WAVE SIM++

Wave Simulation Using Modern C++

Victor Barros
Yan Cheng

Motivation of this project

- One one hand, in today’s classrooms, scientific computation is taught mostly in Python
- Students often believe C++ is expert-friendly and beginner-unfriendly. This is a deep-rooted idea that their teachers probably believe as well.
- On the other hand, programs for scientific computation, developed by experienced scientists and engineers, are often unreadable to novice researchers

- Numpy achieved such success because its many features are well-designed
- With old C++, reproducing these features are highly nontrivial
- Challenge 1: Can we use modern C++ to easily create features similar to what Numpy offers?
- Challenge 2: Can these new features make C++ programs for scientific computation more readable while maintaining the high performance?

In our project, we focus on the wave simulation

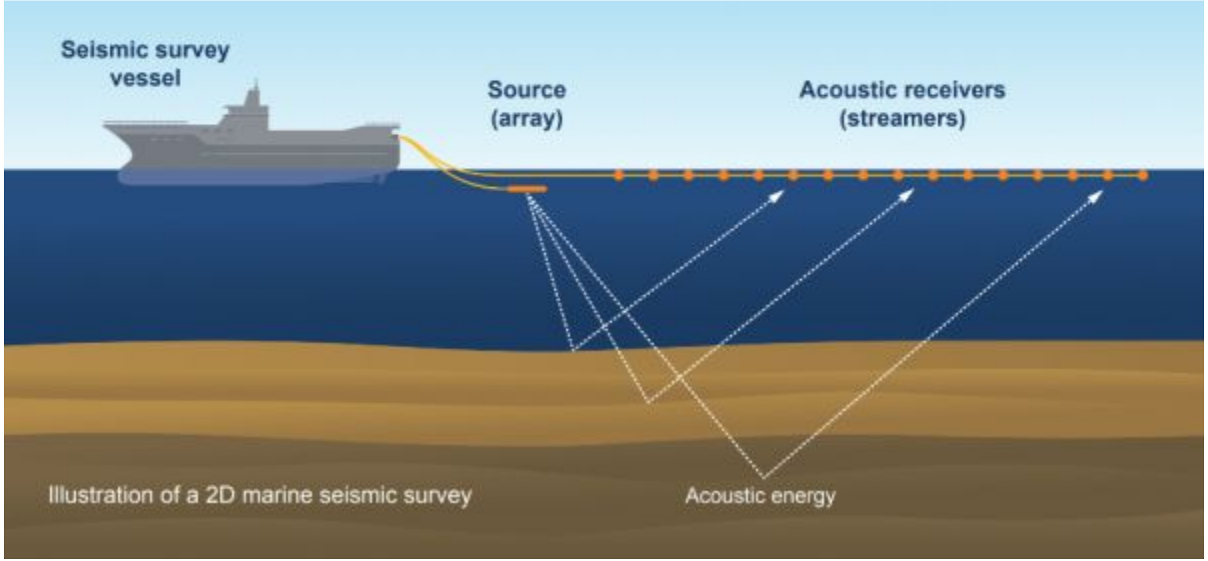
Wave simulation has application in geophysics, medical imaging, etc.

- This requires solving the following equation numerically

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

- The numerical algorithm we use is the finite-difference time-domain (FDTD) method.

Example: application in exploration geophysics



Numerical Algorithm

```python
def ricker(i_s, j_s, f=10, amp=1e0, shift=0.1, extra=False):
    """
    creates a source function (ricker pulse signal)
    f(t,x,y) = g(t) delta(x-x_s) delta(y-y_s)

    parameters:
    x_s, y_s: indices of location of the signal
    f: peak frequency
    amplitude: the default is 1
    shift: shifts the center to left or right

    return:
    3d array (discretized f(t,x,y))
    """

    t = np.linspace(tmin, tmax, nt)
    pft2 = (np.pi * f * (t - shift))**2
    r = amp * (1 - 2 * pft2) * np.exp(-pft2)

    if (extra == True):
      pft2 = (np.pi * f_M * (t - 2 * shift))**2
      r += 0.1 * amp * (1 - 2 * pft2) * np.exp(-pft2)

    x = np.zeros(nx)
    z = np.zeros(nz)
    x[i_s] = 1.0
    z[j_s] = 1.0
    R, X, Z = np.meshgrid(r, x, z, sparse=True, indexing='ij')

    return R*X*Z
```

```cpp
namespace waveSimCore
{
    //! Get the Ricker wavelet as a 3D Array
    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;
    }
}
```

