np::sqrt (
            const boost::multi_array< T, ND > & input_array )  [inline]
```

Implements the numpy sqrt function on multi arrays.

Definition at line 274 of file np.hpp.

```
00275     {
00276         std::function<T(T)> func = (T(*)(T))std::sqrt;
00277         return element_wise_apply(input_array, func);
00278     }
```

6.1.4.19 sqrt() [2/2]

```
template<class T >
T np::sqrt (
            const T input )  [inline]
```

Implements the numpy sqrt function on scalars.

Definition at line 282 of file np.hpp.

```
00283     {
00284         return std::sqrt(input);
00285     }
```

Chapter 7

File Documentation

7.1 coeff.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_COEFF_HPP
00006 #define WAVESIMC_COEFF_HPP
00007
00008 #include "CustomLibraries/np.hpp"
00009 #include <math.h>
00010
00011
00012 boost::multi_array<double, 2> get_sigma_1(boost::multi_array<double, 1> x, double dx, int nx, int nz,
00013                                           double c_max, int n=10, double R=1e-3, int m=2)
00014 {
00015     boost::multi_array<double, 2> sigma_1 = np::zeros(nx, nz);
00016     const double PML_width = n * dx;
00017     const double sigma_max = - c_max * log(R) * (m+1) / (PML_width**(m+1));
00018
00019     // TODO: max: find the maximum element in 1D array
00020     const double x_0 = max(x) - PML_width;
00021
00022     // each column of sigma_1 is a 1D array named "polynomial"
00023     boost::multi_array<double, 1> polynomial = np::zeros(nx);
00024     for (int i=0; i<nx; i++)
00025     {
00026         if (x[i] > x_0)
00027         {
00028             // TODO: Does math.h have an absolute value function?
00029             polynomial[i] = sigma_max * abs(x[i] - x_0)**m;
00030             polynomial[nx-i] = polynomial[i];
00031         }
00032         else
00033         {
00034             polynomial[i] = 0;
00035         }
00036     }
00037
00038     // Copy 1D array into each column of 2D array
00039     for (int i=0; i<nx; i++)
00040         for (int j=0; i<nz; j++)
00041             sigma_1[i][j] = polynomial[i];
00042
00043     return sigma_1;
00044 }
00045
00046
00047
00048 boost::multi_array<double, 2> get_sigma_2(boost::multi_array<double, 1> z, double dz, int nx, int nz,
00049                                           double c_max, int n=10, double R=1e-3, int m=2)
00050 {
00051     boost::multi_array<double, 2> sigma_2 = np::zeros(nx, nz);
00052     const double PML_width = n * dz;
00053     const double sigma_max = - c_max * log(R) * (m+1) / (PML_width**(m+1));
00054
00055     // TODO: max: find the maximum element in 1D array
00056     const double z_0 = max(z) - PML_width;
00057
00058     // each column of sigma_1 is a 1D array named "polynomial"
```

```
00059     boost::multi_array<double, 1> polynomial = np::zeros(nz);
00060     for (int j=0; j<nz; j++)
00061     {
00062         if (z[j] > z_0)
00063         {
00064             // TODO: Does math.h have an absolute value function?
00065             polynomial[j] = sigma_max * abs(z[j] - z_0)**m;
00066             polynomial[nz-j] = polynomial[j];
00067         }
00068         else
00069         {
00070             polynomial[j] = 0;
00071         }
00072     }
00073
00074     // Copy 1D array into each column of 2D array
00075     for (int i=0; i<nx; i++)
00076         for (int j=0; i<nz; j++)
00077             sigma_1[i][j] = polynomial[j];
00078
00079     return sigma_2;
00080 }
00081
00082 #endif //WAVESIMC_COEFF_HPP
```

7.2 computational.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_COMPUTATIONAL_HPP
00006 #define WAVESIMC_COMPUTATIONAL_HPP
00007
00008 boost::multi_array<double, 2> get_profile()
00009 {
00010
00011 }
00012
00013 #endif //WAVESIMC_COMPUTATIONAL_HPP
```

7.3 helper_func.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_HELPER_FUNC_HPP
00006 #define WAVESIMC_HELPER_FUNC_HPP
00007
00008 #include "CustomLibraries/np.hpp"
00009
00010 boost::multi_array<double, 2> dfdx(boost::multi_array<double, 2> f, double dx)
00011 {
00012     std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dx, dx});
00013     return grad_f[0];
00014 }
00015
00016 boost::multi_array<double, 2> dfdz(boost::multi_array<double, 2> f, double dz)
00017 {
00018     std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dz, dz});
00019     return grad_f[1];
00020 }
00021
00022 boost::multi_array<double, 2> d2fdx2(boost::multi_array<double, 2> f, double dx)
00023 {
00024     boost::multi_array<double, 2> f_x = dfdx(f, dx);
00025     boost::multi_array<double, 2> f_xx = dfdx(f_x, dx);
00026     return f_xx;
00027 }
00028
00029 boost::multi_array<double, 2> d2fdz2(boost::multi_array<double, 2> f, double dz)
00030 {
00031     boost::multi_array<double, 2> f_z = dfdz(f, dz);
00032     boost::multi_array<double, 2> f_zz = dfdx(f_z, dz);
00033     return f_zz;
00034 }
00035
```

```
00036 boost::multi_array<double, 2> divergence(boost::multi_array<double, 2> f1, boost::multi_array<double,
       2> f2,
00037                                          double dx, double dz)
00038 {
