WaveSimC

0.8

Generated by Doxygen 1.9.6

Chapter 1

Main Page

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.

Please check the Readme file for more information.

1.1.1 Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

1.1.2 License

Chapter 2

README

2.1 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.

2.1.1 <a href="https://wavesimc.vbpage.net/" >Detailed documentation</a>

2.1.2 Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

2.1.3 Acknowledgments

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

2.2 Theory

2.2.1 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}
```

2.2.2 References

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

2.2.3 Design Philosophy

2.2.3.1 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.

2.2.4 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.

2.3 Building

2.3.1 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
```

2.3.2 Build the project

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

2.3.3 Running

```
./Main
```

2.3.4 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 3

Module Index

3.1 Modules

Here is a list of all modules:

Chapter 4

Namespace Index

4.1 Namespace List

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

Chapter 5

File Index

5.1 File List

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

Chapter 6

Module Documentation

6.1 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.

6.1.1 Detailed Description

6.1.2 Function Documentation

6.1.2.1 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 504 of file np.hpp.

```
00505 {
00506     std::function<T(T, T)> func = std::multiplies<T>();
00507     return np::element_wise_duo_apply(lhs, rhs, func);
00508 }
```

6.1.2.2 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 520 of file np.hpp.

```
00521 {
00522     return rhs * lhs;
00523 }
```

6.1.2.3 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 512 of file np.hpp.

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

6.1.2.4 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 528 of file np.hpp.

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

6.1.2.5 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 545 of file np.hpp.

```
{
    return rhs + lhs;
}
```

6.1.2.6 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 536 of file np.hpp.

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

6.1.2.7 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 553 of file np.hpp.

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

6.1.2.8 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 570 of file np.hpp.

```
{
    return rhs - lhs;
}
```

6.1.2.9 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 561 of file np.hpp.

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

6.1.2.10 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 578 of file np.hpp.

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

6.1.2.11 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 595 of file np.hpp.

```
{
    return rhs / lhs;
}
```

6.1.2.12 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 586 of file np.hpp.

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

Chapter 7

Namespace Documentation

7.1 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<typename T , long unsigned int ND>
  requires std::is_floating_point<T>
  ::value constexpr std::vector< boost::multi_array< T, ND > > gradient (boost::multi_array< T, ND > inArray, std::initializer_list< T > args)
- boost::multi_array< double, 1 > linspace (double start, double stop, long unsigned int num)

  Implements the numpy linspace function.
- template<typename T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr std::vector< boost::multi_array< T, ND > > meshgrid (const boost::multi_array< T, 1 >(&cinput)[ND], bool sparsing=false, indexing indexing_type=xy)
- template<class T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr 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>
  requires std::is_arithmetic<T>
  ::value constexpr 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 >
  requires std::is_arithmetic<T>
  ::value constexpr T sqrt (const T input)

  Implements the numpy sqrt function on scalars.
- template<class T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr 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 >
  requires std::is_arithmetic<T>
  ::value constexpr T exp (const T input)

  Implements the numpy exp function on scalars.
- template<class T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr 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 >
  requires std::is_arithmetic<T>
  ::value constexpr T log (const T input)

  Implements the numpy log function on scalars.
- template<class T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr 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 >
  requires std::is_arithmetic<T>
  ::value constexpr T pow (const T input, const T exponent)

  Implements the numpy pow function on scalars.
- template<class T , long unsigned int ND>
  constexpr 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.
- template<class T , class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
  requires std::is_arithmetic<T>
  constexpr T min (T input1, Ts... inputs)

  Implements the numpy min function for an variadic number of arguments.
- template<typename T >
  requires std::is_arithmetic<T>
  ::value constexpr T abs (T input)

  Implements the numpy abs function for a scalar.
- template<typename T , long unsigned int ND>
  requires std::is_arithmetic<T>
  ::value constexpr boost::multi_array< T, ND - 1 > slice (boost::multi_array< T, ND > const &input_array, std::size_t slice_index)

  Slices the array through one dimension and returns a ND - 1 dimensional array.

7.1.1 Detailed Description

Custom implementation of numpy in C++.

7.1.2 Typedef Documentation

7.1.2.1 ndArrayValue

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

Definition at line 22 of file np.hpp.

7.1.3 Enumeration Type Documentation

7.1.3.1 indexing

```
enum np::indexing
```

Definition at line 171 of file np.hpp.

```
{
    xy,
    ij
};
```

7.1.4 Function Documentation

7.1.4.1 abs()

```
template<typename T >
requires std::is_arithmetic<T>
::value constexpr T np::abs (
             T input )  [inline], [constexpr]
```

Implements the numpy abs function for a scalar.

Definition at line 463 of file np.hpp.

```
{
    return std::abs(input);
}
```

7.1.4.2 element_wise_apply()

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

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.

```
{

    // Create output array copying extents
    using arrayIndex = boost::multi_array<double, ND>::index;
    using ndIndexArray = boost::array<arrayIndex, ND>;
    boost::detail::multi_array::extent_gen<ND> output_extents;
    std::vector<size_t> shape_list;
    for (std::size_t i = 0; i < ND; i++)
    {
        shape_list.push_back(input_array.shape()[i]);
    }
    std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
    boost::multi_array<T, ND> output_array(output_extents);

    // Looping through the elements of the output array
    const T *p = input_array.data();
    ndIndexArray index;
    for (std::size_t i = 0; i < input_array.num_elements(); i++)
    {
        index = getIndexArray(input_array, p);
        output_array(index) = func(input_array(index));
        ++p;
    }
    return output_array;
}
```

7.1.4.3 element_wise_duo_apply()

```
template<class T , long unsigned int ND>
constexpr 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 )  [inline], [constexpr]
```

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.

```
{
    // Create output array copying extents
    using arrayIndex = boost::multi_array<double, ND>::index;
    using ndIndexArray = boost::array<arrayIndex, ND>;
    boost::detail::multi_array::extent_gen<ND> output_extents;
    std::vector<size_t> shape_list;
    for (std::size_t i = 0; i < ND; i++)
    {
        shape_list.push_back(lhs.shape()[i]);
    }
    std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
    boost::multi_array<T, ND> output_array(output_extents);

    // Looping through the elements of the output array
    const T *p = lhs.data();
    ndIndexArray index;
    for (std::size_t i = 0; i < lhs.num_elements(); i++)
    {
        index = getIndexArray(lhs, p);
        output_array(index) = func(lhs(index), rhs(index));
        ++p;
    }
    return output_array;
}
```

7.1.4.4 exp() [1/2]

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

Implements the numpy exp function on multi arrays.

