WaveSimC

1.0

Generated by Doxygen 1.9.6

Chapter 1

Main Page

1.1 COMSW4995 Final Project: WaveSimC

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

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

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

Please check the Readme file for more information.

1.1.1 Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

1.1.2 License

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 WaveSimCExec
```

And then run it with the following command:

```
./WaveSimCExec
```

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:

```
./WaveSimCExec -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.

```
 {c++}
// 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

```
{c++}
// Create the source
boost::multi_array<double, 3> f = waveSimCore::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 = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);
```

Then we can proceed to solve the wave equation using the wave solver.

```
{c++}
// Solve the wave equation
boost::multi_array<double, 3> u = waveSimCore::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:

```
{c++}
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.

```
{c++}
// 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:

```
{c++}
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

```
 {c++}
// 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.

```
 {c++}
// 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: WaveSimC

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

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

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

`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 WaveSimCExec
```

3.3.4 Running the executable

```
./WaveSimCExec
```

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 ../WaveSimC-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.

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

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.

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

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.

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

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.

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

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.

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

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.

```
{
    boost::multi_array<double, 2> c(boost::extents[nx][nz]);

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

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

    double x_0 = xmax / 2.0;
    double z_0 = zmax / 2.0;
```

```
        for (int i = 0; i < nx; i++)
        {
            for (int j = 0; j < nz; j++)
            {
                if (np::pow(XZ[0][i][j] - x_0, 2.0) + np::pow(XZ[1][i][j] - z_0, 2.0) <= np::pow(r, 2.0))
                    c[i][j] = 3.0;
                else
                    c[i][j] = 3.0;
            }
        }

        return c;
    }
```

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.

```
    {
        boost::multi_array<double, 2> sigma_1(boost::extents[nx][nz]);
        const double PML_width = n * dx;

        const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);

        const double x_0 = np::max(x) - PML_width;

        boost::multi_array<double, 1> polynomial(boost::extents[nx]);

        for (int i = 0; i < nx; i++)
        {
            if (x[i] > x_0)
            {
                polynomial[i] = sigma_max * np::pow(np::abs(x[i] - x_0), m);
                polynomial[nx - 1 - i] = polynomial[i];
            }
            else
            {
                polynomial[i] = 0;
            }
        }

        for (int i = 0; i < nx; i++)
            for (int j = 0; j < nz; j++)
                sigma_1[i][j] = polynomial[i];

        return sigma_1;
    }
```

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.

```
    {
        boost::multi_array<double, 2> sigma_2(boost::extents[nx][nz]);
        const double PML_width = n * dz;
        const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);

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

        boost::multi_array<double, 1> polynomial(boost::extents[nz]);
        for (int j = 0; j < nz; j++)
        {
            if (z[j] > z_0)
            {
                polynomial[j] = sigma_max * np::pow(np::abs(z[j] - z_0), m);
                polynomial[nz - 1 - j] = polynomial[j];
            }
            else
            {
                polynomial[j] = 0;
            }
        }

        for (int i = 0; i < nx; i++)
            for (int j = 0; j < nz; j++)
                sigma_2[i][j] = polynomial[j];

        return sigma_2;
    }
```

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.

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

```

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

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

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

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

        const boost::multi_array<double, 1> axis[3] = {r, x, z};
        std::vector<boost::multi_array<double, 3>> RXZ = np::meshgrid(axis, false, np::ij);
        boost::multi_array<double, 3> source = RXZ[0] * RXZ[1] * RXZ[2];

        return source;
    }
```

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.

```
    {

        const boost::multi_array<double, 2> CX = np::pow(c * dt / dx, 2.0);
        const boost::multi_array<double, 2> CZ = np::pow(c * dt / dz, 2.0);
        const double Cf = np::pow(dt, 2.0);

        int dimensions_1[] = {nt, nx, nz};
        boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
        int dimensions_4[] = {nx, nz};
        boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
        int dimensions_5[] = {nx, nz};
        boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);

        for (int n = 1; n < nt - 1; n++)
        {
            for (int i = 1; i < nx - 1; i++)
            {
                for (int j = 1; j < nz - 1; j++)
                {
                    u_xx[i][j] = u[n][i + 1][j] + u[n][i - 1][j] - 2.0 * u[n][i][j];
                    u_zz[i][j] = u[n][i][j + 1] + u[n][i][j - 1] - 2.0 * u[n][i][j];
                }
            }

            // ! Update u
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    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];
                }
            }

            // Dirichlet boundary condition
            for (int i = 0; i < nx; i++)
```

```
            {
                u[n + 1][i][0] = 0.0;
                u[n + 1][i][nz - 1] = 0.0;
            }
            for (int j = 0; j < nz; j++)
            {
                u[n + 1][0][j] = 0.0;
                u[n + 1][nx - 1][j] = 0.0;
            }
        }
        return u;
    }
```

