DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of a tool for postprocessing of simulated earthquake data

Florian Hecher

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Development of a tool for postprocessing of simulated earthquake data

Entwicklung eines Tools zur Nachbearbeitung von simulierten Erdbebendaten

Author:	Florian Hecher
Supervisor:	Univ.-Prof. Dr. Michael Bader
Advisor:	Leonhard Rannabauer
Submission Date:	15.09.2017

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.09.2017 Florian Hecher

Acknowledgments

First I want to thank Prof. Dr. Michael Bader for giving me the opportunity to write this thesis and for providing me with such an interesting topic. I also want to greatly express my gratitude towards Leonhard Rannabauer who provided me with a lot of guidance and helped me with the problems that occurred throughout the development of this work. Furthermore, I want to thank Carsten Uphoff for his explanations and his help on the allocation problem.

Abstract

As they are really difficult to predict, simulating tsunamis is an important task in estimating their impact. This simulation should be well connected to a previous earthquake simulation. For the earthquake simulation tool SeisSol and the tsunami simulation tool sam(oa)^2 this connection is currently quite poor, as SeisSols output format doesn't match the input requirements of sam(oa)^2. This thesis aims to resolve this issue by converting SeisSols triangular mesh to sam(oa)^{2}s rectangular grid. This conversion is done via integral over the original data for the rectangles area. The result is a program with a low runtime that provides a conversion with high accuracy and enables precise simulations of possible tsunamis.

Contents

1 Introduction

Tsunamis are immense natural disasters and as seen in [1] have the potential to cause billions of dollars of damage and to kill thousands of people in multiple countries at once. According to [2] tsunamis can be caused by erupting underwater volcanoes, calving icebergs and even meteorites but in most cases they are caused by underwater earthquakes in the ocean. As explained in [3] if such an earthquake has an epicentre near the earths surface, is powerful enough (at least 7,0 on the Richter-scale) and causes a high vertical displacement of the water above its center, a wave is caused, that can spread over large distances but is mostly underwater, meaning it is almost invisible for most of the time and has barely any influence on ships on the open sea. But once this wave reaches shallow areas close to the coast, water gets compressed and can pile up to waves that are several meters high and can reach far into the inland. When tsunamis happen, they don't leave a lot of time to react. Depending on the distance from the land to the epicentre, according to [3] this time is only between a few minutes and half an hour even if the forming tsunami is recognized early. Predicting or simulating Tsunamis is therefore an important part of finding out, whether an area is in danger of being hit by a tsunami or not, how big the damages would be and if buildings and structures could withstand the wave. For this we need to be able to accurately tell how the wave of a Tsunami develops, how far and fast it spreads and how high, fast and strong the waves will be at the coast. It is also crucial to be able to decide whether a certain earthquake will cause a tsunami and how large it will be. Researching and simulating earthquakes and tsunamis should therefore be combined to achieve the best results. In this case for simulating the earthquakes the tool SeisSol is used, which according to [4] is a "high resolution simulation of seismic wave propagation in realistic media with complex geometry". For calculation it uses tetrahedral meshes in 3D space which then get projected to the 2D ground level with simple up or down displacement values for the surface area. This results in a 2D mesh of triangles as structure to represent the data. But the tool sam(oa)^2, standing for SFCs and Adaptive Meshes for Oceanic And Other Applications, found at [5] which is used for the tsunami simulation is working on a structured grid of triangles. The tool created at [6] ASAGI which is a pArallel Server for Adaptive GeoInformation and is used for input reading is also not working on a mesh but with a grid of perfect rectangles. Therefore, simulating an earthquake in SeisSol and directly feeding it into sam(oa)^2 to simulate the corresponding tsunami is currently

not possible. At the moment data for sam(oa)^2 has to be created from scratch with an extra tool developed at [7] or converted from SeisSol by hand via a cumbersome process using multiple smaller tools. Because this is a multiple step process there are multiple points where something in the conversion can go wrong, which makes looking for possible errors difficult. As a result, the process as a whole is tiresome and costly in terms of time and labour. Although there are already a number of tools to handle geographical data like [8] or data based on triangular meshes as proposed in [9], they either work on differently structured input or like [10] produce output of a different format and none of them make it possible to handle the whole conversion in a single step. Some pre- and postprocessing of the data would still be required. Therefore, the need for a tool that is able to convert the output from SeisSol to input readable for sam(oa)^2 in a single conversion has arisen. Through this tool the conversion process will be simplified, easier for the user, possible errors can be located quicker and overall a lot of time can be saved. Having only one tool also makes it easier to change data or a part of it to be able to see how much of an influence such changes have. For example, the grid resolution can be changed without the need to run the originating simulation in SeisSol multiple times. In short, the main goal is to fill the gap between the two programs by developing a tool that converts the output from SeisSol into a structure that can easily be used as an input for sam(oa)^2. The second goal is to alter existing data (e.g. to reduce or increase the resolution) and compare the corresponding results. Designing such a tool is the goal of this thesis. After the introduction the first section will give a brief overview of necessary information about the used libraries. The second chapter introduces the tools that are directly connected to this thesis. In the third chapter the detailed demands for the tool are explained and instructions for its usage are given. This chapter also gives insight to the code and explains the implementation of the program. The evaluation is done in the fourth chapter and different options with which the program can be run are also compared there. The results of the application of the tool to real data from SeisSol and the resulting simulation in sam(oa)^2 is done in the fifth chapter. The sixth chapter gives an outlook on how the program can be further improved. The final chapter is a conclusion of the thesis.

2 Preliminaries

This tool works with and on different data structures and ways of data representation. As these can be and usually are huge amounts of data, we need a special format to efficiently store and transport the data. For this project the data model *Hierarchical Data Format 5* (HDF5), the data format *Network Common Data Form* (NetCDF) and the *extensive data model format* (XDMF) are used. This is because they are already widely spread and well known. The programs this tool was mainly developed for (see chapter 3) also use them as main way of input or output. The following sections will briefly explain what exactly they are and can do, how they work and how they can be used and are used in this thesis.

2.1 HDF5

According to [11] HDF5 is a data model, library, and file format for storing and managing data. It can be used for all the common datatypes but supports user designed custom datatypes as well. As explained in [12] these datatypes can have specifications regarding information such as byte order (endian), size, and floating point representation which enables a full description about how the data is stored, insuring portability to other platforms. HDF5 is able to deal with data of high volume and complex data of any size. Following the description at [13] it is also almost platform independent, because it runs on laptops as well as on massively parallel systems and anything in between, implementing a high-level API with C, C++, Fortran 90, and Java interfaces. Data is stored efficiently and a high I/O speed is reached. As stated in [14] it is also designed to be very flexible in its usage so applications are enabled to evolve in their use of HDF5 and can accommodate new models. HDF5 is even more widespread than it seems at first glance, since it can be used as a basis for other tools like NetCDF or XDMF.

2.1.1 Design

As described at [14], HDF5 has three main parts. The *file format*, the *data model* and the *software working with this format*. While knowing the high-level of the file format is usually necessary to work with HDF5, the low-level details do not need to be known by

the end user, but it can still be helpful to know the basics of how it works. According to its specification at [15] it appears to be a directed graph on high-level view. The nodes of this graph are higher-level HDF5 objects that can be accessed via the HDF5 APIs. It is further specified, that the lower-level of a file consists of a superblock, B-tree nodes, heap blocks, object headers, object data and free space. As stated in [12], the data model is "simple but versatile" as its "supports complex data relationships and dependencies through its grouping and linking mechanisms.", and as described in [14] a HDF5-file is basically a container (or group) that holds a variety of heterogeneous data objects. Such an object is usually either a group which has some metadata on the groups or datasets it contains, or it is a dataset. [14] further explains, that a dataset is a multidimensional array of data elements that also contains supporting metadata and can represent simple things like tables or graphs but also entire documents like a PDF file. The software contains various libraries and APIs to use those libraries. It is written in C, but adaptions for other programming languages also exist. There are also many third-party programs that either use HDF5 as a basis or are extensions to the initial format.

2.1.2 Usage

While there are tools like *h5dump* or *HDFView* to display the content of a HDF5 file, they usually are not enough to actually work with the data. So for reading and writing the APIs can be used. After opening or creating a file the (meta-)data can be read, new dimensions or datasets can be created and filled and attributes can be set. For filling in data the file must not be in *define*-mode as this state is only used for setting metadata. Each of these operations has its own H5*-C-routine. In this thesis, raw HDF5 is used as a part of reading XDMF and its wrapper NetCDF is used for everything else.

2.2 NetCDF

As described at [16] NetCDF is a set of software libraries and self-describing, machine-independent data formats. These formats support the creation, access, and sharing of array-oriented scientific data. NetCDF was developed and gets maintained by Unidata. It is intended to be used to create, share and work on scientific data, that can be represented as multidimensional arrays. The most important advantages of NetCDF are listed at [17] and are its self-describing, portable, scalable, appendable, sharable and archivable concept. [18] alternatively describes it as an interface to a library of data access functions for storing and retrieving data in the form of arrays.

2.2.1 Design

Like HDF5, NetCDF is separated into the *format*, the *data model* and the corresponding *software*. There are four basic formats. The *classic format* was the first one to be developed and is also the basis for the other three. The second format is the *64-bit offset format*, which changed the size of the relative offset from 32 to 64-bit and thus allowing files to be much larger. The *NetCDF-4 format* is the third format and as described in [17] came 2008 with additional features like per-variable compression, multiple unlimited dimensions, more complex data types and overall better performance, whereas the fourth format the *NetCDF-4 classic model format* only has the performance increasements but no additional features. There are two basic models for NetCDF. The *classic model* to which the first, second and fourth file-format belong, and the *enhanced model* or *NetCDF-4 data model* for the third format. Data of the first model can be represented by the second but not vice-versa. The first model is more simple, needs less prerequisites and is more widely used, while the second model allows more complex data structures, bigger files and has additional features but can still do everything the first model can. Similar to HDF5 the NetCDF data models are also based on multidimensional arrays of a set datatype. The files are split into a header containing the metadata like dimensions, variables and attributes, and a data part. The second model directly relies on the HDF5 library and according to [19] is therefore a HDF5 file in every way. The software belonging to NetCDF are mainly a variety of functions in C that allow working on NetCDF files. Additional libraries for usage in other programming languages like C++ or Fortran are also provided.

2.2.2 Usage

In its use, NetCDF is very similar to HDF5 as there also exist commands like *ncdump* to show a files content. For actualy working on the file the commands *nc_open* (or *nc_create* when creating a new file), *nc_put**, *nc_get** and *nc_close* are used. A file consists of three different types of data. Dimensions consisting of a name and size, variables (depending on the dimensions) and attributes which are additional information about a variable or the file as a whole. In this thesis NetCDF is used as an input-method for the ASAGI- and GEBCO-type files as well as the single form of output. Although the program is in C++, only functions from the C-library of NetCDF were used.

2.3 XDMF

XDMF differs from HDF5 and NetCDF as it is not working directly on the data itself but is combining XML and HDF5 to reach a format that is understood by many widely-used

programs and allows a quick and easy transfer of huge amounts of data. XML is used to store the *light data*, meaning it is small and can be passed between modules easily. According to [20] it is used to transfer information about the data because regardless of its size every dataset can be described completely using the information of number type (float, integer, etc.), precision, location, rank, and dimensions. This allows the user to inquire information about the size and type of the data before he accesses it so he can do some preparations accordingly. The actual values of the data are described as *heavy data* and are stored in HDF5 or binary files. The name and location of the heavy data is also stored in the XML part. In special cases where the heavy data is relatively small, it can also directly be written into the XDMF file, like demonstrated in [21], but this option does not get used in this thesis.

2.3.1 Design

As HDF5 is already explained in another chapter and binary files are not used in the tool developed in this thesis, this chapter will focus on the design of the XML part of XDMF. Next to standard XML options like *Elements*, *XClude*, *XPath* and *Entities* that are described in [22], there are additional XDMF Elements like the *XDMF-version*, that gets specified at the beginning. Next to a *Name* or *Reference* attribute every XDMF Element has at least one *Domain* containing *Grid* elements that themselves have one *Topology* and one *Geometry* to specify the type of grid, its grid points, and how the points are connected to form the grid. To specify values, their type, amount and size a *DataItem* element is used. This can be:

- Uniform - this is the default. A single array of values.
- Collection - a one dimension array of DataItems
- Tree - a hierarchical structure of DataItems
- HyperSlab - contains two data items. The first selects the start, stride and count indexes of the second DataItem.
- Coordinates - contains two DataItems. The first selects the parametric coordinates of the second DataItem.
- Function - calculates an expression.

For this thesis, only the DataItems *Uniform*, *Collection* and *HyperSlab* are used.

2.3.2 Usage

While the XDMF document itself can be read with a variety of standard readers or libraries, for the referenced files HDF5-reading is necessary again. While parsing the XML-tree most of the information is stored in the XML-attributes for the nodes and only the filenames and paths are stored as information of the nodes themselves. This thesis uses XMDF-version 2.0 and contains one Domain with a *Triangle-Topology*, a *XYZ-Geometry*, multiple standard DataItems for the file paths and one Collection of Grids which contain information about the individual timestamps of the provided data. The exact specifications are described in 4.2.2.

3 Related Tools

The tool developed in this thesis works on earthquake and tsunami-related scientific data and converts the ways of representation from one type to another. It is highly adapted to the tools, that produce said data or use it as an input. Knowing what these tools do, how they work and especially what kind of data they need and produce is therefore useful in understanding the goals and ways of this thesis. This chapter will give a quick overview of the three programs, that are most relevant for this tool. For a detailed explanation of capabilitys, usage and implementation the respective documentations should be viewed.

3.1 SeisSol

This is the program producing the data that is used as input. According to [4] SeisSol is a software using "High resolution simulation of seismic wave propagation in realistic media with complex geometry" to simulate earthquakes, wave propagation and dynamic rupture in materials of different elasticity. Predicting an earthquake is still quite difficult so SeisSol is working with observations from geodetic and seismological data to simulate possible scenarios that might occur as a result of subduction. It is also used to predict ground motion in the aftermath of an earthquake and as a predictor for possible volcanic eruptions. As stated in [23] its software has been optimized to reach high peak performance and strong scaling up to 90% parallel efficiency and 45% floating point peak efficiency on simulations performed on the SuperMUC machine. [24] and [25] further explain, that it uses the "Arbitrary high-order accurate DERivative Discontinuous Galerkin (ADER-DG) method", meaning it solves the (elastic) wave equation, which is a partial differential equation, by multiplying it with a test function and then integrating it over a tetrahedral element. The tetrahedral meshes are static and unstructured according to [26], but as explained in [4] fully adaptive through "smooth refining and coarsening strategies". Following the explanation from [27], together with the high-order accuracy the ADER-method allows, this allows seismic waves to be simulated with minimal errors even over great distances and in complex geometries. [28] further explains, that for writing output the data is reduced to first order and projected down to 2D, meaning it will only display the up and down displacement values on the surface and skip higher-order information. The output geometry of a

unstructured mesh of triangles resembles the fixed but unstructured geometry of the three dimensional mesh the simulations were done on. This grid gets stored in a HDF5 file. The format that is used to store the data is to store all the single points with their x, y, and z-coordinates separately and then connect them with a table that stores the corner points for each triangle. For the computation here only the first 2 matter, the z-value is a leftover from the 3D computation and is not needed, as the relevant z-values are the displacement values for the triangles which are stored separately into a third table. The metadata for how to combine the data correctly and additional information gets stored into a single XDMF file.