00039     boost::multi_array<double, 2> f_x = dfdx(f1, dx);
00040     boost::multi_array<double, 2> f_z = dfdz(f2, dz);
00041     // TODO: use element-wize add
00042     div = f1 + f2;
00043     return div;
00044 }
00045
00046
00047 #endif //WAVESIMC_HELPER_FUNC_HPP
```

7.4 solver.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_SOLVER_HPP
00006 #define WAVESIMC_SOLVER_HPP
00007
00008 #include "CustomLibraries/np.hpp"
00009 #include "helper_func.hpp"
00010
00011 boost::multi_array<double, 3> wave_solver(boost::multi_array<double, 2> c,
00012                                           double dt, double dx, double dz, int nt, int nx, int nz,
00013                                           boost::multi_array<double, 3> f,
00014                                           boost::multi_array<double, 2> sigma_1,
       boost::multi_array<double, 2> sigma_2)
00015 {
00016     // TODO: "same shape" functionality of np::zeros
00017     boost::multi_array<double, 3> u = np::zeros(nt, nx, nz);
00018     boost::multi_array<double, 2> u_xx = np::zeros(nx, ny);
00019     boost::multi_array<double, 2> u_zz = np::zeros(nx, ny);
00020     boost::multi_array<double, 2> q_1 = np::zeros(nx, ny);
00021     boost::multi_array<double, 2> q_2 = np::zeros(nx, ny);
00022
00023     // TODO: make multiplication between scalar and boost::multi_array<double, 2> work
00024     // Basically we need to make * and ** work
00025     const boost::multi_array<double, 2> C1 = 1 + dt * (sigma_1 + sigma_2)/((double) 2);
00026     // Question: Is ((double) 2) necessary?
00027     const boost::multi_array<double, 2> C2 = sigma_1 * sigma_2 * (dt**2) - 2;
00028     const boost::multi_array<double, 2> C3 = 1 - dt*(sigma_1 + sigma_2)/2;
00029     const boost::multi_array<double, 2> C4 = (dt*c)**2;
00030     const boost::multi_array<double, 2> C5 = 1 + dt*sigma_1/2;
00031     const boost::multi_array<double, 2> C6 = 1 + dt*sigma_2/2;
00032     const boost::multi_array<double, 2> C7 = 1 - dt*sigma_1/2;
00033     const boost::multi_array<double, 2> C8 = 1 - dt*sigma_2/2;
00034
00035     for (int n = 0; n < nt; n++)
00036     {
00037         u_xx = d2fdx2(u[n], dx);
00038         u_zz = d2fdz2(u[n], dz);
00039
00040         u[n+1] = (C4*(u_xx/(dx**2) + u_zz/(dz**2) - divergence(q_1*sigma_1, q_2*sigma_2, dx, dz)
00041                       + sigma_2*dfdx(q_1, dx) + sigma_1*dfdz(q_2, dz) + f[n]) -
00042                   C2 * u[n] - C3 * u[n-1]) / C1;
00043
00044         q_1 = (dt*dfdx(u[n], dx) + C7*q_1) / C5;
00045         q_2 = (dt*dfdz(u[n], dx) + C8*q_2) / C6;
00046
00047         // Dirichlet boundary condition
00048         for (int i = 0; i < nx; i++)
00049         {
00050             u[n+1][i][0] = 0;
00051             u[n+1][i][nx-1] = 0;
00052         }
00053         for (int j = 0; j < nz; j++)
00054         {
00055             u[n+1][0][j] = 0;
00056             u[n+1][nz-1][j] = 0;
00057         }
00058     }
00059     return u;
00060 }
00061
00062 #endif //WAVESIMC_SOLVER_HPP
```

7.5 source.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_SOURCE_HPP
00006 #define WAVESIMC_SOURCE_HPP
00007
00008
00009 boost::multi_array<double, 3> ricker(int i_s, int j_s, double f=10, double amp=1e0, double shift=0.1)
00010 {
00011     const double pi = 3.141592654;
00012
00013     boost::multi_array<double, 1> t = np::linspace(tmin, tmax, nt);
00014
00015     // TODO: element-wise operators
00016     boost::multi_array<double, 1> pft2 = (pi * f * (t - shift))**2;
00017     boost::multi_array<double, 1> r = amp * (1 - 2 * pft2) * exp(-pft2);
00018
00019     boost::multi_array<double, 1> x = np.zeros(nx);
00020     boost::multi_array<double, 1> z = np.zeros(nz);
00021     x[i_s] = 1.0;
00022     z[j_s] = 1.0;
00023     boost::multi_array<double, 3> TXZ = np::meshgrid(r, x, z, sparse=True, indexing='ij');
00024
00025     return TXZ;
00026 }
00027
00028 #endif //WAVESIMC_SOURCE_HPP
```

7.6 wave.cpp

```
00001 // For the core algorithm, we need six functionalities:
00002 // 1) create the computational domain,
00003 // 2) create a velocity profile (1 & 2 can be put together)
00004 // 3) create attenuation coefficients,
00005 // 4) create source functions,
00006 // 5) helper functions to compute eg. df/dx
00007 // 6) use all above to create a solver function for wave equation
00008
00009 // Standard IO libraries
00010 #include <iostream>
00011 #include <fstream>
00012 #include "CustomLibraries/np.hpp"
00013
00014 #include <math.h>
00015
00016 #include "solver.hpp"
00017 #include "computational.hpp"
00018 #include "coeff.hpp"
00019 #include "source.hpp"
00020 #include "helper_func.hpp"
00021
00022
00023 int main()
00024 {
00025     double dx, dy, dz, dt;
00026     dx = 1.0;
00027     dy = 1.0;
00028     dz = 1.0;
00029     dt = 1.0;
00030     std::vector<boost::multi_array<double, 4» my_arrays = np::gradient(A, {dx, dy, dz, dt});
00031     return 0;
00032 }
```

7.7 np.hpp

```
00001 #ifndef NP_H_
00002 #define NP_H_
00003
00004 #include "boost/multi_array.hpp"
00005 #include "boost/array.hpp"
00006 #include "boost/cstdlib.hpp"
