WaveSimPP

1.0

Chapter 1

Main Page

1.1 COMSW4995 Final Project: WaveSimPP

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

This proejct is licensed under the MIT License - see the LICENSE.md file for details

Chapter 2

Tutorial

2.1 Using the executable

If all you want is to run one simulation with custom parameters, you can use the executable that is generated when you build the project. The executable is located in the build folder.

To build the executable you can use the following commands:
```
mkdir build
cd build
cmake ..
make WaveSimPPExec
```

And then run it with the following command:
```
./WaveSimPPExec
```

The executable will ask you for the parameters of the simulation. A complete description of the parameters can be found in the the help menu of the executable. To see the help menu you can run the executable with the -h flag:
```
./WaveSimPPExec -h
```

2.2 Creating a wave simulation and solving it

The source code of this tutorial can be found in src/examples/wave_solver_with_animation.cpp

The first part of creating the wave simulation is to define the constants for the simulation. These constants are the number of grid points in the x and z directions, the number of time steps, the differentiation values, the domain, the source parameters, and the source location.

It is important to pay attention to the fact that the plane is XZ not XY due to geophysical conventions.
```
// Define the constants for the simulation
// Number of x and z grid points
int nx = 100;
int nz = 100;
// Number of time steps
int nt = 1000;
// Differentiation values
double dx = 0.01;
double dz = 0.01;
double dt = 0.001;
// Define the domain
double xmin = 0.0;
double xmax = nx * dx;
double zmin = 0.0;
double zmax = nz * dz;
double tmin = 0.0;
double tmax = nt * dt;
// Define the source parameters
```

```
double f_M = 10.0;
double amp = 1e0;
double shift = 0.1;
// Source location
int source_is = 50;
int source_js = 50;
```

The next step is to create the source object and the velocity profile
```
// Create the source
boost::multi_array<double, 3> f = waveSimPPCore::ricker(source_is, source_js, f_M, amp, shift, tmin, tmax,
    nt, nx, nz);
// Create the velocity profile
double r = 150.0;
boost::multi_array<double, 2> vel = waveSimPPCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
```

Then we can proceed to solve the wave equation using the wave solver.
```
// Solve the wave equation
boost::multi_array<double, 3> u = waveSimPPCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
```

u is the multi_array that contains the result of the simulation. It has the shape (nt, nx, nz). That means that the first index is the time step, the second index is the x grid point, and the third index is the z grid point.

You can access the result of a specific time step by simply using:
```
boost::multi_array<double, 2> u_at_time_20 = u[20];
```

2.3 Obtaining the results

To see the results you have two main options:

- Use the wavePlotter::Plotter class to plot the results and/or create an animation.

- Export all the frames to a series of csv files for processing in another program.

2.3.1 Plotting the results using the wavePlotter::Plotter class

You should do some light processing to conver the domain from a boost::multi_array to a matplot::vector_2d. This is done using the np::convert_to_matplot function.

After that you can convert the result of each frame to a matplot::vector_2d and plot it using the wavePlotter::Plotter class.
```
// Define the number of different levels for the contour plot
int num_levels = 100;
// Create the levels for the contour plot based on the min and max values of u
double min_u = np::min(u);
double max_u = np::max(u);
std::vector<double> levels = matplot::linspace(min_u, max_u, num_levels);
// Create the x and z axis for the contour plot and convert them to matplot format
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» XcZ = np::meshgrid(axis, false, np::xy);
matplot::vector_2d Xp = np::convert_to_matplot(XcZ[0]);
matplot::vector_2d Zp = np::convert_to_matplot(XcZ[1]);
```

From this point on you can plot it as you wish, for example, if you want a filled contour plot you can do:
```
matplot::vector_2d Up = np::convert_to_matplot(this->u[frame_index]);
matplot::contourf(this->Xp, this->Zp, Up, this->levels);
matplot::show();
```

Another option is to pass the data to the wavePlotter:Plotter class and use the plot function to render all frames
```
// Create the plotter object and animate the results
wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
// If you want to render a specific frame, use this:
// my_plotter.renderFrame(int frame_index);
// Renders the entire animation from start_frame to end_frame
int start_frame = 20;
int end_frame = nt - 1;
int fps = 30;
my_plotter.animate("example-wave.mp4", start_frame, end_frame, fps);
```

The animation will be saved in . and the frames will be saved to ./output

2.3.2 Exporting the results to csv files

You can export the results to a series of individual csv files using the wavePlotter::Plotter class.

```
// Create the plotter object and animate the results
wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
// If you want to export a specific frame, use this:
// my_plotter.exportFrame(int frame_index);
// Exports the entire simulation from start_frame to end_frame
int start_frame = 20;
int end_frame = nt - 1;
my_plotter.exportAllFrames(start_frame, end_frame);
```

The frames will be saved to ./output

Each frame is saved as a csv file with the following format:

```
data at 0_0, data at 0_1, data at 0_2, ..., data at 0_nz
data at 1_0, data at 1_1, data at 1_2, ..., data at 1_nz
data at 2_0, data at 2_1, data at 2_2, ..., data at 2_nz
.
.
.
```

If you want to import the data into a python program you can use the following code:

```
frames = []
for i in range(999):
    filename = "output/frame_" + f'{i:08}' +".csv"
    frames.append(pd.read_csv(filename))
```

Please refer to src/examples/PythonLoadingExample.ipynb for a complete example on how to load the data into a python program and render it using matplotlib (of course you can use any other library to render the data).

Chapter 3

README

3.1 COMSW4995 Final Project: WaveSimPP

Click Here to Access the Github Repo

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.

Detailed documentation

3.1.1 Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

3.1.2 Acknowledgments

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

3.2 Theory

3.2.1 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}
```

3.2.2 References

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

3.2.3 Design Philosophy

3.2.3.1 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.

3.2.4   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.

3.3   Building

Please be aware that since this library uses a few C++ 20 features it is only been tested on gcc-11 and above. It is possible that it will work on other compilers but it is not guaranteed.

3.3.1   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
```

3.3.2   Install Matplotplusplus

This is the library used to generate graphics in the project. To be able to compile this project you must have it installed in your system. First install its dependencies:
```
sudo apt-get install gnuplot
```

or in Mac:
```
brew install gnuplot
```

Then install the library itself by cloning from source:
```
cd src/ExternalLibraries
git clone https://github.com/alandefreitas/matplotplusplus
cd matplotplusplu
mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="-O2" -DBUILD_EXAMPLES=OFF -DBUILD_TESTS=OFF
sudo cmake --build . --parallel 2 --config Release
sudo cmake --install .
```

If you are using clang on mac, make sure to force CMAKE to use gcc by adding the following flag to the first cmake command:
```
-DCMAKE_C_COMPILER=/usr/bin/gcc -DCMAKE_CXX_COMPILER=/usr/bin/g++
```

(or equivalent paths depending on where your gcc is installed)

3.3.3 Building the project

```
mkdir build
cd build
cmake ..
make all
```

You can also build only the executable by running:
```
make WaveSimPPExec
```

3.3.4 Running the executable

```
./WaveSimPPExec
```

Use the help flag -h to see the available runtime options and the full list and description of the parameters.

3.3.5 Building the documentation

Docs building script:
```
./compileDocs.sh
```

Manually:
```
doxygen dconfig
cd documentation/latex
pdflatex refman.tex
cp refman.pdf ../WaveSimPP-1.0-doc.pdf
```

Chapter 4

Module Index

4.1 Modules

Here is a list of all modules:

Chapter 5

Namespace Index

5.1 Namespace List

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

Chapter 6

Class Index

6.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 7

File Index

7.1 File List

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

Chapter 8

Module Documentation

8.1 WaveSimCore

Namespaces

- namespace waveSimCore

Functions

- boost::multi_array< double, 2 > waveSimCore::get_sigma_1 (boost::multi_array< double, 1 > x, double dx, int nx, int nz, double c_max, int n=15, double R=1e-3, double m=2.0)
- boost::multi_array< double, 2 > waveSimCore::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 > waveSimCore::get_profile (double xmin, double xmax, double zmin, double zmax, int nx, int nz, double r)

  Get the velocity profile of the model as a 2D Array.
- boost::multi_array< double, 2 > waveSimCore::dfdx (boost::multi_array< double, 2 > f, double dx)

  Takes the partial derivative of a 2D matrix f with respect to x.
- boost::multi_array< double, 2 > waveSimCore::dfdz (boost::multi_array< double, 2 > f, double dz)

  Takes the partial derivative of a 2D matrix f with respect to z.
- boost::multi_array< double, 2 > waveSimCore::d2fdx2 (boost::multi_array< double, 2 > f, double dx)

  Takes the second partial derivative of a 2D matrix f with respect to x.
- boost::multi_array< double, 2 > waveSimCore::d2fdz2 (boost::multi_array< double, 2 > f, double dz)

  Takes the second partial derivative of a 2D matrix f with respect to z.
- boost::multi_array< double, 2 > waveSimCore::divergence (boost::multi_array< double, 2 > f1, boost↩
  ::multi_array< double, 2 > f2, double dx, double dz)
- boost::multi_array< double, 3 > waveSimCore::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, 3 > waveSimCore::wave_solver_complex (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)
- boost::multi_array< double, 3 > waveSimCore::ricker (int i_s, int j_s, double f, double amp, double shift, double tmin, double tmax, int nt, int nx, int nz)

  Get the Ricker wavelet as a 3D Array.

8.1.1 Detailed Description

8.1.2 Function Documentation

8.1.2.1 d2fdx2()

```
boost::multi_array< double, 2 > waveSimCore::d2fdx2 (
            boost::multi_array< double, 2 > f,
            double dx )
```

Takes the second partial derivative of a 2D matrix f with respect to x.

Definition at line 32 of file helper_func.hpp.

```
00033      {
00034          boost::multi_array<double, 2> df = dfdx(f, dx);
00035          boost::multi_array<double, 2> df2 = dfdx(df, dx);
00036          return df2;
00037      }
```

8.1.2.2 d2fdz2()

```
boost::multi_array< double, 2 > waveSimCore::d2fdz2 (
            boost::multi_array< double, 2 > f,
            double dz )
```

Takes the second partial derivative of a 2D matrix f with respect to z.

Definition at line 40 of file helper_func.hpp.

```
00041      {
00042          boost::multi_array<double, 2> df = dfdz(f, dz);
00043          boost::multi_array<double, 2> df2 = dfdz(df, dz);
00044          return df2;
00045      }
```

8.1.2.3 dfdx()

```
boost::multi_array< double, 2 > waveSimCore::dfdx (
            boost::multi_array< double, 2 > f,
            double dx )
```

Takes the partial derivative of a 2D matrix f with respect to x.

Definition at line 18 of file helper_func.hpp.

```
00019      {
00020          std::vector<boost::multi_array<double, 2> grad_f = np::gradient(f, {dx, dx});
00021          return grad_f[0];
00022      }
```

8.1.2.4 dfdz()

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

Takes the partial derivative of a 2D matrix f with respect to z.

Definition at line 25 of file helper_func.hpp.

```
00026      {
00027          std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dz, dz});
00028          return grad_f[1];
00029      }
```

8.1.2.5 divergence()

```
boost::multi_array< double, 2 > waveSimCore::divergence (
            boost::multi_array< double, 2 > f1,
            boost::multi_array< double, 2 > f2,
            double dx,
            double dz )
```

Takes the divergence of a 2D matrices fx,fz with respect to x and z
Returns dfx/dx + dfz/dz

Definition at line 49 of file helper_func.hpp.

```
00051      {
00052          boost::multi_array<double, 2> f_x = dfdx(f1, dx);
00053          boost::multi_array<double, 2> f_z = dfdz(f2, dz);
00054          boost::multi_array<double, 2> div = f_x + f_z;
00055          return div;
00056      }
```

8.1.2.6 get_profile()

```
boost::multi_array< double, 2 > waveSimCore::get_profile (
            double xmin,
            double xmax,
            double zmin,
            double zmax,
            int nx,
            int nz,
            double r )
```

Get the velocity profile of the model as a 2D Array.

Definition at line 16 of file computational.hpp.

```
00017      {
00018          boost::multi_array<double, 2> c(boost::extents[nx][nz]);
00019
00020          boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
00021          boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);
00022
00023          const boost::multi_array<double, 1> axis[2] = {x, z};
00024          std::vector<boost::multi_array<double, 2» XZ = np::meshgrid(axis, false, np::ij);
00025
00026          double x_0 = xmax / 2.0;
00027          double z_0 = zmax / 2.0;
```

```
00028
00029          for (int i = 0; i < nx; i++)
00030          {
00031              for (int j = 0; j < nz; j++)
00032              {
00033                  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))
00034                      c[i][j] = 3.0;
00035                  else
00036                      c[i][j] = 3.0;
00037              }
00038          }
00039
00040          return c;
00041      }
```

8.1.2.7 get_sigma_1()

```
boost::multi_array< double, 2 > waveSimCore::get_sigma_1 (
              boost::multi_array< double, 1 > x,
              double dx,
              int nx,
              int nz,
              double c_max,
              int n = 15,
              double R = 1e-3,
              double m = 2.0 )
```

Returns the sigma 1 coefficient for the PML boundary conditions
sigma_1 and sigma_2 are the coefficients for the x and z directions respectively
Check Johnson, Steven G. (2021).