Definition at line 289 of file np.hpp.

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

7.1.4.5 exp() [2/2]

```
template<class T >
requires std::is_arithmetic<T>
::value constexpr T np::exp (
             const T input )  [inline], [constexpr]
```

Implements the numpy exp function on scalars.

Definition at line 297 of file np.hpp.

```
{
    return std::exp(input);
}
```

7.1.4.6 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.

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

7.1.4.7 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.

```
{
    Val = xform(Val);
}
```

7.1.4.8 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.

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

7.1.4.9 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.

```
{
    while (begin != end)
    {
        for_each(type_dispatch, *begin, xform);
        ++begin;
    }
}
```

7.1.4.10 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.

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

7.1.4.11 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.

```
{
    using indexType = boost::multi_array<ndArrayValue, ND>::index;
    boost::array<indexType, ND> _index;
    for (unsigned int dir = 0; dir < ND; dir++)
    {
        _index[dir] = getIndex(m, requestedElement, dir);
    }

    return _index;
}
```

7.1.4.12 gradient()

```
template<typename T , long unsigned int ND>
requires std::is_floating_point<T>
::value constexpr std::vector< boost::multi_array< T, ND > > np::gradient (
              boost::multi_array< T, ND > inArray,
              std::initializer_list< T > 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.

```
    {
        // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions of the array");
        using arrayIndex = boost::multi_array<T, ND>::index;

        using ndIndexArray = boost::array<arrayIndex, ND>;

        // constexpr std::size_t n = sizeof...(Args);
        std::size_t n = args.size();
        // std::tuple<Args...> store(args...);
        std::vector<T> arg_vector = args;
        boost::multi_array<T, ND> my_array;
        std::vector<boost::multi_array<T, ND» output_arrays;
        for (std::size_t i = 0; i < n; i++)
        {
            boost::multi_array<T, ND> dfdh = inArray;
            output_arrays.push_back(dfdh);
        }

        ndArrayValue *p = inArray.data();
        ndIndexArray index;
        for (std::size_t i = 0; i < inArray.num_elements(); i++)
        {
            index = getIndexArray(inArray, p);
            /*
            std::cout « "Index: ";
            for (std::size_t j = 0; j < n; j++)
            {
                std::cout « index[j] « " ";
            }
            std::cout « "\n";
            */
            // Calculating the gradient now
            // j is the axis/dimension
            for (std::size_t j = 0; j < n; j++)
            {
                ndIndexArray index_high = index;
                T dh_high;
                if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
                {
                    index_high[j] += 1;
                    dh_high = arg_vector[j];
                }
                else
                {
                    dh_high = 0;
                }
                ndIndexArray index_low = index;
                T dh_low;
                if (index_low[j] > 0)
                {
                    index_low[j] -= 1;
                    dh_low = arg_vector[j];
                }
                else
                {
                    dh_low = 0;
                }

                T dh = dh_high + dh_low;
                T gradient = (inArray(index_high) - inArray(index_low)) / dh;
                // std::cout « gradient « "\n";
                output_arrays[j](index) = gradient;
            }
            // std::cout « " value = " « inArray(index) « "  check = " « *p « std::endl;
            ++p;
        }
        return output_arrays;
    }
```

7.1.4.13 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.

```
    {
        double step = (stop - start) / (num - 1);
        boost::multi_array<double, 1> output(boost::extents[num]);
        for (std::size_t i = 0; i < num; i++)
        {
            output[i] = start + i * step;
        }
        return output;
    }
```

7.1.4.14 log() [1/2]

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

Implements the numpy log function on multi arrays.

Definition at line 304 of file np.hpp.

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

7.1.4.15 log() [2/2]

```
template<class T >
requires std::is_arithmetic<T>
::value constexpr T np::log (
            const T input )  [inline], [constexpr]
```

Implements the numpy log function on scalars.

Definition at line 312 of file np.hpp.

```
    {
        return std::log(input);
    }
```

7.1.4.16 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 384 of file np.hpp.

```
    {
        T max = 0;
        bool max_not_set = true;
        const T *data_pointer = input_array.data();
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            T element = *data_pointer;
            if (max_not_set || element > max)
            {
                max = element;
                max_not_set = false;
            }
            ++data_pointer;
        }
        return max;
    }
```

7.1.4.17 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 404 of file np.hpp.

```
    {
        T max = input1;
        for (T input : {inputs...})
        {
            if (input > max)
            {
                max = input;
            }
        }
        return max;
    }
```

7.1.4.18 meshgrid()

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

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.

```
    {
        using arrayIndex = boost::multi_array<T, ND>::index;
        using oneDArrayIndex = boost::multi_array<T, 1>::index;
        using ndIndexArray = boost::array<arrayIndex, ND>;
        std::vector<boost::multi_array<T, ND» output_arrays;
        boost::multi_array<T, 1> ci[ND];
        // Copy elements of cinput to ci, do the proper inversions
        for (std::size_t i = 0; i < ND; i++)
        {
            std::size_t source = i;
            if (indexing_type == xy && (ND == 3 || ND == 2))
            {
                switch (i)
                {
                case 0:
                    source = 1;
                    break;
                case 1:
                    source = 0;
                    break;
                default:
                    break;
                }
            }
            ci[i] = boost::multi_array<T, 1>();
            ci[i].resize(boost::extents[cinput[source].num_elements()]);
            ci[i] = cinput[source];
        }
        // Deducing the extents of the N-Dimensional output
        boost::detail::multi_array::extent_gen<ND> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 0; i < ND; i++)
        {
            shape_list.push_back(ci[i].shape()[0]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());

        // Creating the output arrays
        for (std::size_t i = 0; i < ND; i++)
        {
            boost::multi_array<T, ND> output_array(output_extents);
            ndArrayValue *p = output_array.data();
            ndIndexArray index;
            // Looping through the elements of the output array
            for (std::size_t j = 0; j < output_array.num_elements(); j++)
            {
                index = getIndexArray(output_array, p);
                oneDArrayIndex index_1d;
                index_1d = index[i];
                output_array(index) = ci[i][index_1d];
                ++p;
            }
            output_arrays.push_back(output_array);
        }
        return output_arrays;
    }
```

7.1.4.19 min() [1/2]

```
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 419 of file np.hpp.

```
    {
        T min = 0;
        bool min_not_set = true;
        const T *data_pointer = input_array.data();
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            T element = *data_pointer;
            if (min_not_set || element < min)
            {
                min = element;
                min_not_set = false;
            }
            ++data_pointer;
        }
        return min;
    }
```

7.1.4.20 min() [2/2]

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

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

Definition at line 439 of file np.hpp.

```
    {
        T min = input1;
        for (T input : {inputs...})
        {
            if (input < min)
            {
                min = input;
            }
        }
        return min;
    }

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND>
 abs(boost::multi_array<T, ND> const &input_array)
    {
        std::function<T(T)> abs_func = [](T input)
        { return std::abs(input); };
        return element_wise_apply(input_array, abs_func);
    }
```

7.1.4.21 pow() [1/2]

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

Implements the numpy pow function on multi arrays.