00007 #include <type_traits>
00008 #include <cassert>
00009 #include <iostream>
00010 #include <functional>
00011 #include <type_traits>
```

```
00012
00019 namespace np
00020 {
00021
00022     typedef double ndArrayValue;
00023
00025     template <std::size_t ND>
00026     inline boost::multi_array<ndArrayValue, ND>::index
00027     getIndex(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement,
      const unsigned short int direction)
00028     {
00029         int offset = requestedElement - m.origin();
00030         return (offset / m.strides()[direction] % m.shape()[direction] + m.index_bases()[direction]);
00031     }
00032
00034     template <std::size_t ND>
00035     inline boost::array<typename boost::multi_array<ndArrayValue, ND>::index, ND>
00036     getIndexArray(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement)
00037     {
00038         using indexType = boost::multi_array<ndArrayValue, ND>::index;
00039         boost::array<indexType, ND> _index;
00040         for (unsigned int dir = 0; dir < ND; dir++)
00041         {
00042             _index[dir] = getIndex(m, requestedElement, dir);
00043         }
00044
00045         return _index;
00046     }
00047
00050     template <typename Array, typename Element, typename Functor>
00051     inline void for_each(const boost::type<Element> &type_dispatch,
00052                          Array A, Functor &xform)
00053     {
00054         for_each(type_dispatch, A.begin(), A.end(), xform);
00055     }
00056
00058     template <typename Element, typename Functor>
00059     inline void for_each(const boost::type<Element> &, Element &Val, Functor &xform)
00060     {
00061         Val = xform(Val);
00062     }
00063
00065     template <typename Element, typename Iterator, typename Functor>
00066     inline void for_each(const boost::type<Element> &type_dispatch,
00067                          Iterator begin, Iterator end,
00068                          Functor &xform)
00069     {
00070         while (begin != end)
00071         {
00072             for_each(type_dispatch, *begin, xform);
00073             ++begin;
00074         }
00075     }
00076
00079     template <typename Array, typename Functor>
00080     inline void for_each(Array &A, Functor xform)
00081     {
00082         // Dispatch to the proper function
00083         for_each(boost::type<typename Array::element>(), A.begin(), A.end(), xform);
00084     }
00085
00089     template <long unsigned int ND>
00090     inline constexpr std::vector<boost::multi_array<double, ND>> gradient(boost::multi_array<double,
      ND> inArray, std::initializer_list<double> args)
00091     {
00092         // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions
      of the array");
00093         using arrayIndex = boost::multi_array<double, ND>::index;
00094
00095         using ndIndexArray = boost::array<arrayIndex, ND>;
00096
00097         // constexpr std::size_t n = sizeof...(Args);
00098         std::size_t n = args.size();
00099         // std::tuple<Args...> store(args...);
00100         std::vector<double> arg_vector = args;
00101         boost::multi_array<double, ND> my_array;
00102         std::vector<boost::multi_array<double, ND>> output_arrays;
00103         for (std::size_t i = 0; i < n; i++)
00104         {
00105             boost::multi_array<double, ND> dfdh = inArray;
00106             output_arrays.push_back(dfdh);
00107         }
00108
00109         ndArrayValue *p = inArray.data();
00110         ndIndexArray index;
00111         for (std::size_t i = 0; i < inArray.num_elements(); i++)
00112         {
```

```
00113             index = getIndexArray(inArray, p);
00114             /*
00115             std::cout « "Index: ";
00116             for (std::size_t j = 0; j < n; j++)
00117             {
00118                 std::cout « index[j] « " ";
00119             }
00120             std::cout « "\n";
00121             */
00122             // Calculating the gradient now
00123             // j is the axis/dimension
00124             for (std::size_t j = 0; j < n; j++)
00125             {
00126                 ndIndexArray index_high = index;
00127                 double dh_high;
00128                 if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
00129                 {
00130                     index_high[j] += 1;
00131                     dh_high = arg_vector[j];
00132                 }
00133                 else
00134                 {
00135                     dh_high = 0;
00136                 }
00137                 ndIndexArray index_low = index;
00138                 double dh_low;
00139                 if (index_low[j] > 0)
00140                 {
00141                     index_low[j] -= 1;
00142                     dh_low = arg_vector[j];
00143                 }
00144                 else
00145                 {
00146                     dh_low = 0;
00147                 }
00148 
00149                 double dh = dh_high + dh_low;
00150                 double gradient = (inArray(index_high) - inArray(index_low)) / dh;