Definition at line 21 of file coeff.hpp.

```
00023      {
00024          boost::multi_array<double, 2> sigma_1(boost::extents[nx][nz]);
00025          const double PML_width = n * dx;
00026
00027          const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);
00028
00029          const double x_0 = np::max(x) - PML_width;
00030
00031          boost::multi_array<double, 1> polynomial(boost::extents[nx]);
00032
00033          for (int i = 0; i < nx; i++)
00034          {
00035              if (x[i] > x_0)
00036              {
00037                  polynomial[i] = sigma_max * np::pow(np::abs(x[i] - x_0), m);
00038                  polynomial[nx - 1 - i] = polynomial[i];
00039              }
00040              else
00041              {
00042                  polynomial[i] = 0;
00043              }
00044          }
00045
00046          for (int i = 0; i < nx; i++)
00047              for (int j = 0; j < nz; j++)
00048                  sigma_1[i][j] = polynomial[i];
00049
00050          return sigma_1;
00051      }
```

8.1.2.8 get_sigma_2()

```
boost::multi_array< double, 2 > waveSimCore::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 )
```

Returns the sigma 2 coefficient for the PML boundary conditions
sigma_1 and sigma_2 are the coefficients for the x and z directions respectively
Check Johnson, Steven G. (2021).

Definition at line 56 of file coeff.hpp.

```
00058    {
00059        boost::multi_array<double, 2> sigma_2(boost::extents[nx][nz]);
00060        const double PML_width = n * dz;
00061        const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);
00062
00063        const double z_0 = np::max(z) - PML_width;
00064
00065        boost::multi_array<double, 1> polynomial(boost::extents[nz]);
00066        for (int j = 0; j < nz; j++)
00067        {
00068            if (z[j] > z_0)
00069            {
00070                polynomial[j] = sigma_max * np::pow(np::abs(z[j] - z_0), m);
00071                polynomial[nz - 1 - j] = polynomial[j];
00072            }
00073            else
00074            {
00075                polynomial[j] = 0;
00076            }
00077        }
00078
00079        for (int i = 0; i < nx; i++)
00080            for (int j = 0; j < nz; j++)
00081                sigma_2[i][j] = polynomial[j];
00082
00083        return sigma_2;
00084    }
```

8.1.2.9 ricker()

```
boost::multi_array< double, 3 > waveSimCore::ricker (
            int i_s,
            int j_s,
            double f,
            double amp,
            double shift,
            double tmin,
            double tmax,
            int nt,
            int nx,
            int nz )
```

Get the Ricker wavelet as a 3D Array.

Definition at line 14 of file source.hpp.

```
00016    {
00017        const double pi = 3.141592654;
```

```
00018
00019          boost::multi_array<double, 1> t = np::linspace(tmin, tmax, nt);
00020          boost::multi_array<double, 1> pft2 = np::pow(pi * f * (t - shift), 2.0);
00021          boost::multi_array<double, 1> r = amp * (1.0 - 2.0 * pft2) * np::exp(-1.0 * pft2);
00022
00023          int dimensions_x[] = {nx};
00024          boost::multi_array<double, 1> x = np::zeros<double>(dimensions_x);
00025
00026          int dimensions_z[] = {nz};
00027          boost::multi_array<double, 1> z = np::zeros<double>(dimensions_z);
00028
00029          x[i_s] = 1.0;
00030          z[j_s] = 1.0;
00031
00032          const boost::multi_array<double, 1> axis[3] = {r, x, z};
00033          std::vector<boost::multi_array<double, 3>> RXZ = np::meshgrid(axis, false, np::ij);
00034          boost::multi_array<double, 3> source = RXZ[0] * RXZ[1] * RXZ[2];
00035
00036          return source;
00037      }
```

8.1.2.10 wave_solver()

```
boost::multi_array< double, 3 > waveSimCore::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 )
```

This function solves the wave equation using the methods explained in the readme
The boundary conditions are so that waves bounce at the boundary

Definition at line 27 of file solver.hpp.

```
00030      {
00031
00032          const boost::multi_array<double, 2> CX = np::pow(c * dt / dx, 2.0);
00033          const boost::multi_array<double, 2> CZ = np::pow(c * dt / dz, 2.0);
00034          const double Cf = np::pow(dt, 2.0);
00035
00036          int dimensions_1[] = {nt, nx, nz};
00037          boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
00038          int dimensions_4[] = {nx, nz};
00039          boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
00040          int dimensions_5[] = {nx, nz};
00041          boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);
00042
00043          for (int n = 1; n < nt - 1; n++)
00044          {
00045              for (int i = 1; i < nx - 1; i++)
00046              {
00047                  for (int j = 1; j < nz - 1; j++)
00048                  {
00049                      u_xx[i][j] = u[n][i + 1][j] + u[n][i - 1][j] - 2.0 * u[n][i][j];
00050                      u_zz[i][j] = u[n][i][j + 1] + u[n][i][j - 1] - 2.0 * u[n][i][j];
00051                  }
00052              }
00053
00054              // ! Update u
00055              for (int i = 0; i < nx; i++)
00056              {
00057                  for (int j = 0; j < nz; j++)
00058                  {
00059                      u[n + 1][i][j] = 2.0 * u[n][i][j] + CX[i][j] * u_xx[i][j] + CZ[i][j] * u_zz[i][j]
      + Cf * f[n][i][j] - u[n - 1][i][j];
00060                  }
00061              }
00062
00063              // Dirichlet boundary condition
00064              for (int i = 0; i < nx; i++)
```

```
00065                    {
00066                        u[n + 1][i][0] = 0.0;
00067                        u[n + 1][i][nz - 1] = 0.0;
00068                    }
00069                    for (int j = 0; j < nz; j++)
00070                    {
00071                        u[n + 1][0][j] = 0.0;
00072                        u[n + 1][nx - 1][j] = 0.0;
00073                    }
00074            }
00075            return u;
00076    }
```

8.1.2.11   wave_solver_complex()

```
boost::multi_array< double, 3 > waveSimCore::wave_solver_complex (
            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 )
```

Solves the wave equation using a more complex solver compared to wave_solver
Also has different boundary conditions. In this one the boundary extends to infinity

Definition at line 22 of file solver_complex.hpp.

```
00026    {
00027
00028            const boost::multi_array<double, 2> C1 = 1.0 + (dt * (sigma_1 + sigma_2) * 0.5);
00029            const boost::multi_array<double, 2> C2 = (sigma_1 * sigma_2 * np::pow(dt, 2.0)) - 2.0;
00030            const boost::multi_array<double, 2> C3 = 1.0 - (dt * (sigma_1 + sigma_2) * 0.5);
00031            const boost::multi_array<double, 2> C4 = np::pow(dt * c, 2.0);
00032            const boost::multi_array<double, 2> C5 = 1.0 + (dt * sigma_1 * 0.5);
00033            const boost::multi_array<double, 2> C6 = 1.0 + (dt * sigma_2 * 0.5);
00034            const boost::multi_array<double, 2> C7 = 1.0 - (dt * sigma_1 * 0.5);
00035            const boost::multi_array<double, 2> C8 = 1.0 - (dt * sigma_2 * 0.5);
00036
00037            int dimensions_1[] = {nt, nx, nz};
00038            boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
00039
00040            int dimensions_2[] = {nx, nz};
00041            boost::multi_array<double, 2> q_1 = np::zeros<double>(dimensions_2);
00042            int dimensions_3[] = {nx, nz};
00043            boost::multi_array<double, 2> q_2 = np::zeros<double>(dimensions_3);
00044
00045            int dimensions_4[] = {nx, nz};
00046            boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
00047            int dimensions_5[] = {nx, nz};
00048            boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);
00049
00050            boost::multi_array<double, 2> f_n(boost::extents[nx][nz]);
00051            boost::multi_array<double, 2> u_n(boost::extents[nx][nz]);
00052            boost::multi_array<double, 2> u_n_1(boost::extents[nx][nz]);
00053
00054            for (int n = 1; n < nt - 1; n++)
00055            {
00056
00057                for (int i = 0; i < nx; i++)
00058                {
00059                    for (int j = 0; j < nz; j++)
00060                    {
00061                        f_n[i][j] = f[n][i][j];
00062                        u_n[i][j] = u[n][i][j];
00063                        u_n_1[i][j] = u[n - 1][i][j];
00064                    }
00065                }
00066
```

```
00067                boost::multi_array<double, 2> div = divergence(q_1 * sigma_1, q_2 * sigma_2, dx, dz);
00068                boost::multi_array<double, 2> dq_1dx = dfdx(q_1, dx);
00069                boost::multi_array<double, 2> dq_2dz = dfdz(q_2, dz);
00070                u_xx = d2fdx2(u_n, dx); // (nx, nz)
00071                u_zz = d2fdz2(u_n, dz); // (nx, nz)
00072                boost::multi_array<double, 2> dudx = dfdx(u_n, dx);
00073                boost::multi_array<double, 2> dudz = dfdz(u_n, dz);
00074
00075                //        for (int i = 1; i < nx-1; i++)
00076                //        {
00077                //            for (int j = 1; j < nz-1; j++)
00078                //            {
00079                //                u_xx[i][j] = u_n[i+1][j] + u_n[i-1][j] - 2.0 * u_n[i][j];
00080                //                u_zz[i][j] = u_n[i][j+1] + u_n[i][j-1] - 2.0 * u_n[i][j];
00081                //            }
00082                //        }
00083
00084                // ! Update u
00085                for (int i = 0; i < nx; i++)
00086                {
00087                    for (int j = 0; j < nz; j++)
00088                    {
00089                        u[n + 1][i][j] = (C4[i][j] * ((u_xx[i][j] / np::pow(dx, 2.0)) + (u_zz[i][j] /
    np::pow(dz, 2.0)) - div[i][j] + sigma_2[i][j] * dq_1dx[i][j] + sigma_1[i][j] * dq_2dz[i][j] +
    f_n[i][j]) - (C2[i][j] * u_n[i][j]) - (C3[i][j] * u_n_1[i][j])) / C1[i][j];
00090                    }
00091                }
00092
00093                // ! Update q_1, q_2
00094                for (int i = 0; i < nx; i++)
00095                {
00096                    for (int j = 0; j < nz; j++)
00097                    {
00098                        q_1[i][j] = (dt * dudx[i][j] + C7[i][j] * q_1[i][j]) / C5[i][j];
00099                        q_2[i][j] = (dt * dudz[i][j] + C8[i][j] * q_2[i][j]) / C6[i][j];
00100                    }
00101                }
00102                //
00103                // Dirichlet boundary condition
00104                for (int i = 0; i < nx; i++)
00105                {
00106                    u[n + 1][i][0] = 0.0;
00107                    u[n + 1][i][nz - 1] = 0.0;
00108                }
00109                for (int j = 0; j < nz; j++)
00110                {
00111                    u[n + 1][0][j] = 0.0;
00112                    u[n + 1][nx - 1][j] = 0.0;
00113                }
00114            }
00115        return u;
00116    }
```

8.2 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.

- matplot::vector_2d np::convert_to_matplot (const boost::multi_array< double, 2 > &arr)

  Convert a 2D boost::multi_array to a matplot::vector_2d.

8.2.1 Detailed Description

8.2.2 Function Documentation

8.2.2.1 convert_to_matplot()

```
matplot::vector_2d np::convert_to_matplot (
            const boost::multi_array< double, 2 > & arr )  [inline]
```

Convert a 2D boost::multi_array to a matplot::vector_2d.