Definition at line 319 of file np.hpp.

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

7.1.4.22 pow() [2/2]

```
template<class T >
requires std::is_arithmetic<T>
::value constexpr T np::pow (
            const T input,
            const T exponent )  [inline], [constexpr]
```

Implements the numpy pow function on scalars.

Definition at line 328 of file np.hpp.

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

7.1.4.23 slice()

```
template<typename T , long unsigned int ND>
requires std::is_arithmetic<T>
::value constexpr boost::multi_array< T, ND - 1 > np::slice (
            boost::multi_array< T, ND > const & input_array,
            std::size_t slice_index )  [inline], [constexpr]
```

Slices the array through one dimension and returns a ND - 1 dimensional array.

Definition at line 470 of file np.hpp.

```
    {

        // Deducing the extents of the N-Dimensional output
        boost::detail::multi_array::extent_gen<ND - 1> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 1; i < ND; i++)
        {
            shape_list.push_back(input_array.shape()[i]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());

        boost::multi_array<T, ND - 1> output_array(output_extents);

        const T *p = input_array.data();
        boost::array<std::size_t, ND> index;
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            index = getIndexArray(input_array, p);
            output_array(index) = input_array[slice_index](index);
            p++;
        }
        return output_array;
    }
```

7.1.4.24 sqrt() [1/2]

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

Implements the numpy sqrt function on multi arrays.

Definition at line 274 of file np.hpp.

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

7.1.4.25 sqrt() [2/2]

```
template<class T >
requires std::is_arithmetic<T>
::value constexpr T np::sqrt (
            const T input )  [inline], [constexpr]
```

Implements the numpy sqrt function on scalars.

Definition at line 282 of file np.hpp.

```
{
    return std::sqrt(input);
}
```

7.1.4.26 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 365 of file np.hpp.

```
{
    // Deducing the extents of the N-Dimensional output
    boost::detail::multi_array::extent_gen<ND> output_extents;
    std::vector<size_t> shape_list;
    for (std::size_t i = 0; i < ND; i++)
    {
        shape_list.push_back(dimensions_input[i]);
    }
    std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
    // Applying a function to return zero always to all of its elements
    boost::multi_array<T, ND> output_array(output_extents);
    std::function<T(T)> zero_func = [](T input)
    { return 0; };
    return element_wise_apply(output_array, zero_func);
}
```

Chapter 8

File Documentation

8.1 coeff.hpp

```
//
// Created by Yan Cheng on 11/28/22.
//

#ifndef WAVESIMC_COEFF_HPP
#define WAVESIMC_COEFF_HPP

#include "CustomLibraries/np.hpp"
#include <math.h>


boost::multi_array<double, 2> get_sigma_1(boost::multi_array<double, 1> x, double dx, int nx, int nz,
                                          double c_max, int n=10, double R=1e-3, double m=2.0)
{
    boost::multi_array<double, 2> sigma_1(boost::extents[nx][nz]);
    const double PML_width = n * dx;

    const double sigma_max = - c_max * log(R) * (m+1) / np::pow(PML_width, (double) m+1);

    const double x_0 = np::max(x) - PML_width;
    boost::multi_array<double, 1> polynomial(boost::extents[nx]);

    for (int i=0; i < nx; i++)
    {
        if (x[i] > x_0)
        {
            polynomial[i] = np::pow(sigma_max * np::abs(x[i] - x_0), (double) m);
            polynomial[nx-i] = polynomial[i];
        }
        else
        {
            polynomial[i] = 0;
        }
    }
    // Copy 1D array into each column of 2D array
    for (int i=0; i<nx; i++)
        for (int j=0; j<nz; j++)
            sigma_1[i][j] = polynomial[i];

    return sigma_1;
}



boost::multi_array<double, 2> get_sigma_2(boost::multi_array<double, 1> z, double dz, int nx, int nz,
                                          double c_max, int n=10, double R=1e-3, double m=2.0)
{
    boost::multi_array<double, 2> sigma_2(boost::extents[nx][nz]);
    const double PML_width = n * dz;
    const double sigma_max = - c_max * log(R) * (m+1) / np::pow(PML_width, (double) m+1);

    const double z_0 = np::max(z) - PML_width;
    std::cout « z_0 ;

    boost::multi_array<double, 1> polynomial(boost::extents[nz]);
    for (int j=0; j<nz; j++)
    {
        if (z[j] > z_0)
        {
            // TODO: Does math.h have an absolute value function?
            polynomial[j] = np::pow(sigma_max * np::abs(z[j] - z_0), (double) m);
            polynomial[nz-j] = polynomial[j];
        }
        else
        {
            polynomial[j] = 0;
        }
    }

    // Copy 1D array into each column of 2D array
    for (int i=0; i<nx; i++)
        for (int j=0; j<nz; j++)
            sigma_2[i][j] = polynomial[j];

    return sigma_2;
}

#endif //WAVESIMC_COEFF_HPP
```

8.2 computational.hpp

```
//
// Created by Yan Cheng on 11/28/22.
//

#ifndef WAVESIMC_COMPUTATIONAL_HPP
#define WAVESIMC_COMPUTATIONAL_HPP

boost::multi_array<double, 2> get_profile(double xmin, double xmax, double zmin, double zmax, int nx,
 int nz)
{
    boost::multi_array<double, 2> c(boost::extents[nx][nz]);

    boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
    boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);

    const boost::multi_array<double, 1> axis[2] = {x, z};
    std::vector<boost::multi_array<double, 2» XZ = np::meshgrid(axis, false, np::xy);

    double x_0 = xmax / 2.0;
    double z_0 = zmax / 2.0;
    double r = 0.2;

    for (int i = 0; i < nx; i++)
    {
        for (int j = 0; j < nz; j++)
        {
            if (np::pow(XZ[0][i][j]-x_0, 2.0) + np::pow(XZ[1][i][j]-z_0, 2.0) <= np::pow(r, 2.0))
                c[i][j] = 3000;
            else
                c[i][j] = 2500;
        }
    }

    return c;
}

#endif //WAVESIMC_COMPUTATIONAL_HPP
```