00151                 // std::cout « gradient « "\n";
00152                 output_arrays[j](index) = gradient;
00153             }
00154             // std::cout « " value = " « inArray(index) « "  check = " « *p « std::endl;
00155             ++p;
00156         }
00157         return output_arrays;
00158     }
00159 
00161     inline boost::multi_array<double, 1> linspace(double start, double stop, long unsigned int num)
00162     {
00163         double step = (stop - start) / (num - 1);
00164         boost::multi_array<double, 1> output(boost::extents[num]);
00165         for (std::size_t i = 0; i < num; i++)
00166         {
00167             output[i] = start + i * step;
00168         }
00169         return output;
00170     }
00171 
00172     enum indexing
00173     {
00174         xy,
00175         ij
00176     };
00177 
00183     template <long unsigned int ND>
00184     inline std::vector<boost::multi_array<double, ND» meshgrid(const boost::multi_array<double, 1>
       (&cinput)[ND], bool sparsing = false, indexing indexing_type = xy)
00185     {
00186         using arrayIndex = boost::multi_array<double, ND>::index;
00187         using ndIndexArray = boost::array<arrayIndex, ND>;
00188         std::vector<boost::multi_array<double, ND» output_arrays;
00189         boost::multi_array<double, 1> ci[ND];
00190         // Copy elements of cinput to ci, do the proper inversions
00191         for (std::size_t i = 0; i < ND; i++)
00192         {
00193             std::size_t source = i;
00194             if (indexing_type == xy && (ND == 3 || ND == 2))
00195             {
00196                 switch (i)
00197                 {
00198                 case 0:
00199                     source = 1;
00200                     break;
00201                 case 1:
00202                     source = 0;
00203                     break;
00204                 default:
```

```
00205                     break;
00206                 }
00207             }
00208             ci[i] = boost::multi_array<double, 1>();
00209             ci[i].resize(boost::extents[cinput[source].num_elements()]);
00210             ci[i] = cinput[source];
00211         }
00212         // Deducing the extents of the N-Dimensional output
00213         boost::detail::multi_array::extent_gen<ND> output_extents;
00214         std::vector<size_t> shape_list;
00215         for (std::size_t i = 0; i < ND; i++)
00216         {
00217             shape_list.push_back(ci[i].shape()[0]);
00218         }
00219         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00220 
00221         // Creating the output arrays
00222         for (std::size_t i = 0; i < ND; i++)
00223         {
00224             boost::multi_array<double, ND> output_array(output_extents);
00225             ndArrayValue *p = output_array.data();
00226             ndIndexArray index;
00227             // Looping through the elements of the output array
00228             for (std::size_t j = 0; j < output_array.num_elements(); j++)
00229             {
00230                 index = getIndexArray(output_array, p);
00231                 boost::multi_array<double, 1>::index index_1d;
00232                 index_1d = index[i];
00233                 output_array(index) = ci[i][index_1d];
00234                 ++p;
00235             }
00236             output_arrays.push_back(output_array);
00237         }
00238         return output_arrays;
00239     }
00240 
00242     template <class T, long unsigned int ND>
00243     inline boost::multi_array<T, ND> element_wise_apply(const boost::multi_array<T, ND> &input_array,
    std::function<T(T)> func)
00244     {
00245 
00246         // Create output array copying extents
00247         using arrayIndex = boost::multi_array<double, ND>::index;
00248         using ndIndexArray = boost::array<arrayIndex, ND>;
00249         boost::detail::multi_array::extent_gen<ND> output_extents;
00250         std::vector<size_t> shape_list;
00251         for (std::size_t i = 0; i < ND; i++)
00252         {
00253             shape_list.push_back(input_array.shape()[i]);
00254         }
00255         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00256         boost::multi_array<T, ND> output_array(output_extents);
00257 
00258         // Looping through the elements of the output array
00259         const T *p = input_array.data();
00260         ndIndexArray index;
00261         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00262         {
00263             index = getIndexArray(input_array, p);
00264             output_array(index) = func(input_array(index));
00265             ++p;
00266         }
00267         return output_array;
00268     }
00269 
00270     // Complex operations
00271 
00273     template <class T, long unsigned int ND>
00274     inline boost::multi_array<T, ND> sqrt(const boost::multi_array<T, ND> &input_array)
00275     {
00276         std::function<T(T)> func = (T(*)(T))std::sqrt;
00277         return element_wise_apply(input_array, func);
00278     }
00279 