- np::linspace
- arithmetic operations, np::pow, np::exp
- np::meshgrid: e.g. use 1D arrays x, y to create f(x,y) = x^2 + xy.

```python
def dfdx(fx, dx):
    dfdx, _ = np.gradient(fx, dx, dz)
    return dfdx

def dfdz(fz, dz):
    _, dfdz = np.gradient(fz, dx, dz)
    return dfdz

def divergence(fx, fz, dx, dz):
    dfdx, _ = np.gradient(fx, dx, dz)
    _, dfdz = np.gradient(fz, dx, dz)
    return dfdx + dfdz

def laplacian(f, dx, dz):
    dfdx, dfdz = np.gradient(f, dx, dz)
    d2fdx2, d2fdxdz = np.gradient(dfdx, dx, dz)
    d2fdzdx, d2fdz2 = np.gradient(dfdz, dx, dz)
    return d2fdx2 + d2fdz2

def d2dt2(f, dt):
    dfdt, _, _ = np.gradient(f, dt, dt, dt)
    d2fdt2, _, _ = np.gradient(dfdt, dt, dt, dt)
    return d2fdt2

def dfdt(f, dt):
    dfdt, _, _ = np.gradient(f, dt, dt, dt)
    return dfdt
```

```cpp
namespace waveSimCore
{
    //! Takes the partial derivative of a 2D matrix f with respect to x
    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];
    }

    //! Takes the partial derivative of a 2D matrix f with respect to z
    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];
    }

    //! Takes the second partial derivative of a 2D matrix f with respect to x
    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;
    }

    //! Takes the second partial derivative of a 2D matrix f with respect to z
    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;
    }

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

}
```

- np::gradient

np: Comparison between np and Numpy

```python
def test_toy_problem():
    x = np.linspace(0,1,100)
    y = np.linspace(0,1,100)
    XcY = np.meshgrid(x,y,sparse=False,indexing="ij")
    dx = 1.0/100.0
    dy = 1.0/100.0
    f = np.power(XcY[0],2.0) + XcY[0] * np.power(XcY[1],1.0)
    gradf = np.gradient(f,dx,dy)
    i = 10;
    j = 20;
    print( "df/dx of f(x,y) = x^2 + xy at x = " + str(x[i]) + " and y = "
+  str(y[j]) + " is equal to " +  str(gradf[0][i][j]))
```

→

```cpp
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);


    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 = 1.0 / 100.0;
    double 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);

    std::vector<boost::multi_array<double, 2>> gradf = np::gradient(f,
{dx, dy});

    int i = 10;
    int 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";
}
```

Result:

df/dx of f(x,y) = x^2 + xy at x = 0.101 and y = 0.202 is equal to 0.408

Implementation of np

np: a numpy-like library

Problems with current options:

- Limited to 2 or 3 dimensions
- Inconsistent behavior
- Lack of compile-time safety
- Limited in data types (double only)