3.2 Sam(oa)^2

This is the software that in the end simulates the actual tsunamis and therefore is the program for which the output is produced. According to [29], its main application are the simulation of tsunami wave propagation and the simulation of two-phase porous media flow, but it can be used on all finite-element-type or finite-volume-type applications that are based on matrix-free, element-oriented formulations. This is possible, because the complexity of the underlying process is hidden from the application. Similar to SeisSol it solves partial differential equations (in case of the tsunami simulation these are the shallow water equations) and its framework is designed to do so in a highly parallelized way. Although sam(oa)^2 is basically a 2D simulation because oftentimes movement along the third dimension is so small that it can be ignored, this is not always the case so in [30] support for 2.5D grids was added to also be able to calculate 3D applications. To be able to simulate tsunamis with maximum accuracy in a minimal amount of time sam(oa)^2 uses the Sierpinski space-filling curve as a basis for its adaptive triangular meshes. The initial grid consists of one rectangle that consists of two triangles which are spereated along the diagonal line. As explained in [29], these two base triangles then get further split via *newest vertex bisection* until they are small enough for accurate calculation. If a triangle is split even further or merged together with another triangle is decided dynamically in each timestep for each triangle as stated in [31]. Because of the Sierpinski space-filling curve an order can then be established for the semi-structured triangles as seen in [32], which is a big difference to SeisSol, which had an unstructured mesh. This order is then used to traverse the grid in a fast an memory-efficient way. Sam(oa)^2 can be used for small experimental examples directly, but for bigger computations and therefore bigger and more complex input the ASAGI tool is necessery.

3.3 ASAGI

This tool is used by both SeisSol and Sam(oa)^2 to process their input and is therefore the program whos input format the output of the conversion tool must match. As this tool converts from SeisSol to Sam(oa)^2 input constraints for SeisSol stated at [33] and [34] will be ignored in this section. According to [6] ASAGI is "a pArallel Server for Adaptive GeoInformation" meaning it was designed to read NetCDF input files for big parallel simulations that work on geographical data based on a cartesian grid. It is an interface for large-scale geo-simulation applications that provides various benefits without end applications noticing. As [35] explains, these programs work with "dynamically adaptive mesh refinement", which means ASAGI can enable them to work more efficiently, by giving each node only the data it needs, copying necessery parts of the dataset to multiple nodes and deleting information that is no longer needed to achieve a balanced workload across all compute nodes. ASAGI works in three steps: The first one is to load all the input data and synchronize processes with the application. The second is the data access while the simulation is running to ensure good workload distribution. The last step only frees all the used ressources. Despite the different mesh data structures the underlying application can have, ASAGI always works on its own cartesian grid for calculations. [36] states, that of these grids there are three different types: FULL(default), CACHE and PASS_THROUGH. FULL means the whole file is loaded at the beginning and not touched in the simulation. With CACHE nothing is loaded initially, but a block is put into the cache when an element is requested. PASS_THROUGH will not load anything but access the file every single time instead. ASAGI also makes it possible to have multiple resolutions at once, but this option is not used here. For grid positions a cell-centered grid is chosen over the vertex-centered grid.

4 Conversion

This tool can convert 3 different types of earthquake- or bathymetry-data (ASAGI-NetCDF, GEBCO-NetCDF and SeisSol-XDMF) into a NetCDF-file that can be used in a tsunami simulation. The previous chapters have explained how the software this program is based upon works and what some related tools are and what they do. This chapter is split into three parts. First it will be explained what the exact demands for the developed tool are and why these demands exist. Then instructions on how it has to be used are given and all the different options that can be set are presented. This includes their effect on the programs behaviour and runtime as well as their effect on the resulting output. Lastly a detailed look into the implementation of the tool is given and it will be explained how it works internally.

4.1 Demands

This section describes all the requirements and demands to the software. In short, core goal of this thesis is to develop a tool, that takes output from SeisSol and converts it to a data structure readable by ASAGI and sam(oa)^2. In order to do so, a XDMF-file and its corresponding data on a triangular mesh need to be read and transformed into a NetCDF-file, that holds a representation of the same data on a rectangular grid. (See 4.2.1 and 4.2.2 for the exact specifications.) In addition, it needs to be possible to freely select only a section of the input area and to choose the grids resolution of the output file. The tool must also handle all different orientations of the three-dimensional input grid, where east-north-up is the most common. While the XDMF-files usually provide bathymetry data in the z-values of the single points, this data has a fixed resolution. In order to get bathymetry data with selectable resolution for the simulation, an additional sources is needed. Therefore, the tool must be able to read GEBCO-style NetCDF-files and transform them into the same kind of NetCDF file it also produces as output for XDMF. (Again see 4.2.1 and 4.2.2 for the exact specifications.) For this a transformation from latitude and longitude coordinates on the earths sphere to a flat 2D grid is necessary. This is because the grid of the ASAGI output defines the location of its points with meters so the latitude and longitude coordinates have to first be mapped onto this grid before the corresponding values can be used in a simulation. This transformation must support two kinds of conversion methods to achieve a high

accuracy for different resolutions in the output grid. The first method has to always take the closest input point available as output point and is useful when the resolution of input and output grid are about the same, because in this case taking multiple values into account would blur the result. The second method is for output grids with a far lower resolution than the input. In order to not loose information this method has to take all input-points within a certain radius into account. This radius needs to be freely selectable. As last requirement, the tool must be able to change the resolution of an output-style NetCDF file with a single timestamp. This makes it possible to simply scale the result of a previous transformation of a GEBCO-file to a new resolution and the time-consuming transformation does not have to be done again.

4.2 Program Usage

In this section an explanation on how the program is used is given. It will start by describing the output format and continue by listing the specifications of the three possible input formats. In the last part, all the currently existing program flags are listed with their program requirements, parameter demands, behaviour and effect on the conversion process and the output file. Note that any form of usage or input that doesn't match these instructions and requirements can result in undefined behaviour or errors and is therefore disadvised.

4.2.1 Output

There is currently only one type of output. It is a NetCDF file designed to be readable by ASAGI and it has the following specifications:

- Two dimensions: x and y (in that order).
- Three variables: *double x(x)*[1], *double y(y)* and *float z(y,x)* or *double z(y,x)* (in that order). The precision of the variable z depends on the precision of the input. All variable units are meters with x pointing east, y pointing north and z pointing upwards. For XDMF-input the variable z changes its dependencies to *float z(time,y,x)* or *double z(time,y,x)* respectively.
- Each variable has the attribute *actual_range* with z having the additional attribute *_FillValue*.

[1] The x before the parenthesis is the name of the variable and the x in parenthesis is the name of the dimension the variable depends on. y and z work the same way. This means the three variables have no dependencies on each other and in theory are totally unrelated. A connection only occurs due to the interpretation by the reading program.

```
netcdf Standard_ASAGI_output_file {
dimensions:
        x = 1000 ;
        y = 1600 ;
variables:
        double x(x) ;
                x:actual_range = -250000., 250000. ;
        double y(y) ;
                y:actual_range = -400000., 400000. ;
        float z(y, x) ;
                z:_FillValue = NaNf ;
                z:actual_range = -6.48760175704956, 16.1780223846436 ;
// global attributes:
        :Conventions = "COARDS/CF-1.0" ;
        :title = "Standard_ASAGI_output_file.nc" ;
        :GMT_version = "5.0.0_r9703M [64-bit]" ;
        :node_offset = 1 ;
}
```

Figure 4.1: Standard ASAGI file with float precision

- Four global attributes: *Conventions* (default set to "COARDS/CF-1.0"), *title*, *GMT_version*, *node_offset* (always set to 1).
- If the Input was in SeisSol-XDMF format there are the additional dimension *time* and the additional variable *double time(time)* with the attributes *long_name* (always set to "time"), *actual_range* and *units* (always set to "seconds since 0000-1-1 0:0:0").

The difference in output format can be seen in Fig. 4.1 and 4.2.

4.2.2 Input

There are three different file types and corresponding formats, that can be used as input. Depending on the input, the exact output format can differ and necessary flags or their effect can change.

ASAGI-NetCDF

This is the simplest input type and the one with the least options. It is expected to be a NetCDF-file and to have the same format as the standard output (see 4.2.1) without

```
netcdf Modified_ASAGI_output_file {
dimensions:
        x = 40 ;
        y = 40 ;
        time = 11 ;
variables:
        double x(x) ;
                x:actual_range = 2250., 21750. ;
        double y(y) ;
                y:actual_range = -9750., 9750. ;
        double z(time, y, x) ;
                z:actual_range = -0.233631199933331, 0.233631199933333 ;
                z:_FillValue = NaNf ;
        double time(time) ;
                time:long_name = "time" ;
                time:actual_range = 0., 5. ;
                time:units = "seconds since 000-1-1 0:0:0:"
// global attributes:
        :Conventions = "COARDS/CF-1.0" ;
        :title = "Modified_ASAGI_output_file.nc" ;
        :GMT_version = "none specified" ;
        :node_offset = 1 ;
}
```

Figure 4.2: Output for XDMF input with double precision

the *time* dimension. For this type, data can only get scaled but no transformations are done.

GEBCO-NetCDF

This format is based on the "General Bathymetric Chart of the Oceans" GEBCO_2014 Grid in 30 arcsecond intervals from [37]. This grid is an aggregation of multiple bathymetric datasets into one and represents the distance from sea-level to the earths ground. Instead of (x,y)-coordinates it uses latitude and longitude in 30 arcsecond intervals. One interval is approximately 926 meters for the latitude as all meridians have the same length, but changes drastically for the longitude depending on where you are, because the distance that is covered per degree decreases towards the poles. At the equator, it's also approximately 926 meters for 30 arcseconds but this gets less, the further one is away from the equator. This input is used to get the bathymetry data needed for a tsunami simulation. For this a projection from a spheres surface to a flat 2D grid is done (see 4.3.3 for details). It is possible to choose between two different modes of conversion, that change how many input points are taken into account for one output point. A minimal example is shown in fig 4.3. The GEBCO format is a NetCDF file with the following specifications[2]:

- Two dimensions: *lat* and *lon* (in that order).
- Three variables: *short elevation(lat,lon)*, *double lat(lat)* and *double lon(lon)* (in that order). The units for *lat* and *lon* are absolute latitude and longitude coordinates on the earths sphere, while *elevation* is measured in meters and points upwards. *elevation* can also have the precision levels of "float" or "double", with the latter one also producing output in "double" precision.
- Four global attributes: *Conventions*, *title*, *node_offset* and *history* (without particular order).

SeisSol-XDMF

The XDMF input format is the most complex one of the three formats and must meet the most requirements. In contrast to the other 2 input types, the XDMF input file does not contain the data itself but specifies all the files where the actual data is stored. This can be done with one or more HDF5 files. Data is represented via an unstructured triangular-mesh, that gets stored by listing all single data-points, triangles as point-triples and actual z-values separately. A minimal example of a XDMF file and

[2]Data acquired from [37] has a lot of additional information in its attributes, but these are not needed

```
netcdf Minimal_GEBCO_input_file {
dimensions:
       lat = 5476 ;
       lon = 8156 ;
variables:
       short elevation(lat, lon) ;
       double lat(lat) ;
       double lon(lon) ;
// global attributes:
       :Conventions = "CF-1.0" ;
       :title = "The GEBCO_2014 Grid" ;
       :history = "This is version 20150318 of the data set." ;
       :node_offset = 1 ;
}
```

Figure 4.3: Minimal GEBCO input file

its underlying HDF5 file are shown in Fig. 4.4[3] and 4.5. The HDF5 files must meet the following requirements:

- Data is split into the three parts *geometry*, *connect* and *W*. These parts are usually all inside one single file, but can also be separated into two or three files.
- *geometry* must be of type double. It lists all single data points with their respective XYZ-values (in that order, all values for one point are listed before the next point begins).
- *connect* must be of type uint64. It connects all triangles one after another, by listing three corner points of each respective triangle. Corner-points are represented by their position in *geometry* data, starting with index 0.
- *W* must either be of type float or double according to the XDMF file. It contains z-values for the triangles in all timestamps. Triangles are ordered the same way as in *connect*, but each timestamp is listed separately, meaning all data for timestamp 0 is stored, before timestamp 1 starts.

The XDMF-file has a XML structure where all relevant information is inside the

[3] In this example everything that is not needed by this tool was removed, so other programs that usually can read XDMF might not be able to read this file anymore

Xdmf.Domain-branch. This branch must contain the following subbranches and information:

- Exactly one *Topology* which needs the XML-Attribute *NumberOfElements* to indicate the number of existing triangles. This branch contains subbranches with further information:
  - One *DataItem* with the path to the location of the data connecting the triangles. This path must end in ":/connect".
- One *Geometry* with the XML-Attribute *NumberOfElements* indicating the number of single points in the dataset. This branch contains subbranches with further information:
  - One *DataItem* containing the path to the XYZ-data of the single-points. This path must end in ":/geometry"
- One *DataItem* containing the path to the z values of the triangles. This path must end in ":/W".
- This *DataItem* needs a XML-Attribute *Precision* to indicate the precision of the datatype. Possible values are 4 (for float) or 8 (for double).
- One *Grid* which contains the information about the timestamps. This branch contains subbranches with further information:
  - Each timestamp has its own *Grid* which needs a subtree *Time* with the XML-attribute *Value*.
  - This value represent the point in time the timestamp is from.
- Any additional information is currently ignored.

4.2.3 Flags

As this tool does not only have one single function but a variety of calculations it can perform, there are a number of flags needed to get the expected results. These flags can influence runtime, precision, how the input is handled, which output is produced, which calculations or projections are done and set specific values inside the calculation or the output. This chapter will explain the effects of each flag. Fig. 4.6 shows an example output for the *tpv-16-surface.xdmf* input scenario and the figures 4.7, 4.8, 4.9 and 4.10 show the impact of their respective flags.

The program is started by calling *Convert.exe* with its input parameters:

```
-<Xdmf Version="2.0">
    -<Domain>
        -<Topology NumberOfElements="45164">
            <DataItem>XDMF_example.h5:/connect</DataItem>
         </Topology>
        -<Geometry NumberOfElements="22685">
              <DataItem>XDMF_example.h5:/geometry</DataItem>
          </Geometry>
         <DataItem Precision="8">XDMF_example.h5:/W</DataItem>
         -<Grid>
            -<Grid>
                 -<Time Value="0"/>
              </Grid>
            +<Grid></Grid>
            +<Grid></Grid>
            +<Grid></Grid>
          </Grid>
     </Domain>
 </Xdmf>
```

Figure 4.4: Minimal XDMF input file

```
netcdf XDMF_example {
dimensions:
        phony_dim_0 = UNLIMITED ; // (4 currently)
        phony_dim_1 = 45164 ;
        phony_dim_2 = 3 ;
        phony_dim_3 = 22685 ;
variables:
        double W(phony_dim_0, phony_dim_1) ;
        uint64 connect(phony_dim_1, phony_dim_2) ;
        double geometry(phony_dim_3, phony_dim_2) ;
}
```

Figure 4.5: Corresponding HDF5 file

−−help: Dominates all other flags. No calculating is done, but a short help-message about every single flag is written.

