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 459 of file np.hpp.

```
00460 {
00461     std::function<T(T, T)> func = std::multiplies<T>();
00462     return np::element_wise_duo_apply(lhs, rhs, func);
00463 }
```

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 475 of file np.hpp.

```
00476 {
00477     return rhs * lhs;
00478 }
```

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 467 of file np.hpp.

```
00468 {
00469     std::function<T(T)> func = [lhs](T item)
00470     { return lhs * item; };
00471     return np::element_wise_apply(rhs, func);
00472 }
```

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 483 of file np.hpp.

```
00484 {
00485     std::function<T(T, T)> func = std::plus<T>();
00486     return np::element_wise_duo_apply(lhs, rhs, func);
00487 }
```

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 500 of file np.hpp.

```
00501 {
00502     return rhs + lhs;
00503 }
```

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 491 of file np.hpp.

```
00492 {
00493     std::function<T(T)> func = [lhs](T item)
00494     { return lhs + item; };
00495     return np::element_wise_apply(rhs, func);
00496 }
```

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 508 of file np.hpp.

```
00509 {
00510     std::function<T(T, T)> func = std::minus<T>();
00511     return np::element_wise_duo_apply(lhs, rhs, func);
00512 }
```

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 525 of file np.hpp.

```
00526 {
00527     return rhs - lhs;
00528 }
```

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 516 of file np.hpp.

```
00517 {
00518     std::function<T(T)> func = [lhs](T item)
00519     { return lhs - item; };
00520     return np::element_wise_apply(rhs, func);
00521 }
```

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 533 of file np.hpp.

```
00534 {
00535     std::function<T(T, T)> func = std::divides<T>();
00536     return np::element_wise_duo_apply(lhs, rhs, func);
00537 }
```

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 550 of file np.hpp.

```
00551 {
00552     return rhs / lhs;
00553 }
```

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 541 of file np.hpp.

```
00542 {
00543     std::function<T(T)> func = [lhs](T item)
00544     { return lhs / item; };
00545     return np::element_wise_apply(rhs, func);
00546 }
```

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)
- template<typename T , typename inT , long unsigned int ND>
  requires std::is_integral<inT>
  ::value &&std::is_arithmetic< T >::value constexpr boost::multi_array< T, ND > zeros (inT(&dimensions_input)[ND])

  Implements the numpy zeros function for an n-dimensionl multi array.
- template<typename T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr T max (boost::multi_array< T, ND > const &input_array)

  Implements the numpy max function for an n-dimensionl multi array.
- template<class T , class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
  requires std::is_arithmetic<T>
  ::value constexpr T max (T input1, Ts... inputs)

  Implements the numpy max function for an variadic number of arguments.
- template<typename T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr T min (boost::multi_array< T, ND > const &input_array)

  Implements the numpy min function for an n-dimensionl multi array.

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 171 of file np.hpp.

```
00172     {
00173         xy,
00174         ij
00175     };
```

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 242 of file np.hpp.

```
00243     {
00244
00245         // Create output array copying extents
00246         using arrayIndex = boost::multi_array<double, ND>::index;
00247         using ndIndexArray = boost::array<arrayIndex, ND>;
00248         boost::detail::multi_array::extent_gen<ND> output_extents;
00249         std::vector<size_t> shape_list;
00250         for (std::size_t i = 0; i < ND; i++)
00251         {
00252             shape_list.push_back(input_array.shape()[i]);
```

```
00253         }
00254         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00255         boost::multi_array<T, ND> output_array(output_extents);
00256 
00257         // Looping through the elements of the output array
00258         const T *p = input_array.data();
00259         ndIndexArray index;
00260         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00261         {
00262             index = getIndexArray(input_array, p);
00263             output_array(index) = func(input_array(index));
00264             ++p;
00265         }
00266         return output_array;
00267     }
```