Definition at line 16 of file np_to_matplot.hpp.

```
00017     {
00018         std::vector<double> x = matplot::linspace(0, 0, arr.shape()[0]);
00019         std::vector<double> y = matplot::linspace(00, 0, arr.shape()[1]);
00020         matplot::vector_2d result = std::get<0>(matplot::meshgrid(x, y));
```

```
00021          // std::cout « "arr.shape()[0] = " « arr.shape()[0] « " arr.shape()[1] = " « arr.shape()[1] «
    std::endl;
00022          for (size_t i = 0; i < arr.shape()[0]; i++)
00023          {
00024              for (size_t j = 0; j < arr.shape()[1]; j++)
00025              {
00026                  result[i][j] = arr[i][j];
00027              }
00028          }
00029          return result;
00030      }
```

8.2.2.2 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 }
```

8.2.2.3 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 }
```

8.2.2.4 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.

```
00513 {
00514      std::function<T(T)> func = [lhs](T item)
00515      { return lhs * item; };
00516      return np::element_wise_apply(rhs, func);
00517 }
```

8.2.2.5 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.

```
00529 {
00530     std::function<T(T, T)> func = std::plus<T>();
00531     return np::element_wise_duo_apply(lhs, rhs, func);
00532 }
```

8.2.2.6 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.

```
00546 {
00547     return rhs + lhs;
00548 }
```

8.2.2.7 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.

```
00537 {
00538     std::function<T(T)> func = [lhs](T item)
00539     { return lhs + item; };
00540     return np::element_wise_apply(rhs, func);
00541 }
```

8.2.2.8 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.

```
00554 {
00555     std::function<T(T, T)> func = std::minus<T>();
00556     return np::element_wise_duo_apply(lhs, rhs, func);
00557 }
```

8.2.2.9 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.

```
00571 {
00572     return rhs - lhs;
00573 }
```

8.2.2.10 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.

```
00562 {
00563     std::function<T(T)> func = [lhs](T item)
00564     { return lhs - item; };
00565     return np::element_wise_apply(rhs, func);
00566 }
```

8.2.2.11 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.

```
00579 {
00580     std::function<T(T, T)> func = std::divides<T>();
00581     return np::element_wise_duo_apply(lhs, rhs, func);
00582 }
```

8.2.2.12 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.

```
00596 {
00597     std::function<T(T)> func = [rhs](T item)
00598     { return item / rhs; };
00599     return np::element_wise_apply(lhs, func);
00600 }
```

8.2.2.13 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.

```
00587 {
00588     std::function<T(T)> func = [lhs](T item)
00589     { return lhs / item; };
00590     return np::element_wise_apply(rhs, func);
00591 }
```

8.3 WavePlotter

Namespaces

- namespace wavePlotter

    Custom plotter class.

Classes

- class wavePlotter::Plotter

    This class is used to plot the wave field TODO: make it multithreaded.

Functions

- wavePlotter::Plotter::Plotter (const boost::multi_array< double, 3 > &u, const matplot::vector_2d &Xp, const matplot::vector_2d &Zp, int num_levels, int nt)

    Constructor.
- void wavePlotter::Plotter::renderFrame (int index)

    Renders a frame of the wave field to a image on disk.
- void wavePlotter::Plotter::renderAllFrames (int begin_frame_index, int end_frame_index)

    Renders all frames of the wave field to form an animation to be saved on disk.
- void wavePlotter::Plotter::animate (std::string output_file_name, int begin_frame_index, int end_frame_index, int frame_rate)

    Renders a complete video animation of the wave field.
- void wavePlotter::Plotter::exportFrame (int index)

    Export a frame of the wave field to a .csv format for external use.
- void wavePlotter::Plotter::exportAllFrames (int begin_frame_index, int end_frame_index)

    Export all frames of the wave field to a .csv format for external use.
- void wavePlotter::Plotter::setSaveDirectory (std::string save_directory)

    Set the save directory for the rendered frames.

8.3.1 Detailed Description

8.3.2 Function Documentation

8.3.2.1 animate()

```
void wavePlotter::Plotter::animate (
            std::string output_file_name,
            int begin_frame_index,
            int end_frame_index,
            int frame_rate ) [inline]
```

Renders a complete video animation of the wave field.

Definition at line 56 of file wavePlotter.hpp.

```
00057        {
00058            renderAllFrames(begin_frame_index, end_frame_index);
00059            std::string ffmpeg_render_command = "ffmpeg -framerate " + std::to_string(frame_rate) + "
      -pattern_type glob -i '" + save_directory + "/*.png' -c:v libx264 -pix_fmt yuv420p " +
      output_file_name;
00060            std::system(ffmpeg_render_command.c_str());
00061        }
```

8.3.2.2 exportAllFrames()

```
void wavePlotter::Plotter::exportAllFrames (
              int begin_frame_index,
              int end_frame_index )  [inline]
```

Export all frames of the wave field to a .csv format for external use.

Definition at line 82 of file wavePlotter.hpp.

```
00083          {
00084              for (int i = begin_frame_index; i < end_frame_index; i++)
00085              {
00086                  exportFrame(i);
00087              }
00088          }
```

8.3.2.3 exportFrame()

```
void wavePlotter::Plotter::exportFrame (
              int index )  [inline]
```

Export a frame of the wave field to a .csv format for external use.

Definition at line 63 of file wavePlotter.hpp.

```
00064          {
00065              matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
00066              std::ofstream outfile;
00067              outfile.open(save_directory + "/frame_" + format_num(index) + ".csv");
00068              for (std::size_t i = 0; i < Up.size(); i++)
00069              {
00070                  for (std::size_t j = 0; j < Up[i].size(); j++)
00071                  {
00072                      outfile « Up[i][j];
00073                      if (j != Up[i].size() - 1)
00074                          outfile « ",";
00075                  }
00076                  outfile « "\n";
00077              }
00078              outfile.close();
00079          }
```

8.3.2.4 Plotter()

```
wavePlotter::Plotter::Plotter (
              const boost::multi_array< double, 3 > & u,
              const matplot::vector_2d & Xp,
              const matplot::vector_2d & Zp,
              int num_levels,
              int nt )  [inline]
```

Constructor.

Definition at line 25 of file wavePlotter.hpp.

```
00026          {
00027              this->u.resize(boost::extents[u.shape()[0]][u.shape()[1]][u.shape()[2]]);
00028              this->u = u;
00029              this->Xp = Xp;
00030              this->Zp = Zp;
00031              this->num_levels = num_levels;
00032              this->nt = nt;
00033              double min_u = np::min(u);
00034              double max_u = np::max(u);
00035              std::cout « "min_u = " « min_u « " max_u = " « max_u « "\n";
00036              this->levels = matplot::linspace(min_u, max_u, num_levels);
00037          }
```

8.3.2.5 renderAllFrames()

```
void wavePlotter::Plotter::renderAllFrames (
             int begin_frame_index,
             int end_frame_index ) [inline]
```

Renders all frames of the wave field to form an animation to be saved on disk.

Definition at line 47 of file wavePlotter.hpp.
```
00048         {
00049             for (int i = begin_frame_index; i < end_frame_index; i++)
00050             {
00051                 renderFrame(i);
00052             }
00053         }
```

8.3.2.6 renderFrame()

```
void wavePlotter::Plotter::renderFrame (
             int index ) [inline]
```

Renders a frame of the wave field to a image on disk.

Definition at line 39 of file wavePlotter.hpp.
```
00040         {
00041             matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
00042             matplot::contourf(this->Xp, this->Zp, Up, this->levels);
00043             matplot::save(save_directory + "/contourf_" + format_num(index) + ".png");
00044         }
```

8.3.2.7 setSaveDirectory()

```
void wavePlotter::Plotter::setSaveDirectory (
             std::string save_directory ) [inline]
```

Set the save directory for the rendered frames.

Definition at line 91 of file wavePlotter.hpp.
```
00092         {
00093             this->save_directory = save_directory;
00094         }
```

Chapter 9

Namespace Documentation

9.1   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.

- matplot::vector_2d convert_to_matplot (const boost::multi_array< double, 2 > &arr)

      Convert a 2D boost::multi_array to a matplot::vector_2d.

9.1.1 Detailed Description

Custom implementation of numpy in C++.

9.1.2 Typedef Documentation

9.1.2.1 ndArrayValue

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

Definition at line 22 of file np.hpp.

9.1.3 Enumeration Type Documentation

9.1.3.1 indexing

```
enum np::indexing
```

Definition at line 172 of file np.hpp.

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

9.1.4 Function Documentation

9.1.4.1 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.

```
00464    {
00465        return std::abs(input);
00466    }
```

9.1.4.2 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.

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

9.1.4.3 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.

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

9.1.4.4 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.

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

9.1.4.5 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.

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

9.1.4.6 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.
```
00081      {
00082          // Dispatch to the proper function
00083          for_each(boost::type<typename Array::element>(), A.begin(), A.end(), xform);
00084      }
```

9.1.4.7 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.
```
00060      {
00061          Val = xform(Val);
00062      }
```

9.1.4.8 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.
```
00053      {
00054          for_each(type_dispatch, A.begin(), A.end(), xform);
00055      }
```

9.1.4.9 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.

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

9.1.4.10 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.

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

9.1.4.11 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.

```
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     }
```

9.1.4.12 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 Uses ij indexing Todo: Implement xy indexing

Definition at line 90 of file np.hpp.

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

9.1.4.13 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 161 of file np.hpp.
```
00162      {
00163          double step = (stop - start) / (num - 1);
00164          boost::multi_array<double, 1> output(boost::extents[num]);
00165          for (std::size_t i = 0; i < num; i++)
00166          {
00167              output[i] = start + i * step;
00168          }
00169          return output;
00170      }
```

9.1.4.14 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.
```
00305      {
00306          std::function<T(T)> func = std::log<T>();
00307          return element_wise_apply(input_array, func);
00308      }
```

9.1.4.15 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.
```
00313      {
00314          return std::log(input);
00315      }
```

9.1.4.16 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.

```
00385     {
00386         T max = 0;
00387         bool max_not_set = true;
00388         const T *data_pointer = input_array.data();
00389         for (std::size_t i = 0; i < input_array.num_elements(); i++)
00390         {
00391             T element = *data_pointer;
00392             if (max_not_set || element > max)
00393             {
00394                 max = element;
00395                 max_not_set = false;
00396             }
00397             ++data_pointer;
00398         }
00399         return max;
00400     }
```

9.1.4.17 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.

```
00405     {
00406         T max = input1;
00407         for (T input : {inputs...})
00408         {
00409             if (input > max)
00410             {
00411                 max = input;
00412             }
00413         }
00414         return max;
00415     }
```

9.1.4.18 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 184 of file np.hpp.

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

9.1.4.19   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.

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

```
00425            {
00426                T element = *data_pointer;
00427                if (min_not_set || element < min)
00428                {
00429                    min = element;
00430                    min_not_set = false;
00431                }
00432                ++data_pointer;
00433            }
00434            return min;
00435    }
```

9.1.4.20 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.
```
00440    {
00441        T min = input1;
00442        for (T input : {inputs...})
00443        {
00444            if (input < min)
00445            {
00446                min = input;
00447            }
00448        }
00449        return min;
00450    }
00451
00453    template <typename T, long unsigned int ND>
00454    requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND>
        abs(boost::multi_array<T, ND> const &input_array)
00455    {
00456        std::function<T(T)> abs_func = [](T input)
00457        { return std::abs(input); };
00458        return element_wise_apply(input_array, abs_func);
00459    }
```

9.1.4.21 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.
```
00320    {
00321        std::function<T(T)> pow_func = [exponent](T input)
00322        { return std::pow(input, exponent); };
00323        return element_wise_apply(input_array, pow_func);
00324    }
```

9.1.4.22 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.

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

9.1.4.23 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.

```
00471    {
00472
00473        // Deducing the extents of the N-Dimensional output
00474        boost::detail::multi_array::extent_gen<ND − 1> output_extents;
00475        std::vector<size_t> shape_list;
00476        for (std::size_t i = 1; i < ND; i++)
00477        {
00478            shape_list.push_back(input_array.shape()[i]);
00479        }
00480        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00481
00482        boost::multi_array<T, ND − 1> output_array(output_extents);
00483
00484        const T *p = input_array.data();
00485        boost::array<std::size_t, ND> index;
00486        for (std::size_t i = 0; i < input_array.num_elements(); i++)
00487        {
00488            index = getIndexArray(input_array, p);
00489            output_array(index) = input_array[slice_index](index);
00490            p++;
00491        }
00492        return output_array;
00493    }
```

9.1.4.24 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.

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

9.1.4.25 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.

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

9.1.4.26 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.

```
00366    {
00367        // Deducing the extents of the N-Dimensional output
00368        boost::detail::multi_array::extent_gen<ND> output_extents;
00369        std::vector<size_t> shape_list;
00370        for (std::size_t i = 0; i < ND; i++)
00371        {
00372            shape_list.push_back(dimensions_input[i]);
00373        }
00374        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00375        // Applying a function to return zero always to all of its elements
00376        boost::multi_array<T, ND> output_array(output_extents);
00377        std::function<T(T)> zero_func = [](T input)
00378        { return 0; };
00379        return element_wise_apply(output_array, zero_func);
00380    }
```

9.2 wavePlotter Namespace Reference

Custom plotter class.

Classes

- class Plotter

    This class is used to plot the wave field TODO: make it multithreaded.

9.2.1 Detailed Description

Custom plotter class.