−−i: Name of the input file. Must be specified.

−−o: Name of the output file. Default is the name of the input file+"_copy.nc"

−−it: Type of input. There are 3 possible options: "ASAGI", "GEBCO" and "XDMF". The input is expected to be formatted like specified in 4.2.2. Default is "ASAGI".

−−ot: Type of the output. As there is currently only one possible output-format, this option is ignored and set to "ASAGI" as specified in 4.2.1.

−−dx and −−dy: Difference from x to x+1 and from y to y+1 in the output grid in meters. The smaller the value, the slower the computation, but the higher the resolution of the output grid. Defaults to 500.

−−origin: Two parameters. The origin(0,0) of your grid where everything will be built around in (y,x) for XDMF and in (lat,lon) for GEBCO (format 123.75 instead of 123°45′.). As default the (approximated) center of the grid is taken[4]. For XDMF values of this point are kept, meaning for example if (0,24) is the center in the input, it will also be the center-point of your output. For GEBCO this points coordinates will change to (0,0). This can be changed with "−−neworigin". As a result, the point (0,0) is not necessarily the center of the output grid. It is advised to always also set "−−domain" when using this flag, otherwise some parts of the output grid might not have valid input data to get their values from. (see Fig. 4.10)

−−domain: Four parameters. Absolute distance from the root-point to the edges of the grid in kilometers. Default is half of the distance from edge to edge (approximated)[5]. In this case the zero-point will also always be the (approximated) center of the grid. Order of values are west, north, east, south. All values must not be negative. (see Fig. 4.10)

−−ignore: After reading input a check is done, wether data is available for all points of the requested output-grid. If not, the program aborts the conversion and in some cases might also crash. If you want to try anyway, set this flag but

[4] Although it looks like it, the input is not actually rectangular. Therefore, the middle of the latitude values and the middle of the longitude values are taken. The resulting point is the approximated center

[5] This distance is constant for latitude/y but changes for longitude/x depending on which latitude it is measured on. Here the lowest latitude value is taken.

keep in mind, that this might result in an error in the program and points without available data as a basis might have arbitrary values.

−−fast: Must be set for XDMF but is optional for ASAGI and GEBCO. The program will run much faster but will also need a lot of RAM. Make sure you have enough space for the computation. Otherwise an allocation-error can occur. More details about this can be found in 4.3.1

There are several flags specific to the single input types:

ASAGI: No special flags

GEBCO: GEBCO is converted to ASAGI via 2 different methods and therefore has special flags.

−−converttype: Two possible values. "sphere_convert" (which is the default) means for a single output-point the closest point in the input-grid is taken. "sphere_convert_clouds" means for one output-point all input-points within a certain cutoff-radius are taken and weighted depending on their distance from the input-point (see 4.3 for the exact calculation). Behaviour for points with no data within the cutoff-radius is undefined!

−−cutoff: The cutoff radius in meters. Default is 1000.

−−neworigin: Two parameters. As a default the new center-point will always be at (0,0) but this can be changed with this flag. The first parameter is the y-value, the second one is the x-value. The units are kilometers.

−−EPSG: For the special case of the center being at latitude 88.5113 and longitude 0, an alternative way of calculating the latitude and longitude for the new grids points is provided with the PROJ.4 tool. If not set, the law of Haversines is used as default method.

XDMF: Each output-point represents the rectangle of the grid where he is in the center. For each of the triangles their impact on a point is based on how much they overlap with the rectangle.

−−down: The z-values will be interpreted to point downwards. (Normally the z-values are interpreted to point upwards.) Note: The Output will still be pointing upwards! (see Fig. 4.7)

−−XYdirections: Eight possible options. As a default the X and Y values are interpreted to point east and north respectively ("en"). This can be changed to

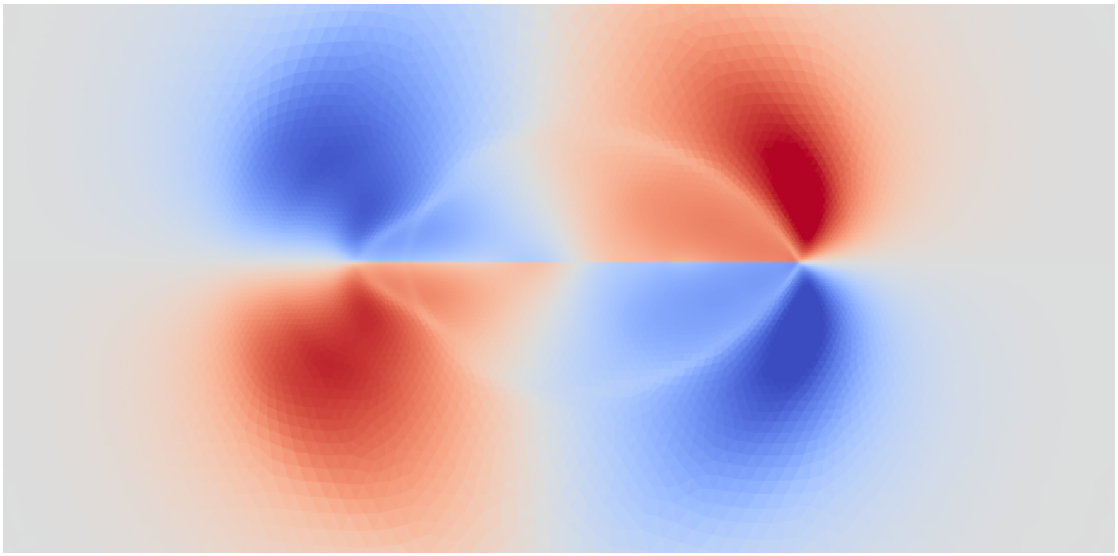


Figure 4.6: Example output without special flags

"en","ew","nw","ne","wn","ws","sw" or "se"[6]. Note: The Output will still be "en"! (see Fig. 4.9)

−−split: Prints every timestamp into its own file (without the *time*-dimension) instead of everything into a single one.

−−reverse: Iterate over triangles instead of rectangles. This is faster in most cases, as rectangles are sorted whereas triangles are not (see 4.3 for the exact implementation).

−−check: Calculates and prints the integral over the whole input and output for each individual timestamp. Note: If the output area was changed with "−−domain", no useful information is produced.

−−correct: Checks if the input data contains triangles multiple times and removes the duplicates. (see Fig. 4.8 for an uncorrected example)

−−boost: Uses the boost-library to calculate overlap between triangles and rectangles.

[6] e, w, n and s are abbreviations for east, west, north and south respectively.

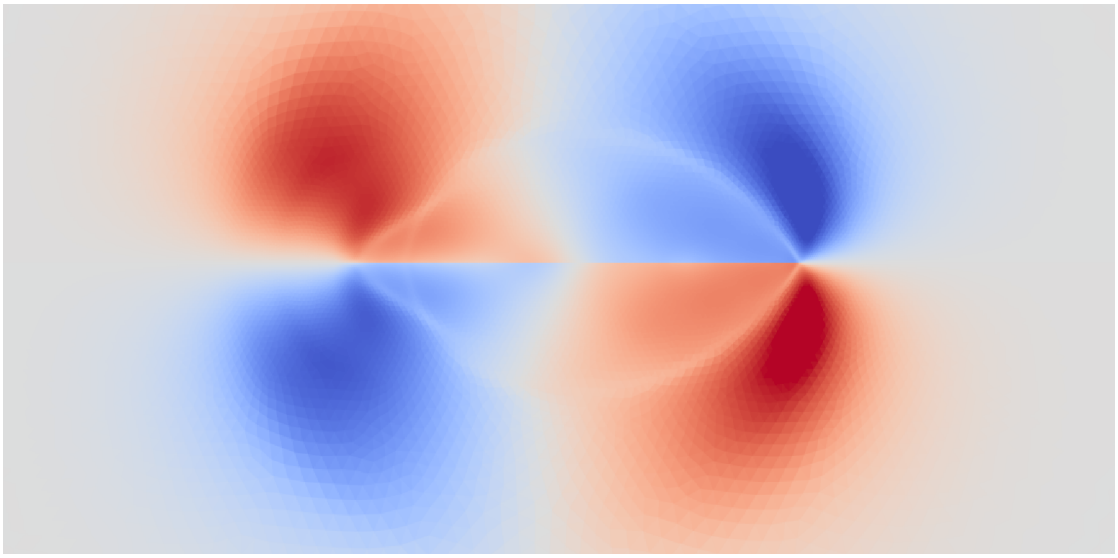


Figure 4.7: Influence of –down flag

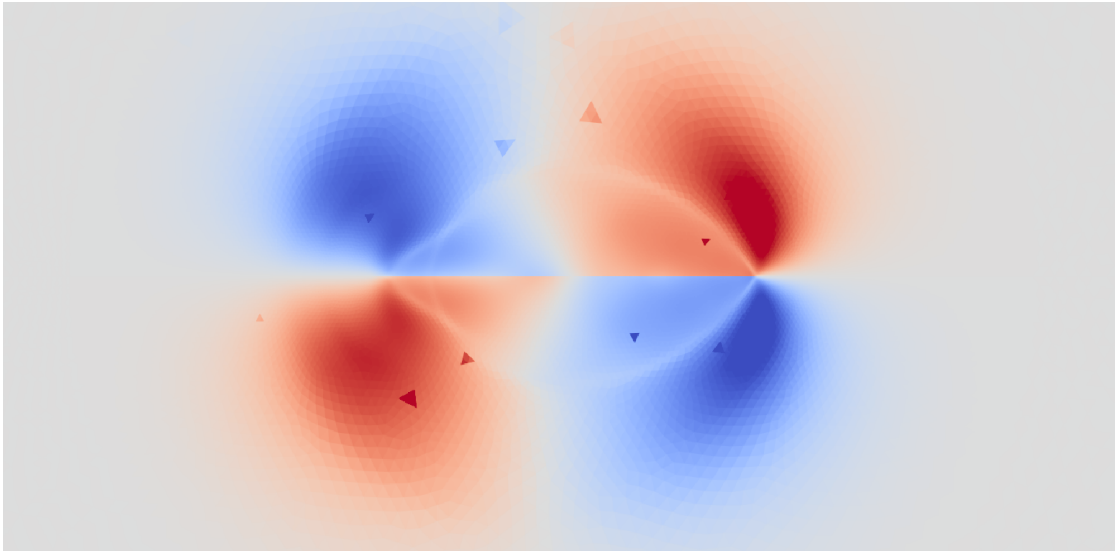


Figure 4.8: Influence of no –correct flag

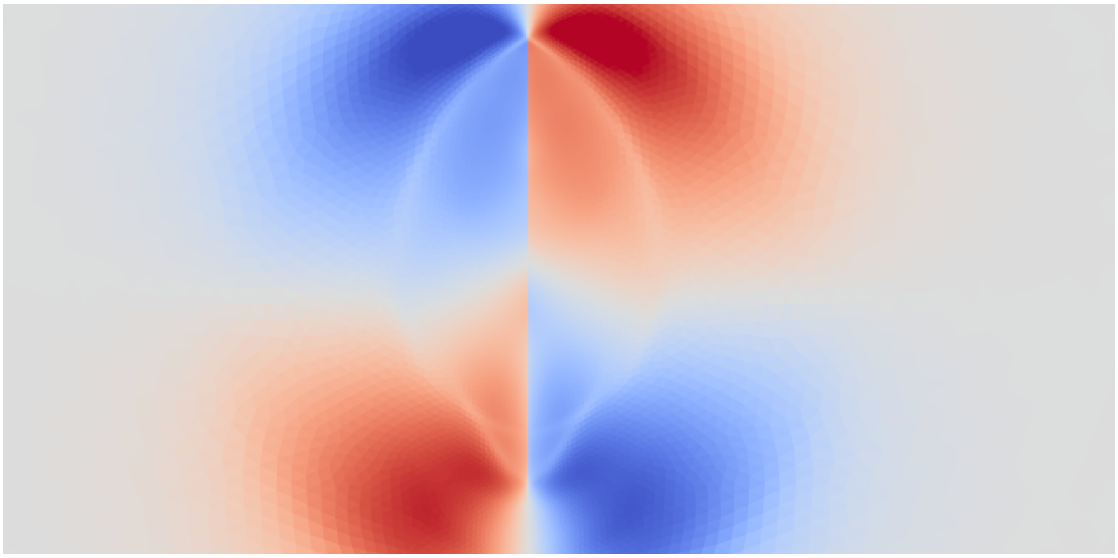


Figure 4.9: Influence of –XYdirections ne flag

Figure 4.10: Influence of –origin 0 0 –domain 20 20 20 20 flags

4.3 Implementation

The previous two chapters explained what this program is doing and its usage. This chapter will give detailed insight about the internal workings, what the mathematical basis for the calculations are and how they are implemented. Together with 4.2 it is a good starting point for subsequent developers to increase performance and/or precision or to add new functions (see chapter 7 for suggestions). After carefully reading this description one should have a rough idea about the structure and flow of information in the program. Every subsection will explain a specific part of the program and the files that are connected with it. A flow chart of the program is shown in Fig. 4.15.

4.3.1 General procedure

This chapter will describe the parts of the program, that dont have one specific task assigned to them, but are responsible for the basic structure of it. This means they have an impact on which functions are called and which computations are done. They dont do any calculating themselves.

User Input

Input is gathered right at the start in *Cmdline.cpp* using the boost library's *program_options*. After describing all possible flags at the beginning, every single one is checked for separately and if not present, a default value is set. If no input file is present and "−−help" is not set, an error is thrown. An error is also thrown if a flag has the wrong number of parameters, an unknown flag is set or the input has bad syntax. The information gets stored in the struct *info* (see 4.3.6).

Program Flow

The *Main.cpp* file and its *main* function are the core of this program. No calculations are done there, but the workflow is structured by calling necessary functions according to the user input. It is determined which of the two available modes (described below) gets used. Depending on which mode is used, the point in the program and the way reading and writing is done differs, but the actual calculations stay the same. For the input type *XDMF* there exist the additional option *check*, which prints the result of a comparison of the integral over the complete data area of the input and output. Due to imprecisions in the area-calculation the check is done twice for the output. First for the area that the rectangles are supposed to have and then for the area that actually got calculated.

Modes of computation - Fast

The procedure of *fast*-mode is quite simple. First the input is read by loading all the data into arrays. This way the computation will be done quickly as every needed value is already available and no time is lost reading from a file during the calculations. It has the downside, that a lot of RAM is used. This amount is impacted by the size and resolution of the output grid as well as size of the provided input. For the output this is at least $(size\_x\_dimension + size\_y\_dimension + size\_x\_dimension \cdot size\_y\_dimension) \cdot 8$ Bytes. For the input it is 8 Bytes for every provided input value[7]. Currently if there is not enough RAM available, this mode crashes with a *bad alloc* error. After reading, the conversion is done and at last some context information, like the dimension names and *actual_range* attributes, for the output file is set and the output gets written. Every single one of those three steps has multiple functions that get called depending on the input type and conversion type. The used structs are *sdata*, *cdata* and *file* (For which you can see a description in 4.3.6).