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 336 of file np.hpp.

```
00337     {
00338         // Create output array copying extents
00339         using arrayIndex = boost::multi_array<double, ND>::index;
00340         using ndIndexArray = boost::array<arrayIndex, ND>;
00341         boost::detail::multi_array::extent_gen<ND> output_extents;
00342         std::vector<size_t> shape_list;
00343         for (std::size_t i = 0; i < ND; i++)
00344         {
00345             shape_list.push_back(lhs.shape()[i]);
00346         }
00347         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00348         boost::multi_array<T, ND> output_array(output_extents);
00349 
00350         // Looping through the elements of the output array
00351         const T *p = lhs.data();
00352         ndIndexArray index;
00353         for (std::size_t i = 0; i < lhs.num_elements(); i++)
00354         {
00355             index = getIndexArray(lhs, p);
00356             output_array(index) = func(lhs(index), rhs(index));
00357             ++p;
00358         }
00359         return output_array;
00360     }
```

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 288 of file np.hpp.

```
00289     {
00290         std::function<T(T)> func = (T(*)(T))std::exp;
00291         return element_wise_apply(input_array, func);
00292     }
```

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 296 of file np.hpp.

```
00297     {
00298         return std::exp(input);
00299     }
```

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

Definition at line 89 of file np.hpp.

```
00090     {
00091         // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions
      of the array");
00092         using arrayIndex = boost::multi_array<double, ND>::index;
00093 
00094         using ndIndexArray = boost::array<arrayIndex, ND>;
00095 
00096         // constexpr std::size_t n = sizeof...(Args);
00097         std::size_t n = args.size();
00098         // std::tuple<Args...> store(args...);
00099         std::vector<double> arg_vector = args;
00100         boost::multi_array<double, ND> my_array;
00101         std::vector<boost::multi_array<double, ND» output_arrays;
00102         for (std::size_t i = 0; i < n; i++)
00103         {
00104             boost::multi_array<double, ND> dfdh = inArray;
00105             output_arrays.push_back(dfdh);
00106         }
00107 
00108         ndArrayValue *p = inArray.data();
00109         ndIndexArray index;
00110         for (std::size_t i = 0; i < inArray.num_elements(); i++)
00111         {
00112             index = getIndexArray(inArray, p);
00113             /*
00114             std::cout « "Index: ";
00115             for (std::size_t j = 0; j < n; j++)
00116             {
00117                 std::cout « index[j] « " ";
00118             }
00119             std::cout « "\n";
00120             */
00121             // Calculating the gradient now
00122             // j is the axis/dimension
00123             for (std::size_t j = 0; j < n; j++)
00124             {
00125                 ndIndexArray index_high = index;
00126                 double dh_high;
00127                 if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
00128                 {
00129                     index_high[j] += 1;
00130                     dh_high = arg_vector[j];
00131                 }
00132                 else
00133                 {
00134                     dh_high = 0;
00135                 }
00136                 ndIndexArray index_low = index;
00137                 double dh_low;
00138                 if (index_low[j] > 0)
00139                 {
00140                     index_low[j] -= 1;
00141                     dh_low = arg_vector[j];
00142                 }
00143                 else
00144                 {
00145                     dh_low = 0;
00146                 }
00147 
00148                 double dh = dh_high + dh_low;
00149                 double gradient = (inArray(index_high) - inArray(index_low)) / dh;
00150                 // std::cout « gradient « "\n";
00151                 output_arrays[j](index) = gradient;
00152             }
00153             // std::cout « " value = " « inArray(index) « "  check = " « *p « std::endl;
00154             ++p;
00155         }
00156         return output_arrays;
00157     }
```

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 160 of file np.hpp.

```
00161     {
00162         double step = (stop - start) / (num - 1);
00163         boost::multi_array<double, 1> output(boost::extents[num]);
00164         for (std::size_t i = 0; i < num; i++)
00165         {
00166             output[i] = start + i * step;
00167         }
00168         return output;
00169     }
```

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 303 of file np.hpp.

```
00304     {
00305         std::function<T(T)> func = std::log<T>();
00306         return element_wise_apply(input_array, func);
00307     }
```

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 311 of file np.hpp.

```
00312     {
00313         return std::log(input);
00314     }
```

6.1.4.15 max() [1/2]

```
template<typename T , long unsigned int ND>
requires std::is_arithmetic<T>
::value constexpr T np::max (
            boost::multi_array< T, ND > const & input_array )  [inline], [constexpr]
```

Implements the numpy max function for an n-dimensionl multi array.