9.3 waveSimCore Namespace Reference

Functions

- 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=15, double R=1e-3, double m=2.0)
- 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 > get_profile (double xmin, double xmax, double zmin, double zmax, int nx, int nz, double r)

    Get the velocity profile of the model as a 2D Array.
- boost::multi_array< double, 2 > dfdx (boost::multi_array< double, 2 > f, double dx)

    Takes the partial derivative of a 2D matrix f with respect to x.
- boost::multi_array< double, 2 > dfdz (boost::multi_array< double, 2 > f, double dz)

    Takes the partial derivative of a 2D matrix f with respect to z.
- boost::multi_array< double, 2 > d2fdx2 (boost::multi_array< double, 2 > f, double dx)

    Takes the second partial derivative of a 2D matrix f with respect to x.
- boost::multi_array< double, 2 > d2fdz2 (boost::multi_array< double, 2 > f, double dz)

    Takes the second partial derivative of a 2D matrix f with respect to z.
- 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, 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, 3 > wave_solver_complex (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)
- 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)

    Get the Ricker wavelet as a 3D Array.

9.3.1 Detailed Description

This namespace contains all the core algorithm for solving wave equation
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

Chapter 10

Class Documentation

10.1 wavePlotter::Plotter Class Reference

This class is used to plot the wave field TODO: make it multithreaded.

```
#include <wavePlotter.hpp>
```

Public Member Functions

- Plotter (const boost::multi_array< double, 3 > &u, const matplot::vector_2d &Xp, const matplot::vector_2d &Zp, int num_levels, int nt)

    Constructor.
- void renderFrame (int index)

    Renders a frame of the wave field to a image on disk.
- void renderAllFrames (int begin_frame_index, int end_frame_index)

    Renders all frames of the wave field to form an animation to be saved on disk.
- void animate (std::string output_file_name, int begin_frame_index, int end_frame_index, int frame_rate)

    Renders a complete video animation of the wave field.
- void exportFrame (int index)

    Export a frame of the wave field to a .csv format for external use.
- void exportAllFrames (int begin_frame_index, int end_frame_index)

    Export all frames of the wave field to a .csv format for external use.
- void setSaveDirectory (std::string save_directory)

    Set the save directory for the rendered frames.

10.1.1 Detailed Description

This class is used to plot the wave field TODO: make it multithreaded.

Plotter class

Definition at line 21 of file wavePlotter.hpp.

The documentation for this class was generated from the following file:

- src/CustomLibraries/wavePlotter.hpp

Chapter 11

File Documentation

11.1   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
00016 namespace waveSimCore
00017 {
00021     boost::multi_array<double, 2> get_sigma_1(boost::multi_array<double, 1> x, double dx, int nx, int
    nz,
00022                                                double c_max, int n = 15, double R = 1e-3, double m =
    2.0)
00023     {
00024         boost::multi_array<double, 2> sigma_1(boost::extents[nx][nz]);
00025         const double PML_width = n * dx;
00026
00027         const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);
00028
00029         const double x_0 = np::max(x) - PML_width;
00030
00031         boost::multi_array<double, 1> polynomial(boost::extents[nx]);
00032
00033         for (int i = 0; i < nx; i++)
00034         {
00035             if (x[i] > x_0)
00036             {
00037                 polynomial[i] = sigma_max * np::pow(np::abs(x[i] - x_0), m);
00038                 polynomial[nx - 1 - i] = polynomial[i];
00039             }
00040             else
00041             {
00042                 polynomial[i] = 0;
00043             }
00044         }
00045
00046         for (int i = 0; i < nx; i++)
00047             for (int j = 0; j < nz; j++)
00048                 sigma_1[i][j] = polynomial[i];
00049
00050         return sigma_1;
00051     }
00052
00056     boost::multi_array<double, 2> get_sigma_2(boost::multi_array<double, 1> z, double dz, int nx, int
    nz,
00057                                                double c_max, int n = 10, double R = 1e-3, double m =
    2.0)
00058     {
00059         boost::multi_array<double, 2> sigma_2(boost::extents[nx][nz]);
00060         const double PML_width = n * dz;
00061         const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);
00062
00063         const double z_0 = np::max(z) - PML_width;
00064
00065         boost::multi_array<double, 1> polynomial(boost::extents[nz]);
```

```
00066              for (int j = 0; j < nz; j++)
00067              {
00068                  if (z[j] > z_0)
00069                  {
00070                      polynomial[j] = sigma_max * np::pow(np::abs(z[j] - z_0), m);
00071                      polynomial[nz - 1 - j] = polynomial[j];
00072                  }
00073                  else
00074                  {
00075                      polynomial[j] = 0;
00076                  }
00077              }
00078
00079              for (int i = 0; i < nx; i++)
00080                  for (int j = 0; j < nz; j++)
00081                      sigma_2[i][j] = polynomial[j];
00082
00083              return sigma_2;
00084          }
00085 }
00086 #endif // WAVESIMC_COEFF_HPP
```

11.2 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
00013 namespace waveSimCore
00014 {
00016      boost::multi_array<double, 2> get_profile(double xmin, double xmax, double zmin, double zmax, int
     nx, int nz, double r)
00017      {
00018          boost::multi_array<double, 2> c(boost::extents[nx][nz]);
00019
00020          boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
00021          boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);
00022
00023          const boost::multi_array<double, 1> axis[2] = {x, z};
00024          std::vector<boost::multi_array<double, 2» XZ = np::meshgrid(axis, false, np::ij);
00025
00026          double x_0 = xmax / 2.0;
00027          double z_0 = zmax / 2.0;
00028
00029          for (int i = 0; i < nx; i++)
00030          {
00031              for (int j = 0; j < nz; j++)
00032              {
00033                  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))
00034                      c[i][j] = 3.0;
00035                  else
00036                      c[i][j] = 3.0;
00037              }
00038          }
00039
00040          return c;
00041      }
00042 }
00043 #endif // WAVESIMC_COMPUTATIONAL_HPP
```

11.3 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
00015 namespace waveSimCore
00016 {
00018      boost::multi_array<double, 2> dfdx(boost::multi_array<double, 2> f, double dx)
00019      {
```

```
00020            std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dx, dx});
00021            return grad_f[0];
00022       }
00023
00025       boost::multi_array<double, 2> dfdz(boost::multi_array<double, 2> f, double dz)
00026       {
00027            std::vector<boost::multi_array<double, 2» grad_f = np::gradient(f, {dz, dz});
00028            return grad_f[1];
00029       }
00030
00032       boost::multi_array<double, 2> d2fdx2(boost::multi_array<double, 2> f, double dx)
00033       {
00034            boost::multi_array<double, 2> df = dfdx(f, dx);
00035            boost::multi_array<double, 2> df2 = dfdx(df, dx);
00036            return df2;
00037       }
00038
00040       boost::multi_array<double, 2> d2fdz2(boost::multi_array<double, 2> f, double dz)
00041       {
00042            boost::multi_array<double, 2> df = dfdz(f, dz);
00043            boost::multi_array<double, 2> df2 = dfdz(df, dz);
00044            return df2;
00045       }
00046
00049       boost::multi_array<double, 2> divergence(boost::multi_array<double, 2> f1,
      boost::multi_array<double, 2> f2,
00050                                          double dx, double dz)
00051       {
00052            boost::multi_array<double, 2> f_x = dfdx(f1, dx);
00053            boost::multi_array<double, 2> f_z = dfdz(f2, dz);
00054            boost::multi_array<double, 2> div = f_x + f_z;
00055            return div;
00056       }
00057
00058 }
00059 #endif // WAVESIMC_HELPER_FUNC_HPP
```

11.4   solver.hpp

```
00001 //
00002 // Created by Yan Cheng on 12/6/22.
00003 //
00004
00005 #ifndef CORETESTS_CPP_SOLVER2_HPP
00006 #define CORETESTS_CPP_SOLVER2_HPP
00007
00008 #include "CustomLibraries/np.hpp"
00009
00023 namespace waveSimCore
00024 {
00027       boost::multi_array<double, 3> wave_solver(boost::multi_array<double, 2> c,
00028                                          double dt, double dx, double dz, int nt, int nx, int nz,
00029                                          boost::multi_array<double, 3> f)
00030       {
00031
00032            const boost::multi_array<double, 2> CX = np::pow(c * dt / dx, 2.0);
00033            const boost::multi_array<double, 2> CZ = np::pow(c * dt / dz, 2.0);
00034            const double Cf = np::pow(dt, 2.0);
00035
00036            int dimensions_1[] = {nt, nx, nz};
00037            boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
00038            int dimensions_4[] = {nx, nz};
00039            boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
00040            int dimensions_5[] = {nx, nz};
00041            boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);
00042
00043            for (int n = 1; n < nt - 1; n++)
00044            {
00045                for (int i = 1; i < nx - 1; i++)
00046                {
00047                    for (int j = 1; j < nz - 1; j++)
00048                    {
00049                        u_xx[i][j] = u[n][i + 1][j] + u[n][i - 1][j] - 2.0 * u[n][i][j];
00050                        u_zz[i][j] = u[n][i][j + 1] + u[n][i][j - 1] - 2.0 * u[n][i][j];
00051                    }
00052                }
00053
00054                // ! Update u
00055                for (int i = 0; i < nx; i++)
00056                {
00057                    for (int j = 0; j < nz; j++)
00058                    {
00059                        u[n + 1][i][j] = 2.0 * u[n][i][j] + CX[i][j] * u_xx[i][j] + CZ[i][j] * u_zz[i][j]
      + Cf * f[n][i][j] - u[n - 1][i][j];
```

```
00060                    }
00061                }
00062
00063            // Dirichlet boundary condition
00064            for (int i = 0; i < nx; i++)
00065            {
00066                u[n + 1][i][0] = 0.0;
00067                u[n + 1][i][nz - 1] = 0.0;
00068            }
00069            for (int j = 0; j < nz; j++)
00070            {
00071                u[n + 1][0][j] = 0.0;
00072                u[n + 1][nx - 1][j] = 0.0;
00073            }
00074        }
00075        return u;
00076    }
00077 }
00078 #endif // CORETESTS_CPP_SOLVER2_HPP
```

11.5 solver_complex.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_SOLVER_COMPLEX_HPP
00006 #define WAVESIMC_SOLVER_COMPLEX_HPP
00007
00008 #include "CustomLibraries/np.hpp"
00009 #include "helper_func.hpp"
00010
00011 #include <cmath>
00012
00018 namespace waveSimCore
00019 {
00022     boost::multi_array<double, 3> wave_solver_complex(boost::multi_array<double, 2> c,
00023                                                       double dt, double dx, double dz, int nt, int nx,
    int nz,
00024                                                       boost::multi_array<double, 3> f,
00025                                                       boost::multi_array<double, 2> sigma_1,
    boost::multi_array<double, 2> sigma_2)
00026    {
00027
00028        const boost::multi_array<double, 2> C1 = 1.0 + (dt * (sigma_1 + sigma_2) * 0.5);
00029        const boost::multi_array<double, 2> C2 = (sigma_1 * sigma_2 * np::pow(dt, 2.0)) - 2.0;
00030        const boost::multi_array<double, 2> C3 = 1.0 - (dt * (sigma_1 + sigma_2) * 0.5);
00031        const boost::multi_array<double, 2> C4 = np::pow(dt * c, 2.0);
00032        const boost::multi_array<double, 2> C5 = 1.0 + (dt * sigma_1 * 0.5);
00033        const boost::multi_array<double, 2> C6 = 1.0 + (dt * sigma_2 * 0.5);
00034        const boost::multi_array<double, 2> C7 = 1.0 - (dt * sigma_1 * 0.5);
00035        const boost::multi_array<double, 2> C8 = 1.0 - (dt * sigma_2 * 0.5);
00036
00037        int dimensions_1[] = {nt, nx, nz};
00038        boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
00039
00040        int dimensions_2[] = {nx, nz};
00041        boost::multi_array<double, 2> q_1 = np::zeros<double>(dimensions_2);
00042        int dimensions_3[] = {nx, nz};
00043        boost::multi_array<double, 2> q_2 = np::zeros<double>(dimensions_3);
00044
00045        int dimensions_4[] = {nx, nz};
00046        boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
00047        int dimensions_5[] = {nx, nz};
00048        boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);
00049
00050        boost::multi_array<double, 2> f_n(boost::extents[nx][nz]);
00051        boost::multi_array<double, 2> u_n(boost::extents[nx][nz]);
00052        boost::multi_array<double, 2> u_n_1(boost::extents[nx][nz]);
00053
00054        for (int n = 1; n < nt - 1; n++)
00055        {
00056
00057            for (int i = 0; i < nx; i++)
00058            {
00059                for (int j = 0; j < nz; j++)
00060                {
00061                    f_n[i][j] = f[n][i][j];
00062                    u_n[i][j] = u[n][i][j];
00063                    u_n_1[i][j] = u[n - 1][i][j];
00064                }
00065            }
00066
```

```
00067                 boost::multi_array<double, 2> div = divergence(q_1 * sigma_1, q_2 * sigma_2, dx, dz);
00068                 boost::multi_array<double, 2> dq_1dx = dfdx(q_1, dx);
00069                 boost::multi_array<double, 2> dq_2dz = dfdz(q_2, dz);
00070                 u_xx = d2fdx2(u_n, dx); // (nx, nz)
00071                 u_zz = d2fdz2(u_n, dz); // (nx, nz)
00072                 boost::multi_array<double, 2> dudx = dfdx(u_n, dx);
00073                 boost::multi_array<double, 2> dudz = dfdz(u_n, dz);
00074
00075                 //        for (int i = 1; i < nx-1; i++)
00076                 //        {
00077                 //            for (int j = 1; j < nz-1; j++)
00078                 //            {
00079                 //                u_xx[i][j] = u_n[i+1][j] + u_n[i-1][j] - 2.0 * u_n[i][j];
00080                 //                u_zz[i][j] = u_n[i][j+1] + u_n[i][j-1] - 2.0 * u_n[i][j];
00081                 //            }
00082                 //        }
00083
00084                 // ! Update u
00085                 for (int i = 0; i < nx; i++)
00086                 {
00087                     for (int j = 0; j < nz; j++)
00088                     {
00089                         u[n + 1][i][j] = (C4[i][j] * ((u_xx[i][j] / np::pow(dx, 2.0)) + (u_zz[i][j] /
     np::pow(dz, 2.0)) - div[i][j] + sigma_2[i][j] * dq_1dx[i][j] + sigma_1[i][j] * dq_2dz[i][j] +
     f_n[i][j]) - (C2[i][j] * u_n[i][j]) - (C3[i][j] * u_n_1[i][j])) / C1[i][j];
00090                     }
00091                 }
00092
00093                 // ! Update q_1, q_2
00094                 for (int i = 0; i < nx; i++)
00095                 {
00096                     for (int j = 0; j < nz; j++)
00097                     {
00098                         q_1[i][j] = (dt * dudx[i][j] + C7[i][j] * q_1[i][j]) / C5[i][j];
00099                         q_2[i][j] = (dt * dudz[i][j] + C8[i][j] * q_2[i][j]) / C6[i][j];
00100                     }
00101                 }
00102                 //
00103                 // Dirichlet boundary condition
00104                 for (int i = 0; i < nx; i++)
00105                 {
00106                     u[n + 1][i][0] = 0.0;
00107                     u[n + 1][i][nz - 1] = 0.0;
00108                 }
00109                 for (int j = 0; j < nz; j++)
00110                 {
00111                     u[n + 1][0][j] = 0.0;
00112                     u[n + 1][nx - 1][j] = 0.0;
00113                 }
00114             }
00115         return u;
00116     }
00117 }
00118 #endif // WAVESIMC_SOLVER_HPP
```