Modes of computation - Slow

The *slow*-mode requires only minimal amounts of RAM so it can also be run on local machines. This is achieved, by reading every single value from the file exactly when it is needed instead of storing them. Values are also immediately written to the output-file when possible and not stored anywhere. It is likely, that many values have to be read multiple times so this mode is quite inefficient. Reading from a file is also significantly slower compared to reading from an array. As a result of this procedure no separate functions for reading, calculating and writing exist. Different functions only get called based on the input type and conversion type, meaning in contrast to *fast*-mode the conversion functions also do the reading and writing. This also means that the output file has to be created before the conversion starts. At this point not all necessary information is available, so an additional function has to be called right before the end to compensate for that. Only the struct *file* is used. Note: There is currently no support for XDMF-input in *slow*-mode.

4.3.2 Input reading

The file *Input.cpp* and its array counterpart *AInput.cpp* perform the reading of input files, context information and in case of *AInput* also its data.

[7] This includes the position values for the x and y dimension in ASAGI and GEBCO as well as the *geometry* and *connect* values for XDMF

ASAGI

After opening the input file the length in the *x*-dimension (index 0 in the NetCDF-file, see 4.2.1) and *y*-dimension(index 1) are read. Then information about the three variables is gathered and the level of precision(see 4.3.6) is set. If the z-variable is of type double it is considered as high precision. Everything else is considered as low precision. As next step, all important global attributes and their length are stored. At this point *Input* is done, but *AInput* continues by reading the data. In case of low precision, the z-variable has to be read one by one as the array used for calculation is always of type double.

GEBCO

After opening the input file the length of the *x/longitude*-dimension(index 1 in the NetCDF-file, see 4.2.2) and *y/latitude*-dimension(index 0) are read. Then information about the three variables is gathered and missing information needed for the output and the level of precision are set. If the z-variable *elevation* is of type double it is considered as high precision. Everything else is considered as low precision. From this point on the procedure is the same as for ASAGI. Attributes are read and *AInput* loads the data.

XDMF

As XDMF is only available in the *fast* version, its input reading only exists in *AInput*. For reading the xml-document the property-tree part from the boost-library is used. It reads the number of triangles, the number of single-points and their respective file and location. The location of the z-values *W* and the level of precision are also stored. Next the number of timestamps and the time-values of them are read. At last the respective files for the needed parts of the data are opened and the data is loaded into the arrays.

4.3.3 Conversion

The two files *Converts.cpp* and its array based counterpart *AConverts.cpp* are the heart of the program, as they do the actual work in converting one type of data to another. The main difference again is, that *AConverts* works on arrays which contain all the data while *Converts* loads a value directly from the given file when it needs it and immediately writes the result into the output file. *Converts* also does not support XDMF as input. For all conversion types it might be necessary to slightly adjust the "dx" and "dy" values to exactly fit into the new grid after its size is calculated. For this the highest possible number that is smaller than the "dx"/"dy"-value and a divisor of the

size in x or y directions is calculated. They will never be made larger so the resolution is always at least as high as the user defined.

Simplescale

As the output is always of type ASAGI, input of type ASAGI is only changed a bit. The grid the data is based on will be rescaled to a new resolution. It can be both higher or lower than the original one. The algorithm deciding which value to put into the cells for the new grid simply takes the point from the original grid, that is closest to the new one. After calculating the new values for the attributes this function is already done.

Sphere to xy

Despite looking like a grid on first glance, input of type GEBCO is based on the earths sphere and therefore has to be converted to an actual flat rectangular grid first. This conversion also allows the user to clip only a part of the input by setting a centerpoint(definded by coordinates in latitude and longitude) and the distance to the edges in west, north, east and south direction in kilometers. If this is not the case the whole domain is converted. In contrast to ASAGI, at this point we need to check if the area the user wants to clip is actually available in the input. If not, an error is thrown. This check can be ignored with "−−ignore" flag. After finishing preparations the conversion is started. Since the input grid is on a sphere and given in degrees, the cartesian distances "dx" or "dy" need to be converted to find the position of a neighbouring point. To locate the corresponding point in the original grid to a point in the new grid, we start with latitude and longitude coordinates of the bottom left corner and calculate the new points coordinates in two nested for-loops. The explanation and formulas for this conversions are described in 4.3.5. The outer one walks "dx" meters east and represents the bottom edge of our output grid. The inner loop iterates over the resulting points and walks "dy" meters north for each of them. This way every output point gets its respective latitude and longitude coordinates. This can be seen in Fig. 4.11.
For the value assignment there are two conversion types, *simple* and *cloud*, each having its own function. At last the new attribute values are set and the function returns.

Converttype simple: The *simple* method directly takes the value from the point of the original grid which is closest to the point of the new grid by accessing its z-value directly via its index. This method is most useful, when the resolution of the input grid and the output grid are about the same, or if the output grids resolution is higher, as in these cases the information gets projected most accurately.

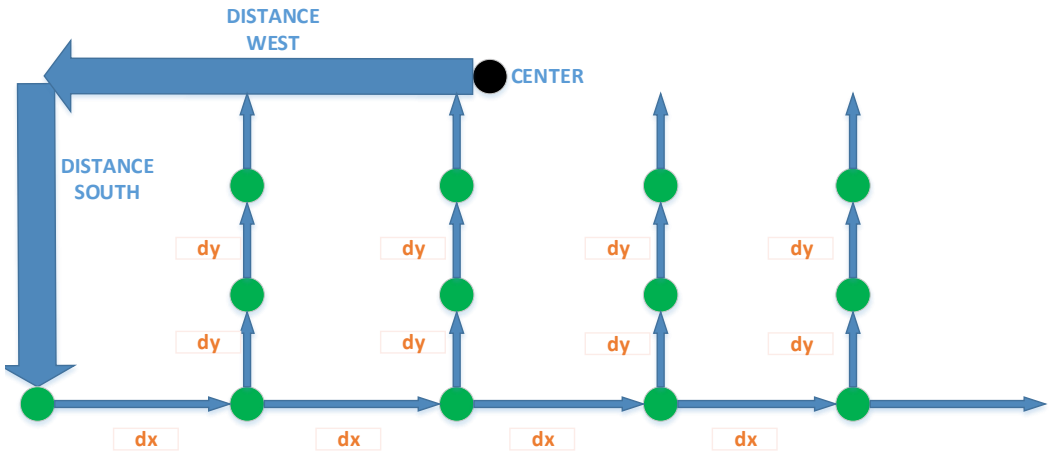


Figure 4.11: Visualization of GEBCO conversion

Converttype cloud: The *cloud* method takes every point from the original grid into account, that is within a given cutoff radius from the from the point of the new grid. The impact of the original grids points is based on their respective distance to the new grids point. The value is calculated according to this formula:

$$z(x) = \frac{\sum_{x_n} z(x_n) * (r - \mid x - x_n \mid_2)}{\sum_{x_n} r - \mid x - x_n \mid_2)}$$

with x being the new grids point, x_n the points from the original grid within the cutoff radius, $z()$ their z-value and r the cutoff radius.
The *cloud* method uses the *diamond*-algorithm which is much more complicated than the algorithm of *simple*. It uses the closest point from the original grid as starting value (calculated like in *simple*). This point has an absolute index-distance of zero. If it is already outside the cutoff radius its value is taken and the algorithm terminates. If it is inside, all the points from the original grid with an absolute distance for the indexes of one are checked if they are inside the cutoff radius. Next all points with an absolute distance for the indexes of two are checked. This continues as long as there is at least one point from the original grid with an absolute distance for the indexes of x that is still inside the cutoff radius. If that is not the case it is guaranteed, that all points with a larger an absolute distance for the indexes are also outside the cutoff radius. The principle of this is shown in Fig. 4.12. Finally, all found points and their z-values are added according to the formula, the new value is set and the algorithm terminates. The reason this algorithm has the name *diamond*, is that the order it checks all points with the same index distance x in is shaped like a diamond. From the center the algorithm

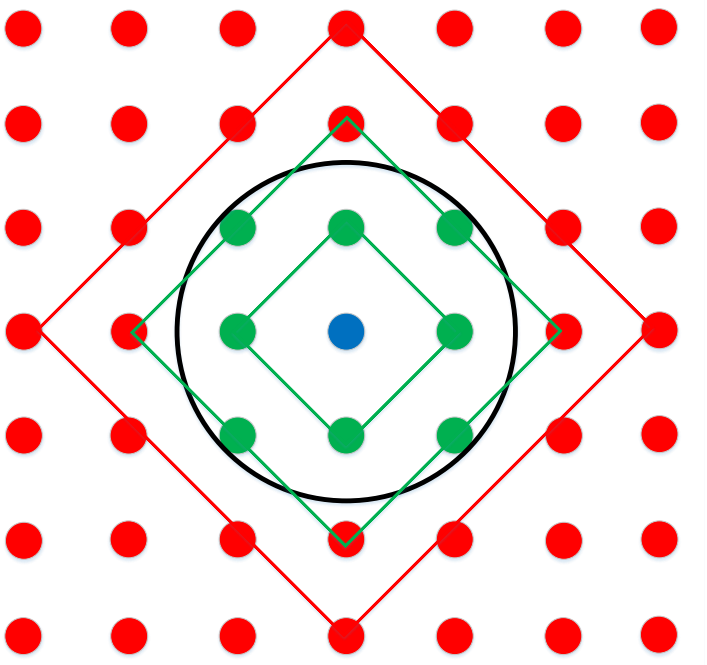


Figure 4.12: Visualization of diamond algorithm

goes x steps (meaning indexes in the original grid) to the northernmost point and from there the next point is one step to the east and one step to the south. This continues until the easternmost point is reached. From there we go to the southernmost point, the westernmost point and back to the northernmost point. With this strategy, calculating the actual index-distance of any point is not necessary.

Check data range: This function calculates the latitude and longitude coordinates of the 4 corners of the requested output grid and checks whether there is any provided data within their cutoff radius. If yes, it is guaranteed, that all the other points will also have values to work with. If not, the function returns with "false", meaning the check for data failed.

Convert XDMF

This conversion is only available in *fast* mode (see 4.3.1 for reference) and therefore only implemented in *AConverts.cpp*. The conversion of XDMF is the most complicated one, as the input is neither a grid, like in ASAGI, nor similar to a grid, like GEBCO, but a mesh of triangles. These triangles are given by a collection of points in *geometry*, whose points

form triangles in *connect* for which the values are stored in the field *W*. XDMF also has multiple timestamps that have to be taken into account in the conversion (see 4.2.2 for the exact specification). The strategy after which the new z-values are calculated is simple. Every new point represents a rectangle with size $dx \cdot dy$ and overlaps with one or more triangles. The value of these triangles is multiplied by the percentage of the rectangle they cover. This way the resulting value is the integral over the whole area of the rectangle. The difficulty lies in calculating the respective overlap of a rectangle and a triangle and doing so efficiently. One can iterate over the rectangles and check every triangle (which is the standard way) or iterate over the triangles and check every rectangle. The latter is usually faster, as the rectangles are sorted and therefore some precalculations can be done which give a rough idea about the area the triangle is in. This way not all rectangles have to be checked. After the conversion, the values for the attributes are set and the function returns.

Precalculations: Before actually converting, a few preparations have to be done, as the input can have some special unwanted characteristics as a result of how the data was produced. It is possible, that *connect* contains some triangles multiple times. These duplicates have to be removed. This is done by setting the *connect*-values of unneeded copies to "-1", excluding them from being considered in the conversion. It is also possible that the x-axis and y-axis do not point east and north respectively, as they are guaranteed to do in the other two input types. This cannot be detected by the program and has to be specified by the user, so both axis can be turned around to point in the wanted direction. Like for GEBCO, the user can specify a certain area he wants to clip so it has to be checked, if he actually also provided data for the requested area. If not, an error is thrown. This check can be ignored with the "−−ignore" flag.

Convert simple: Fills the z-values for the output grid. This is done by iterating over the output grids points. Each point representing the area of a rectangle with itself in the center and the distance "dx/2" to the left and right as well as "dy/2" to the top and bottom. This way all the rectangles have the same size. For every rectangle every single triangle is checked for its overlap with the rectangle either with a calculation based on the boost library or a self-implemented approach. The difference is explained in 4.3.5. Next for each timestamp the respective value of the triangle is multiplied by the percentage of the rectangles area it covers and added to the rectangles value for this timestamp. This is possible, since the position of the triangles and rectangles don't change over time, meaning the overlap also stays the same for every timestep. For each rectangle, the accumulated overlapping areas are stored. Due to precision errors these areas will sometimes not perfectly match the exact intended rectangle area, so

this array can later be used for calculating the absolute error for the whole conversion. As the area for all rectangles is a constant and the overlapping percentage is calculated by dividing the area of the overlap by the area of a rectangle, time can be saved, by multiplying the triangles values with the overlapping area and only dividing the final sum once, after all triangles have been traversed.

Convert reversed: The reversed conversion works in a similar way to the standard one. The actual calculation of the overlapping areas, percentages, checksums and z-values is the same as in 4.3.3, but this algorithm iterates over the triangles instead of the rectangles. In contrast to the unstructured triangular grid, the rectangles are sorted and therefore it is possible to determine an area for each triangle in which every rectangle it overlaps with has to lie in. This area is called the *boundary rectangle*. The size of this area is in most cases only a fraction of the entire grid, which means a lot of rectangles do not have to be checked resulting in a significant performance increasement. The boundaries of said area are determined by the highest and lowest x and y value of the three triangles corner-points. The corresponding indexes in the output grid to these boundaries are taken as start and ending for the rectangles that have to be checked.

Check XDMF data range: This function checks if the northernmost, easternmost, southernmost and westernmost requested coordinates are inside the given data range.

Check double triangles: Due to SeisSol being highly parallelized, some parts of the data used have to be duplicated, which results in some triangles being listed multiple times in the *connect* list. These duplicates need to be removed. In order to do that, this function iterates over the *connect* array where three consecutive values represent one triangle. For example the indexes 0, 1 and 2 belong together as well as the indexes 3, 4, 5 etc. Each of these triples is compared with every other triple to check if they have the same values. As the values represent the points from the geometry this means, that a triangle is contained multiple times, so the values of one triple are set to (-1, -1, -1). These duplicates are later ignored in the conversion. This algorithm has a runtime of $O(n^2)$, which is far too slow for large datasets, but an efficient way of hashing can be used, because duplicated triangles are guaranteed to have all three values in the exact same order. An array the size of number of single points, containing lists of pairs is created and the *connect* array is traversed only once. The first value of every triangle is used as index for the list array. The corresponding list is the checked for a pair, containing the second and third value. If one is found, the three values of the active triangles are set to "-1". If not, a pair containing the second and third value is added to

the list on this index. This speeds the process up to almost O(n) runtime. The function returns with "true" if the data was correct (meaning it has not found a duplicate) or with "false" if there were duplicates.

4.3.4 Output writing

The two files*Output.cpp* and its array counterpart *AOutput.cpp* are responsible for writing the newly calculated data to a NetCDF-file with ASAGI-like structure. The difference between the two files is, that AOutput works on arrays that get passed to it while Output does not fill in any data but only sets the files structure and attributes.

Output in slow mode

The *slow* conversion mode works directly on files. Therefore, an output file has to be created before the conversion is started. For this the output writing is split into two functions. The first one *asagioutputcreate* creates the file and sets the dimensions and variables as well as some of the attributes. It is called in the beginning of the conversion functions. The second function is *asagioutputfinish* and it sets all the attributes for which the data was not known when creating the file.