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.

```
    {

        const boost::multi_array<double, 2> C1 = 1.0 + (dt * (sigma_1 + sigma_2) * 0.5);
        const boost::multi_array<double, 2> C2 = (sigma_1 * sigma_2 * np::pow(dt, 2.0)) - 2.0;
        const boost::multi_array<double, 2> C3 = 1.0 - (dt * (sigma_1 + sigma_2) * 0.5);
        const boost::multi_array<double, 2> C4 = np::pow(dt * c, 2.0);
        const boost::multi_array<double, 2> C5 = 1.0 + (dt * sigma_1 * 0.5);
        const boost::multi_array<double, 2> C6 = 1.0 + (dt * sigma_2 * 0.5);
        const boost::multi_array<double, 2> C7 = 1.0 - (dt * sigma_1 * 0.5);
        const boost::multi_array<double, 2> C8 = 1.0 - (dt * sigma_2 * 0.5);

        int dimensions_1[] = {nt, nx, nz};
        boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);

        int dimensions_2[] = {nx, nz};
        boost::multi_array<double, 2> q_1 = np::zeros<double>(dimensions_2);
        int dimensions_3[] = {nx, nz};
        boost::multi_array<double, 2> q_2 = np::zeros<double>(dimensions_3);

        int dimensions_4[] = {nx, nz};
        boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
        int dimensions_5[] = {nx, nz};
        boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);

        boost::multi_array<double, 2> f_n(boost::extents[nx][nz]);
        boost::multi_array<double, 2> u_n(boost::extents[nx][nz]);
        boost::multi_array<double, 2> u_n_1(boost::extents[nx][nz]);

        for (int n = 1; n < nt - 1; n++)
        {

            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    f_n[i][j] = f[n][i][j];
                    u_n[i][j] = u[n][i][j];
                    u_n_1[i][j] = u[n - 1][i][j];
                }
            }

```

```
            boost::multi_array<double, 2> div = divergence(q_1 * sigma_1, q_2 * sigma_2, dx, dz);
            boost::multi_array<double, 2> dq_1dx = dfdx(q_1, dx);
            boost::multi_array<double, 2> dq_2dz = dfdz(q_2, dz);
            u_xx = d2fdx2(u_n, dx); // (nx, nz)
            u_zz = d2fdz2(u_n, dz); // (nx, nz)
            boost::multi_array<double, 2> dudx = dfdx(u_n, dx);
            boost::multi_array<double, 2> dudz = dfdz(u_n, dz);

            //        for (int i = 1; i < nx-1; i++)
            //        {
            //            for (int j = 1; j < nz-1; j++)
            //            {
            //                u_xx[i][j] = u_n[i+1][j] + u_n[i-1][j] - 2.0 * u_n[i][j];
            //                u_zz[i][j] = u_n[i][j+1] + u_n[i][j-1] - 2.0 * u_n[i][j];
            //            }
            //        }

            // ! Update u
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    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];
                }
            }

            // ! Update q_1, q_2
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    q_1[i][j] = (dt * dudx[i][j] + C7[i][j] * q_1[i][j]) / C5[i][j];
                    q_2[i][j] = (dt * dudz[i][j] + C8[i][j] * q_2[i][j]) / C6[i][j];
                }
            }
            //
            // Dirichlet boundary condition
            for (int i = 0; i < nx; i++)
            {
                u[n + 1][i][0] = 0.0;
                u[n + 1][i][nz - 1] = 0.0;
            }
            for (int j = 0; j < nz; j++)
            {
                u[n + 1][0][j] = 0.0;
                u[n + 1][nx - 1][j] = 0.0;
            }
        }
        return u;
    }
```

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.

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

```
        // std::cout « "arr.shape()[0] = " « arr.shape()[0] « " arr.shape()[1] = " « arr.shape()[1] «
    std::endl;
        for (size_t i = 0; i < arr.shape()[0]; i++)
        {
            for (size_t j = 0; j < arr.shape()[1]; j++)
            {
                result[i][j] = arr[i][j];
            }
        }
        return result;
    }
```

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.

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

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.

```
{
    return rhs * lhs;
}
```

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

```
        {
            renderAllFrames(begin_frame_index, end_frame_index);
            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;
            std::system(ffmpeg_render_command.c_str());
        }
```

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.

```
        {
            for (int i = begin_frame_index; i < end_frame_index; i++)
            {
                renderFrame(i);
            }
        }
```

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.

```
        {
            matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
            matplot::contourf(this->Xp, this->Zp, Up, this->levels);
            matplot::save(save_directory + "/contourf_" + format_num(index) + ".png");
        }
```

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.

```
        {
            this->save_directory = save_directory;
        }
```

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.