Definition at line 383 of file np.hpp.

```
00384     {
00385         T max = 0;
00386         bool max_not_set = true;
00387         const T *data_pointer = input_array.data();
00388         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00389         {
00390             T element = *data_pointer;
00391             if (max_not_set || element > max)
00392             {
00393                 max = element;
00394                 max_not_set = false;
00395             }
00396             ++data_pointer;
00397         }
00398         return max;
00399     }
```

6.1.4.16 max() [2/2]

```
template<class T , class...  Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
requires std::is_arithmetic<T>
::value constexpr T np::max (
            T input1,
            Ts...  inputs )  [inline], [constexpr]
```

Implements the numpy max function for an variadic number of arguments.

Definition at line 403 of file np.hpp.

```
00404     {
00405         T max = input1;
00406         for (T input : {inputs...})
00407         {
00408             if (input > max)
00409             {
00410                 max = input;
00411             }
00412         }
00413         return max;
00414     }
```

6.1.4.17 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 183 of file np.hpp.

```
00184     {
00185         using arrayIndex = boost::multi_array<double, ND>::index;
00186         using ndIndexArray = boost::array<arrayIndex, ND>;
00187         std::vector<boost::multi_array<double, ND» output_arrays;
00188         boost::multi_array<double, 1> ci[ND];
00189         // Copy elements of cinput to ci, do the proper inversions
00190         for (std::size_t i = 0; i < ND; i++)
00191         {
00192             std::size_t source = i;
00193             if (indexing_type == xy && (ND == 3 || ND == 2))
00194             {
00195                 switch (i)
00196                 {
00197                 case 0:
00198                     source = 1;
00199                     break;
00200                 case 1:
00201                     source = 0;
00202                     break;
00203                 default:
00204                     break;
00205                 }
00206             }
00207             ci[i] = boost::multi_array<double, 1>();
00208             ci[i].resize(boost::extents[cinput[source].num_elements()]);
00209             ci[i] = cinput[source];
00210         }
00211         // Deducing the extents of the N-Dimensional output
00212         boost::detail::multi_array::extent_gen<ND> output_extents;
00213         std::vector<size_t> shape_list;
00214         for (std::size_t i = 0; i < ND; i++)
00215         {
00216             shape_list.push_back(ci[i].shape()[0]);
00217         }
00218         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00219 
00220         // Creating the output arrays
00221         for (std::size_t i = 0; i < ND; i++)
00222         {
00223             boost::multi_array<double, ND> output_array(output_extents);
00224             ndArrayValue *p = output_array.data();
00225             ndIndexArray index;
00226             // Looping through the elements of the output array
00227             for (std::size_t j = 0; j < output_array.num_elements(); j++)
00228             {
00229                 index = getIndexArray(output_array, p);
00230                 boost::multi_array<double, 1>::index index_1d;
00231                 index_1d = index[i];
00232                 output_array(index) = ci[i][index_1d];
00233                 ++p;
00234             }
00235             output_arrays.push_back(output_array);
00236         }
00237         return output_arrays;
00238     }
```

6.1.4.18 min()

```
template<typename T , long unsigned int ND>
requires std::is_arithmetic<T>
::value constexpr T np::min (
            boost::multi_array< T, ND > const & input_array )  [inline], [constexpr]
```

Implements the numpy min function for an n-dimensionl multi array.

Definition at line 418 of file np.hpp.

```
00419     {
00420         T min = 0;
00421         bool min_not_set = true;
00422         const T *data_pointer = input_array.data();
00423         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00424         {
00425             T element = *data_pointer;
00426             if (min_not_set || element < min)
00427             {
```

```
00428                 min = element;
00429                 min_not_set = false;
00430             }
00431             ++data_pointer;
00432         }
00433         return min;
00434     }
```

6.1.4.19 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 318 of file np.hpp.

```
00319     {
00320         std::function<T(T)> pow_func = [exponent](T input)
00321         { return std::pow(input, exponent); };
00322         return element_wise_apply(input_array, pow_func);
00323     }
```

6.1.4.20 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 327 of file np.hpp.

```
00328     {
00329         return std::pow(input, exponent);
00330     }
```

6.1.4.21 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 273 of file np.hpp.

```
00274     {
00275         std::function<T(T)> func = (T(*)(T))std::sqrt;
00276         return element_wise_apply(input_array, func);
00277     }
```

6.1.4.22 sqrt() [2/2]

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

Implements the numpy sqrt function on scalars.