00281     template <class T>
00282     inline T sqrt(const T input)
00283     {
00284         return std::sqrt(input);
00285     }
00286 
00288     template <class T, long unsigned int ND>
00289     inline boost::multi_array<T, ND> exp(const boost::multi_array<T, ND> &input_array)
00290     {
00291         std::function<T(T)> func = (T(*)(T))std::exp;
00292         return element_wise_apply(input_array, func);
00293     }
00294 
```

```
00296     template <class T>
00297     inline T exp(const T input)
00298     {
00299         return std::exp(input);
00300     }
00301
00303     template <class T, long unsigned int ND>
00304     inline boost::multi_array<T, ND> log(const boost::multi_array<T, ND> &input_array)
00305     {
00306         std::function<T(T)> func = std::log<T>();
00307         return element_wise_apply(input_array, func);
00308     }
00309
00311     template <class T>
00312     inline T log(const T input)
00313     {
00314         return std::log(input);
00315     }
00316
00318     template <class T, long unsigned int ND>
00319     inline boost::multi_array<T, ND> pow(const boost::multi_array<T, ND> &input_array, const T
    exponent)
00320     {
00321         std::function<T(T)> pow_func = [exponent](T input)
00322         { return std::pow(input, exponent); };
00323         return element_wise_apply(input_array, pow_func);
00324     }
00325
00327     template <class T>
00328     inline T pow(const T input, const T exponent)
00329     {
00330         return std::pow(input, exponent);
00331     }
00332
00336     template <class T, long unsigned int ND>
00337     boost::multi_array<T, ND> element_wise_duo_apply(boost::multi_array<T, ND> const &lhs,
    boost::multi_array<T, ND> const &rhs, std::function<T(T, T)> func)
00338     {
00339         // Create output array copying extents
00340         using arrayIndex = boost::multi_array<double, ND>::index;
00341         using ndIndexArray = boost::array<arrayIndex, ND>;
00342         boost::detail::multi_array::extent_gen<ND> output_extents;
00343         std::vector<size_t> shape_list;
00344         for (std::size_t i = 0; i < ND; i++)
00345         {
00346             shape_list.push_back(lhs.shape()[i]);
00347         }
00348         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00349         boost::multi_array<T, ND> output_array(output_extents);
00350
00351         // Looping through the elements of the output array
00352         const T *p = lhs.data();
00353         ndIndexArray index;
00354         for (std::size_t i = 0; i < lhs.num_elements(); i++)
00355         {
00356             index = getIndexArray(lhs, p);
00357             output_array(index) = func(lhs(index), rhs(index));
00358             ++p;
00359         }
00360         return output_array;
00361     }
00362
00364     template <typename T, typename inT, long unsigned int ND>
00365     inline constexpr boost::multi_array<T, ND> zeros(inT (&dimensions_input)[ND]) requires
    std::is_integral<inT>::value && std::is_arithmetic<T>::value
00366     {
00367         // Deducing the extents of the N-Dimensional output
00368         boost::detail::multi_array::extent_gen<ND> output_extents;
00369         std::vector<size_t> shape_list;
00370         for (std::size_t i = 0; i < ND; i++)
00371         {
00372             shape_list.push_back(dimensions_input[i]);
00373         }
00374         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00375         // Applying a function to return zero always to all of its elements
00376         boost::multi_array<T, ND> output_array(output_extents);
00377         std::function<T(T)> zero_func = [](T input)
00378         { return 0; };
00379         return element_wise_apply(output_array, zero_func);
00380     }
00381
00383     template <typename T, long unsigned int ND>
00384     inline constexpr T max(boost::multi_array<T, ND> const &input_array) requires
    std::is_arithmetic<T>::value
00385     {
00386         T max = 0;
00387         bool max_not_set = true;
```

```
00388         const T *data_pointer = input_array.data();
00389         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00390         {
00391             T element = *data_pointer;
00392             if (max_not_set || element > max)
00393             {
00394                 max = element;
00395                 max_not_set = false;
00396             }
00397             ++data_pointer;
00398         }
00399         return max;
00400     }
00401
00403     template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
00404     inline constexpr T max(T input1, Ts... inputs) requires std::is_arithmetic<T>::value
00405     {
00406         T max = input1;
00407         for (T input : {inputs...})
00408         {
00409             if (input > max)
00410             {
00411                 max = input;
00412             }
00413         }
00414         return max;
00415     }
00416
00418     template <typename T, long unsigned int ND>
00419     inline constexpr T min(boost::multi_array<T, ND> const &input_array) requires