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

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.

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

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.

```
{

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

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

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.

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

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

        return _index;
    }
```

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.

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

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

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

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

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

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.

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

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.

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

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.

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

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.

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

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.

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

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.

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

    // Creating the output arrays
    for (std::size_t i = 0; i < ND; i++)
    {
        boost::multi_array<T, ND> output_array(output_extents);
        ndArrayValue *p = output_array.data();
        ndIndexArray index;
        // Looping through the elements of the output array
        for (std::size_t j = 0; j < output_array.num_elements(); j++)
        {
            index = getIndexArray(output_array, p);
            oneDArrayIndex index_1d;
            index_1d = index[i];
            output_array(index) = ci[i][index_1d];
            ++p;
        }
        output_arrays.push_back(output_array);
    }
    if (indexing_type == xy && (ND == 3 || ND == 2))
    {
        std::swap(output_arrays[0], output_arrays[1]);
    }
    return output_arrays;
}
```

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.

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

```
    {
        T element = *data_pointer;
        if (min_not_set || element < min)
        {
            min = element;
            min_not_set = false;
        }
        ++data_pointer;
    }
    return min;
}
```

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.

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

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

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.

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

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.

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

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.

```
{

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

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

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

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.

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

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.

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

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.

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

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

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

#ifndef WAVESIMC_COEFF_HPP
#define WAVESIMC_COEFF_HPP

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

namespace waveSimCore
{
    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> sigma_1(boost::extents[nx][nz]);
        const double PML_width = n * dx;

        const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);

        const double x_0 = np::max(x) - PML_width;

        boost::multi_array<double, 1> polynomial(boost::extents[nx]);

        for (int i = 0; i < nx; i++)
        {
            if (x[i] > x_0)
            {
                polynomial[i] = sigma_max * np::pow(np::abs(x[i] - x_0), m);
                polynomial[nx - 1 - i] = polynomial[i];
            }
            else
            {
                polynomial[i] = 0;
            }
        }

        for (int i = 0; i < nx; i++)
            for (int j = 0; j < nz; j++)
                sigma_1[i][j] = polynomial[i];

        return sigma_1;
    }

    boost::multi_array<double, 2> get_sigma_2(boost::multi_array<double, 1> z, double dz, int nx, int nz,
                                               double c_max, int n = 10, double R = 1e-3, double m = 2.0)
    {
        boost::multi_array<double, 2> sigma_2(boost::extents[nx][nz]);
        const double PML_width = n * dz;
        const double sigma_max = -c_max * log(R) * (m + 1.0) / np::pow(PML_width, m + 1.0);

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

        boost::multi_array<double, 1> polynomial(boost::extents[nz]);
```

```
        for (int j = 0; j < nz; j++)
        {
            if (z[j] > z_0)
            {
                polynomial[j] = sigma_max * np::pow(np::abs(z[j] - z_0), m);
                polynomial[nz - 1 - j] = polynomial[j];
            }
            else
            {
                polynomial[j] = 0;
            }
        }

        for (int i = 0; i < nx; i++)
            for (int j = 0; j < nz; j++)
                sigma_2[i][j] = polynomial[j];

        return sigma_2;
    }
}
#endif // WAVESIMC_COEFF_HPP
```

11.2 computational.hpp

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

#ifndef WAVESIMC_COMPUTATIONAL_HPP
#define WAVESIMC_COMPUTATIONAL_HPP

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

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

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

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

        for (int i = 0; i < nx; i++)
        {
            for (int j = 0; j < nz; j++)
            {
                if (np::pow(XZ[0][i][j] - x_0, 2.0) + np::pow(XZ[1][i][j] - z_0, 2.0) <= np::pow(r,
  2.0))
                    c[i][j] = 3.0;
                else
                    c[i][j] = 3.0;
            }
        }

        return c;
    }
}
#endif // WAVESIMC_COMPUTATIONAL_HPP
```

11.3 helper_func.hpp

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

#ifndef WAVESIMC_HELPER_FUNC_HPP
#define WAVESIMC_HELPER_FUNC_HPP

#include "CustomLibraries/np.hpp"

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

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

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

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

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

    boost::multi_array<double, 2> divergence(boost::multi_array<double, 2> f1,
 boost::multi_array<double, 2> f2,
                                             double dx, double dz)
    {
        boost::multi_array<double, 2> f_x = dfdx(f1, dx);
        boost::multi_array<double, 2> f_z = dfdz(f2, dz);
        boost::multi_array<double, 2> div = f_x + f_z;
        return div;
    }