11.6 source.hpp

```
00001 //
00002 // Created by Yan Cheng on 11/28/22.
00003 //
00004
00005 #ifndef WAVESIMC_SOURCE_HPP
00006 #define WAVESIMC_SOURCE_HPP
00011 namespace waveSimCore
00012 {
00014     boost::multi_array<double, 3> ricker(int i_s, int j_s, double f, double amp, double shift,
00015                                          double tmin, double tmax, int nt, int nx, int nz)
00016     {
00017         const double pi = 3.141592654;
00018
00019         boost::multi_array<double, 1> t = np::linspace(tmin, tmax, nt);
00020         boost::multi_array<double, 1> pft2 = np::pow(pi * f * (t - shift), 2.0);
00021         boost::multi_array<double, 1> r = amp * (1.0 - 2.0 * pft2) * np::exp(-1.0 * pft2);
00022
00023         int dimensions_x[] = {nx};
00024         boost::multi_array<double, 1> x = np::zeros<double>(dimensions_x);
00025
00026         int dimensions_z[] = {nz};
00027         boost::multi_array<double, 1> z = np::zeros<double>(dimensions_z);
00028
00029         x[i_s] = 1.0;
00030         z[j_s] = 1.0;
00031
```

```
00032          const boost::multi_array<double, 1> axis[3] = {r, x, z};
00033          std::vector<boost::multi_array<double, 3> RXZ = np::meshgrid(axis, false, np::ij);
00034          boost::multi_array<double, 3> source = RXZ[0] * RXZ[1] * RXZ[2];
00035
00036          return source;
00037      }
00038 }
00039 #endif // WAVESIMC_SOURCE_HPP
```

11.7 wave.cpp

```
00001 // Standard IO libraries
00002 #include <iostream>
00003 #include <fstream>
00004
00005 #include "CustomLibraries/np.hpp"
00006
00007 #include <math.h>
00008
00009 #include "solver.hpp"
00010 #include "computational.hpp"
00011 #include "coeff.hpp"
00012 #include "source.hpp"
00013 #include "helper_func.hpp"
00014
00015 int main()
00016 {
00017      double dx, dy, dz, dt;
00018      dx = 1.0;
00019      dy = 1.0;
00020      dz = 1.0;
00021      dt = 1.0;
00022      std::vector<boost::multi_array<double, 4> my_arrays = np::gradient(A, {dx, dy, dz, dt});
00023      return 0;
00024 }
```