Definition at line 281 of file np.hpp.

```
00282     {
00283         return std::sqrt(input);
00284     }
```

6.1.4.23 zeros()

```
template<typename T , typename inT , long unsigned int ND>
requires std::is_integral<inT>
::value &&std::is_arithmetic< T >::value constexpr boost::multi_array< T, ND > np::zeros (
             inT(&) dimensions_input[ND] )  [inline], [constexpr]
```

Implements the numpy zeros function for an n-dimensionl multi array.

Definition at line 364 of file np.hpp.

```
00365     {
00366         // Deducing the extents of the N-Dimensional output
00367         boost::detail::multi_array::extent_gen<ND> output_extents;
00368         std::vector<size_t> shape_list;
00369         for (std::size_t i = 0; i < ND; i++)
00370         {
00371             shape_list.push_back(dimensions_input[i]);
00372         }
00373         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00374         // Applying a function to return zero always to all of its elements
00375         boost::multi_array<T, ND> output_array(output_extents);
00376         std::function<T(T)> zero_func = [](T input)
00377         { return 0; };
00378         return element_wise_apply(output_array, zero_func);
00379     }
```

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
00088     template <long unsigned int ND>
00089     inline constexpr std::vector<boost::multi_array<double, ND>> gradient(boost::multi_array<double,
      ND> inArray, std::initializer_list<double> args)
00090     {
00091         // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions
      of the array");
00092         using arrayIndex = boost::multi_array<double, ND>::index;
00093
00094         using ndIndexArray = boost::array<arrayIndex, ND>;
00095
00096         // constexpr std::size_t n = sizeof...(Args);
00097         std::size_t n = args.size();
00098         // std::tuple<Args...> store(args...);
00099         std::vector<double> arg_vector = args;
00100         boost::multi_array<double, ND> my_array;
00101         std::vector<boost::multi_array<double, ND>> output_arrays;
00102         for (std::size_t i = 0; i < n; i++)
00103         {
00104             boost::multi_array<double, ND> dfdh = inArray;
00105             output_arrays.push_back(dfdh);
00106         }
00107
00108         ndArrayValue *p = inArray.data();
00109         ndIndexArray index;
00110         for (std::size_t i = 0; i < inArray.num_elements(); i++)
00111         {
```

```
00112             index = getIndexArray(inArray, p);
00113             /*
00114             std::cout « "Index: ";
00115             for (std::size_t j = 0; j < n; j++)
00116             {
00117                 std::cout « index[j] « " ";
00118             }
00119             std::cout « "\n";
00120             */
00121             // Calculating the gradient now
00122             // j is the axis/dimension
00123             for (std::size_t j = 0; j < n; j++)
00124             {
00125                 ndIndexArray index_high = index;
00126                 double dh_high;
00127                 if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
00128                 {
00129                     index_high[j] += 1;
00130                     dh_high = arg_vector[j];
00131                 }
00132                 else
00133                 {
00134                     dh_high = 0;
00135                 }
00136                 ndIndexArray index_low = index;
00137                 double dh_low;
00138                 if (index_low[j] > 0)
00139                 {
00140                     index_low[j] -= 1;
00141                     dh_low = arg_vector[j];
00142                 }
00143                 else
00144                 {
00145                     dh_low = 0;
00146                 }
00147
00148                 double dh = dh_high + dh_low;
00149                 double gradient = (inArray(index_high) - inArray(index_low)) / dh;
00150                 // std::cout « gradient « "\n";
00151                 output_arrays[j](index) = gradient;
00152             }
00153             // std::cout « " value = " « inArray(index) « "  check = " « *p « std::endl;
00154             ++p;
00155         }
00156         return output_arrays;
00157     }
00158
00160     inline boost::multi_array<double, 1> linspace(double start, double stop, long unsigned int num)
00161     {
00162         double step = (stop - start) / (num - 1);
00163         boost::multi_array<double, 1> output(boost::extents[num]);
00164         for (std::size_t i = 0; i < num; i++)
00165         {
00166             output[i] = start + i * step;
00167         }
00168         return output;
00169     }
00170
00171     enum indexing
00172     {
00173         xy,
00174         ij
00175     };
00176
00182     template <long unsigned int ND>
00183     inline std::vector<boost::multi_array<double, ND>> meshgrid(const boost::multi_array<double, 1>
        (&cinput)[ND], bool sparsing = false, indexing indexing_type = xy)
00184     {
00185         using arrayIndex = boost::multi_array<double, ND>::index;
00186         using ndIndexArray = boost::array<arrayIndex, ND>;
00187         std::vector<boost::multi_array<double, ND>> output_arrays;
00188         boost::multi_array<double, 1> ci[ND];
00189         // Copy elements of cinput to ci, do the proper inversions
00190         for (std::size_t i = 0; i < ND; i++)
00191         {
00192             std::size_t source = i;
00193             if (indexing_type == xy && (ND == 3 || ND == 2))
00194             {
00195                 switch (i)
00196                 {
00197                 case 0:
00198                     source = 1;
00199                     break;
00200                 case 1:
00201                     source = 0;
00202                     break;
00203                 default:
```