8.3 helper_func.hpp

```
//
// Created by Yan Cheng on 11/28/22.
//

#ifndef WAVESIMC_HELPER_FUNC_HPP
#define WAVESIMC_HELPER_FUNC_HPP

#include "CustomLibraries/np.hpp"

boost::multi_array<double, 2> dfdx(boost::multi_array<double, 2> f, double dx)
{
    std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dx, dx});
    return grad_f[0];
}

boost::multi_array<double, 2> dfdz(boost::multi_array<double, 2> f, double dz)
```

```
{
    std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dz, dz});
    return grad_f[1];
}

boost::multi_array<double, 2> d2fdx2(boost::multi_array<double, 2> f, double dx)
{
    boost::multi_array<double, 2> df = dfdx(f, dx);
    boost::multi_array<double, 2> df2 = dfdx(df, dx);
    return df2;
}

boost::multi_array<double, 2> d2fdz2(boost::multi_array<double, 2> f, double dz)
{
    boost::multi_array<double, 2> df = dfdz(f, dz);
    boost::multi_array<double, 2> df2 = dfdz(df, dz);
    return df2;
}

boost::multi_array<double, 2> divergence(boost::multi_array<double, 2> f1, boost::multi_array<double,
 2> f2,
                                         double dx, double dz)
{
    boost::multi_array<double, 2> f_x = dfdx(f1, dx);
    boost::multi_array<double, 2> f_z = dfdz(f2, dz);
    // TODO: use element-wize add
    boost::multi_array<double, 2> div = f_x + f_z;
    return div;
}


#endif //WAVESIMC_HELPER_FUNC_HPP
```

8.4 solver.hpp

```
//
// Created by Yan Cheng on 11/28/22.
//

#ifndef WAVESIMC_SOLVER_HPP
#define WAVESIMC_SOLVER_HPP

#include "CustomLibraries/np.hpp"
#include "helper_func.hpp"

boost::multi_array<double, 3> wave_solver(boost::multi_array<double, 2> c,
                                          double dt, double dx, double dz, int nt, int nx, int nz,
                                          boost::multi_array<double, 3> f,
                                          boost::multi_array<double, 2> sigma_1,
 boost::multi_array<double, 2> sigma_2)
{
    // TODO: "same shape" functionality of np::zeros
    boost::multi_array<double, 3> u(boost::extents[nt][nx][nz]);
    boost::multi_array<double, 2> u_xx(boost::extents[nx][nz]);
    boost::multi_array<double, 2> u_zz(boost::extents[nx][nz]);
    boost::multi_array<double, 2> q_1(boost::extents[nx][nz]);
    boost::multi_array<double, 2> q_2(boost::extents[nx][nz]);

    const boost::multi_array<double, 2> C1 = 1.0 + dt * (sigma_1 + sigma_2) / 2.0;
    const boost::multi_array<double, 2> C2 = sigma_1 * sigma_2 * np::pow(dt, 2.0) - 2.0;
    const boost::multi_array<double, 2> C3 = 1.0 - dt * (sigma_1 + sigma_2) / 2.0;
    const boost::multi_array<double, 2> C4 = np::pow(dt * c, 2.0);
    const boost::multi_array<double, 2> C5 = 1.0 + dt * sigma_1 / 2.0;
    const boost::multi_array<double, 2> C6 = 1.0 + dt * sigma_2 / 2.0;
    const boost::multi_array<double, 2> C7 = 1.0 - dt * sigma_1 / 2.0;
    const boost::multi_array<double, 2> C8 = 1.0 - dt * sigma_2 / 2.0;

    for (int n = 0; n < nt; n++)
    {
        u_xx = d2fdx2(u[n], dx);
        u_zz = d2fdz2(u[n], dz);

        u[n+1] = (C4 * ((u_xx / np::pow(dx, 2.0)) + (u_zz / np::pow(dz, 2.0))
                    - divergence(q_1 * sigma_1, q_2 * sigma_2, dx, dz)
                    + (sigma_2 * dfdx(q_1, dx)) + (sigma_1 * dfdz(q_2, dz) + f[n]))
                    - (C2 * u[n]) - (C3 * u[n-1])) / C1;

        q_1 = (dt * dfdx(u[n], dx) + C7 * q_1) / C5;
        q_2 = (dt * dfdz(u[n], dx) + C8 * q_2) / C6;

        // Dirichlet boundary condition
        for (int i = 0; i < nx; i++)
        {
            u[n+1][i][0] = 0;
            u[n+1][i][nx-1] = 0;
        }
        for (int j = 0; j < nz; j++)
        {
            u[n+1][0][j] = 0;
            u[n+1][nz-1][j] = 0;
        }
    }
    return u;
}

#endif //WAVESIMC_SOLVER_HPP
```

8.5 source.hpp

```
//
// Created by Yan Cheng on 11/28/22.
//

#ifndef WAVESIMC_SOURCE_HPP
#define WAVESIMC_SOURCE_HPP


boost::multi_array<double, 3> ricker(int i_s, int j_s, double f, double amp, double shift,
                                     double tmin, double tmax, int nt, int nx, int nz)
{
    const double pi = 3.141592654;

    boost::multi_array<double, 1> t = np::linspace(tmin, tmax, nt);
    boost::multi_array<double, 1> pft2 = np::pow(pi * f * (t - shift), 2.0);
    boost::multi_array<double, 1> r = amp * (1.0 - 2.0 * pft2) * np::exp(-1.0 * pft2);

    int dimensions_x[] = {nx};
    boost::multi_array<double, 1> x = np::zeros<double>(dimensions_x);

    int dimensions_z[] = {nz};
    boost::multi_array<double, 1> z = np::zeros<double>(dimensions_z);

    x[i_s] = 1.0;
    z[j_s] = 1.0;

    const boost::multi_array<double, 1> axis[3] = {r, x, z};
    std::vector<boost::multi_array<double, 3» RXZ = np::meshgrid(axis, false, np::xy);

    boost::multi_array<double, 3> source = RXZ[0] * RXZ[1] * RXZ[2];

    return source;
}

#endif //WAVESIMC_SOURCE_HPP
```