Output in fast mode

The *fast*-conversion type works on arrays and has all data loaded into the RAM. Because of this one single call of the function *aasagioutput* is enough to create the file with all the dimensions, variables and attributes. This function can then also immediately fill in all data, but its structure is more complicated than the straightforward functions of *Output.cpp* due to some optional flags and characteristics the XDMF input has. These are the *time*-dimension of XDMF which neither the input ASAGI nor GEBCO has and the *–split* flag, which changes how this difference is handled. If it is set, every single timestamp of XDMF gets its own file and a counting number is added to this files name. Without this flag the standard way of output is adding the dimension *time* to the output file and the dependencies of the z-variable change from (y,x) to (time,y,x). Because of this the output file differs slightly from the standard format (see 4.2.1). So on various points through the method it has to be checked what the input was in order to set the correct output. *AOutput.cpp* also has a second function, but it is only used to set some hardcoded values that are needed to create the output file correctly.

4.3.5 Supporting calculations

This chapter is about the *Helpers.cpp* file which contains a variety of functions that are needed by different parts of the program. They got extracted to make it easier to call them from everywhere. While the smaller ones are mostly self-explanatory and will therefore only be briefly summarized, the bigger ones can have a large impact and are explained in detail.

Finding Maxima and Minima

For *slow*-mode no values are stored so maxima and minima need to be found from a given variable with one dimension in *dminimum* and *dmaximum* or two dimensions in *doubleminimum*, *doublemaximum*, *floatminimum* and *floatmaximum*) from a NetCDF file. Getting the minimum and maximum values for the single-points from a XDMF input is a bit more complicated since they are all stored into one single array. Here all three coordinates for single points are stored sequentially, which is why this structure has to be taken into account. So for XDMF *axdmfxmax*, *axdmfxmin*, *axdmfymax*, *axdmfymin*, *axdmfzmax* and *axdmfzmin* are used to only check every third value.

Area calculation

For the conversion of XDMF areas of rectangles and triangles are needed.

Rectangles: While the area of a rectangle is easily calculated by multiplying *dx* and *dy* which are the length of the rectangles sides, a second option is provided which constructs a rectangle polygon with the boost library and returns its area value.

Triangles: The area of a triangle is calculated with the xy-coordinates of the corner-points as inputs according to the following formula described at [38]:

$$\frac{|\ (A_x \cdot (B_y - C_y) + B_x \cdot (C_y - A_y) + C_y \cdot (A_y - B_y))\ |}{2}$$

Calculations on the earths sphere

Four functions are used only when converting GEBCO. Based one the Haversine-formula for great-circle distances and the approach from [39], the *distancelatlon* function calculates the distance of two points on a sphere. The input are two points x and y with their respective latitude lat_x, lat_y and longitude lon_x, lon_y values. The output is

the distance in meters. The following formula is used:

$$\arcsin(\frac{\sqrt{dx \cdot dx + dy \cdot dy + dz \cdot dz}}{2}) \cdot 2 \cdot 6371000$$

with

$$dz = \sin(lat_x) - \sin(lat_y)$$
$$dx = \cos(lon_z) \cdot \cos(lat_x) - \cos(lat_y)$$
$$dy = \sin(lon_z) \cdot \cos(lat_x)$$
$$lon_z = lon_x - lon_y$$

For calculating the latitude and longitude of a point with a fixed distance and bearing from the starting point *calculatenewlat* and *calculatenewlon* are used. They use these formulas from [40] for latitude :

$$\varphi_2 = \arcsin(\sin(\varphi_1) \cdot \cos(\delta) + cos(\varphi_1) \cdot \sin(\delta) \cdot \cos(\theta))$$

and longitude:

$$\lambda_2 = \lambda_1 + \arctan 2(\sin(\theta) \cdot \sin(\delta) \cdot \cos(\varphi_1), \cos(\delta) - \sin(\varphi_1) \cdot \sin(\varphi_2))$$

where φ is latitude, λ is longitude, θ is the bearing (clockwise from north), δ is the angular distance d/R; d is the distance travelled and R the earth's radius.
For the special case of latitude 88.5113 and longitude 0 as a center, the *convert_proj* method can be used as an alternative to the *calculatenewlat* and *calculatenewlon* methods. It is based on the PROJ.4 tool from [41], takes the cartesian distances from the center as an input and converts them to their respective coordinate values.

Check integral

Only gets called when "−−check" is set for XDMF. For asserting that the conversion was accurate to a certain degree, this functions calculates the integral of z-values over the complete input and output data. It returns 3 different values for each timestamp. The integral over all triangles, the integral over all rectangles using the area they are supposed to have and the integral over all rectangles using the area that actually got calculated for them. The latter two are not identical due to precision errors when calculating overlaps.

Check overlap

Since XDMF is based on triangles and the output on rectangles, overlaps between those two forms need to be calculated in the conversion process. There are two separate functions for calculating these areas. One relies on the boost-geometry-library while the other uses a self-implemented approach based on the Sutherland-Hodgman algorithm as explained at [42]. Both approaches are compared in regard to runtime and accuracy in chapter 5.

Check overlap with boost: After creating two polygons (a rectangle and a triangle) from the input and ensuring they are set correctly, it is checked whether the triangle completely covers the rectangle (in which case the area of the rectangle is returned) or vice versa (then the triangle area is returned). In case of a non-complete overlap, an intersection polygon is calculated and its area is returned.

Check overlap with self-implementation: This function has much more complicated code in comparison to the boost approach but works much more precise and faster. The algorithm used is an adapted version of the Sutherland-Hodgman algorithm for clipping polygons, where the triangle is used as subject-Polygon and the rectangle as clipping-Polygon. A pseudocode of this can be found in Fig. 4.13. The presented Pseudocode is a simplified version from Wikipedia[43], that gives a quick and easy to understand overview, but a more detailed explanation can be found at [42] and [44].
The code complexity and runtime can both be drastically reduced due to the adaption of two given facts: First, the calculated intersection is always between a rectangle and a triangle so a general approach is not necessary and the code can be optimized for this single case. Second, the rectangle is always parallel to the coordinate-axes which makes checking for crossing-points of two lines very easy. As illustrated in Fig. 4.14 for the left and right edge of the rectangle the slope of the triangle edge can simply be multiplied with the distance to the rectangles edge on the x-axis. The top and bottom edge work similarly. This way no complex calculations regarding the crossing point of two lines are needed. The result is always either empty (in which case zero is returned) or a convex intersection polygon. This polygon is divided into separate triangles with one fixed corner always being the first point of the polygon. The area of these triangles can then be calculated, added and returned.

Get indexes

In order to save time for the XDMF conversion, a boundary rectangle is calculated, which guarantees, that all overlapping rectangles for a given triangle are inside it. For

```
List outputList = subjectPolygon;
  for (Edge clipEdge in clipPolygon) do
    List inputList = outputList;
    outputList.clear();
    Point S = inputList.last;
    for (Point E in inputList) do
      if (E inside clipEdge) then
        if (S not inside clipEdge) then
          outputList.add(ComputeIntersection(S,E,clipEdge));
        end if
        outputList.add(E);
      else if (S inside clipEdge) then
        outputList.add(ComputeIntersection(S,E,clipEdge));
      end if
      S = E;
    done
  done
```

Figure 4.13: Pseudo-Code for the adapted Sutherland-Hodgman algorithm from Wikipedia[43]. More detailed explanations can be found at [42] and [44].

this the three corner points of this triangle and their maximum and minimum x and y are used. To find out with which rectangles this boundary overlaps, the edges of the boundary are used to determine the x- and y-indexes of the rectangles that lie on the edge of the boundary. For this we need the size of the grid, the starting values of the grid and the dx and dy values. For one index the distance from the edge to the starting value for the respective dimension is taken and divided by the delta-value. As the index is an integer, for the left and bottom edge the value is rounded down, the top and right edge are rounded up. This results in the 4 wanted indexes which are then returned via the struct *indexes*.

4.3.6 h-files and level of precision

Information.h

This file contains all structs that are used in the computation. These are *programinformation* for the storing users input and which flags where set, *fileinformation* and *afileinformation* which contain information about the input and output files for *slow*-mode and *fast*-mode respectively, *asimpledata* containing the data for ASAGI and GEBCO input and output as well as XDMF output, *acomplexdata* for the XDMF input data, *indexes* for the triangle boundary and *point* which is used for the self-implemented checkoverlap

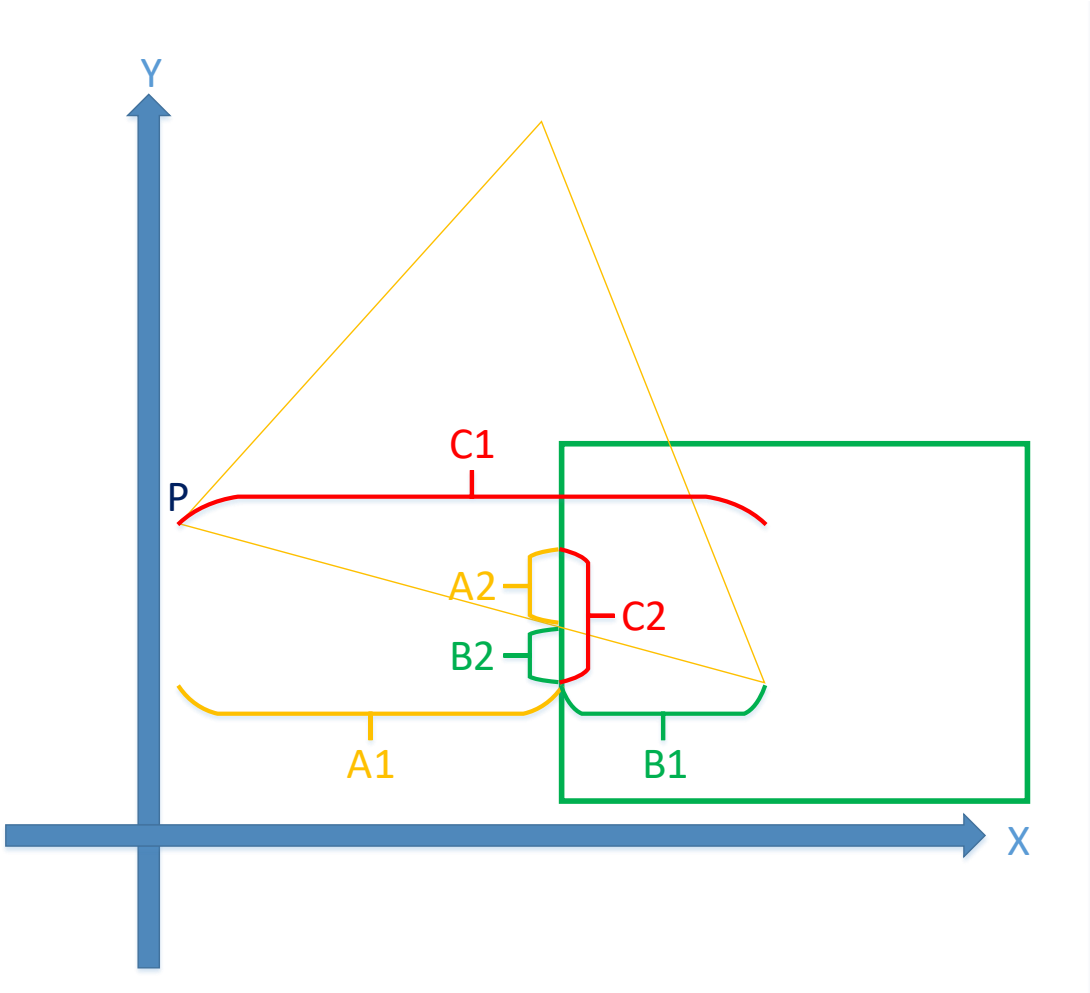


Figure 4.14: Visualization of the adaption for Sutherland-Hodgman algorithm

Figure explanation: The relation of A:B:C is the same in both directions and the x value of the point of intersection is the x value of the clipping edge and therefore known without calculations. As a result the distance from point P to the clipping edge (A1) is calculated by subtracting the x value of P from the x value of the point of intersection. This is then multiplied with the slope of the triangle edge $\frac{C2}{C1} = \frac{A2}{A1} = \frac{B2}{B1}$ to get the distance A2 which is added to the y value of the starting point P to get the y value of the point of intersection.

function.

Alloc.h

A small file containing the *splitdouble* class which enables splitting a single *double* array into multiple arrays without making it necessary to change the rest of the implementation. The result of this is, that the RAM can be used much more efficient since it is no longer needed to find one big continuous block.

Level of Precision

The data the program works on is always of type double, but if the input did not have this kind of precision (e.g. float or short), the output is adjusted accordingly and gets downgraded to type float.

Testing

Test.cpp contains some tests mainly for the *Helpers* functions. It is not involved in the program itself.

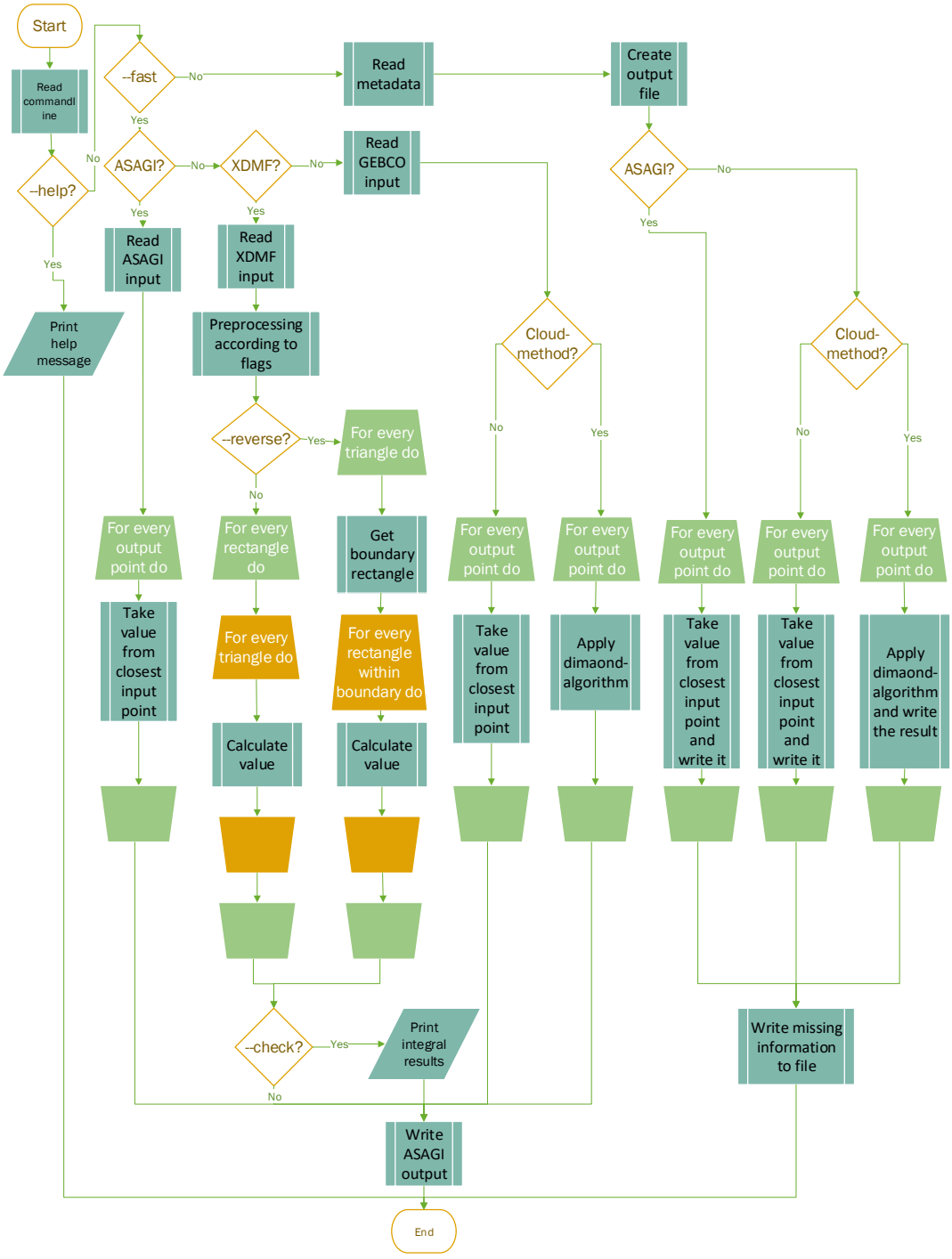


Figure 4.15: Program flowchart

5 Evaluation of different program options

As shown in the previous chapter, this tool provides multiple options in how the input can be handled and converted. These decisions can have a great impact on runtime and accuracy of the program. This chapter will evaluate how the program is doing in terms of speed and accuracy and a comparison of the most relevant flags in this regard will be done.