}
#endif // WAVESIMC_HELPER_FUNC_HPP
```

11.4 solver.hpp

```
//
// Created by Yan Cheng on 12/6/22.
//

#ifndef CORETESTS_CPP_SOLVER2_HPP
#define CORETESTS_CPP_SOLVER2_HPP

#include "CustomLibraries/np.hpp"

namespace waveSimCore
{
    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)
    {

        const boost::multi_array<double, 2> CX = np::pow(c * dt / dx, 2.0);
        const boost::multi_array<double, 2> CZ = np::pow(c * dt / dz, 2.0);
        const double Cf = np::pow(dt, 2.0);

        int dimensions_1[] = {nt, nx, nz};
        boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);
        int dimensions_4[] = {nx, nz};
        boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
        int dimensions_5[] = {nx, nz};
        boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);

        for (int n = 1; n < nt - 1; n++)
        {
            for (int i = 1; i < nx - 1; i++)
            {
                for (int j = 1; j < nz - 1; j++)
                {
                    u_xx[i][j] = u[n][i + 1][j] + u[n][i - 1][j] - 2.0 * u[n][i][j];
                    u_zz[i][j] = u[n][i][j + 1] + u[n][i][j - 1] - 2.0 * u[n][i][j];
                }
            }

            // ! Update u
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    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];
```

```
                    }
                }

                // Dirichlet boundary condition
                for (int i = 0; i < nx; i++)
                {
                    u[n + 1][i][0] = 0.0;
                    u[n + 1][i][nz - 1] = 0.0;
                }
                for (int j = 0; j < nz; j++)
                {
                    u[n + 1][0][j] = 0.0;
                    u[n + 1][nx - 1][j] = 0.0;
                }
            }
            return u;
        }
}
#endif // CORETESTS_CPP_SOLVER2_HPP
```

11.5 solver_complex.hpp

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

#ifndef WAVESIMC_SOLVER_COMPLEX_HPP
#define WAVESIMC_SOLVER_COMPLEX_HPP

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

#include <cmath>

namespace waveSimCore
{
    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)
    {

        const boost::multi_array<double, 2> C1 = 1.0 + (dt * (sigma_1 + sigma_2) * 0.5);
        const boost::multi_array<double, 2> C2 = (sigma_1 * sigma_2 * np::pow(dt, 2.0)) - 2.0;
        const boost::multi_array<double, 2> C3 = 1.0 - (dt * (sigma_1 + sigma_2) * 0.5);
        const boost::multi_array<double, 2> C4 = np::pow(dt * c, 2.0);
        const boost::multi_array<double, 2> C5 = 1.0 + (dt * sigma_1 * 0.5);
        const boost::multi_array<double, 2> C6 = 1.0 + (dt * sigma_2 * 0.5);
        const boost::multi_array<double, 2> C7 = 1.0 - (dt * sigma_1 * 0.5);
        const boost::multi_array<double, 2> C8 = 1.0 - (dt * sigma_2 * 0.5);

        int dimensions_1[] = {nt, nx, nz};
        boost::multi_array<double, 3> u = np::zeros<double>(dimensions_1);

        int dimensions_2[] = {nx, nz};
        boost::multi_array<double, 2> q_1 = np::zeros<double>(dimensions_2);
        int dimensions_3[] = {nx, nz};
        boost::multi_array<double, 2> q_2 = np::zeros<double>(dimensions_3);

        int dimensions_4[] = {nx, nz};
        boost::multi_array<double, 2> u_xx = np::zeros<double>(dimensions_4);
        int dimensions_5[] = {nx, nz};
        boost::multi_array<double, 2> u_zz = np::zeros<double>(dimensions_5);

        boost::multi_array<double, 2> f_n(boost::extents[nx][nz]);
        boost::multi_array<double, 2> u_n(boost::extents[nx][nz]);
        boost::multi_array<double, 2> u_n_1(boost::extents[nx][nz]);

        for (int n = 1; n < nt - 1; n++)
        {

            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    f_n[i][j] = f[n][i][j];
                    u_n[i][j] = u[n][i][j];
                    u_n_1[i][j] = u[n - 1][i][j];
                }
            }