11.8 np.hpp

```
00001 #ifndef NP_H_
00002 #define NP_H_
00003
00004 #include "boost/multi_array.hpp"
00005 #include "boost/array.hpp"
00006 #include "boost/cstdlib.hpp"
00007 #include <type_traits>
00008 #include <cassert>
00009 #include <iostream>
00010 #include <functional>
00011 #include <type_traits>
00012
00019 namespace np
00020 {
00021
00022      typedef double ndArrayValue;
00023
00025      template <std::size_t ND>
00026      inline boost::multi_array<ndArrayValue, ND>::index
00027      getIndex(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement,
        const unsigned short int direction)
00028      {
00029          int offset = requestedElement - m.origin();
00030          return (offset / m.strides()[direction] % m.shape()[direction] + m.index_bases()[direction]);
00031      }
00032
00034      template <std::size_t ND>
00035      inline boost::array<typename boost::multi_array<ndArrayValue, ND>::index, ND>
00036      getIndexArray(const boost::multi_array<ndArrayValue, ND> &m, const ndArrayValue *requestedElement)
00037      {
00038          using indexType = boost::multi_array<ndArrayValue, ND>::index;
00039          boost::array<indexType, ND> _index;
00040          for (unsigned int dir = 0; dir < ND; dir++)
00041          {
00042              _index[dir] = getIndex(m, requestedElement, dir);
00043          }
00044
00045          return _index;
00046      }
00047
00050      template <typename Array, typename Element, typename Functor>
```

```
00051      inline void for_each(const boost::type<Element> &type_dispatch,
00052                           Array A, Functor &xform)
00053      {
00054          for_each(type_dispatch, A.begin(), A.end(), xform);
00055      }
00056
00058      template <typename Element, typename Functor>
00059      inline void for_each(const boost::type<Element> &, Element &Val, Functor &xform)
00060      {
00061          Val = xform(Val);
00062      }
00063
00065      template <typename Element, typename Iterator, typename Functor>
00066      inline void for_each(const boost::type<Element> &type_dispatch,
00067                           Iterator begin, Iterator end,
00068                           Functor &xform)
00069      {
00070          while (begin != end)
00071          {
00072              for_each(type_dispatch, *begin, xform);
00073              ++begin;
00074          }
00075      }
00076
00079      template <typename Array, typename Functor>
00080      inline void for_each(Array &A, Functor xform)
00081      {
00082          // Dispatch to the proper function
00083          for_each(boost::type<typename Array::element>(), A.begin(), A.end(), xform);
00084      }
00085
00089      template <typename T, long unsigned int ND>
00090      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)
00091      {
00092          // static_assert(args.size() == ND, "Number of arguments must match the number of dimensions
      of the array");
00093          using arrayIndex = boost::multi_array<T, ND>::index;
00094
00095          using ndIndexArray = boost::array<arrayIndex, ND>;
00096
00097          // constexpr std::size_t n = sizeof...(Args);
00098          std::size_t n = args.size();
00099          // std::tuple<Args...> store(args...);
00100          std::vector<T> arg_vector = args;
00101          boost::multi_array<T, ND> my_array;
00102          std::vector<boost::multi_array<T, ND>> output_arrays;
00103          for (std::size_t i = 0; i < n; i++)
00104          {
00105              boost::multi_array<T, ND> dfdh = inArray;
00106              output_arrays.push_back(dfdh);
00107          }
00108
00109          ndArrayValue *p = inArray.data();
00110          ndIndexArray index;
00111          for (std::size_t i = 0; i < inArray.num_elements(); i++)
00112          {
00113              index = getIndexArray(inArray, p);
00114              /*
00115              std::cout « "Index: ";
00116              for (std::size_t j = 0; j < n; j++)
00117              {
00118                  std::cout « index[j] « " ";
00119              }
00120              std::cout « "\n";
00121              */
00122              // Calculating the gradient now
00123              // j is the axis/dimension
00124              for (std::size_t j = 0; j < n; j++)
00125              {
00126                  ndIndexArray index_high = index;
00127                  T dh_high;
00128                  if ((long unsigned int)index_high[j] < inArray.shape()[j] - 1)
00129                  {
00130                      index_high[j] += 1;
00131                      dh_high = arg_vector[j];
00132                  }
00133                  else
00134                  {
00135                      dh_high = 0;
00136                  }
00137                  ndIndexArray index_low = index;
00138                  T dh_low;
00139                  if (index_low[j] > 0)
00140                  {
00141                      index_low[j] -= 1;
00142                      dh_low = arg_vector[j];
```

```
00143                    }
00144                    else
00145                    {
00146                        dh_low = 0;
00147                    }
00148
00149                    T dh = dh_high + dh_low;
00150                    T gradient = (inArray(index_high) - inArray(index_low)) / dh;
00151                    // std::cout << gradient << "\n";
00152                    output_arrays[j](index) = gradient;
00153                }
00154                // std::cout << " value = " << inArray(index) << "  check = " << *p << std::endl;
00155                ++p;
00156            }
00157        return output_arrays;
00158    }
00159
00161    inline boost::multi_array<double, 1> linspace(double start, double stop, long unsigned int num)
00162    {
00163        double step = (stop - start) / (num - 1);
00164        boost::multi_array<double, 1> output(boost::extents[num]);
00165        for (std::size_t i = 0; i < num; i++)
00166        {
00167            output[i] = start + i * step;
00168        }
00169        return output;
00170    }
00171
00172    enum indexing
00173    {
00174        xy,
00175        ij
00176    };
00177
00183    template <typename T, long unsigned int ND>
00184    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)
00185    {
00186        using arrayIndex = boost::multi_array<T, ND>::index;
00187        using oneDArrayIndex = boost::multi_array<T, 1>::index;
00188        using ndIndexArray = boost::array<arrayIndex, ND>;
00189        std::vector<boost::multi_array<T, ND>> output_arrays;
00190        boost::multi_array<T, 1> ci[ND];
00191        // Copy elements of cinput to ci, do the proper inversions
00192        for (std::size_t i = 0; i < ND; i++)
00193        {
00194            std::size_t source = i;
00195            if (indexing_type == xy && (ND == 3 || ND == 2))
00196            {
00197                if (i == 0)
00198                    source = 1;
00199                else if (i == 1)
00200                    source = 0;
00201                else
00202                    source = i;
00203            }
00204            ci[i] = boost::multi_array<T, 1>();
00205            ci[i].resize(boost::extents[cinput[source].num_elements()]);
00206            ci[i] = cinput[source];
00207        }
00208        // Deducing the extents of the N-Dimensional output
00209        boost::detail::multi_array::extent_gen<ND> output_extents;
00210        std::vector<size_t> shape_list;
00211        for (std::size_t i = 0; i < ND; i++)
00212        {
00213            shape_list.push_back(ci[i].shape()[0]);
00214        }
00215        std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00216
00217        // Creating the output arrays
00218        for (std::size_t i = 0; i < ND; i++)
00219        {
00220            boost::multi_array<T, ND> output_array(output_extents);
00221            ndArrayValue *p = output_array.data();
00222            ndIndexArray index;
00223            // Looping through the elements of the output array
00224            for (std::size_t j = 0; j < output_array.num_elements(); j++)
00225            {
00226                index = getIndexArray(output_array, p);
00227                oneDArrayIndex index_1d;
00228                index_1d = index[i];
00229                output_array(index) = ci[i][index_1d];
00230                ++p;
00231            }
00232            output_arrays.push_back(output_array);
00233        }
```

```
00234                if (indexing_type == xy && (ND == 3 || ND == 2))
00235                {
00236                    std::swap(output_arrays[0], output_arrays[1]);
00237                }
00238                return output_arrays;
00239          }
00240
00242       template <class T, long unsigned int ND>
00243       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)
00244          {
00245
00246                // Create output array copying extents
00247                using arrayIndex = boost::multi_array<double, ND>::index;
00248                using ndIndexArray = boost::array<arrayIndex, ND>;
00249                boost::detail::multi_array::extent_gen<ND> output_extents;
00250                std::vector<size_t> shape_list;
00251                for (std::size_t i = 0; i < ND; i++)
00252                {
00253                    shape_list.push_back(input_array.shape()[i]);
00254                }
00255                std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00256                boost::multi_array<T, ND> output_array(output_extents);
00257
00258                // Looping through the elements of the output array
00259                const T *p = input_array.data();
00260                ndIndexArray index;
00261                for (std::size_t i = 0; i < input_array.num_elements(); i++)
00262                {
00263                    index = getIndexArray(input_array, p);
00264                    output_array(index) = func(input_array(index));
00265                    ++p;
00266                }
00267                return output_array;
00268          }
00269
00270       // Complex operations
00271
00273       template <class T, long unsigned int ND>
00274       requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> sqrt(const
        boost::multi_array<T, ND> &input_array)
00275          {
00276                std::function<T(T)> func = (T(*)(T))std::sqrt;
00277                return element_wise_apply(input_array, func);
00278          }
00279
00281       template <class T>
00282       requires std::is_arithmetic<T>::value inline constexpr T sqrt(const T input)
00283          {
00284                return std::sqrt(input);
00285          }
00286
00288       template <class T, long unsigned int ND>
00289       requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> exp(const
        boost::multi_array<T, ND> &input_array)
00290          {
00291                std::function<T(T)> func = (T(*)(T))std::exp;
00292                return element_wise_apply(input_array, func);
00293          }
00294
00296       template <class T>
00297       requires std::is_arithmetic<T>::value inline constexpr T exp(const T input)
00298          {
00299                return std::exp(input);
00300          }
00301
00303       template <class T, long unsigned int ND>
00304       requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND> log(const
        boost::multi_array<T, ND> &input_array)
00305          {
00306                std::function<T(T)> func = std::log<T>();
00307                return element_wise_apply(input_array, func);
00308          }
00309
00311       template <class T>
00312       requires std::is_arithmetic<T>::value inline constexpr T log(const T input)
00313          {
00314                return std::log(input);
00315          }
00316
00318       template <class T, long unsigned int ND>
00319       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)
00320          {
00321                std::function<T(T)> pow_func = [exponent](T input)
00322                { return std::pow(input, exponent); };
00323                return element_wise_apply(input_array, pow_func);
```

```
00324        }
00325
00327        template <class T>
00328        requires std::is_arithmetic<T>::value inline constexpr T pow(const T input, const T exponent)
00329        {
00330            return std::pow(input, exponent);
00331        }
00332
00336        template <class T, long unsigned int ND>
00337        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)
00338        {
00339            // Create output array copying extents
00340            using arrayIndex = boost::multi_array<double, ND>::index;
00341            using ndIndexArray = boost::array<arrayIndex, ND>;
00342            boost::detail::multi_array::extent_gen<ND> output_extents;
00343            std::vector<size_t> shape_list;
00344            for (std::size_t i = 0; i < ND; i++)
00345            {
00346                shape_list.push_back(lhs.shape()[i]);
00347            }
00348            std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00349            boost::multi_array<T, ND> output_array(output_extents);
00350
00351            // Looping through the elements of the output array
00352            const T *p = lhs.data();
00353            ndIndexArray index;
00354            for (std::size_t i = 0; i < lhs.num_elements(); i++)
00355            {
00356                index = getIndexArray(lhs, p);
00357                output_array(index) = func(lhs(index), rhs(index));
00358                ++p;
00359            }
00360            return output_array;
00361        }
00362
00364        template <typename T, typename inT, long unsigned int ND>
00365        requires std::is_integral<inT>::value && std::is_arithmetic<T>::value inline constexpr
     boost::multi_array<T, ND> zeros(inT (&dimensions_input)[ND])
00366        {
00367            // Deducing the extents of the N-Dimensional output
00368            boost::detail::multi_array::extent_gen<ND> output_extents;
00369            std::vector<size_t> shape_list;
00370            for (std::size_t i = 0; i < ND; i++)
00371            {
00372                shape_list.push_back(dimensions_input[i]);
00373            }
00374            std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00375            // Applying a function to return zero always to all of its elements
00376            boost::multi_array<T, ND> output_array(output_extents);
00377            std::function<T(T)> zero_func = [](T input)
00378            { return 0; };
00379            return element_wise_apply(output_array, zero_func);
00380        }
00381
00383        template <typename T, long unsigned int ND>
00384        requires std::is_arithmetic<T>::value inline constexpr T max(boost::multi_array<T, ND> const
     &input_array)
00385        {
00386            T max = 0;
00387            bool max_not_set = true;
00388            const T *data_pointer = input_array.data();
00389            for (std::size_t i = 0; i < input_array.num_elements(); i++)
00390            {
00391                T element = *data_pointer;
00392                if (max_not_set || element > max)
00393                {
00394                    max = element;
00395                    max_not_set = false;
00396                }
00397                ++data_pointer;
00398            }
00399            return max;
00400        }
00401
00403        template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)»
00404        requires std::is_arithmetic<T>::value inline constexpr T max(T input1, Ts... inputs)
00405        {
00406            T max = input1;
00407            for (T input : {inputs...})
00408            {
00409                if (input > max)
00410                {
00411                    max = input;
00412                }
00413            }
00414            return max;
```

```
00415        }
00416
00418        template <typename T, long unsigned int ND>
00419        requires std::is_arithmetic<T>::value inline constexpr T min(boost::multi_array<T, ND> const
        &input_array)
00420        {
00421            T min = 0;
00422            bool min_not_set = true;
00423            const T *data_pointer = input_array.data();
00424            for (std::size_t i = 0; i < input_array.num_elements(); i++)
00425            {
00426                T element = *data_pointer;
00427                if (min_not_set || element < min)
00428                {
00429                    min = element;
00430                    min_not_set = false;
00431                }
00432                ++data_pointer;
00433            }
00434            return min;
00435        }
00436
00438        template <class T, class... Ts, class = std::enable_if_t<(std::is_same_v<T, Ts> && ...)>>
00439        inline constexpr T min(T input1, Ts... inputs) requires std::is_arithmetic<T>::value
00440        {
00441            T min = input1;
00442            for (T input : {inputs...})
00443            {
00444                if (input < min)
00445                {
00446                    min = input;
00447                }
00448            }
00449            return min;
00450        }
00451
00453        template <typename T, long unsigned int ND>
00454        requires std::is_arithmetic<T>::value inline constexpr boost::multi_array<T, ND>
        abs(boost::multi_array<T, ND> const &input_array)
00455        {
00456            std::function<T(T)> abs_func = [](T input)
00457            { return std::abs(input); };
00458            return element_wise_apply(input_array, abs_func);
00459        }
00460
00462        template <typename T>
00463        requires std::is_arithmetic<T>::value inline constexpr T abs(T input)
00464        {
00465            return std::abs(input);
00466        }
00467
00469        template <typename T, long unsigned int ND>
00470        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)
00471        {
00472
00473            // Deducing the extents of the N-Dimensional output
00474            boost::detail::multi_array::extent_gen<ND - 1> output_extents;
00475            std::vector<size_t> shape_list;
00476            for (std::size_t i = 1; i < ND; i++)
00477            {
00478                shape_list.push_back(input_array.shape()[i]);
00479            }
00480            std::copy(shape_list.begin(), shape_list.end(), output_extents.ranges_.begin());
00481
00482            boost::multi_array<T, ND - 1> output_array(output_extents);
00483
00484            const T *p = input_array.data();
00485            boost::array<std::size_t, ND> index;
00486            for (std::size_t i = 0; i < input_array.num_elements(); i++)
00487            {
00488                index = getIndexArray(input_array, p);
00489                output_array(index) = input_array[slice_index](index);
00490                p++;
00491            }
00492            return output_array;
00493        }
00494
00495 }
00496
00497 // Override of operators in the boost::multi_array class to make them more np-like
00498 // Basic operators
00499 // All of the are element-wise
00500
00501 // Multiplication operator
00503 template <class T, long unsigned int ND>
00504 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, boost::multi_array<T,
```

```
      ND> const &rhs)
00505 {
00506     std::function<T(T, T)> func = std::multiplies<T>();
00507     return np::element_wise_duo_apply(lhs, rhs, func);
00508 }
00509
00511 template <class T, long unsigned int ND>
00512 inline boost::multi_array<T, ND> operator*(T const &lhs, boost::multi_array<T, ND> const &rhs)
00513 {
00514     std::function<T(T)> func = [lhs](T item)
00515     { return lhs * item; };
00516     return np::element_wise_apply(rhs, func);
00517 }
00519 template <class T, long unsigned int ND>
00520 inline boost::multi_array<T, ND> operator*(boost::multi_array<T, ND> const &lhs, T const &rhs)
00521 {
00522     return rhs * lhs;
00523 }
00524
00525 // Plus operator
00527 template <class T, long unsigned int ND>
00528 boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
      const &rhs)
00529 {
00530     std::function<T(T, T)> func = std::plus<T>();
00531     return np::element_wise_duo_apply(lhs, rhs, func);
00532 }
00533
00535 template <class T, long unsigned int ND>
00536 inline boost::multi_array<T, ND> operator+(T const &lhs, boost::multi_array<T, ND> const &rhs)
00537 {
00538     std::function<T(T)> func = [lhs](T item)
00539     { return lhs + item; };
00540     return np::element_wise_apply(rhs, func);
00541 }
00542
00544 template <class T, long unsigned int ND>
00545 inline boost::multi_array<T, ND> operator+(boost::multi_array<T, ND> const &lhs, T const &rhs)
00546 {
00547     return rhs + lhs;
00548 }
00549
00550 // Subtraction operator
00552 template <class T, long unsigned int ND>
00553 boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
      const &rhs)
00554 {
00555     std::function<T(T, T)> func = std::minus<T>();
00556     return np::element_wise_duo_apply(lhs, rhs, func);
00557 }
00558
00560 template <class T, long unsigned int ND>
00561 inline boost::multi_array<T, ND> operator-(T const &lhs, boost::multi_array<T, ND> const &rhs)
00562 {
00563     std::function<T(T)> func = [lhs](T item)
00564     { return lhs - item; };
00565     return np::element_wise_apply(rhs, func);
00566 }
00567
00569 template <class T, long unsigned int ND>
00570 inline boost::multi_array<T, ND> operator-(boost::multi_array<T, ND> const &lhs, T const &rhs)
00571 {
00572     return rhs - lhs;
00573 }
00574
00575 // Division operator
00577 template <class T, long unsigned int ND>
00578 boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, boost::multi_array<T, ND>
      const &rhs)
00579 {
00580     std::function<T(T, T)> func = std::divides<T>();
00581     return np::element_wise_duo_apply(lhs, rhs, func);
00582 }
00583
00585 template <class T, long unsigned int ND>
00586 inline boost::multi_array<T, ND> operator/(T const &lhs, boost::multi_array<T, ND> const &rhs)
00587 {
00588     std::function<T(T)> func = [lhs](T item)
00589     { return lhs / item; };
00590     return np::element_wise_apply(rhs, func);
00591 }
00592
00594 template <class T, long unsigned int ND>
00595 inline boost::multi_array<T, ND> operator/(boost::multi_array<T, ND> const &lhs, T const &rhs)
00596 {
00597     std::function<T(T)> func = [rhs](T item)
00598     { return item / rhs; };
```

```
00599      return np::element_wise_apply(lhs, func);
00600 }
00601
00603 #endif
```

11.9   np_to_matplot.hpp

```
00001 #ifndef NPTOMATPLOT_H_
00002 #define NPTOMATPLOT_H_
00003
00004 #include <matplot/matplot.h>
00005 #include <thread>
00006 #include "boost/multi_array.hpp"
00007 #include "boost/array.hpp"
00013 namespace np
00014 {
00016     inline matplot::vector_2d convert_to_matplot(const boost::multi_array<double, 2> &arr)
00017     {
00018         std::vector<double> x = matplot::linspace(0, 0, arr.shape()[0]);
00019         std::vector<double> y = matplot::linspace(00, 0, arr.shape()[1]);
00020         matplot::vector_2d result = std::get<0>(matplot::meshgrid(x, y));
00021         // std::cout « "arr.shape()[0] = " « arr.shape()[0] « " arr.shape()[1] = " « arr.shape()[1] «
    std::endl;
00022         for (size_t i = 0; i < arr.shape()[0]; i++)
00023         {
00024             for (size_t j = 0; j < arr.shape()[1]; j++)
00025             {
00026                 result[i][j] = arr[i][j];
00027             }
00028         }
00029         return result;
00030     }
00031 }
00032 #endif
```