      std::is_arithmetic<T>::value
00420     {
00421         T min = 0;
00422         bool min_not_set = true;
00423         const T *data_pointer = input_array.data();
00424         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00425         {
00426             T element = *data_pointer;
00427             if (min_not_set || element < min)
00428             {
00429                 min = element;
00430                 min_not_set = false;
00431             }
00432             ++data_pointer;
00433         }
00434         return min;
00435     }
00436
00438     template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
00439     inline constexpr T min(T input1, Ts... inputs) requires std::is_arithmetic<T>::value
00440     {
00441         T min = input1;
00442         for (T input : {inputs...})
00443         {
00444             if (input < min)
00445             {
00446                 min = input;
00447             }
00448         }
00449         return min;
00450     }
00451 }
00452
00453 // Override of operators in the boost::multi_array class to make them more np-like
00454 // Basic operators
00455 // All of the are element-wise
00456
00457 // Multiplication operator
00459 template <class T, long unsigned int ND>
00460 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, boost::multi_array<T,
      ND> const &rhs)
00461 {
00462     std::function<T(T, T)> func = std::multiplies<T>();
00463     return np::element_wise_duo_apply(lhs, rhs, func);
00464 }
00465
00467 template <class T, long unsigned int ND>
00468 inline boost::multi_array<T, ND> operator*(T const &lhs, boost::multi_array<T, ND> const &rhs)
00469 {
00470     std::function<T(T)> func = [lhs](T item)
00471     { return lhs * item; };
00472     return np::element_wise_apply(rhs, func);
00473 }
00475 template <class T, long unsigned int ND>
00476 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, T const &rhs)
00477 {
00478     return rhs * lhs;
```

```
00479 }
00480
00481 // Plus operator
00483 template <class T, long unsigned int ND>
00484 boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00485 {
00486     std::function<T(T, T)> func = std::plus<T>();
00487     return np::element_wise_duo_apply(lhs, rhs, func);
00488 }
00489
00491 template <class T, long unsigned int ND>
00492 inline boost::multi_array<T, ND> operator+(T const &lhs, boost::multi_array<T, ND> const &rhs)
00493 {
00494     std::function<T(T)> func = [lhs](T item)
00495     { return lhs + item; };
00496     return np::element_wise_apply(rhs, func);
00497 }
00498
00500 template <class T, long unsigned int ND>
00501 inline boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, T const &rhs)
00502 {
00503     return rhs + lhs;
00504 }
00505
00506 // Subtraction operator
00508 template <class T, long unsigned int ND>
00509 boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00510 {
00511     std::function<T(T, T)> func = std::minus<T>();
00512     return np::element_wise_duo_apply(lhs, rhs, func);
00513 }
00514
00516 template <class T, long unsigned int ND>
00517 inline boost::multi_array<T, ND> operator-(T const &lhs, boost::multi_array<T, ND> const &rhs)
00518 {
00519     std::function<T(T)> func = [lhs](T item)
00520     { return lhs - item; };
00521     return np::element_wise_apply(rhs, func);
00522 }
00523
00525 template <class T, long unsigned int ND>
00526 inline boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, T const &rhs)
00527 {
00528     return rhs - lhs;
00529 }
00530
00531 // Division operator
00533 template <class T, long unsigned int ND>
00534 boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00535 {
00536     std::function<T(T, T)> func = std::divides<T>();
00537     return np::element_wise_duo_apply(lhs, rhs, func);
00538 }
00539
00541 template <class T, long unsigned int ND>
00542 inline boost::multi_array<T, ND> operator/(T const &lhs, boost::multi_array<T, ND> const &rhs)
00543 {
00544     std::function<T(T)> func = [lhs](T item)
00545     { return lhs / item; };
00546     return np::element_wise_apply(rhs, func);
00547 }
00548
00550 template <class T, long unsigned int ND>
00551 inline boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, T const &rhs)
00552 {
00553     return rhs / lhs;
00554 }
00555
00557 #endif
```

7.8 main.cpp

```
00001 #include <iostream>
00002 #include <string>
00003 #include "ExternalLibraries/cxxopts.hpp"
00004 #include "CustomLibraries/np.hpp"
00005
00006 // Command line arguments
00007 cxxopts::Options options("WaveSimC", "A wave propagation simulator written in C++ for seismic data
       processing.");
```

```
00008 int main(int argc, char *argv[])
00009 {
00010     // Parse command line arguments
00011     options.add_options()("d,debug", "Enable debugging")("i,input_file", "Input file path",
      cxxopts::value<std::string>())("o,output_file", "Output file path",
      cxxopts::value<std::string>())("v,verbose", "Verbose output",
      cxxopts::value<bool>()->default_value("false"));
00012     auto result = options.parse(argc, argv);