8.6 wave.cpp

```
// For the core algorithm, we need six functionalities:
// 1) create the computational domain,
// 2) create a velocity profile (1 & 2 can be put together)
// 3) create attenuation coefficients,
// 4) create source functions,
// 5) helper functions to compute eg. df/dx
// 6) use all above to create a solver function for wave equation

// Standard IO libraries
#include <iostream>
#include <fstream>
#include "CustomLibraries/np.hpp"

#include <math.h>

#include "solver.hpp"
#include "computational.hpp"
#include "coeff.hpp"
#include "source.hpp"
#include "helper_func.hpp"


int main()
{
    double dx, dy, dz, dt;
    dx = 1.0;
    dy = 1.0;
    dz = 1.0;
    dt = 1.0;
    std::vector<boost::multi_array<double, 4» my_arrays = np::gradient(A, {dx, dy, dz, dt});
    return 0;
}
```

8.7 np.hpp

```
#ifndef NP_H_
#define NP_H_

#include "boost/multi_array.hpp"
#include "boost/array.hpp"
#include "boost/cstdlib.hpp"
#include <type_traits>
#include <cassert>
#include <iostream>
#include <functional>
#include <type_traits>

namespace np
{

    typedef double ndArrayValue;

    template <std::size_t ND>
    inline boost::multi_array<ndArrayValue, ND>::index
    getIndex(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement, const unsigned short int direction)
    {
        int offset = requestedElement - m.origin();
        return (offset / m.strides()[direction] % m.shape()[direction] + m.index_bases()[direction]);
    }

    template <std::size_t ND>
    inline boost::array<typename boost::multi_array<ndArrayValue, ND>::index, ND>
    getIndexArray(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement)
    {
        using indexType = boost::multi_array<ndArrayValue, ND>::index;
        boost::array<indexType, ND> _index;
        for (unsigned int dir = 0; dir < ND; dir++)
        {
            _index[dir] = getIndex(m, requestedElement, dir);
        }

        return _index;
    }

    template <typename Array, typename Element, typename Functor>
    inline void for_each(const boost::type<Element> &type_dispatch,
                         Array A, Functor &xform)
    {
        for_each(type_dispatch, A.begin(), A.end(), xform);
    }

    template <typename Element, typename Functor>
    inline void for_each(const boost::type<Element> &, Element &Val, Functor &xform)
    {
        Val = xform(Val);
    }

    template <typename Element, typename Iterator, typename Functor>
    inline void for_each(const boost::type<Element> &type_dispatch,
                         Iterator begin, Iterator end,
                         Functor &xform)
    {
        while (begin != end)
        {
            for_each(type_dispatch, *begin, xform);
            ++begin;
        }
    }

    template <typename Array, typename Functor>
    inline void for_each(Array &A, Functor xform)
    {
        // Dispatch to the proper function
        for_each(boost::type<typename Array::element>(), A.begin(), A.end(), xform);
    }

    template <typename T, long unsigned int ND>
    requires std::is_floating_point<T>::value inline constexpr std::vector<boost::multi_array<T, ND>> gradient(boost::multi_array<T, ND> inArray, std::initializer_list<T> args)
    {
        // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions of the array");
        using arrayIndex = boost::multi_array<T, ND>::index;

        using ndIndexArray = boost::array<arrayIndex, ND>;

        // constexpr std::size_t n = sizeof...(Args);
        std::size_t n = args.size();
        // std::tuple<Args...> store(args...);
        std::vector<T> arg_vector = args;
        boost::multi_array<T, ND> my_array;
        std::vector<boost::multi_array<T, ND>> output_arrays;
        for (std::size_t i = 0; i < n; i++)
        {
            boost::multi_array<T, ND> dfdh = inArray;
            output_arrays.push_back(dfdh);
        }

        ndArrayValue *p = inArray.data();
        ndIndexArray index;
        for (std::size_t i = 0; i < inArray.num_elements(); i++)
        {
            index = getIndexArray(inArray, p);
            /*
            std::cout << "Index: ";
            for (std::size_t j = 0; j < n; j++)
            {
                std::cout << index[j] << " ";
            }
            std::cout << "\n";
            */
            // Calculating the gradient now
            // j is the axis/dimension
            for (std::size_t j = 0; j < n; j++)
            {
                ndIndexArray index_high = index;
                T dh_high;
                if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
                {
                    index_high[j] += 1;
                    dh_high = arg_vector[j];
                }
                else
                {
                    dh_high = 0;
                }
                ndIndexArray index_low = index;
                T dh_low;
                if (index_low[j] > 0)
                {
                    index_low[j] -= 1;
                    dh_low = arg_vector[j];
                }
                else
                {
                    dh_low = 0;
                }

                T dh = dh_high + dh_low;
                T gradient = (inArray(index_high) - inArray(index_low)) / dh;
                // std::cout << gradient << "\n";
                output_arrays[j](index) = gradient;
            }
            // std::cout << " value = " << inArray(index) << "  check = " << *p << std::endl;
            ++p;
        }
        return output_arrays;
    }

    inline boost::multi_array<double, 1> linspace(double start, double stop, long unsigned int num)
    {
        double step = (stop - start) / (num - 1);
        boost::multi_array<double, 1> output(boost::extents[num]);
        for (std::size_t i = 0; i < num; i++)
        {
            output[i] = start + i * step;
        }
        return output;
    }

    enum indexing
    {
        xy,
        ij
    };

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr std::vector<boost::multi_array<T, ND»
meshgrid(const boost::multi_array<T, 1> (&cinput)[ND], bool sparsing = false, indexing indexing_type
= xy)
    {
        using arrayIndex = boost::multi_array<T, ND>::index;
        using oneDArrayIndex = boost::multi_array<T, 1>::index;
        using ndIndexArray = boost::array<arrayIndex, ND>;
        std::vector<boost::multi_array<T, ND» output_arrays;
        boost::multi_array<T, 1> ci[ND];
        // Copy elements of cinput to ci, do the proper inversions
        for (std::size_t i = 0; i < ND; i++)
        {
            std::size_t source = i;
            if (indexing_type == xy && (ND == 3 || ND == 2))
            {
                switch (i)
                {
                case 0:
                    source = 1;
                    break;
                case 1:
                    source = 0;
                    break;
                default:
                    break;
                }
            }
            ci[i] = boost::multi_array<T, 1>();
            ci[i].resize(boost::extents[cinput[source].num_elements()]);
            ci[i] = cinput[source];
        }
        // Deducing the extents of the N-Dimensional output
        boost::detail::multi_array::extent_gen<ND> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 0; i < ND; i++)
        {
            shape_list.push_back(ci[i].shape()[0]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());

        // Creating the output arrays
        for (std::size_t i = 0; i < ND; i++)
        {
            boost::multi_array<T, ND> output_array(output_extents);
            ndArrayValue *p = output_array.data();
            ndIndexArray index;
            // Looping through the elements of the output array
            for (std::size_t j = 0; j < output_array.num_elements(); j++)
            {
                index = getIndexArray(output_array, p);
                oneDArrayIndex index_1d;
                index_1d = index[i];
                output_array(index) = ci[i][index_1d];
                ++p;
            }
            output_arrays.push_back(output_array);
        }
        return output_arrays;
    }

    template <class T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND>
element_wise_apply(const boost::multi_array<T, ND> &input_array, std::function<T(T)> func)
    {

        // Create output array copying extents
        using arrayIndex = boost::multi_array<double, ND>::index;
        using ndIndexArray = boost::array<arrayIndex, ND>;
        boost::detail::multi_array::extent_gen<ND> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 0; i < ND; i++)
        {
            shape_list.push_back(input_array.shape()[i]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
        boost::multi_array<T, ND> output_array(output_extents);

        // Looping through the elements of the output array
        const T *p = input_array.data();
        ndIndexArray index;
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            index = getIndexArray(input_array, p);