5.1 Speed

5.1.1 Comparison fast and slow mode

The comparison for the runtime of *fast* and *slow* mode was done on a single node of the dual socket Intel SandyBridge-EP Xeon E5-2670 from the MAC cluster which according to [45] has 128 GB RAM. Two separate comparisons are done for the input types ASAGI and GEBCO. For XDMF its flags are examined. For this the quad socket Intel Westmere-EX Xeon E7-4830 with 512 GB RAM from the MAC cluster specified at [45] is used.

Comparison for ASAGI

Both modes were compared on two different input files and four different output resolutions. The input files were of type ASAGI and had 1.600.000 (small) and 112.000.000 (big) values respectively. The input grid resolution was 500m in both directions. The output grids got scaled to 50000m, 5000m, 500m and 50m[1] for resolution in both directions. This means the first two outputs have a lower resolution, the third one has the same and the fourth one a higher resolution than the original. Results are shown in Fig. 5.1. The difference in the two modes for ASAGI type files is barely visible. This is because the value calculation for ASAGI is one simple division to get an index and as a result both modes spend most of their time reading and writing. The only noticeable difference is with very small output grids compared to input grids. In those cases the

[1] Due to runtime restrictions this resolution got stopped early and scaled to the full runtime based on the percentage of conversion, that was done at that point.

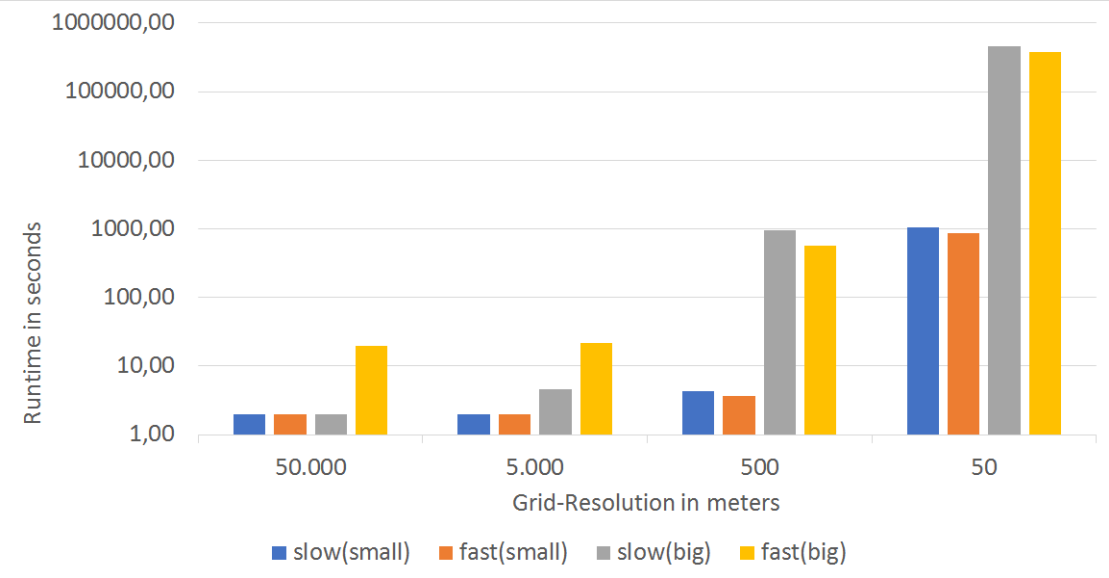


Figure 5.1: Comparison of *fast* and *slow* mode regarding runtime for ASAGI

fast mode wastes time loading all input values into the RAM, since most of them are not needed. For scaling an ASAGI file the choice of mode therefore has not much impact.

Comparison for GEBCO

The comparison for GEBCO type files was done with a single input file from [37] centered around Sumatra and an interval of 30 arc-seconds (about 927m). Here *fast* and *slow* mode are compared again, but also the runtime of the two different approaches available for GEBCO are compared. These are the standard version which takes the closest input point for its output point value and the cloud-method, which takes everything within a given cutoff radius. For the latter method two different radiuses are compared. The first one is 1000 meters, which is slightly more than the resolution of the input grid. The second one is 5000 meters[2]. The runtime results are shown in Fig. 5.2. In contrast to ASAGI, for GEBCO the difference between *fast* and *slow* mode is clearly visible. For very small outputs the *slow* mode is still slightly better, because it takes the values it needs. This stays true for the standard conversion method, since it works very similar to ASAGI, but for the cloud-method *fast* mode is the better choice by far. This is because for this method one input point can have an impact on multiple

[2]Due to runtime restrictions the *slow* mode for this radius got stopped early and scaled to the full runtime based on the percentage of conversion, that was done at that point.

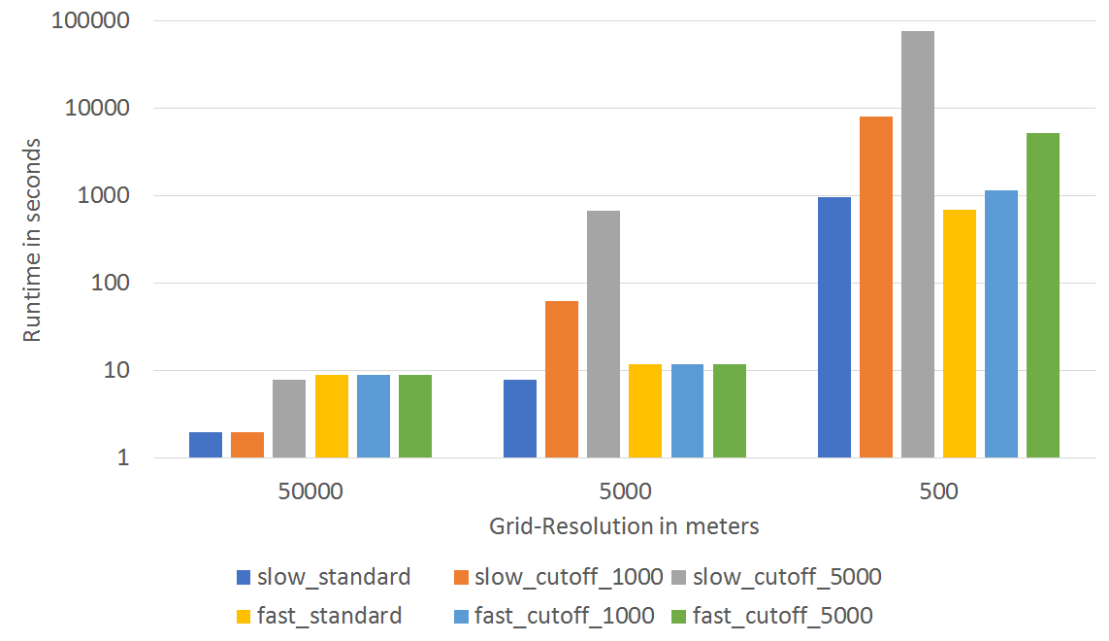


Figure 5.2: Comparison of different program options for GEBCO regarding runtime

output points and is therefore also needed multiple times resulting in many more accesses than just reading and writing once. The benefit of having the data loaded into the RAM becomes greater, the better the resolution and the larger the cutoff radius gets. For example for the best resolution of 500 meters 85% of the time is saved for the cutoff radius 1000 meters and 93% for 5000 meters. So in summary choosing a mode for the standard conversion has almost no impact, but for the cloud-method choosing *fast* mode has massive runtime benefits.

5.1.2 Additional options for XDMF

Since the XDMF input type does not exist in *slow* mode, no comparison can be done. But compared to ASAGI and GEBCO, XDMF has some additional flags that require further computing and therefore also additional time. First one standard computation is run with the *–reverse* flag being active. This is then compared to the same run with one additional flag set. For the small input with 45164 triangles and 11 time steps a resolution of 500 meters for the output grid was chosen. The large input with 7068432 triangles and 1001 time steps has an output grid with a resolution of 5000 meters. The results can be seen in Fig. 5.3. The standard reversed conversion is the fastest for both small and large inputs.

The *–correct* only flag has negligible impact on runtime, but guarantees a correct input with no duplicates. It should therefore always be set.
The *–split* flag increases the runtime slightly for small files, because additional time is spent creating files, but this also becomes negligible for large files. Therefore, the decision on setting this flag is only impacted by the wanted output format and not by runtime.
While *–check* causes only insignificant runtime increase for small files it can cause a big increase for large files. This gets up to 43% additional runtime for the evaluation scenario. As this flag is only useful as long as no special domain was chosen, creates additional runtime and has no special benefits for the calculation itself it should be reserved for testing.
The *–boost* flag causes the program to use an approach based on the boost library instead of a self-implemented one. As this approach is not specialized for the scenario it causes a significant runtime increase for both small and large files. In this case it went from three to 70 seconds for the small input file and from 38 to 109 minutes for the large input file. So when prioritizing runtime, the self-implemented approach should be chosen over the boost version.
The last option examined is not setting the *–reverse* flag. This causes enormous runtime increasement for both small and large files. The runtime for the small file went up from three seconds to one and a half hours. For the large file the evaluation had to be stopped due to runtime restrictions, but is estimated to have gone up from about 38 minutes to several weeks. Unless the speed of the standard conversion gets increased significantly (e.g. by parallelization or precalculation, see 7.1.3 and 7.2.2) it cannot be considered a viable option compared to the reversed approach.

5.2 Accuracy

Even more important than a programs runtime is its accuracy in calculating results. Because the ASAGI conversion does not calculate values, an accuracy analysis for it doesn't make sense. The GEBCO and XDMF conversions however do their own value calculation and have therefore to be tested for accuracy. There exist separate methods for both inputs.

5.2.1 GEBCO

For GEBCO objectively measuring accuracy inside the application proves to be quite difficult, as there is no way to tell, if the calculated results correctly represent the actual data. The only way to test the results is to do a full tsunami simulation and compare the results with the real world data. Doing this however is not completely reliable, as

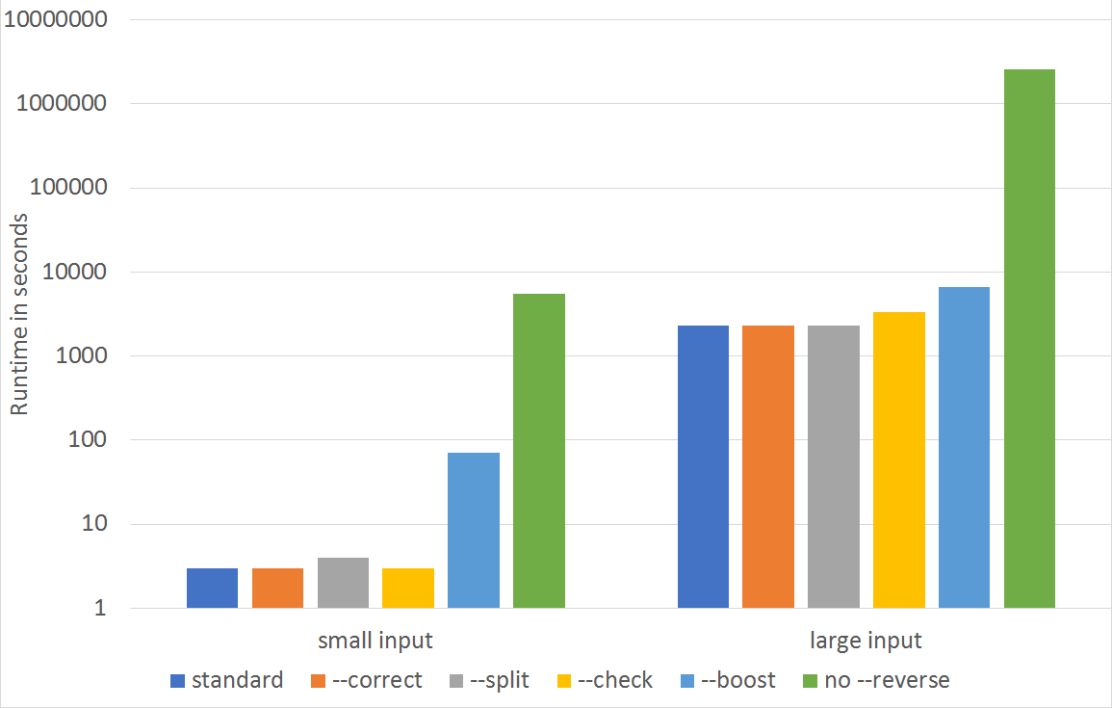


Figure 5.3: Comparison of different flags for XDMF conversion regarding runtime

there are other potential sources for errors in the earthquake input, the simulation itself and the inaccuracy in measurement of real world data.

Conversion Types

As most of the output-points do not exactly lie on top of an input point, either the closest value has to be taken or a separate value has to be calculated via interpolation. None of those two approaches is perfectly accurate and the actual value is unknown. The accuracy can however be improved by choosing the correct program flags. For GEBCOs 30-arsecond interval grid, every output grid with a resolution better than 1000 meters should choose the standard conversion method or the cloud-method with a cutoff radius of about 1000 meters at maximum. This way every input-value will have an impact and every output point gets the value it is most likely to have. For output grids with a resolution worse than 1000 meters, the cloud-method with a cutoff radius about half the resolution is the best choice. For these grids, information for some input points would get lost otherwise. This way the output points still don't have their perfect value, but are representing the area around them in a quite accurate way.

Conversion Methods

The results for the comparison of the two conversion methods are shown in chapter 6. While the conversion with proj4 almost perfectly aligns with the real data, the haversine conversion has a slight shift in timing and altitude.

5.2.2 XDMF

In contrast to GEBCO, the XDMF conversion allows a mathematical check for accuracy, as no information gets lost in the conversion process and no interpolation is done, as the values for the rectangles get calculated via an integral of the input values for their area. Because of this an integral of the complete input domain and the complete output domain are supposed to have the exact same value. This can be checked with the *–check* flag. The actual accuracy differs for the self-implemented approach and the one using the boost library. Both are compared using the small example input that was also used for runtime measurements in 5.1.2. The conversion is done with a resolution of 50 meters in both directions.