```
00204                     break;
00205                 }
00206             }
00207             ci[i] = boost::multi_array<double, 1>();
00208             ci[i].resize(boost::extents[cinput[source].num_elements()]);
00209             ci[i] = cinput[source];
00210         }
00211         // Deducing the extents of the N-Dimensional output
00212         boost::detail::multi_array::extent_gen<ND> output_extents;
00213         std::vector<size_t> shape_list;
00214         for (std::size_t i = 0; i < ND; i++)
00215         {
00216             shape_list.push_back(ci[i].shape()[0]);
00217         }
00218         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00219 
00220         // Creating the output arrays
00221         for (std::size_t i = 0; i < ND; i++)
00222         {
00223             boost::multi_array<double, ND> output_array(output_extents);
00224             ndArrayValue *p = output_array.data();
00225             ndIndexArray index;
00226             // Looping through the elements of the output array
00227             for (std::size_t j = 0; j < output_array.num_elements(); j++)
00228             {
00229                 index = getIndexArray(output_array, p);
00230                 boost::multi_array<double, 1>::index index_1d;
00231                 index_1d = index[i];
00232                 output_array(index) = ci[i][index_1d];
00233                 ++p;
00234             }
00235             output_arrays.push_back(output_array);
00236         }
00237         return output_arrays;
00238     }
00239 
00241     template <class T, long unsigned int ND>
00242     inline boost::multi_array<T, ND> element_wise_apply(const boost::multi_array<T, ND> &input_array,
        std::function<T(T)> func)
00243     {
00244 
00245         // Create output array copying extents
00246         using arrayIndex = boost::multi_array<double, ND>::index;
00247         using ndIndexArray = boost::array<arrayIndex, ND>;
00248         boost::detail::multi_array::extent_gen<ND> output_extents;
00249         std::vector<size_t> shape_list;
00250         for (std::size_t i = 0; i < ND; i++)
00251         {
00252             shape_list.push_back(input_array.shape()[i]);
00253         }
00254         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00255         boost::multi_array<T, ND> output_array(output_extents);
00256 
00257         // Looping through the elements of the output array
00258         const T *p = input_array.data();
00259         ndIndexArray index;
00260         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00261         {
00262             index = getIndexArray(input_array, p);
00263             output_array(index) = func(input_array(index));
00264             ++p;
00265         }
00266         return output_array;
00267     }
00268 
00269     // Complex operations
00270 
00272     template <class T, long unsigned int ND>
00273     inline boost::multi_array<T, ND> sqrt(const boost::multi_array<T, ND> &input_array)
00274     {
00275         std::function<T(T)> func = (T(*)(T))std::sqrt;
00276         return element_wise_apply(input_array, func);
00277     }
00278 
00280     template <class T>
00281     inline T sqrt(const T input)
00282     {
00283         return std::sqrt(input);
00284     }
00285 
00287     template <class T, long unsigned int ND>
00288     inline boost::multi_array<T, ND> exp(const boost::multi_array<T, ND> &input_array)
00289     {
00290         std::function<T(T)> func = (T(*)(T))std::exp;
00291         return element_wise_apply(input_array, func);
00292     }
00293 
```