            output_array(index) = func(input_array(index));
            ++p;
        }
        return output_array;
    }

    // Complex operations

    template <class T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> sqrt(const
 boost::multi_array<T, ND> &input_array)
    {
        std::function<T(T)> func = (T(*)(T))std::sqrt;
        return element_wise_apply(input_array, func);
    }

    template <class T>
    requires std::is_arithmetic<T>::value inline constexpr T sqrt(const T input)
    {
        return std::sqrt(input);
    }

    template <class T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> exp(const
 boost::multi_array<T, ND> &input_array)
    {
        std::function<T(T)> func = (T(*)(T))std::exp;
        return element_wise_apply(input_array, func);
    }

    template <class T>
    requires std::is_arithmetic<T>::value inline constexpr T exp(const T input)
    {
        return std::exp(input);
    }

    template <class T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> log(const
 boost::multi_array<T, ND> &input_array)
    {
        std::function<T(T)> func = std::log<T>();
        return element_wise_apply(input_array, func);
    }

    template <class T>
    requires std::is_arithmetic<T>::value inline constexpr T log(const T input)
    {
        return std::log(input);
    }

    template <class T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> pow(const
 boost::multi_array<T, ND> &input_array, const T exponent)
    {
        std::function<T(T)> pow_func = [exponent](T input)
        { return std::pow(input, exponent); };
        return element_wise_apply(input_array, pow_func);
    }

    template <class T>
    requires std::is_arithmetic<T>::value inline constexpr T pow(const T input, const T exponent)
    {
        return std::pow(input, exponent);
    }

    template <class T, long unsigned int ND>
    inline constexpr 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)
    {
        // Create output array copying extents
        using arrayIndex = boost::multi_array<double, ND>::index;
        using ndIndexArray = boost::array<arrayIndex, ND>;
        boost::detail::multi_array::extent_gen<ND> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 0; i < ND; i++)
        {
            shape_list.push_back(lhs.shape()[i]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
        boost::multi_array<T, ND> output_array(output_extents);

        // Looping through the elements of the output array
        const T *p = lhs.data();
        ndIndexArray index;
        for (std::size_t i = 0; i < lhs.num_elements(); i++)
        {
            index = getIndexArray(lhs, p);
            output_array(index) = func(lhs(index), rhs(index));
            ++p;
        }
        return output_array;
    }

    template <typename T, typename inT, long unsigned int ND>
    requires std::is_integral<inT>::value && std::is_arithmetic<T>::value inline constexpr
 boost::multi_array<T, ND> zeros(inT (&dimensions_input)[ND])
    {
        // Deducing the extents of the N-Dimensional output
        boost::detail::multi_array::extent_gen<ND> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 0; i < ND; i++)
        {
            shape_list.push_back(dimensions_input[i]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
        // Applying a function to return zero always to all of its elements
        boost::multi_array<T, ND> output_array(output_extents);
        std::function<T(T)> zero_func = [](T input)
        { return 0; };
        return element_wise_apply(output_array, zero_func);
    }

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr T max(boost::multi_array<T, ND> const
 &input_array)
    {
        T max = 0;
        bool max_not_set = true;
        const T *data_pointer = input_array.data();
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            T element = *data_pointer;
            if (max_not_set || element > max)
            {
                max = element;
                max_not_set = false;
            }
            ++data_pointer;
        }
        return max;
    }

    template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)»
    requires std::is_arithmetic<T>::value inline constexpr T max(T input1, Ts... inputs)
    {
        T max = input1;
        for (T input : {inputs...})
        {
            if (input > max)
            {
                max = input;
            }
        }
        return max;
    }

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr T min(boost::multi_array<T, ND> const
 &input_array)
    {
        T min = 0;
        bool min_not_set = true;
        const T *data_pointer = input_array.data();
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            T element = *data_pointer;
            if (min_not_set || element < min)
            {
                min = element;
                min_not_set = false;
            }
            ++data_pointer;
        }
        return min;
    }

    template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)»
    inline constexpr T min(T input1, Ts... inputs) requires std::is_arithmetic<T>::value
    {
        T min = input1;
        for (T input : {inputs...})
        {
            if (input < min)
            {
                min = input;
            }
        }
        return min;
    }

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> abs(boost::multi_array<T, ND> const &input_array)
    {
        std::function<T(T)> abs_func = [](T input)
        { return std::abs(input); };
        return element_wise_apply(input_array, abs_func);
    }

    template <typename T>
    requires std::is_arithmetic<T>::value inline constexpr T abs(T input)
    {
        return std::abs(input);
    }

    template <typename T, long unsigned int ND>
    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND - 1> slice(boost::multi_array<T, ND> const &input_array, std::size_t slice_index)
    {

        // Deducing the extents of the N-Dimensional output
        boost::detail::multi_array::extent_gen<ND - 1> output_extents;
        std::vector<size_t> shape_list;
        for (std::size_t i = 1; i < ND; i++)
        {
            shape_list.push_back(input_array.shape()[i]);
        }
        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());

        boost::multi_array<T, ND - 1> output_array(output_extents);

        const T *p = input_array.data();
        boost::array<std::size_t, ND> index;
        for (std::size_t i = 0; i < input_array.num_elements(); i++)
        {
            index = getIndexArray(input_array, p);
            output_array(index) = input_array[slice_index](index);
            p++;
        }
        return output_array;
    }

}

// Override of operators in the boost::multi_array class to make them more np-like
// Basic operators
// All of the are element-wise

// Multiplication operator
template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND> const &rhs)
{
    std::function<T(T, T)> func = std::multiplies<T>();
    return np::element_wise_duo_apply(lhs, rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator*(T const &lhs, boost::multi_array<T, ND> const &rhs)
{
    std::function<T(T)> func = [lhs](T item)
    { return lhs * item; };
    return np::element_wise_apply(rhs, func);
}
template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, T const &rhs)
{
    return rhs * lhs;
}

// Plus operator
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)
{
    std::function<T(T, T)> func = std::plus<T>();
    return np::element_wise_duo_apply(lhs, rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator+(T const &lhs, boost::multi_array<T, ND> const &rhs)
{
    std::function<T(T)> func = [lhs](T item)
    { return lhs + item; };
    return np::element_wise_apply(rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, T const &rhs)
{
    return rhs + lhs;
}

// Subtraction operator
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)
{
    std::function<T(T, T)> func = std::minus<T>();
    return np::element_wise_duo_apply(lhs, rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator-(T const &lhs, boost::multi_array<T, ND> const &rhs)
{
    std::function<T(T)> func = [lhs](T item)
    { return lhs - item; };
    return np::element_wise_apply(rhs, func);
}

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

// Division operator
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)
{
    std::function<T(T, T)> func = std::divides<T>();
    return np::element_wise_duo_apply(lhs, rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator/(T const &lhs, boost::multi_array<T, ND> const &rhs)
{
    std::function<T(T)> func = [lhs](T item)
    { return lhs / item; };
    return np::element_wise_apply(rhs, func);
}

template <class T, long unsigned int ND>
inline boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, T const &rhs)
{
    return rhs / lhs;
}

#endif
```