```

```
            boost::multi_array<double, 2> div = divergence(q_1 * sigma_1, q_2 * sigma_2, dx, dz);
            boost::multi_array<double, 2> dq_1dx = dfdx(q_1, dx);
            boost::multi_array<double, 2> dq_2dz = dfdz(q_2, dz);
            u_xx = d2fdx2(u_n, dx); // (nx, nz)
            u_zz = d2fdz2(u_n, dz); // (nx, nz)
            boost::multi_array<double, 2> dudx = dfdx(u_n, dx);
            boost::multi_array<double, 2> dudz = dfdz(u_n, dz);

            //        for (int i = 1; i < nx-1; i++)
            //        {
            //            for (int j = 1; j < nz-1; j++)
            //            {
            //                u_xx[i][j] = u_n[i+1][j] + u_n[i-1][j] - 2.0 * u_n[i][j];
            //                u_zz[i][j] = u_n[i][j+1] + u_n[i][j-1] - 2.0 * u_n[i][j];
            //            }
            //        }

            // ! Update u
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    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];
                }
            }

            // ! Update q_1, q_2
            for (int i = 0; i < nx; i++)
            {
                for (int j = 0; j < nz; j++)
                {
                    q_1[i][j] = (dt * dudx[i][j] + C7[i][j] * q_1[i][j]) / C5[i][j];
                    q_2[i][j] = (dt * dudz[i][j] + C8[i][j] * q_2[i][j]) / C6[i][j];
                }
            }
            //
            // Dirichlet boundary condition
            for (int i = 0; i < nx; i++)
            {
                u[n + 1][i][0] = 0.0;
                u[n + 1][i][nz - 1] = 0.0;
            }
            for (int j = 0; j < nz; j++)
            {
                u[n + 1][0][j] = 0.0;
                u[n + 1][nx - 1][j] = 0.0;
            }
        }
        return u;
    }
}
#endif // WAVESIMC_SOLVER_HPP
```

11.6 source.hpp

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

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

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

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

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

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

```

```
        const boost::multi_array<double, 1> axis[3] = {r, x, z};
        std::vector<boost::multi_array<double, 3» RXZ = np::meshgrid(axis, false, np::ij);
        boost::multi_array<double, 3> source = RXZ[0] * RXZ[1] * RXZ[2];

        return source;
    }
}
#endif // WAVESIMC_SOURCE_HPP
```

11.7 wave.cpp

```
// Standard IO libraries
#include <iostream>
#include <fstream>

#include "CustomLibraries/np.hpp"

#include <math.h>

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

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

11.8 np.hpp

```
#ifndef NP_H_
#define NP_H_

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

namespace np
{

    typedef double ndArrayValue;

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

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

        return _index;
    }

    template <typename Array, typename Element, typename Functor>
```

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

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

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

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

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

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

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

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

```
                }
                else
                {
                    dh_low = 0;
                }

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

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

    enum indexing
    {
        xy,
        ij
    };

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

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

```
        if (indexing_type == xy && (ND == 3 || ND == 2))
        {
            std::swap(output_arrays[0], output_arrays[1]);
        }
        return output_arrays;
    }

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

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

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

    // Complex operations

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

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

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

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

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

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

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

```
    }

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

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

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

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

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

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

```cpp
    }

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

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

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

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

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

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

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

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

}

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

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

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

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

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

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

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

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

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

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

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

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

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

```
    return np::element_wise_apply(lhs, func);
}

#endif
```

11.9 np_to_matplot.hpp

```
#ifndef NPTOMATPLOT_H_
#define NPTOMATPLOT_H_

#include <matplot/matplot.h>
#include <thread>
#include "boost/multi_array.hpp"
#include "boost/array.hpp"
namespace np
{
    inline matplot::vector_2d convert_to_matplot(const boost::multi_array<double, 2> &arr)
    {
        std::vector<double> x = matplot::linspace(0, 0, arr.shape()[0]);
        std::vector<double> y = matplot::linspace(00, 0, arr.shape()[1]);
        matplot::vector_2d result = std::get<0>(matplot::meshgrid(x, y));
        // std::cout « "arr.shape()[0] = " « arr.shape()[0] « " arr.shape()[1] = " « arr.shape()[1] «
    std::endl;
        for (size_t i = 0; i < arr.shape()[0]; i++)
        {
            for (size_t j = 0; j < arr.shape()[1]; j++)
            {
                result[i][j] = arr[i][j];
            }
        }
        return result;
    }
}
#endif
```