```
00295     template <class T>
00296     inline T exp(const T input)
00297     {
00298         return std::exp(input);
00299     }
00300
00302     template <class T, long unsigned int ND>
00303     inline boost::multi_array<T, ND> log(const boost::multi_array<T, ND> &input_array)
00304     {
00305         std::function<T(T)> func = std::log<T>();
00306         return element_wise_apply(input_array, func);
00307     }
00308
00310     template <class T>
00311     inline T log(const T input)
00312     {
00313         return std::log(input);
00314     }
00315
00317     template <class T, long unsigned int ND>
00318     inline boost::multi_array<T, ND> pow(const boost::multi_array<T, ND> &input_array, const T
    exponent)
00319     {
00320         std::function<T(T)> pow_func = [exponent](T input)
00321         { return std::pow(input, exponent); };
00322         return element_wise_apply(input_array, pow_func);
00323     }
00324
00326     template <class T>
00327     inline T pow(const T input, const T exponent)
00328     {
00329         return std::pow(input, exponent);
00330     }
00331
00335     template <class T, long unsigned int ND>
00336     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)
00337     {
00338         // Create output array copying extents
00339         using arrayIndex = boost::multi_array<double, ND>::index;
00340         using ndIndexArray = boost::array<arrayIndex, ND>;
00341         boost::detail::multi_array::extent_gen<ND> output_extents;
00342         std::vector<size_t> shape_list;
00343         for (std::size_t i = 0; i < ND; i++)
00344         {
00345             shape_list.push_back(lhs.shape()[i]);
00346         }
00347         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00348         boost::multi_array<T, ND> output_array(output_extents);
00349
00350         // Looping through the elements of the output array
00351         const T *p = lhs.data();
00352         ndIndexArray index;
00353         for (std::size_t i = 0; i < lhs.num_elements(); i++)
00354         {
00355             index = getIndexArray(lhs, p);
00356             output_array(index) = func(lhs(index), rhs(index));
00357             ++p;
00358         }
00359         return output_array;
00360     }
00361
00363     template <typename T, typename inT, long unsigned int ND>
00364     requires std::is_integral<inT>::value && std::is_arithmetic<T>::value inline constexpr
    boost::multi_array<T, ND> zeros(inT (&dimensions_input)[ND])
00365     {
00366         // Deducing the extents of the N-Dimensional output
00367         boost::detail::multi_array::extent_gen<ND> output_extents;
00368         std::vector<size_t> shape_list;
00369         for (std::size_t i = 0; i < ND; i++)
00370         {
00371             shape_list.push_back(dimensions_input[i]);
00372         }
00373         std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00374         // Applying a function to return zero always to all of its elements
00375         boost::multi_array<T, ND> output_array(output_extents);
00376         std::function<T(T)> zero_func = [](T input)
00377         { return 0; };
00378         return element_wise_apply(output_array, zero_func);
00379     }
00380
00382     template <typename T, long unsigned int ND>
00383     requires std::is_arithmetic<T>::value inline constexpr T max(boost::multi_array<T, ND> const
    &input_array)
00384     {
00385         T max = 0;
00386         bool max_not_set = true;
```