8.8 main.cpp

```
#include <iostream>
#include <string>
#include "ExternalLibraries/cxxopts.hpp"
#include "CustomLibraries/np.hpp"

// Command line arguments
cxxopts::Options options("WaveSimC", "A wave propagation simulator written in C++ for seismic data processing.");
int main(int argc, char *argv[])
{
    // Parse command line arguments
    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"));
    auto result = options.parse(argc, argv);

    std::cout « "Hello World"
              « "\n";
}
```

8.9 CoreTests.cpp

```
//
// Created by Yan Cheng on 12/2/22.
//

#include <boost/multi_array.hpp>
#include <boost/array.hpp>
#include "CustomLibraries/np.hpp"
#include <cassert>
#include <iostream>

#include "CoreAlgorithm/helper_func.hpp"
#include "CoreAlgorithm/coeff.hpp"
#include "CoreAlgorithm/source.hpp"
#include "CoreAlgorithm/computational.hpp"
//#include "CoreAlgorithm/solver.hpp"

void test_(){
    boost::multi_array<double, 2> sigma_1 = get_sigma_1(np::linspace(0.0, 1.0, 100), 1.0 / 100.0, 100, 100, 3000.0);

        int nx = 100;
        int nz  = 100;
    for (int i = 0; i < nx; i++)
    {
        for (int j = 0; j < nz; j++)
            std::cout « sigma_1[i][j] « " ";
        std::cout « "\n";
    }
}

int main(){
    test_();
}
```

8.10 variadic.cpp

```
#include "boost/multi_array.hpp"
#include "boost/array.hpp"
#include "CustomLibraries/np.hpp"
#include <cassert>
#include <iostream>

void test_gradient()
{
    // Create a 4D array that is 3 x 4 x 2 x 1
    typedef boost::multi_array<double, 4>::index index;
    boost::multi_array<double, 4> A(boost::extents[3][4][2][2]);

    // Assign values to the elements
    int values = 0;
    for (index i = 0; i != 3; ++i)
        for (index j = 0; j != 4; ++j)
            for (index k = 0; k != 2; ++k)
                for (index l = 0; l != 2; ++l)
                    A[i][j][k][l] = values++;

    // Verify values
    int verify = 0;
    for (index i = 0; i != 3; ++i)
        for (index j = 0; j != 4; ++j)
            for (index k = 0; k != 2; ++k)
                for (index l = 0; l != 2; ++l)
                    assert(A[i][j][k][l] == verify++);

    double dx, dy, dz, dt;
    dx = 1.0;
    dy = 1.0;
    dz = 1.0;
    dt = 1.0;
    std::vector<boost::multi_array<double, 4» my_arrays = np::gradient(A, {dx, dy, dz, dt});

    boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
    std::vector<boost::multi_array<double, 1» gradf = np::gradient(x, {1.0});
    for (int i = 0; i < 5; i++)
    {
        std::cout « gradf[0][i] « ",";
    }
    std::cout « "\n";
    // np::print(std::cout, my_arrays[0]);
}