11.10 wavePlotter.hpp

```
#ifndef WAVESOLVER_H_
#define WAVESOLVER_H_
#include <boost/multi_array.hpp>
#include <boost/array.hpp>
#include <fstream>
#include <iostream>
#include "CustomLibraries/np.hpp"
#include "CustomLibraries/np_to_matplot.hpp"

namespace wavePlotter
{
    class Plotter
    {
    public:
        Plotter(const boost::multi_array<double, 3> &u, const matplot::vector_2d &Xp, const
    matplot::vector_2d &Zp, int num_levels, int nt)
        {
            this->u.resize(boost::extents[u.shape()[0]][u.shape()[1]][u.shape()[2]]);
            this->u = u;
            this->Xp = Xp;
            this->Zp = Zp;
            this->num_levels = num_levels;
            this->nt = nt;
            double min_u = np::min(u);
            double max_u = np::max(u);
            std::cout « "min_u = " « min_u « " max_u = " « max_u « "\n";
            this->levels = matplot::linspace(min_u, max_u, num_levels);
        }
        void renderFrame(int index)
        {
            matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
            matplot::contourf(this->Xp, this->Zp, Up, this->levels);
            matplot::save(save_directory + "/contourf_" + format_num(index) + ".png");
        }

        void renderAllFrames(int begin_frame_index, int end_frame_index)
        {
            for (int i = begin_frame_index; i < end_frame_index; i++)
            {
                renderFrame(i);
            }
        }
```

```

        void animate(std::string output_file_name, int begin_frame_index, int end_frame_index, int frame_rate)
        {
            renderAllFrames(begin_frame_index, end_frame_index);
            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;
            std::system(ffmpeg_render_command.c_str());
        }
        void exportFrame(int index)
        {
            matplot::vector_2d Up = np::convert_to_matplot(this->u[index]);
            std::ofstream outfile;
            outfile.open(save_directory + "/frame_" + format_num(index) + ".csv");
            for (std::size_t i = 0; i < Up.size(); i++)
            {
                for (std::size_t j = 0; j < Up[i].size(); j++)
                {
                    outfile « Up[i][j];
                    if (j != Up[i].size() - 1)
                        outfile « ",";
                }
                outfile « "\n";
            }
            outfile.close();
        }

        void exportAllFrames(int begin_frame_index, int end_frame_index)
        {
            for (int i = begin_frame_index; i < end_frame_index; i++)
            {
                exportFrame(i);
            }
        }

        void setSaveDirectory(std::string save_directory)
        {
            this->save_directory = save_directory;
        }

    private:
        std::string format_num(int num, int length = 8)
        {
            std::string str_num = std::to_string(num);

            int str_length = str_num.length();
            for (int i = 0; i < length - str_length; i++)
                str_num = "0" + str_num;
            return str_num;
        }

        boost::multi_array<double, 3> u;
        matplot::vector_2d Xp;
        matplot::vector_2d Zp;
        int num_levels;
        int nt;
        std::vector<double> levels;
        std::string save_directory = "output";
    };

}
#endif
```

11.11 wave_solver_with_animation.cpp

```
#include <boost/multi_array.hpp>
#include <boost/array.hpp>
#include "CustomLibraries/np.hpp"
#include "CustomLibraries/np_to_matplot.hpp"
#include "CustomLibraries/wavePlotter.hpp"
#include <matplot/matplot.h>
#include <cassert>
#include <iostream>
#include <sstream>

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

int main()
```

```
{
    // 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;

    // Create the source
    boost::multi_array<double, 3> f = waveSimCore::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 = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);

    // Solve the wave equation
    boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);

    // 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]);

    // 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 .
    // Frames will be saved to ./output
}
```