11.10   wavePlotter.hpp

```
00001 #ifndef WAVESOLVER_H_
00002 #define WAVESOLVER_H_
00003 #include <boost/multi_array.hpp>
00004 #include <boost/array.hpp>
00005 #include <fstream>
00006 #include <iostream>
00007 #include "CustomLibraries/np.hpp"
00008 #include "CustomLibraries/np_to_matplot.hpp"
00009
00016 namespace wavePlotter
00017 {
00021     class Plotter
00022     {
00023     public:
00025         Plotter(const boost::multi_array<double, 3> &u, const matplot::vector_2d &Xp, const
    matplot::vector_2d &Zp, int num_levels, int nt)
00026         {
00027             this->u.resize(boost::extents[u.shape()[0]][u.shape()[1]][u.shape()[2]]);
00028             this->u = u;
00029             this->Xp = Xp;
00030             this->Zp = Zp;
00031             this->num_levels = num_levels;
00032             this->nt = nt;
00033             double min_u = np::min(u);
00034             double max_u = np::max(u);
00035             std::cout « "min_u = " « min_u « " max_u = " « max_u « "\n";
00036             this->levels = matplot::linspace(min_u, max_u, num_levels);
00037         }
00039         void renderFrame(int index)
00040         {
00041             matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
00042             matplot::contourf(this->Xp, this->Zp, Up, this->levels);
00043             matplot::save(save_directory + "/contourf_" + format_num(index) + ".png");
00044         }
00045
00047         void renderAllFrames(int begin_frame_index, int end_frame_index)
00048         {
00049             for (int i = begin_frame_index; i < end_frame_index; i++)
00050             {
00051                 renderFrame(i);
00052             }
00053         }
```

```
00054
00056          void animate(std::string output_file_name, int begin_frame_index, int end_frame_index, int
    frame_rate)
00057          {
00058              renderAllFrames(begin_frame_index, end_frame_index);
00059              std::string ffmpeg_render_command = "ffmpeg -framerate " + std::to_string(frame_rate) + "
    -pattern_type glob -i '" + save_directory + "/*.png' -c:v libx264 -pix_fmt yuv420p " +
    output_file_name;
00060              std::system(ffmpeg_render_command.c_str());
00061          }
00063          void exportFrame(int index)
00064          {
00065              matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
00066              std::ofstream outfile;
00067              outfile.open(save_directory + "/frame_" + format_num(index) + ".csv");
00068              for (std::size_t i = 0; i < Up.size(); i++)
00069              {
00070                  for (std::size_t j = 0; j < Up[i].size(); j++)
00071                  {
00072                      outfile << Up[i][j];
00073                      if (j != Up[i].size() - 1)
00074                          outfile << ",";
00075                  }
00076                  outfile << "\n";
00077              }
00078              outfile.close();
00079          }
00080
00082          void exportAllFrames(int begin_frame_index, int end_frame_index)
00083          {
00084              for (int i = begin_frame_index; i < end_frame_index; i++)
00085              {
00086                  exportFrame(i);
00087              }
00088          }
00089
00091          void setSaveDirectory(std::string save_directory)
00092          {
00093              this->save_directory = save_directory;
00094          }
00095
00096      private:
00097          std::string format_num(int num, int length = 8)
00098          {
00099              std::string str_num = std::to_string(num);
00100
00101              int str_length = str_num.length();
00102              for (int i = 0; i < length - str_length; i++)
00103                  str_num = "0" + str_num;
00104              return str_num;
00105          }
00106
00107          boost::multi_array<double, 3> u;
00108          matplot::vector_2d Xp;
00109          matplot::vector_2d Zp;
00110          int num_levels;
00111          int nt;
00112          std::vector<double> levels;
00113          std::string save_directory = "output";
00114      };
00115
00116 }
00117 #endif
```

11.11 wave_solver_with_animation.cpp

```
00001 #include <boost/multi_array.hpp>
00002 #include <boost/array.hpp>
00003 #include "CustomLibraries/np.hpp"
00004 #include "CustomLibraries/np_to_matplot.hpp"
00005 #include "CustomLibraries/wavePlotter.hpp"
00006 #include <matplot/matplot.h>
00007 #include <cassert>
00008 #include <iostream>
00009 #include <sstream>
00010
00011 #include "CoreAlgorithm/helper_func.hpp"
00012 #include "CoreAlgorithm/coeff.hpp"
00013 #include "CoreAlgorithm/source.hpp"
00014 #include "CoreAlgorithm/computational.hpp"
00015 #include "CoreAlgorithm/solver.hpp"
00016
00017 int main()
```

```
00018 {
00019     // Define the constants for the simulation
00020
00021     // Number of x and z grid points
00022     int nx = 100;
00023     int nz = 100;
00024     // Number of time steps
00025     int nt = 1000;
00026
00027     // Differentiation values
00028     double dx = 0.01;
00029     double dz = 0.01;
00030     double dt = 0.001;
00031
00032     // Define the domain
00033     double xmin = 0.0;
00034     double xmax = nx * dx;
00035     double zmin = 0.0;
00036     double zmax = nz * dz;
00037     double tmin = 0.0;
00038     double tmax = nt * dt;
00039
00040     // Define the source parameters
00041     double f_M = 10.0;
00042     double amp = 1e0;
00043     double shift = 0.1;
00044
00045     // Source location
00046     int source_is = 50;
00047     int source_js = 50;
00048
00049     // Create the source
00050     boost::multi_array<double, 3> f = waveSimCore::ricker(source_is, source_js, f_M, amp, shift, tmin,
      tmax, nt, nx, nz);
00051
00052     // Create the velocity profile
00053     double r = 150.0;
00054     boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
00055
00056     // Solve the wave equation
00057     boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
00058
00059     // Define the number of different levels for the contour plot
00060     int num_levels = 100;
00061     // Create the levels for the contour plot based on the min and max values of u
00062     double min_u = np::min(u);
00063     double max_u = np::max(u);
00064     std::vector<double> levels = matplot::linspace(min_u, max_u, num_levels);
00065
00066     // Create the x and z axis for the contour plot and convert them to matplot format
00067     boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
00068     boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);
00069     const boost::multi_array<double, 1> axis[2] = {x, z};
00070     std::vector<boost::multi_array<double, 2> XcZ = np::meshgrid(axis, false, np::xy);
00071
00072     matplot::vector_2d Xp = np::convert_to_matplot(XcZ[0]);
00073     matplot::vector_2d Zp = np::convert_to_matplot(XcZ[1]);
00074
00075     // Create the plotter object and animate the results
00076     wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
00077
00078     // If you want to render a specific frame, use this:
00079     // my_plotter.renderFrame(int frame_index);
00080
00081     // Renders the entire animation from start_frame to end_frame
00082     int start_frame = 20;
00083     int end_frame = nt - 1;
00084     int fps = 30;
00085     my_plotter.animate("example-wave.mp4", start_frame, end_frame, fps);
00086
00087     // The animation will be saved in .
00088     // Frames will be saved to ./output
00089 }
```

11.12 main.cpp

```
00001 #include <iostream>
00002 #include <string>
00003 #include "ExternalLibraries/cxxopts.hpp"
00004
00005 #include <boost/multi_array.hpp>
00006 #include <boost/array.hpp>
00007 #include "CustomLibraries/np.hpp"
```

```
00008 #include "CustomLibraries/np_to_matplot.hpp"
00009 #include "CustomLibraries/wavePlotter.hpp"
00010 #include <matplot/matplot.h>
00011 #include <cassert>
00012 #include <sstream>
00013
00014 #include "CoreAlgorithm/helper_func.hpp"
00015 #include "CoreAlgorithm/coeff.hpp"
00016 #include "CoreAlgorithm/source.hpp"
00017 #include "CoreAlgorithm/computational.hpp"
00018 #include "CoreAlgorithm/solver.hpp"
00019
00020 // Command line arguments
00021 cxxopts::Options options("WaveSimPP", "A wave propagation simulator written in C++.");
00022 int main(int argc, char *argv[])
00023 {
00024     // Parse command line arguments
00025     options.add_options()("d,debug", "Enable debugging",
       cxxopts::value<bool>()->default_value("false"));
00026     options.add_options()("animate", "Render an animation at the end",
       cxxopts::value<bool>()->default_value("true"));
00027     options.add_options()("render", "Render each of the frames at the end",
       cxxopts::value<bool>()->default_value("false"));
00028     options.add_options()("export", "Export the data to a series of csv files",
       cxxopts::value<bool>()->default_value("false"));
00029     options.add_options()("o,output_dir", "Output directory path",
       cxxopts::value<std::string>()->default_value("output"));
00030     options.add_options()("output_filename", "Output filename",
       cxxopts::value<std::string>()->default_value("output.mp4"));
00031     options.add_options()("framerate", "Framerate of output video",
       cxxopts::value<int>()->default_value("30"));
00032     options.add_options()("v,verbose", "Verbose output",
       cxxopts::value<bool>()->default_value("false"));
00033     options.add_options()("source_i", "Source i position",
       cxxopts::value<int>()->default_value("50"));
00034     options.add_options()("source_j", "Source j position",
       cxxopts::value<int>()->default_value("50"));
00035     options.add_options()("nt", "Number of time steps", cxxopts::value<int>()->default_value("1000"));
00036     options.add_options()("nx", "Number of x steps", cxxopts::value<int>()->default_value("100"));
00037     options.add_options()("nz", "Number of z steps", cxxopts::value<int>()->default_value("100"));
00038     options.add_options()("dt", "Time step size", cxxopts::value<double>()->default_value("0.001"));
00039     options.add_options()("dx", "x step size", cxxopts::value<double>()->default_value("0.01"));
00040     options.add_options()("dz", "z step size", cxxopts::value<double>()->default_value("0.01"));
00041     options.add_options()("f,frequency", "Frequency of source",
       cxxopts::value<double>()->default_value("10.0"));
00042     options.add_options()("a,amplitude", "Amplitude of source",
       cxxopts::value<double>()->default_value("1.0"));
00043     options.add_options()("s,shift", "Shift of source",
       cxxopts::value<double>()->default_value("0.1"));
00044     options.add_options()("r,radius", "Radius of velocity profile",
       cxxopts::value<double>()->default_value("150.0"));
00045     options.add_options()("l,num_levels", "Number of levels in the filled contour plot",
       cxxopts::value<int>()->default_value("100"));
00046     options.add_options()("h,help", "Print help");
00047     cxxopts::ParseResult result;
00048     try
00049     {
00050         result = options.parse(argc, argv);
00051     }
00052     catch (const cxxopts::exceptions::exception &e)
00053     {
00054         std::cerr << "WaveSimPP: " << e.what() << '\n';
00055         std::cerr << "usage: WaveSimPP [options] ...\n";
00056         return EXIT_FAILURE;
00057     }
00058     if (result.count("help"))
00059     {
00060         std::cout << options.help({"", "Group"}) << std::endl;
00061         return true;
00062     }
00063
00064     // Define the constants for the simulation
00065
00066     // Number of x and z grid points
00067
00068     int nx = result["nx"].as<int>();
00069
00070     int nz = result["nz"].as<int>();
00071     // Number of time steps
00072     int nt = result["nt"].as<int>();
00073
00074     // Differentiation values
00075     double dx = result["dx"].as<double>();
00076     double dz = result["dz"].as<double>();
00077     double dt = result["dt"].as<double>();
00078
00079     // Define the domain
```

```
00080      double xmin = 0.0;
00081      double xmax = nx * dx;
00082      double zmin = 0.0;
00083      double zmax = nz * dz;
00084      double tmin = 0.0;
00085      double tmax = nt * dt;
00086
00087      // Define the source parameters
00088      double f_M = result["frequency"].as<double>();
00089      double amp = result["amplitude"].as<double>();
00090      double shift = result["shift"].as<double>();
00091
00092      // Source location
00093      int source_is = result["source_i"].as<int>();
00094      int source_js = result["source_j"].as<int>();
00095      // Create the source
00096      boost::multi_array<double, 3> f = waveSimCore::ricker(source_is, source_js, f_M, amp, shift, tmin,
      tmax, nt, nx, nz);
00097
00098      // Create the velocity profile
00099      double r = result["radius"].as<double>();
00100
00101      boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
00102
00103      // Solve the wave equation
00104      boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
00105
00106      // Define the number of different levels for the contour plot
00107      int num_levels = result["num_levels"].as<int>();
00108      // Create the levels for the contour plot based on the min and max values of u
00109      double min_u = np::min(u);
00110      double max_u = np::max(u);
00111      std::vector<double> levels = matplot::linspace(min_u, max_u, num_levels);
00112
00113      // Create the x and z axis for the contour plot and convert them to matplot format
00114      boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
00115      boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);
00116      const boost::multi_array<double, 1> axis[2] = {x, z};
00117      std::vector<boost::multi_array<double, 2» XcZ = np::meshgrid(axis, false, np::xy);
00118
00119      matplot::vector_2d Xp = np::convert_to_matplot(XcZ[0]);
00120      matplot::vector_2d Zp = np::convert_to_matplot(XcZ[1]);
00121
00122      // Create the plotter object and animate the results
00123      wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
00124      my_plotter.setSaveDirectory(result["output_filename"].as<std::string>());
00125
00126      // If you want to render a specific frame, use this:
00127      // my_plotter.renderFrame(int frame_index);
00128
00129      // Renders the entire animation from start_frame to end_frame
00130      int start_frame = 0;
00131      int end_frame = nt;
00132      int fps = result["framerate"].as<int>();
00133      if (result["export"].as<bool>())
00134      {
00135          my_plotter.renderAllFrames(start_frame, end_frame);
00136      }
00137      else if (result["animate"].as<bool>())
00138      {
00139          my_plotter.animate(result["output_filename"].as<std::string>(), start_frame, end_frame, fps);
00140      }
00141      else
00142      {
00143          my_plotter.renderAllFrames(start_frame, end_frame);
00144      }
00145 }
```