```

```
void test_meshgrid()
{
    boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> y = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> z = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> t = np::linspace(0, 1, 5);
    const boost::multi_array<double, 1> axis[4] = {x, y, z, t};
    std::vector<boost::multi_array<double, 4» my_arrays = np::meshgrid(axis, false, np::xy);
    // np::print(std::cout, my_arrays[0]);
    int nx = 3;
    int ny = 2;
    boost::multi_array<double, 1> x2 = np::linspace(0, 1, nx);
    boost::multi_array<double, 1> y2 = np::linspace(0, 1, ny);
    const boost::multi_array<double, 1> axis2[2] = {x2, y2};
    std::vector<boost::multi_array<double, 2» my_arrays2 = np::meshgrid(axis2, false, np::xy);
    std::cout « "xv\n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « my_arrays2[0][i][j] « " ";
        }
        std::cout « "\n";
    }
    std::cout « "yv\n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « my_arrays2[1][i][j] « " ";
        }
        std::cout « "\n";
    }
}

void test_complex_operations()
{
    int nx = 3;
    int ny = 2;
    boost::multi_array<double, 1> x = np::linspace(0, 1, nx);
    boost::multi_array<double, 1> y = np::linspace(0, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» my_arrays = np::meshgrid(axis, false, np::xy);
    boost::multi_array<double, 2> A = np::sqrt(my_arrays[0]);
    std::cout « "sqrt\n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « A[i][j] « " ";
        }
        std::cout « "\n";
    }
    std::cout « "\n";
    float a = 100.0;
    float sqa = np::sqrt(a);
    std::cout « "sqrt of " « a « " is " « sqa « "\n";
    std::cout « "exp\n";
    boost::multi_array<double, 2> B = np::exp(my_arrays[0]);
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « B[i][j] « " ";
        }
        std::cout « "\n";
    }

    std::cout « "Power\n";
    boost::multi_array<double, 1> x2 = np::linspace(1, 3, nx);
    boost::multi_array<double, 1> y2 = np::linspace(1, 3, ny);
    const boost::multi_array<double, 1> axis2[2] = {x2, y2};
    std::vector<boost::multi_array<double, 2» my_arrays2 = np::meshgrid(axis2, false, np::xy);
    boost::multi_array<double, 2> C = np::pow(my_arrays2[1], 2.0);
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « C[i][j] « " ";
        }
        std::cout « "\n";
    }
}

void test_equal()
{
    boost::multi_array<double, 1> x = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> y = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> z = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> t = np::linspace(0, 1, 5);
    const boost::multi_array<double, 1> axis[4] = {x, y, z, t};
    std::vector<boost::multi_array<double, 4» my_arrays = np::meshgrid(axis, false, np::xy);
    boost::multi_array<double, 1> x2 = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> y2 = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> z2 = np::linspace(0, 1, 5);
    boost::multi_array<double, 1> t2 = np::linspace(0, 1, 5);
    const boost::multi_array<double, 1> axis2[4] = {x2, y2, z2, t2};
    std::vector<boost::multi_array<double, 4» my_arrays2 = np::meshgrid(axis2, false, np::xy);
    std::cout « "equality test:\n";
    std::cout « (bool)(my_arrays == my_arrays2) « "\n";
}
void test_basic_operations()
{
    int nx = 3;
    int ny = 2;
    boost::multi_array<double, 1> x = np::linspace(0, 1, nx);
    boost::multi_array<double, 1> y = np::linspace(0, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» my_arrays = np::meshgrid(axis, false, np::xy);

    std::cout « "basic operations:\n";

    std::cout « "addition:\n";
    boost::multi_array<double, 2> A = my_arrays[0] + my_arrays[1];

    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « A[i][j] « " ";
        }
        std::cout « "\n";
    }

    std::cout « "multiplication:\n";
    boost::multi_array<double, 2> B = my_arrays[0] * my_arrays[1];

    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « B[i][j] « " ";
        }
        std::cout « "\n";
    }
    double coeff = 3;
    boost::multi_array<double, 1> t = np::linspace(0, 1, nx);
    boost::multi_array<double, 1> t_time_3 = coeff * t;
    boost::multi_array<double, 1> t_time_2 = 2.0 * t;
    std::cout « "t_time_3: ";
    for (int j = 0; j < nx; j++)
    {
        std::cout « t_time_3[j] « " ";
    }
    std::cout « "\n";
    std::cout « "t_time_2: ";
    for (int j = 0; j < nx; j++)
    {
        std::cout « t_time_2[j] « " ";
    }
    std::cout « "\n";
}

void test_zeros()
{
    int nx = 3;
    int ny = 2;
    int dimensions[] = {ny, nx};
    boost::multi_array<double, 2> A = np::zeros<double>(dimensions);
    std::cout « "zeros:\n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « A[i][j] « " ";
        }
        std::cout « "\n";
    }
}

void test_min_max()
{
    int nx = 24;
    int ny = 5;
    boost::multi_array<double, 1> x = np::linspace(0, 10, nx);
    boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» my_array = np::meshgrid(axis, false, np::xy);
    std::cout « "min: " « np::min(my_array[0]) « "\n";
    std::cout « "max: " « np::max(my_array[1]) « "\n";
    std::cout « "max simple: " « np::max(1.0, 2.0, 3.0, 4.0, 5.0) « "\n";
    std::cout « "min simple: " « np::min(1, -2, 3, -4, 5) « "\n";
}

void test_toy_problem()
{
    boost::multi_array<double, 1> x = np::linspace(0, 1, 100);
    boost::multi_array<double, 1> y = np::linspace(0, 1, 100);
    // x = np::pow(x, 2.0);
    // y = np::pow(y, 3.0);

    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);

    double dx, dy;
    dx = 1.0 / 100.0;
    dy = 1.0 / 100.0;

    boost::multi_array<double, 2> f = np::pow(XcY[0], 2.0) + XcY[0] * np::pow(XcY[1], 1.0);

    // g.push_back(np::gradient(XcY[0], {dx, dy}));
    // g.push_back(np::gradient(XcY[1], {dx, dy}));
    std::vector<boost::multi_array<double, 2» gradf = np::gradient(f, {dx, dy});
    // auto [gradfx_x, gradfx_y] = np::gradient(f, {dx, dy});

    int i, j;
    i = 10;
    j = 20;
    std::cout « "df/dx of f(x,y) = x^2 + xy at x = " « x[i] « " and y = " « y[j] « " is equal to " «
 gradf[0][i][j];

    std::cout « "\n";
}

void test_abs()
{
    int nx = 4;
    int ny = 4;
    boost::multi_array<double, 1> x = np::linspace(-1, 1, nx);
    boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
    boost::multi_array<double, 2> abs_f = np::abs(XcY[0]);
    std::cout « "abs_f: \n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « abs_f[i][j] « " ";
        }
        std::cout « "\n";
    }
}

void test_slice()
{
    int nx = 4;
    int ny = 4;
    boost::multi_array<double, 1> x = np::linspace(-1, 1, nx);
    boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
    boost::multi_array<double, 2> f = np::pow(XcY[0], 2.0) + XcY[0] * np::pow(XcY[1], 1.0);
    std::cout « "f: \n";
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « f[i][j] « " ";
        }
        std::cout « "\n";
    }
    std::cout « "f[0]: \n";
    boost::multi_array<double, 1> f_slice = np::slice(f, 0);
    for (int i = 0; i < nx; i++)
    {
        std::cout « f_slice[i] « " ";
    }
    std::cout « "\n";

    std::cout « "f[1]: \n";
    f_slice = np::slice(f, 1);
    for (int i = 0; i < ny; i++)
    {
        std::cout « f_slice[i] « " ";
    }
    std::cout « "\n";
}

int main()
{
    test_gradient();
    test_meshgrid();
    test_complex_operations();
    test_equal();
    test_basic_operations();
    test_zeros();
    test_min_max();
    test_abs();
    test_toy_problem();
    test_slice();
}
```