11.12 main.cpp

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

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

```
#include "CustomLibraries/np_to_matplot.hpp"
#include "CustomLibraries/wavePlotter.hpp"
#include <matplot/matplot.h>
#include <cassert>
#include <sstream>

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

// Command line arguments
cxxopts::Options options("WaveSimC", "A wave propagation simulator written in C++.");
int main(int argc, char *argv[])
{
    // Parse command line arguments
    options.add_options()("d,debug", "Enable debugging", cxxopts::value<bool>()->default_value("false"));
    options.add_options()("animate", "Render an animation at the end", cxxopts::value<bool>()->default_value("true"));
    options.add_options()("render", "Render each of the frames at the end", cxxopts::value<bool>()->default_value("false"));
    options.add_options()("export", "Export the data to a series of csv files", cxxopts::value<bool>()->default_value("false"));
    options.add_options()("o,output_dir", "Output directory path", cxxopts::value<std::string>()->default_value("output"));
    options.add_options()("output_filename", "Output filename", cxxopts::value<std::string>()->default_value("output.mp4"));
    options.add_options()("framerate", "Framerate of output video", cxxopts::value<int>()->default_value("30"));
    options.add_options()("v,verbose", "Verbose output", cxxopts::value<bool>()->default_value("false"));
    options.add_options()("source_i", "Source i position", cxxopts::value<int>()->default_value("50"));
    options.add_options()("source_j", "Source j position", cxxopts::value<int>()->default_value("50"));
    options.add_options()("nt", "Number of time steps", cxxopts::value<int>()->default_value("1000"));
    options.add_options()("nx", "Number of x steps", cxxopts::value<int>()->default_value("100"));
    options.add_options()("nz", "Number of z steps", cxxopts::value<int>()->default_value("100"));
    options.add_options()("dt", "Time step size", cxxopts::value<double>()->default_value("0.001"));
    options.add_options()("dx", "x step size", cxxopts::value<double>()->default_value("0.01"));
    options.add_options()("dz", "z step size", cxxopts::value<double>()->default_value("0.01"));
    options.add_options()("f,frequency", "Frequency of source", cxxopts::value<double>()->default_value("10.0"));
    options.add_options()("a,amplitude", "Amplitude of source", cxxopts::value<double>()->default_value("1.0"));
    options.add_options()("s,shift", "Shift of source", cxxopts::value<double>()->default_value("0.1"));
    options.add_options()("r,radius", "Radius of velocity profile", cxxopts::value<double>()->default_value("150.0"));
    options.add_options()("l,num_levels", "Number of levels in the filled contour plot", cxxopts::value<int>()->default_value("100"));
    options.add_options()("h,help", "Print help");
    cxxopts::ParseResult result;
    try
    {
        result = options.parse(argc, argv);
    }
    catch (const cxxopts::exceptions::exception &e)
    {
        std::cerr « "WaveSimC: " « e.what() « '\n';
        std::cerr « "usage: WaveSimC [options] ...\n";
        return EXIT_FAILURE;
    }
    if (result.count("help"))
    {
        std::cout « options.help({"", "Group"}) « std::endl;
        return true;
    }

    // Define the constants for the simulation

    // Number of x and z grid points

    int nx = result["nx"].as<int>();

    int nz = result["nz"].as<int>();
    // Number of time steps
    int nt = result["nt"].as<int>();

    // Differentiation values
    double dx = result["dx"].as<double>();
    double dz = result["dz"].as<double>();
    double dt = result["dt"].as<double>();

    // 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 = result["frequency"].as<double>();
    double amp = result["amplitude"].as<double>();
    double shift = result["shift"].as<double>();

    // Source location
    int source_is = result["source_i"].as<int>();
    int source_js = result["source_j"].as<int>();
    // Create the source
    boost::multi_array<double, 3> f = waveSimCore::ricker(source_is, source_js, f_M, amp, shift, tmin,
tmax, nt, nx, nz);

    // Create the velocity profile
    double r = result["radius"].as<double>();

    boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);

    // Solve the wave equation
    boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);

    // Define the number of different levels for the contour plot
    int num_levels = result["num_levels"].as<int>();
    // 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]);

    // Create the plotter object and animate the results
    wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
    my_plotter.setSaveDirectory(result["output_filename"].as<std::string>());

    // 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 = 0;
    int end_frame = nt;
    int fps = result["framerate"].as<int>();
    if (result["export"].as<bool>())
    {
        my_plotter.renderAllFrames(start_frame, end_frame);
    }
    else if (result["animate"].as<bool>())
    {
        my_plotter.animate(result["output_filename"].as<std::string>(), start_frame, end_frame, fps);
    }
    else
    {
        my_plotter.renderAllFrames(start_frame, end_frame);
    }
}
```

11.13 CoreTests.cpp

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

#include <boost/multi_array.hpp>
#include <boost/array.hpp>
#include "CustomLibraries/np.hpp"
#include "CustomLibraries/np_to_matplot.hpp"
#include "CustomLibraries/wavePlotter.hpp"
#include <matplot/matplot.h>
#include <cassert>
#include <iostream>
#include <sstream>
```

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

std::string format_num(int num, int length = 8)
{
    std::string str_num = std::to_string(num);

    int str_length = str_num.length();
    for (int i = 0; i < length - str_length; i++)
        str_num = "0" + str_num;
    return str_num;
}

void test_()
{
    int num_levels = 100;
    int nx = 100;
    int nz = 100;
    int nt = 1000;

    double dx = 0.01;
    double dz = 0.01;
    double dt = 0.001;

    double xmin = 0.0;
    double xmax = nx * dx;
    double zmin = 0.0;
    double zmax = nz * dz;
    double tmin = 0.0;
    double tmax = nt * dt;

    double f_M = 10.0;
    double amp = 1e0;
    double shift = 0.1;

    boost::multi_array<double, 3> f = waveSimCore::ricker(50, 50, f_M, amp, shift, tmin, tmax, nt, nx, nz);

    double r = 150.0;
    boost::multi_array<double, 2> vel = waveSimCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);

    boost::multi_array<double, 3> u = waveSimCore::wave_solver(vel, dt, dx, dz, nt, nx, nz, f);
    double min_u = np::min(u);
    double max_u = np::max(u);
    std::cout « "min_u = " « min_u « " max_u = " « max_u « "\n";
    std::vector<double> levels = matplot::linspace(min_u, max_u, num_levels);
    std::cout « u[20][10][10] « "\n";

    // boost::multi_array<double, 2> u20(boost::extents[nx][nz]);
    // for (int i = 0; i < nx; i++)
    // {
    //     for (int j = 0; j < nz; j++)
    //     {
    //         u20[i][j] = u[10][i][j];
    //         std::cout « u20[i][j] « " ";
    //     }
    //     std::cout « "\n";
    // }
    // std::cout « "\n";

    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]);
    // matplot::contourf(Xp, Zp, vel, levels, "", num_levels);
    // matplot::show();

    wavePlotter::Plotter my_plotter(u, Xp, Zp, num_levels, nt);
    // my_plotter.exportAllFrames(0, nt - 1);
    my_plotter.animate("output-test.mp4", 20, nt - 1, 30);

    // for (int i = 10; i < nt - 1; i++)
    // {
    //     matplot::vector_2d Up = np::convert_to_matplot(u[i]);
    //     matplot::contourf(Xp, Zp, Up, levels);
    //     matplot::save("output/contourf_" + format_num(i) + ".png");
    // }
}

int main()
```

```
{
    test_();
}
```