NumCpp

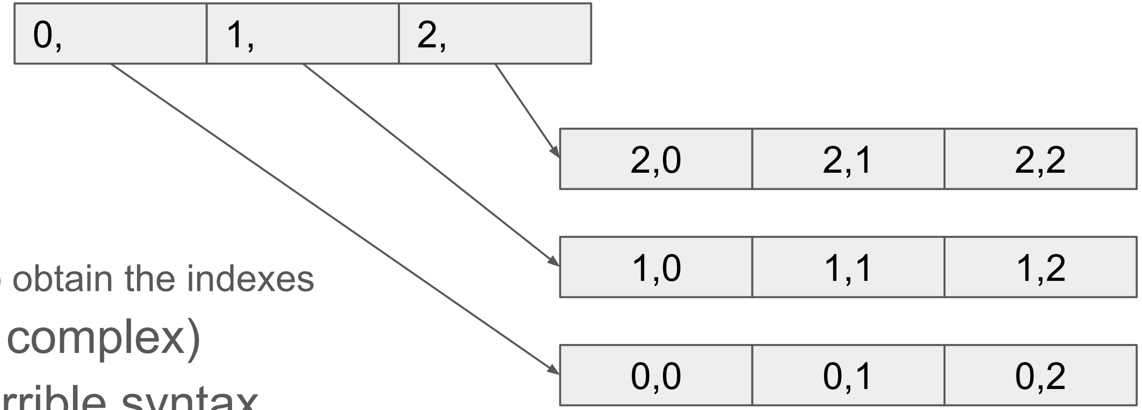
xtensor

np: a numpy-like library

Main features

- Uses boost::multi_array and expands upon it
- Supports n-dimensional arrays and operations
- Very compile-time safe
- Easily expandable
- Very similar syntax to python’s np
- Based on the standard library (+boost)
- Supports all std::is_arithmetic data types (most of the time)

boost::multi_array



- Nested structure of arrays
- Iterator provided
    - Requires division operation to obtain the indexes
- Uses views for slicing (very complex)
- Size allocation makes for terrible syntax
- Margin for improvement

Syntax:

Two dimensional array:

```
boost::multi_array<double, 2> XZ = np::meshgrid(axis, false, np::ij)[0];
```

N-Dimensional array of arbitrary type:

```
boost::multi_array<type, ND> n-dimensional-array;
```

Major modern C++ features used

- Concepts & Constraints
- Templates
- Variadic Templates
- Constexpr
- std::chrono

Example:

```
//! Implements the numpy min function for an n-dimensional multi array
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)
```

Example: Implementing meshgrid

```
//! Implementation of meshgrid
//! TODO: Implement sparsing=true
//! If the indexing type is xy, 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
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;
}
```

Using

Using the executable

```
$./WaveSimPPExec -h
A wave propagation simulator written in C++.
Usage:
  WaveSimPP [OPTION...]

  -d, --debug                 Enable debugging
      --animate               Render an animation at the end (default: true)
      --render                Render each of the frames at the end
      --export                Export the data to a series of csv files
  -o, --output_dir arg        Output directory path (default: output)
      --output_filename arg   Output filename (default: output.mp4)
      --framerate arg         Framerate of output video (default: 30)
  -v, --verbose               Verbose output
      --source_i arg          Source i position (default: 50)
      --source_j arg          Source j position (default: 50)
      --nt arg                Number of time steps (default: 1000)
      --nx arg                Number of x steps (default: 100)
      --nz arg                Number of z steps (default: 100)
      --dt arg                Time step size (default: 0.001)
      --dx arg                x step size (default: 0.01)
      --dz arg                z step size (default: 0.01)
  -f, --frequency arg         Frequency of source (default: 10.0)
  -a, --amplitude arg         Amplitude of source (default: 1.0)
  -s, --shift arg             Shift of source (default: 0.1)
  -r, --radius arg            Radius of velocity profile (default: 150.0)
  -l, --num_levels arg        Number of levels in the filled contour plot
                              (default: 100)
  -h, --help                  Print help
```

Using as a library

```
// Create the source
boost::multi_array<double, 3> f = waveSimPPCore::ricker(source_is, source_js, f_M, amp, shift, tmin,
tmax, nt, nx, nz);

// Create the velocity profile
double r = 150.0;
boost::multi_array<double, 2> vel = waveSimPPCore::get_profile(xmin, xmax, zmin, zmax, nx, nz, r);


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

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

Rendering

Renderers: Internal (Matplot++)

https://github.com/alandefreitas/matplotplusplus

Known issues: it is a limited library

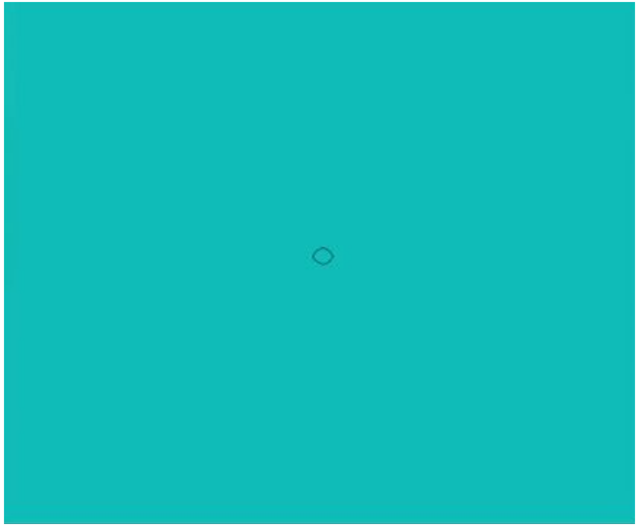
Render all frames as .csv file each:

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

Animating:

```
void wavePlotter::Plotter::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");
    }
```

Result:



Renderers: External (matplotlib)

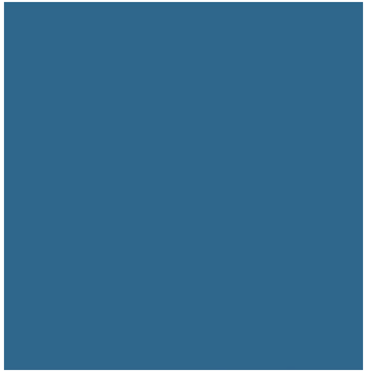
Render all frames as .csv file each:

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

Loading frames in Python:

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

umax = np.max([df.max().max() for df in frames])
umin = np.min([df.min().min() for df in frames])
n_levels = 100
levels = np.linspace(umax,umin,100)
```

Result:



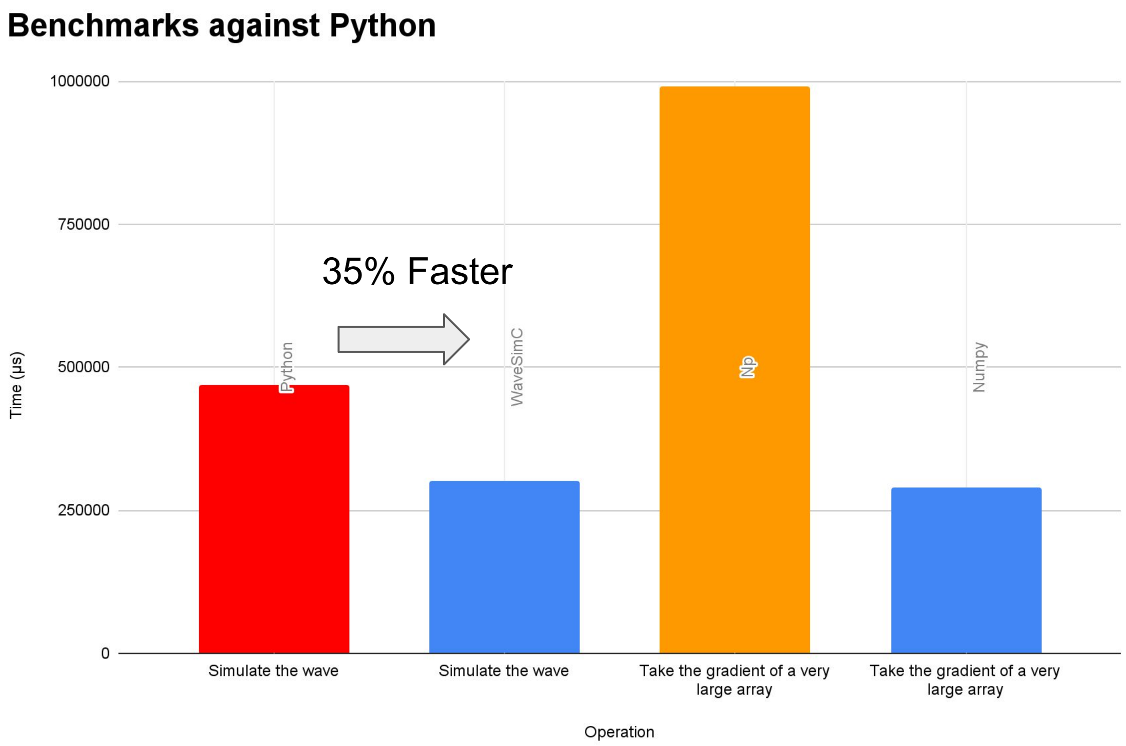
Animating:

```python
def make_cartoon(u, vmin, vmax, num_level=100, skip=10, size=25, interval=80, cmap=None):
    # Insert geometric constants from the original problem here

    x = np.linspace(xmin, xmax, nx)
    z = np.linspace(zmin, zmax, nz)
    X, Z = np.meshgrid(x, z, indexing='ij')
    levels = np.linspace(vmin, vmax, num_level)

    fig, ax = plt.subplots(figsize=(size, size))
    ax.contourf(X, Z, u[0], levels=levels, vmin=umin, vmax=umax, cmap=cmap)
    ax.set_ylabel("z (m)")

    plt.gca().set_aspect('equal', adjustable='box')

    def animate(i):
        i *= skip
        ax.clear()
        ax.text(0.45, 1.05, "t = {0:0.2f}".format(np.round(i*dt, 2)), transform=ax.transAxes)
        ax.contourf(X, Z, u[i], levels=levels, vmin=vmin, vmax=vmax, cmap=cmap)
        ax.invert_yaxis()
        ax.set_xlabel("x (m)")
        ax.set_ylabel("z (m)")

        plt.close()

    return animation.FuncAnimation(fig, animate, interval=interval, frames=int(len(u)/skip))
```