11.13   Benchmarks.cpp

```
00001 //
00002 // Created by Yan Cheng on 12/2/22.
00003 //
00004
00005 #include <boost/multi_array.hpp>
00006 #include <boost/array.hpp>
00007 #include "CustomLibraries/np.hpp"
00008 #include "CustomLibraries/np_to_matplot.hpp"
00009 #include "CustomLibraries/wavePlotter.hpp"
00010 #include <matplot/matplot.h>
00011 #include <cassert>
00012 #include <iostream>
00013 #include <sstream>
```

```
00014 #include <chrono>
00015
00016 #include "CoreAlgorithm/helper_func.hpp"
00017 #include "CoreAlgorithm/coeff.hpp"
00018 #include "CoreAlgorithm/source.hpp"
00019 #include "CoreAlgorithm/computational.hpp"
00020 #include "CoreAlgorithm/solver.hpp"
00021
00022 void benchmark_solver()
00023 {
00024     int nx = 101;
00025     int nz = 101;
00026     int nt = 750;
00027
00028     double dx = 10;
00029     double dz = 10;
00030     double dt = 0.001;
00031
00032     double xmin = 0.0;
00033     double xmax = nx * dx;
00034     double zmin = 0.0;
00035     double zmax = nz * dz;
00036     double tmin = 0.0;
00037     double tmax = nt * dt;
00038
00039     double f_M = 10.0;
00040     double amp = 1e0;
00041     double shift = 0.1;
00042
00043     boost::multi_array<double, 3> f = waveSimCore::ricker(50, 50, f_M, amp, shift, tmin, tmax, nt, nx,
     nz);
00044
00045     double r = 150.0;
00046     boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
00047     auto start = std::chrono::high_resolution_clock::now();
00048     boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
00049     auto stop = std::chrono::high_resolution_clock::now();
00050     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
00051     std::cout « "Time taken by the solving function: " « duration.count() « " microseconds" «
     std::endl;
00052 }
00053
00054 void benchmark_gradient()
00055 {
00056     int sizex = 100;
00057     int sizey = 100;
00058     int sizez = 100;
00059     int sizet = 10;
00060     double dx = 0.1;
00061     double dy = 0.1;
00062     double dz = 0.1;
00063     double dt = 0.1;
00064     int axis[4] = {sizex, sizey, sizez, sizet};
00065     boost::multi_array<double, 4> nd_array = np::zeros<double>(axis);
00066     auto start = std::chrono::high_resolution_clock::now();
00067     auto grad = np::gradient(nd_array, {dx, dy, dz, dt});
00068     auto stop = std::chrono::high_resolution_clock::now();
00069     auto duration = std::chrono::duration_cast<std::chrono::microseconds>(stop - start);
00070     std::cout « "Time taken by the gradient function: " « duration.count() « " microseconds" «
     std::endl;
00071 }
00072
00073 int main()
00074 {
00075     benchmark_solver();
00076     benchmark_gradient();
00077 }
```

11.14 CoreTests.cpp

```
00001 //
00002 // Created by Yan Cheng on 12/2/22.
00003 //
00004
00005 #include <boost/multi_array.hpp>
00006 #include <boost/array.hpp>
00007 #include "CustomLibraries/np.hpp"
00008 #include "CustomLibraries/np_to_matplot.hpp"
00009 #include "CustomLibraries/wavePlotter.hpp"
00010 #include <matplot/matplot.h>
00011 #include <cassert>
00012 #include <iostream>
00013 #include <sstream>
```

```
00014
00015 #include "CoreAlgorithm/helper_func.hpp"
00016 #include "CoreAlgorithm/coeff.hpp"
00017 #include "CoreAlgorithm/source.hpp"
00018 #include "CoreAlgorithm/computational.hpp"
00019 #include "CoreAlgorithm/solver.hpp"
00020
00021 std::string format_num(int num, int length = 8)
00022 {
00023     std::string str_num = std::to_string(num);
00024
00025     int str_length = str_num.length();
00026     for (int i = 0; i < length - str_length; i++)
00027         str_num = "0" + str_num;
00028     return str_num;
00029 }
00030
00031 void test_()
00032 {
00033     int num_levels = 100;
00034     int nx = 100;
00035     int nz = 100;
00036     int nt = 1000;
00037
00038     double dx = 0.01;
00039     double dz = 0.01;
00040     double dt = 0.001;
00041
00042     double xmin = 0.0;
00043     double xmax = nx * dx;
00044     double zmin = 0.0;
00045     double zmax = nz * dz;
00046     double tmin = 0.0;
00047     double tmax = nt * dt;
00048
00049     double f_M = 10.0;
00050     double amp = 1e0;
00051     double shift = 0.1;
00052
00053     boost::multi_array<double, 3> f = waveSimCore::ricker(50, 50, f_M, amp, shift, tmin, tmax, nt, nx,
     nz);
00054
00055     double r = 150.0;
00056     boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
00057
00058     boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
00059     double min_u = np::min(u);
00060     double max_u = np::max(u);
00061     std::cout << "min_u = " << min_u << " max_u = " << max_u << "\n";
00062     std::vector<double> levels = matplot::linspace(min_u, max_u, num_levels);
00063     std::cout << u[20][10][10] << "\n";
00064
00065     // boost::multi_array<double, 2> u20(boost::extents[nx][nz]);
00066     // for (int i = 0; i < nx; i++)
00067     // {
00068     //     for (int j = 0; j < nz; j++)
00069     //     {
00070     //         u20[i][j] = u[10][i][j];
00071     //         std::cout << u20[i][j] << " ";
00072     //     }
00073     //     std::cout << "\n";
00074     // }
00075     // std::cout << "\n";
00076
00077     boost::multi_array<double, 1> x = np::linspace(xmin, xmax, nx);
00078     boost::multi_array<double, 1> z = np::linspace(zmin, zmax, nz);
00079     const boost::multi_array<double, 1> axis[2] = {x, z};
00080     std::vector<boost::multi_array<double, 2» XcZ = np::meshgrid(axis, false, np::xy);
00081
00082     matplot::vector_2d Xp = np::convert_to_matplot(XcZ[0]);
00083     matplot::vector_2d Zp = np::convert_to_matplot(XcZ[1]);
00084     // matplot::contourf(Xp, Zp, vel, levels, "", num_levels);
00085     // matplot::show();
00086
00087     wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
00088     // my_plotter.exportAllFrames(0, nt - 1);
00089     my_plotter.animate("output-test.mp4", 20, nt - 1, 30);
00090
00091     // for (int i = 10; i < nt - 1; i++)
00092     // {
00093     //     matplot::vector_2d Up = np::convert_to_matplot(u[i]);
00094     //     matplot::contourf(Xp, Zp, Up, levels);
00095     //     matplot::save("output/contourf_" + format_num(i) + ".png");
00096     // }
00097 }
00098
00099 int main()
```

```
00100 {
00101     test_();
00102 }
```

11.15 MatPlotTest.cpp

```cpp
00001 #include <matplot/matplot.h>
00002 #include <thread>
00003 #include "boost/multi_array.hpp"
00004 #include "boost/array.hpp"
00005 #include "CustomLibraries/np.hpp"
00006 #include "CustomLibraries/np_to_matplot.hpp"
00007
00008 using namespace matplot;
00009 void test_simple_plot()
00010 {
00011     std::vector<double> x = linspace(-2 * pi, 2 * pi);
00012     std::vector<double> y = linspace(0, 4 * pi);
00013     auto [X, Y] = meshgrid(x, y);
00014     vector_2d Z =
00015         transform(X, Y, [](double x, double y)
00016                   { return sin(x) + cos(y); });
00017     contourf(X, Y, Z, 10);
00018
00019     show();
00020 }
00021
00022 void test_conversion()
00023 {
00024     boost::multi_array<double, 1> x = np::linspace(0, 1, 100);
00025     boost::multi_array<double, 1> y = np::linspace(0, 1, 100);
00026     // x = np::pow(x, 2.0);
00027     // y = np::pow(y, 3.0);
00028
00029     const boost::multi_array<double, 1> axis[2] = {x, y};
00030     std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
00031
00032     double dx, dy;
00033     dx = 1.0 / 100.0;
00034     dy = 1.0 / 100.0;
00035
00036     boost::multi_array<double, 2> f = np::pow(XcY[0], 2.0) + XcY[0] * np::pow(XcY[1], 1.0);
00037
00038     // g.push_back(np::gradient(XcY[0], {dx, dy}));
00039     // g.push_back(np::gradient(XcY[1], {dx, dy}));
00040     std::vector<boost::multi_array<double, 2» gradf = np::gradient(f, {dx, dy});
00041     matplot::vector_2d Xp = np::convert_to_matplot(XcY[0]);
00042     matplot::vector_2d Yp = np::convert_to_matplot(XcY[1]);
00043     matplot::vector_2d X = matplot::meshgrid(linspace(0, 1, 100), linspace(0, 1, 100)).first;
00044     matplot::vector_2d Y = matplot::meshgrid(linspace(0, 1, 100), linspace(0, 1, 100)).second;
00045
00046     matplot::vector_2d Z = np::convert_to_matplot(gradf[1]);
00047     std::cout « "X.size() = " « X.size() « std::endl;
00048     std::cout « "Y.size() = " « Y.size() « std::endl;
00049     std::cout « "Z.size() = " « Z.size() « std::endl;
00050     for (size_t i = 0; i < X.size(); i++)
00051     {
00052         for (size_t j = 0; j < X[i].size(); j++)
00053         {
00054             std::cout « "X[" « i « "][" « j « "] = " « X[i][j] « " | XP[" « i « "][" « j « "] = " «
    Xp[i][j] « " | YP[" « i « "][" « j « "] = " « Yp[i][j] « std::endl;
00055         }
00056     }
00057     contourf(Xp, Yp, Z, 10);
00058     show();
00059 }
00060
00061 int main()
00062 {
00063     // test_simple_plot();
00064     test_conversion();
00065     return 0;
00066 }
```

11.16 variadic.cpp

```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 of f(x,y) = x^2 + xy at x = " « x[i] « " and y = " « y[j] « " is equal to " «
        gradf[0][i][j];
00255
00256        std::cout « "\n";
00257 }
00258
00259 void test_abs()
00260 {
00261        int nx = 4;
00262        int ny = 4;
00263        boost::multi_array<double, 1> x = np::linspace(-1, 1, nx);
```

```
00264       boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
00265       const boost::multi_array<double, 1> axis[2] = {x, y};
00266       std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
00267       boost::multi_array<double, 2> abs_f = np::abs(XcY[0]);
00268       std::cout « "abs_f: \n";
00269       for (int i = 0; i < ny; i++)
00270       {
00271           for (int j = 0; j < nx; j++)
00272           {
00273               std::cout « abs_f[i][j] « " ";
00274           }
00275           std::cout « "\n";
00276       }
00277 }
00278
00279 void test_slice()
00280 {
00281     int nx = 4;
00282     int ny = 4;
00283     boost::multi_array<double, 1> x = np::linspace(-1, 1, nx);
00284     boost::multi_array<double, 1> y = np::linspace(-1, 1, ny);
00285     const boost::multi_array<double, 1> axis[2] = {x, y};
00286     std::vector<boost::multi_array<double, 2» XcY = np::meshgrid(axis, false, np::xy);
00287     boost::multi_array<double, 2> f = np::pow(XcY[0], 2.0) + XcY[0] * np::pow(XcY[1], 1.0);
00288     std::cout « "f: \n";
00289     for (int i = 0; i < ny; i++)
00290     {
00291         for (int j = 0; j < nx; j++)
00292         {
00293             std::cout « f[i][j] « " ";
00294         }
00295         std::cout « "\n";
00296     }
00297     std::cout « "f[0]: \n";
00298     boost::multi_array<double, 1> f_slice = np::slice(f, 0);
00299     for (int i = 0; i < nx; i++)
00300     {
00301         std::cout « f_slice[i] « " ";
00302     }
00303     std::cout « "\n";
00304
00305     std::cout « "f[1]: \n";
00306     f_slice = np::slice(f, 1);
00307     for (int i = 0; i < ny; i++)
00308     {
00309         std::cout « f_slice[i] « " ";
00310     }
00311     std::cout « "\n";
00312 }
00313
00314 int main()
00315 {
00316     test_gradient();
00317     test_meshgrid();
00318     test_complex_operations();
00319     test_equal();
00320     test_basic_operations();
00321     test_zeros();
00322     test_min_max();
00323     test_abs();
00324     test_toy_problem();
00325     test_slice();
00326 }
```