Self-implemented approach

For this case the biggest difference in integral values appears in time step 4 with an expected value of -3.6854463e-8 and a calculated value of -3.6867587e-8 resulting in a

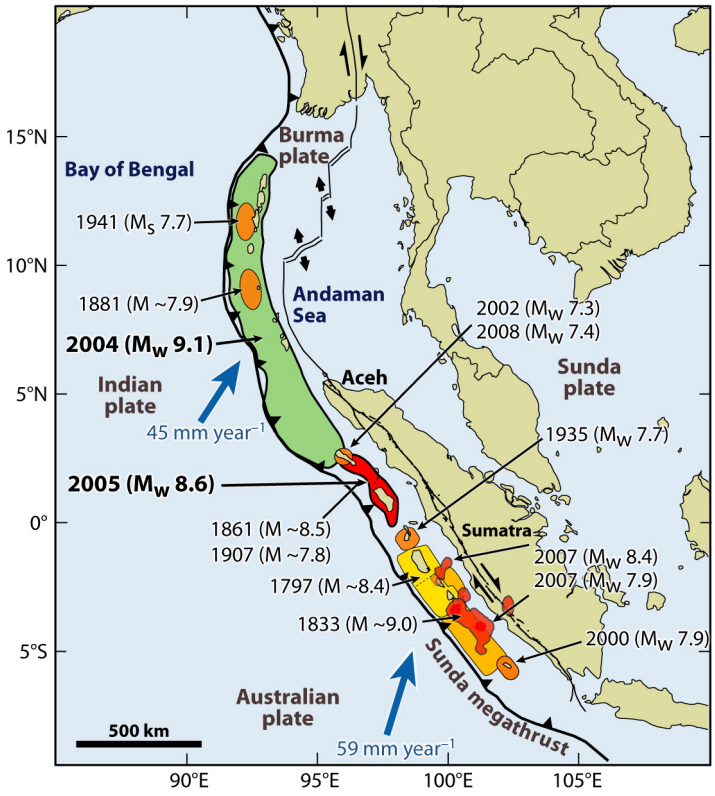
total difference of 1.3124e-11 and a relative error of 3.5610341e-4. With a total number of 3024000 rectangles this is an average error of 4.3399471e-18 and a relative error of 1.1775906e-10 per rectangle.

Boost library

Using the boost library, the results are far too inaccurate for further use in scientific computing, as the same 4th time step as before has a calculated value of 3,6507379e-3 which is nowhere near the expectations. Using the boost library is therefore significantly slower and more inaccurate than the self-implemented approach and is disadvised to use.

6 Application on real data

In this chapter the result of applying this tool to real data is shown. For this the simulated earthquake data from SeisSol for the 2004 Sumatra Earthquake is taken as input. The converted file is then fed into sam(oa)^2 and the result of this tsunami simulation is compared to measured data from the 2004 tsunami. The conversion was done with a resolution of 5000m in both directions using the self-implemented approach for calculating overlaps. As bathymetry data the 30-arcsecond interval data from [37] was taken and clipped with the center at the coordinates (0, 88.5113) and 2000km distance in all four directions. This data was converted with the haversine-method and the proj.4 method using the standard conversion type.
The Sumatra Earthquake appeared west of the north end of sumatra. This can be seen in Fig. 6.1 taken from [46]. The Figs. 6.2, 6.3 and 6.4 show the resulting tsunami after 5, 60 and 120 minutes respectively.
Fig. 6.5 shows the route of a satelite from [47] that measured the waterheight above sealevel in the indian ocean during the tsunami. This data was taken from [48] and can be compared with the simulation. The results of this are shown in Fig. 6.6 for the Haversine-method and in Fig. 6.7 for the PROJ.4 method. The blue dots show the measured data from the satelite and the orange line is the data from the simulation at the same place and time. While the Haversine-method produces a shift in altitude and timing, the PROJ.4-method matches the satelite data almost perfectly, considering that the simulation that was run has some inaccuracies itself. The resolution of the tsunami simulation is 7812.5 meters, which is wider than the real wave was. This results in a lower amplitude and a different timing due to a different acceleration.
These results show, that this tool can be used to replace the current conversion process.



Shearer P, Bürgmann R. 2010.
Annu. Rev. Earth Planet. Sci. 38:103–31

Figure 6.1: The green area is where the 2004 Sumatra earthquake appeared.[46]

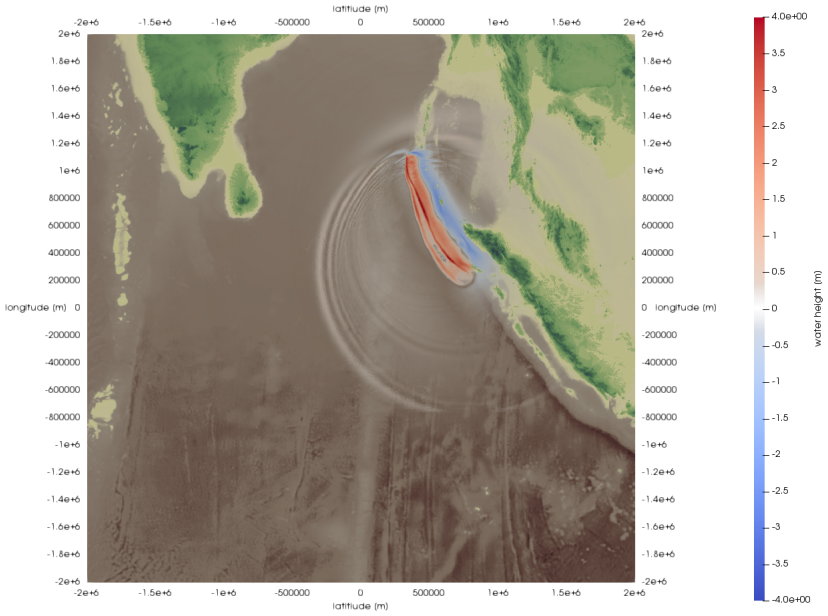


Figure 6.2: The simulated tsunami after 5 minutes

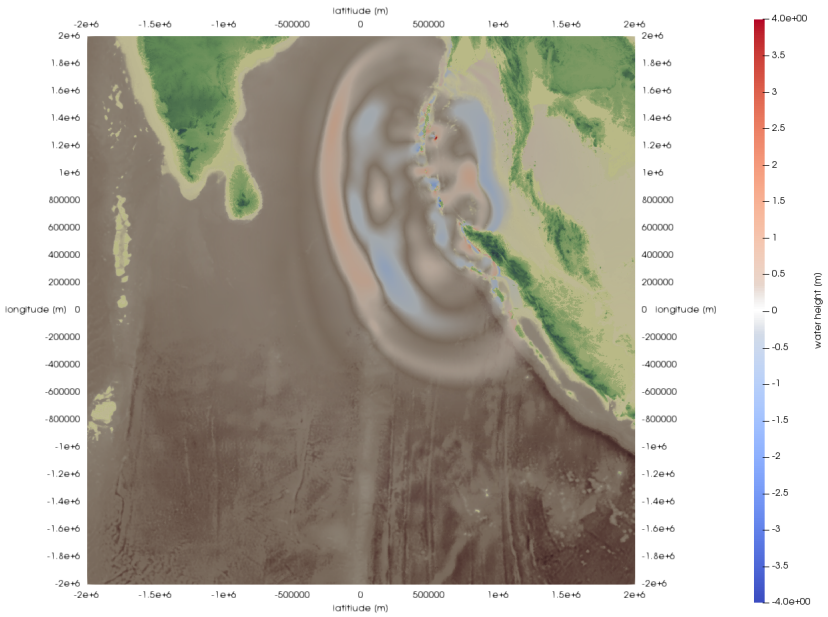


Figure 6.3: The simulated tsunami after 60 minutes

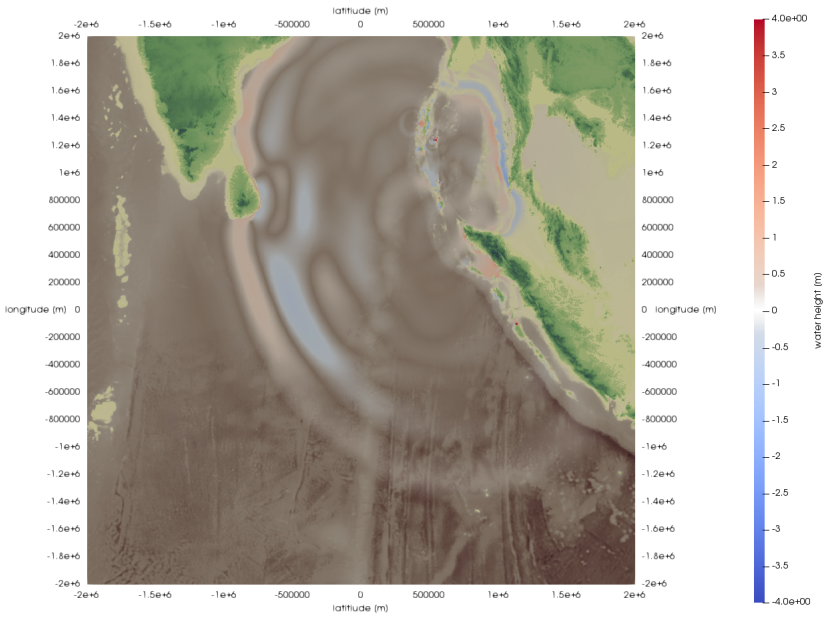


Figure 6.4: The simulated tsunami after 120 minutes

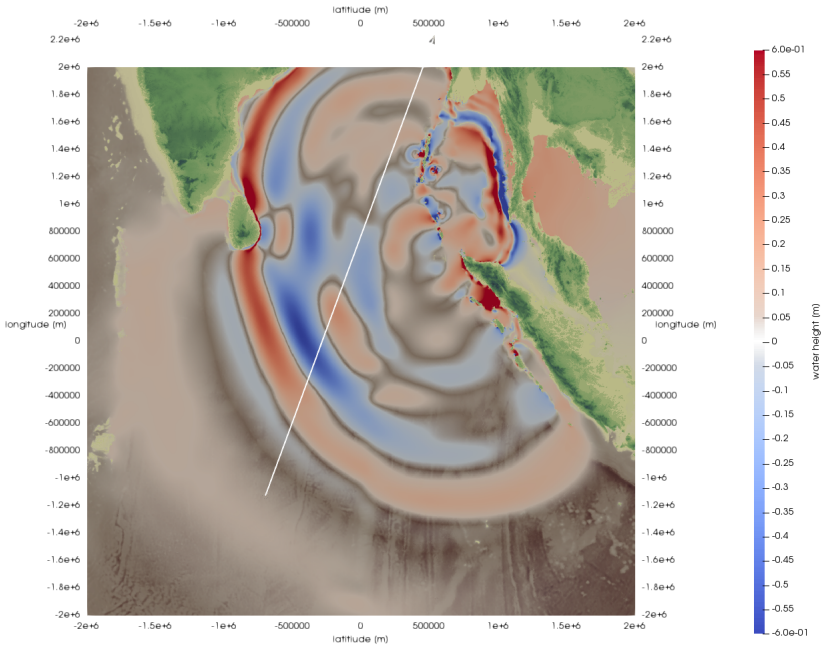


Figure 6.5: Route of the satelite with the tsunamis state after 115 minutes

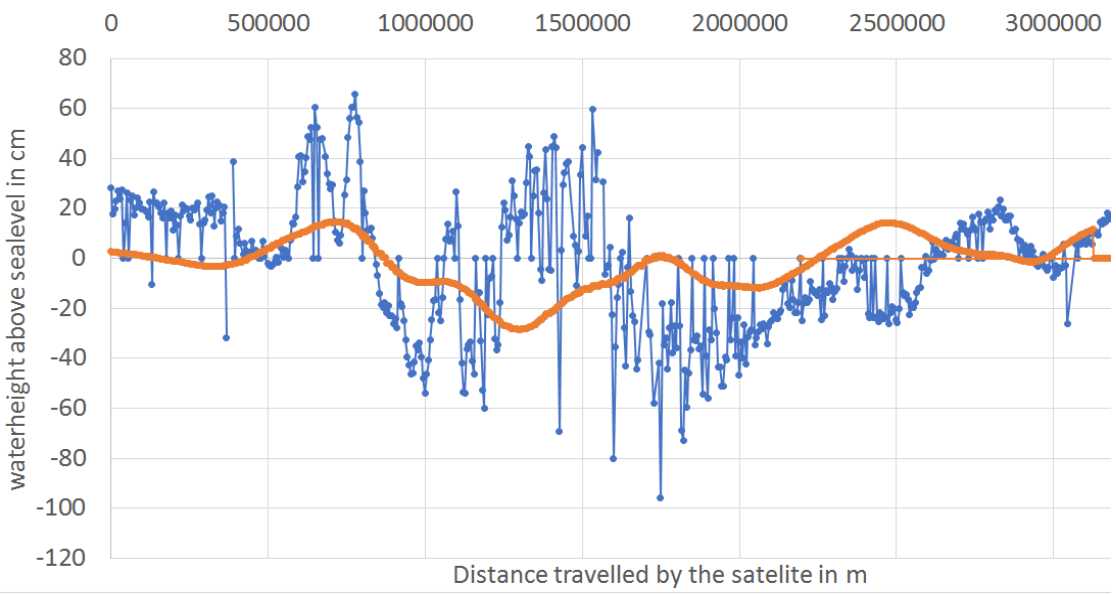


Figure 6.6: Comparison of satelite data to the simulation with bathymetry converted via Haversine method.

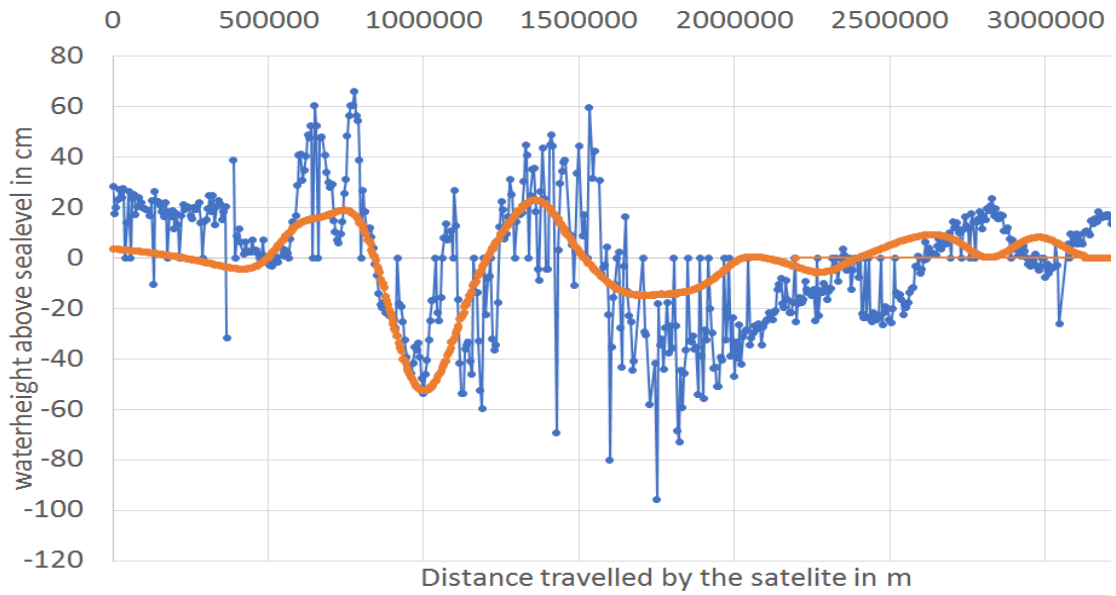


Figure 6.7: Comparison of satelite data to the simulation with bathymetry converted via PROJ.4 method.