Benchmarking

Benchmarking: Wave Solving

System specs:
OS: Ubuntu 20 Running on Windows 11 WSL
CPU: Ryzen 5800H (8 Cores, 16 Threads)
RAM: 32GB DDR4
System Capped at 100W



Infrastructure

Version, Compiler and Build System

C++ 20

GCC 12

- Large amount of C++20 features
- Standard in unix systems
- Free and open source

CMAKE

- Widely used in unix systems
- Compatible with our external libraries
- Easy to use
- Free and open source

Hosting: github

Github repository:
https://github.com/yc3855/COMSW4995

- Automatic checking of every commit through the use of github actions
- Bugs & features managed through github issues
- Open source and we aim to make it community driven

Documentation: Doxygen + Netlify

- Automatic generation of documentation through in-code comments
- Generates PDF and HTML docs
- Support for markdown

Netlify:

- Automatic deployment of static web pages
- Free for simple use

Online docs: https://wavesimc.vbpage.net/

Online docs: https://wavesimc.vbpage.net/

WaveSimPP 1.0

A C library for solving the wave equation and reconstructing the wave field

Main Page | Related Pages | Modules | Namespaces | Classes | Files

Main Page

COMSW4995 Final Project: WaveSimPP

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

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

In addition, a custom implementation of numpy in modern C++ is also included as a header library. That library aims to make c++ more pyth
many utilities to expand the functionality of the library.

Please check the **Readme file** for more information.

Authors

Victor Barros - Undergradute Student - Mechanical Engineering - Columbia University

Yan Cheng - PhD Candidate - Applied Mathematics - Columbia University

License

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

WaveSimPP 1.0

A C library for solving the wave equation and reconstructing the wave field

Main Page | Related Pages | Modules | Namespaces | Classes | Files

Tutorial

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

To build the executable you can use the following commands:

```
mkdir build
cd build
cmake ..
make WaveSimPPExec
```

And then run it with the following command:

```
./WaveSimPPExec
```

The executable will ask you for the parameters of the simulation. A complete description of the parameters can be found in t

```
./WaveSimPPExec -h
```

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 o
location.

It is important to pay attention to the fact that the plane is XZ not XY due to geophysical conventions.

```
// Define the constants for the simulation

// Number of x and z grid points
int nx = 100;
int nz = 100;
// Number of time steps
int nt = 1000;

// Differentiation values
double dx = 0.01;
double dz = 0.01;
double dt = 0.001;

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

// Define the source parameters
double f_M = 10.0;
double amp = 1e0;
```

Online docs: https://wavesimc.vbpage.net/

WAVE SIM++

WaveSimPP 1.0

A C library for solving the wave equation and reconstructing the wave field

Main Page | Related Pages | Modules | Namespaces ▾ | Classes ▾ | Files ▾

wavePlotter 〉 Plotter

wavePlotter::Plotter Class Reference

WavePlotter

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

`#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.

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

◆ 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
)

Implementation of meshgrid TODO: Implement sparsing=true If the indexing type is xx, then reverse the order of the first two elements

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]);
```

Plans for the future

Plans for the future

- Implement a custom multi-dimensional array data structure
  - Improve performance and syntax
  - Would remove the dependency on boost

- Implement a custom rendering library
  - Address limitations of current options
  - Add gpu optimization

- Further optimize functions

Thank you