00013 
00014     std::cout « "Hello World"
00015               « "\n";
00016 }
```

7.9 variadic.cpp

```
00001 #include "boost/multi_array.hpp"
00002 #include "boost/array.hpp"
00003 #include "CustomLibraries/np.hpp"
00004 #include <cassert>
00005 #include <iostream>
00006 
00007 void test_gradient()
00008 {
00009     // Create a 4D array that is 3 x 4 x 2 x 1
00010     typedef boost::multi_array<double, 4>::index index;
00011     boost::multi_array<double, 4> A(boost::extents[3][4][2][2]);
00012 
00013     // Assign values to the elements
00014     int values = 0;
00015     for (index i = 0; i != 3; ++i)
00016         for (index j = 0; j != 4; ++j)
00017             for (index k = 0; k != 2; ++k)
00018                 for (index l = 0; l != 2; ++l)
00019                     A[i][j][k][l] = values++;
00020 
00021     // Verify values
00022     int verify = 0;
00023     for (index i = 0; i != 3; ++i)
00024         for (index j = 0; j != 4; ++j)
00025             for (index k = 0; k != 2; ++k)
00026                 for (index l = 0; l != 2; ++l)
00027                     assert(A[i][j][k][l] == verify++);
00028 
00029     double dx, dy, dz, dt;
00030     dx = 1.0;
00031     dy = 1.0;
00032     dz = 1.0;
00033     dt = 1.0;
00034     std::vector<boost::multi_array<double, 4» my_arrays = np::gradient(A, {dx, dy, dz, dt});
00035 
00036     boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
00037     std::vector<boost::multi_array<double, 1» gradf = np::gradient(x, {1.0});
00038     for (int i = 0; i < 5; i++)
00039     {
00040         std::cout « gradf[0][i] « ",";
00041     }
00042     std::cout « "\n";
00043     // np::print(std::cout, my_arrays[0]);
00044 }
00045 
00046 void test_meshgrid()
00047 {
00048     boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
00049     boost::multi_array<double, 1> y = np::linspace(0, 1, 5);
00050     boost::multi_array<double, 1> z = np::linspace(0, 1, 5);
00051     boost::multi_array<double, 1> t = np::linspace(0, 1, 5);
00052     const boost::multi_array<double, 1> axis[4] = {x, y, z, t};
00053     std::vector<boost::multi_array<double, 4» my_arrays = np::meshgrid(axis, false, np::xy);
00054     // np::print(std::cout, my_arrays[0]);
00055     int nx = 3;
00056     int ny = 2;
00057     boost::multi_array<double, 1> x2 = np::linspace(0, 1, nx);
00058     boost::multi_array<double, 1> y2 = np::linspace(0, 1, ny);
00059     const boost::multi_array<double, 1> axis2[2] = {x2, y2};
00060     std::vector<boost::multi_array<double, 2» my_arrays2 = np::meshgrid(axis2, false, np::xy);
00061     std::cout « "xv\n";
00062     for (int i = 0; i < ny; i++)
00063     {
00064         for (int j = 0; j < nx; j++)
00065         {
00066             std::cout « my_arrays2[0][i][j] « " ";
00067         }
00068         std::cout « "\n";
```

```
00069     }
00070     std::cout « "yv\n";
00071     for (int i = 0; i < ny; i++)
00072     {
00073         for (int j = 0; j < nx; j++)
00074         {
00075             std::cout « my_arrays2[1][i][j] « " ";
00076         }
00077         std::cout « "\n";
00078     }
00079 }
00080
00081 void test_complex_operations()
00082 {
00083     int nx = 3;
00084     int ny = 2;
00085     boost::multi_array<double, 1> x = np::linspace(0, 1, nx);
00086     boost::multi_array<double, 1> y = np::linspace(0, 1, ny);
00087     const boost::multi_array<double, 1> axis[2] = {x, y};
00088     std::vector<boost::multi_array<double, 2» my_arrays = np::meshgrid(axis, false, np::xy);
00089     boost::multi_array<double, 2> A = np::sqrt(my_arrays[0]);
00090     std::cout « "sqrt\n";
00091     for (int i = 0; i < ny; i++)
00092     {
00093         for (int j = 0; j < nx; j++)
00094         {
00095             std::cout « A[i][j] « " ";
00096         }
00097         std::cout « "\n";
00098     }
00099     std::cout « "\n";
00100     float a = 100.0;
00101     float sqa = np::sqrt(a);
00102     std::cout « "sqrt of " « a « " is " « sqa « "\n";
00103     std::cout « "exp\n";
00104     boost::multi_array<double, 2> B = np::exp(my_arrays[0]);
00105     for (int i = 0; i < ny; i++)
00106     {
00107         for (int j = 0; j < nx; j++)
00108         {
00109             std::cout « B[i][j] « " ";
00110         }
00111         std::cout « "\n";
00112     }
00113
00114     std::cout « "Power\n";
00115     boost::multi_array<double, 1> x2 = np::linspace(1, 3, nx);
00116     boost::multi_array<double, 1> y2 = np::linspace(1, 3, ny);
00117     const boost::multi_array<double, 1> axis2[2] = {x2, y2};
00118     std::vector<boost::multi_array<double, 2» my_arrays2 = np::meshgrid(axis2, false, np::xy);
00119     boost::multi_array<double, 2> C = np::pow(my_arrays2[1], 2.0);
00120     for (int i = 0; i < ny; i++)
00121     {
00122         for (int j = 0; j < nx; j++)
00123         {
00124             std::cout « C[i][j] « " ";
00125         }
00126         std::cout « "\n";
00127     }
00128 }
00129
00130 void test_equal()
00131 {
00132     boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
00133     boost::multi_array<double, 1> y = np::linspace(0, 1, 5);