```
00387         const T *data_pointer = input_array.data();
00388         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00389         {
00390             T element = *data_pointer;
00391             if (max_not_set || element > max)
00392             {
00393                 max = element;
00394                 max_not_set = false;
00395             }
00396             ++data_pointer;
00397         }
00398         return max;
00399     }
00400
00402     template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
00403     requires std::is_arithmetic<T>::value inline constexpr T max(T input1, Ts... inputs)
00404     {
00405         T max = input1;
00406         for (T input : {inputs...})
00407         {
00408             if (input > max)
00409             {
00410                 max = input;
00411             }
00412         }
00413         return max;
00414     }
00415
00417     template <typename T, long unsigned int ND>
00418     requires std::is_arithmetic<T>::value inline constexpr T min(boost::multi_array<T, ND> const
      &input_array)
00419     {
00420         T min = 0;
00421         bool min_not_set = true;
00422         const T *data_pointer = input_array.data();
00423         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00424         {
00425             T element = *data_pointer;
00426             if (min_not_set || element < min)
00427             {
00428                 min = element;
00429                 min_not_set = false;
00430             }
00431             ++data_pointer;
00432         }
00433         return min;
00434     }
00435
00437     template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
00438     inline constexpr T min(T input1, Ts... inputs) requires std::is_arithmetic<T>::value
00439     {
00440         T min = input1;
00441         for (T input : {inputs...})
00442         {
00443             if (input < min)
00444             {
00445                 min = input;
00446             }
00447         }
00448         return min;
00449     }
00450 }
00451
00452 // Override of operators in the boost::multi_array class to make them more np-like
00453 // Basic operators
00454 // All of the are element-wise
00455
00456 // Multiplication operator
00458 template <class T, long unsigned int ND>
00459 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, boost::multi_array<T,
      ND> const &rhs)
00460 {
00461     std::function<T(T, T)> func = std::multiplies<T>();
00462     return np::element_wise_duo_apply(lhs, rhs, func);
00463 }
00464
00466 template <class T, long unsigned int ND>
00467 inline boost::multi_array<T, ND> operator*(T const &lhs, boost::multi_array<T, ND> const &rhs)
00468 {
00469     std::function<T(T)> func = [lhs](T item)
00470     { return lhs * item; };
00471     return np::element_wise_apply(rhs, func);
00472 }
00474 template <class T, long unsigned int ND>
00475 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, T const &rhs)
00476 {
00477     return rhs * lhs;
```

```
00478 }
00479
00480 // Plus operator
00482 template <class T, long unsigned int ND>
00483 boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00484 {
00485     std::function<T(T, T)> func = std::plus<T>();
00486     return np::element_wise_duo_apply(lhs, rhs, func);
00487 }
00488
00490 template <class T, long unsigned int ND>
00491 inline boost::multi_array<T, ND> operator+(T const &lhs, boost::multi_array<T, ND> const &rhs)
00492 {
00493     std::function<T(T)> func = [lhs](T item)
00494     { return lhs + item; };
00495     return np::element_wise_apply(rhs, func);
00496 }
00497
00499 template <class T, long unsigned int ND>
00500 inline boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, T const &rhs)
00501 {
00502     return rhs + lhs;
00503 }
00504
00505 // Subtraction operator
00507 template <class T, long unsigned int ND>
00508 boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00509 {
00510     std::function<T(T, T)> func = std::minus<T>();
00511     return np::element_wise_duo_apply(lhs, rhs, func);
00512 }
00513
00515 template <class T, long unsigned int ND>
00516 inline boost::multi_array<T, ND> operator-(T const &lhs, boost::multi_array<T, ND> const &rhs)
00517 {
00518     std::function<T(T)> func = [lhs](T item)
00519     { return lhs - item; };
00520     return np::element_wise_apply(rhs, func);
00521 }
00522
00524 template <class T, long unsigned int ND>
00525 inline boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, T const &rhs)
00526 {
00527     return rhs - lhs;
00528 }
00529
00530 // Division operator
00532 template <class T, long unsigned int ND>
00533 boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
       const &rhs)
00534 {
00535     std::function<T(T, T)> func = std::divides<T>();
00536     return np::element_wise_duo_apply(lhs, rhs, func);
00537 }
00538
00540 template <class T, long unsigned int ND>
00541 inline boost::multi_array<T, ND> operator/(T const &lhs, boost::multi_array<T, ND> const &rhs)
00542 {
00543     std::function<T(T)> func = [lhs](T item)
00544     { return lhs / item; };
00545     return np::element_wise_apply(rhs, func);
00546 }
00547
00549 template <class T, long unsigned int ND>
00550 inline boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, T const &rhs)
00551 {
00552     return rhs / lhs;
00553 }
00554
00556 #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 }
```