11.14 MatPlotTest.cpp

```
#include <matplot/matplot.h>
#include <thread>
#include "boost/multi_array.hpp"
#include "boost/array.hpp"
#include "CustomLibraries/np.hpp"
#include "CustomLibraries/np_to_matplot.hpp"

using namespace matplot;
void test_simple_plot()
{
    std::vector<double> x = linspace(-2 * pi, 2 * pi);
    std::vector<double> y = linspace(0, 4 * pi);
    auto [X, Y] = meshgrid(x, y);
    vector_2d Z =
        transform(X, Y, [](double x, double y)
                  { return sin(x) + cos(y); });
    contourf(X, Y, Z, 10);

    show();
}

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

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

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

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

    // g.push_back(np::gradient(XcY[0], {dx, dy}));
    // g.push_back(np::gradient(XcY[1], {dx, dy}));
    std::vector<boost::multi_array<double, 2» gradf = np::gradient(f, {dx, dy});
    matplot::vector_2d Xp = np::convert_to_matplot(XcY[0]);
    matplot::vector_2d Yp = np::convert_to_matplot(XcY[1]);
    matplot::vector_2d X = matplot::meshgrid(linspace(0, 1, 100), linspace(0, 1, 100)).first;
    matplot::vector_2d Y = matplot::meshgrid(linspace(0, 1, 100), linspace(0, 1, 100)).second;

    matplot::vector_2d Z = np::convert_to_matplot(gradf[1]);
    std::cout « "X.size() = " « X.size() « std::endl;
    std::cout « "Y.size() = " « Y.size() « std::endl;
    std::cout « "Z.size() = " « Z.size() « std::endl;
    for (size_t i = 0; i < X.size(); i++)
    {
        for (size_t j = 0; j < X[i].size(); j++)
        {
            std::cout « "X[" « i « "][" « j « "] = " « X[i][j] « " | XP[" « i « "][" « j « "] = " «
       Xp[i][j] « " | YP[" « i « "][" « j « "] = " « Yp[i][j] « std::endl;
        }
    }
    contourf(Xp, Yp, Z, 10);
    show();
}

int main()
{
    // test_simple_plot();
    test_conversion();
    return 0;
}
```

11.15 variadic.cpp

```
#include "boost/multi_array.hpp"
#include "boost/array.hpp"
#include "CustomLibraries/np.hpp"
```

```
#include <cassert>
#include <iostream>

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

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

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

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

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

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

void test_complex_operations()
{
    int nx = 3;
    int ny = 2;
    boost::multi_array<double, 1> x = np::linspace(0, 1, nx);
    boost::multi_array<double, 1> y = np::linspace(0, 1, ny);
    const boost::multi_array<double, 1> axis[2] = {x, y};
    std::vector<boost::multi_array<double, 2» my_arrays = np::meshgrid(axis, false, np::xy);
    boost::multi_array<double, 2> A = np::sqrt(my_arrays[0]);
    std::cout « "sqrt\n";
```

```
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « A[i][j] « " ";
        }
        std::cout « "\n";
    }
    std::cout « "\n";
    float a = 100.0;
    float sqa = np::sqrt(a);
    std::cout « "sqrt of " « a « " is " « sqa « "\n";
    std::cout « "exp\n";
    boost::multi_array<double, 2> B = np::exp(my_arrays[0]);
    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « B[i][j] « " ";
        }
        std::cout « "\n";
    }

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

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

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

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

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

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

    for (int i = 0; i < ny; i++)
    {
        for (int j = 0; j < nx; j++)
        {
            std::cout « B[i][j] « " ";
```

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

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

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

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

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

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

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

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

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

    std::cout « "\n";
}

void test_abs()
{
    int nx = 4;
    int ny = 4;
    boost::multi_array<double, 1> x = np::linspace(-1, 1, nx);
```

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

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

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

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