00134     boost::multi_array<double, 1> z = np::linspace(0, 1, 5);
00135     boost::multi_array<double, 1> t = np::linspace(0, 1, 5);
00136     const boost::multi_array<double, 1> axis[4] = {x, y, z, t};
00137     std::vector<boost::multi_array<double, 4» my_arrays = np::meshgrid(axis, false, np::xy);
00138     boost::multi_array<double, 1> x2 = np::linspace(0, 1, 5);
00139     boost::multi_array<double, 1> y2 = np::linspace(0, 1, 5);
00140     boost::multi_array<double, 1> z2 = np::linspace(0, 1, 5);
00141     boost::multi_array<double, 1> t2 = np::linspace(0, 1, 5);
00142     const boost::multi_array<double, 1> axis2[4] = {x2, y2, z2, t2};
00143     std::vector<boost::multi_array<double, 4» my_arrays2 = np::meshgrid(axis2, false, np::xy);
00144     std::cout « "equality test:\n";
00145     std::cout « (bool)(my_arrays == my_arrays2) « "\n";
00146 }
00147 void test_basic_operations()
00148 {
00149     int nx = 3;
00150     int ny = 2;
00151     boost::multi_array<double, 1> x = np::linspace(0, 1, nx);
00152     boost::multi_array<double, 1> y = np::linspace(0, 1, ny);
00153     const boost::multi_array<double, 1> axis[2] = {x, y};
00154     std::vector<boost::multi_array<double, 2» my_arrays = np::meshgrid(axis, false, np::xy);
00155
```

```
00156     std::cout « "basic operations:\n";
00157
00158     std::cout « "addition:\n";
00159     boost::multi_array<double, 2> A = my_arrays[0] + my_arrays[1];
00160
00161     for (int i = 0; i < ny; i++)
00162     {
00163         for (int j = 0; j < nx; j++)
00164         {
00165             std::cout « A[i][j] « " ";
00166         }
00167         std::cout « "\n";
00168     }
00169
00170     std::cout « "multiplication:\n";
00171     boost::multi_array<double, 2> B = my_arrays[0] * my_arrays[1];
00172
00173     for (int i = 0; i < ny; i++)
00174     {
00175         for (int j = 0; j < nx; j++)
00176         {
00177             std::cout « B[i][j] « " ";
00178         }
00179         std::cout « "\n";
00180     }
00181     double coeff = 3;
00182     boost::multi_array<double, 1> t = np::linspace(0, 1, nx);
00183     boost::multi_array<double, 1> t_time_3 = coeff * t;
00184     boost::multi_array<double, 1> t_time_2 = 2.0 * t;
00185     std::cout « "t_time_3: ";
00186     for (int j = 0; j < nx; j++)
00187     {
00188         std::cout « t_time_3[j] « " ";
00189     }
00190     std::cout « "\n";
00191     std::cout « "t_time_2: ";
00192     for (int j = 0; j < nx; j++)
00193     {
00194         std::cout « t_time_2[j] « " ";
00195     }
00196     std::cout « "\n";
00197 }
00198
00199 void test_zeros()
00200 {
00201     int nx = 3;
00202     int ny = 2;
00203     int dimensions[] = {ny, nx};
00204     boost::multi_array<double, 2> A = np::zeros<double>(dimensions);
00205     std::cout « "zeros:\n";
00206     for (int i = 0; i < ny; i++)
00207     {
00208         for (int j = 0; j < nx; j++)
00209         {
00210             std::cout « A[i][j] « " ";
00211         }
00212         std::cout « "\n";
00213     }
00214 }
00215
00216 void test_min_max()
00217 {
00218     int nx = 24;
00219     int ny = 5;
00220     boost::multi_array<double, 1> x = np::linspace(0, 10, nx);
00221     boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
00222     const boost::multi_array<double, 1> axis[2] = {x, y};
00223     std::vector<boost::multi_array<double, 2» my_array = np::meshgrid(axis, false, np::xy);
00224     std::cout « "min: " « np::min(my_array[0]) « "\n";
00225     std::cout « "max: " « np::max(my_array[1]) « "\n";
00226     std::cout « "max simple: " « np::max(1.0, 2.0, 3.0, 4.0, 5.0) « "\n";
00227     std::cout « "min simple: " « np::min(1, -2, 3, -4, 5) « "\n";
00228 }
00229
00230 void test_toy_problem()
00231 {
00232     boost::multi_array<double, 1> x = np::linspace(0, 1, 100);
00233     boost::multi_array<double, 1> y = np::linspace(0, 1, 100);
00234     // x = np::pow(x, 2.0);
00235     // y = np::pow(y, 3.0);
00236
00237     const boost::multi_array<double, 1> axis[2] = {x, y};
00238     std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
00239
00240     double dx, dy;
00241     dx = 1.0 / 100.0;
00242     dy = 1.0 / 100.0;
```

```
00243
00244     boost::multi_array<double, 2> f = np::pow(XcY[0], 2.0) + XcY[0] * np::pow(XcY[1], 1.0);
00245
00246     // g.push_back(np::gradient(XcY[0], {dx, dy}));
00247     // g.push_back(np::gradient(XcY[1], {dx, dy}));
00248     std::vector<boost::multi_array<double, 2» gradf = np::gradient(f, {dx, dy});
00249     // auto [gradfx_x, gradfx_y] = np::gradient(f, {dx, dy});
00250
00251     int i, j;
00252     i = 10;
00253     j = 20;
00254     std::cout « "df/dx at x = " « x[i] « " and y = " « y[j] « " is equal to " « gradf[0][i][j];
00255
00256     std::cout « "\n";
00257 }
00258
00259 int main()
00260 {
00261     test_gradient();
00262     test_meshgrid();
00263     test_complex_operations();
00264     test_equal();
00265     test_basic_operations();
00266     test_zeros();
00267     test_min_max();
00268     test_toy_problem();
00269 }
```