7 Possible improvements

While the tool fulfills all its requirements and has a fairly fast runtime, it is still far from optimal and can still be improved in various areas. This chapter presents the six biggest areas where improvement is possible and gives suggestions on how they might be realised aswell as showing problems that might occur.

7.1 Parallelization

Currently there is not a single point in this program, where parallelization is used. But there is actually a lot of potential for runtime improvement, as almost every part of the program can be split in multiple independent tasks.

7.1.1 Reading

The *fast* mode is always working on arrays of type double, but it is not guaranteed that the input is also of type double. Because of that the reading process is already slowed down to reading value by value and then converting every single of those values manually to type double. Every value has its own designated place in the array, so no writing conflicts can occur in this area and multiple threads reading from the same file, but not writing back, also poses no problem. Therefore the reading process can be significantly accelerated by assigning equally sized blocks of the file to different threads.

7.1.2 Writing

Writing in *fast* mode has the same problem as reading. Every value has to be manually converted back to the correct data type and is written on its own and the whole process is therefore quite slow, but is already suited for parallelization. Accessing the values is no problem, as they are only read from an array, but writing might prove more difficult, as multiple threads would need to write on the same file simultaneously. This problem does not occur, when the *-split* flag is set and the output is written to multiple files. In this case assigning every file to its own thread can easily speed up the writing process.

7.1.3 Converting

However the biggest chance for runtime improvement lies in the conversion itself. While the *slow* mode faces the same problem as the writing process in having to write to a single file with multiple processes at once, if that is appropriately solved, the same parallelizatoin as for *fast* mode is possible. Currently the program calculates values for every rectangle sequentially one after another. In regard to race condition there would be no problem in giving every rectangle its own thread. This is because the rectangles are independant of each other and are not changing the input values. The only exception is the *–reverse* flag for XDMF, as it causes the program to iterate over the triangles instead, which results in multiple triangles reading and writing to the same rectangle. In this area synchronisation has to be implemented.

7.2 Optimization

Apart from parallelization there are a few parts of the program that have great potential for performance optimization.

7.2.1 XDMF: Reversed conversion

With the boundary rectangle (see 4.3.3 and 4.3.5 for an explanation), the reversed conversion iterating over the triangles already has optimization built in, but within this boundary can still be a lot of rectangles that do not actually overlapp with the triangle. Fig. 7.1 visualizes the problem. This is especially the case, if the triangle only barely reaches one more index in one direction by a small amount. The whole row of that index will then be added, but with one exception no rectangle is actually overlapping. The correct rectangles can instead be found, by traversing the edges of the triangle from the northernmost corner to the other two and for every new row that is reached, every rectangle between the left edge and the right edge is checked. If one edges end is reached, it is replaced by the remaining third edge. This way only rectangles that are crossed by an edge or inside the triangle are checked and no unnecessery calculations are done. This idea is visualized in Fig. 7.2.

7.2.2 XDMF: Standard conversion

The standard conversion traverses the rectangles and looses a lot of time by checking for overlaps with every single triangle. This is because they are currently not ordered in any way and therefore no precalculations can be done to determine if there is a chance of an overlap. To improve that, the triangles have to either be ordered in a way

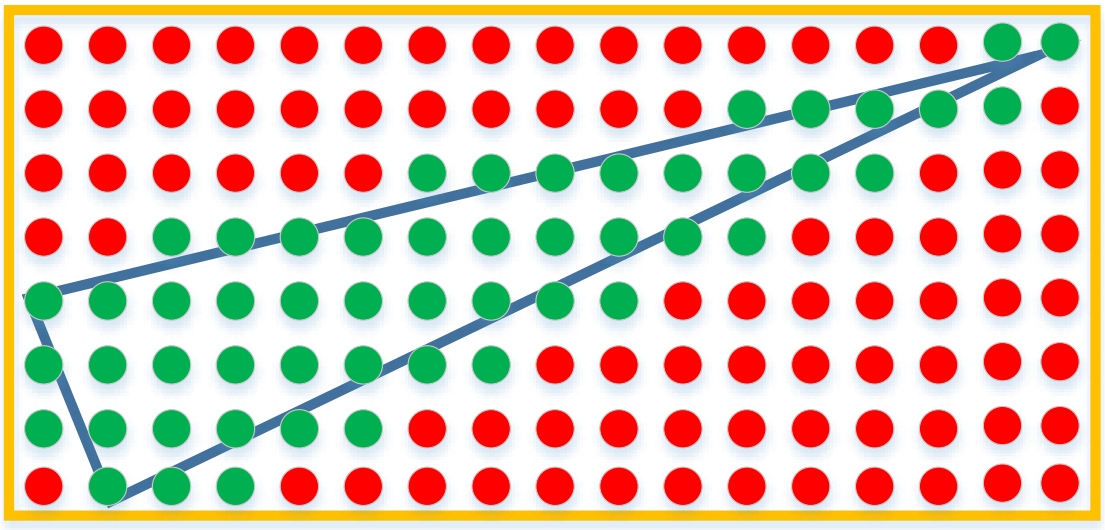


Figure 7.1: Visualization of the boundary rectangle problem.
Green dots represent rectangles that are checked correctly. Red dots represent rectangles that are checked despite not overlapping with the triangle.

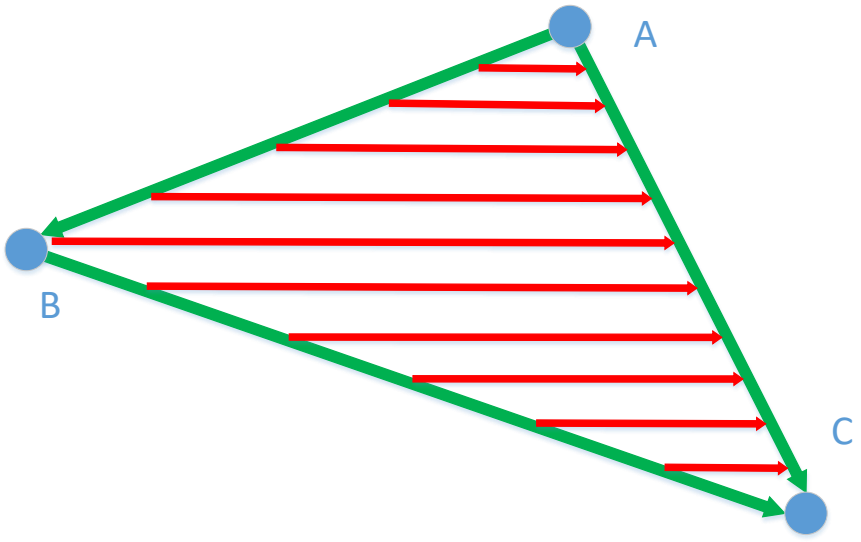


Figure 7.2: Visualization of the traversion algorithm.
The green arrows represent the triangle edges, that get traversed. Each red arrow represents the traversal of all rectangles in that row, who are between the triangles edges. No non-overlapping rectangles get checked.

their resulting index in the array represents their location in the mesh, or a sorting structure over the triangles has to be established. For example the latter one could be a tree that stores information about triangle location. Each node can represent a part of the conversion area and reference subtrees which contain smaller areas or store the triangles in that area. This way for every rectangle the tree can first be used to limit the number of triangles, that have to manually be checked for an overlap. This way, the major drawback of this method in comparison to the reversed version can be resolved.

7.3 Error handling

Although the program has been tested extensively with correct input and well formated files aswell as flags set in accordance with 4.2, there currently exists almost no error handling in the program so its behaviour is undefinded when used incorrectly. This includes things like the setting of nonexisting flags causing a crash or the program trying to write output although the conversion has been aborted due to no data being available in the requested area or the cutoff radius being to small for the given input grid. This information needs to be passed on to different parts of the program, so they can react accordingly. Also some errors like a *std::bad_alloc error* due to not enough RAM being available are currently not caught so the program crashes as a result. These errors need to be caught so the program can safely be stopped and more helpful error messages like how much RAM was requested in total, how much was requested already and what request caused the error can be displayed and the user can act accordingly. Another problem is the handling of missing or ill formated input files. In the first case, the program crashes with an error and the second case is undefinded. Here a mechanism needs to be implemented that ensure, that the first cause is caught and a helpful error message is produced before the program is stopped and that the second case is detected, before also relaying the problem to the user and stopping the program.

7.4 Extension: XDMF in slow mode

The *fast* mode increses performance significantly compared to the *slow* mode but comes with the drawback of needing a lot of RAM, so it can only be run on bigger machines. For GEBCO and ASAGI input files the user has the option choose between the two modes depending on what machine the program is run on. For XDMF input files this option currently does not exist, as its conversion is only available in *fast* mode. To be able to convert XDMF on smaller machines or even local PCs too, the *slow* mode has to be adapted to also contain the conversion from triangles to rectangles. The main change for this would be the introduction of new functions in *Input.cpp* to handle

reading the context information only, *Output.cpp* to set missing information that is not available from the input and *Converts.cpp* for reading, converting and writing the data. There also need to be small changes or additions to *Helpers.cpp* and *Main.cpp* to slightly restructure the general process of the program to adapt the new option.

7.5 Extension: PROJ.4 tool for GEBCO conversion

For the latitude of 0 and longitude of 88.5113 an alternative for calculating the new points coordinates in the GEBCO conversion exists. It uses the PROJ.4 tool from [41] and changes the conversion result in a noticeable way. This is shown in 5.2 and chapter 6. As this option was added very late in the development process, its integration into the program is quite poor from a coding point of view. If this is improved, it also opens up the option of applying this method to other input scenarios. This is rather important, as the PROJ.4 tool achieves more accurate results than the Haversine method.

7.6 Accuracy of Haversine-method

The Haversine-method for GEBCO assumes the earth to be a perfect sphere, but this is not the case. As a result there is a small error in each calculation of latitude and longitude coordinates. In the current approach presented in 4.3.3 these errors accumulate so the result gets more inaccurate, the further one is away from the bottom left corner. This can be improved, by calculating every point in the new grid inpedendently from each other using only the distance from the gridcenter, as it is already implemented for the PROJ.4 approach.

8 Conclusion

In this thesis a tool for postprocessing of earthquake data from SeisSol was developed. This tool is necessery for efficient coupling of the eartquake-simulation SeisSol and the tsunami-simulation sam(oa)^2, as data is currently only transfered by hand in a long complicated process. Those two and all other relevant used tools are explained in chapter 2 and 3. After running this tool, the data is sucessfully converted from SeisSols triangular mesh to a perfectly rectangular grid, that can be read by ASAGI and therefore sam(oa)^2 as described in chapter 4. The second goal was to provide bathymetry data by converting GEBCO-style files to the same ASAGI format. As explained in 4.3.1, this part of the program can also be run on small local machines. Multiple options for the conversion process were tested regarding runtime and precision to ensure it is good enough to replace the current process. In this evaluation in chapter 5 the *fast* mode proved to be superior compared to the *slow* mode regarding runtime for the GEBCO cloud-conversion method, because here the same value needs to be read more than once. Both modes have about the same speed for ASAGI and the standard GEBCO conversion. For XDMF the *reversed* approach is faster in multiple orders of magnitude compared to the standard version. This is because rectangles are ordered while triangles are not and therefore a boundary rectangle can be constructed to limit the number of rectangle-triangle combinations that have to be checked. This gets explained in 4.3.3 and 4.3.5. In regard to precision the self-implemented adaption of the Sutherland-Hodgman algorithm from 4.3.5 is more accurate than the usage of the boost library. It is also faster, since it can exploit certain circumstances of the scenario. To check for potential differences in the resulting tsunami simulation, the tool was also applied to real data in chapter 6. While there was a slight shift in altitude and timing for the bathymetry data converted via the Haversine-method, the simulated data with bathymetry converted with PROJ.4 alligns almost perfectly with the real world data. While the program can still be improved in the areas of runtime and functionality as suggested in chapter 7, the results certainly are promising and it can be used to replace the current process.

List of Figures

Bibliography

[1] J. Pickrell. *Facts and Figures: Asian Tsunami Disaster*. URL: https://www.newscientist.com/article/dn9931-facts-and-figures-asian-tsunami-disaster/.

[2] A. Back. *Tsunamis: how they form*. URL: http://www.australiangeographic.com.au/topics/science-environment/2011/03/tsunamis-how-they-form.

[3] A3M-Mobile-Personal-Protection-GmbH. *The formation of a Tsunami*. URL: http://www.tsunami-alarm-system.com/en/phenomenon-tsunami/phenomenon-tsunami-formation.html.

[4] S. Rettenberger, C. Uphoff, A.-A. Gabriel, B. Madden, S. Wollherr, and T. Ulrich. *SeisSol Homepage*. URL: http://www.seissol.org/.

[5] O. Meister. *Samoa - SFCs and Adaptive Meshes for Oceanic And Other Applications*. URL: https://github.com/meistero/Samoa.

[6] S. Rettenberger. *a pArallel Server for Adaptive GeoInformation*. URL: https://github.com/TUM-I5/ASAGI.

[7] M. Schreiber. *Tsunami - Scripts related to Tsunami simulations*. URL: https://github.com/TUM-I5/tsunami.

[8] *Generic Mapping Tools*. URL: http://gmt.soest.hawaii.edu/projects/gmt.

[9] B. P. Johnston, J. M. Sullivan, and A. Kwasnik. "Automatic conversion of triangular finite element meshes to quadrilateral elements." In: *International Journal for Numerical Methods in Engineering* 31.1 (1991), pp. 67–84. ISSN: 1097-0207. DOI: 10.1002/nme.1620310105.

[10] T. Itoh and K. Shimada. *Finite element modeling method and computer system for converting a triangular mesh surface to a quadrilateral mesh surface*. US Patent 5,774,124. 1998.

[11] HDF-Group. *HDF5 Support Page*. URL: https://support.hdfgroup.org/HDF5/.

[12] HDF-Group. *HDF5 Technologies*. URL: https://support.hdfgroup.org/about/hdf_technologies.html.

[13] HDF-Group. *What is HDF5?* URL: https://support.hdfgroup.org/HDF5/whatishdf5.html.

[14] HDF-Group. *High Level Introduction to HDF5*. URL: https://support.hdfgroup.org/HDF5/Tutor/HDF5Intro.pdf.

[15] HDF-Group. *HDF5 File Format Specification*. URL: https://support.hdfgroup.org/HDF5/doc/H5.format.html.

[16] Unidata. *Network Common Data Form*. URL: https://www.unidata.ucar.edu/software/netcdf/.

[17] Unidata. *NetCDF FAQ*. URL: http://www.unidata.ucar.edu/software/netcdf/docs/faq.html.

[18] Unidata. *NetCDF Introduction*. URL: http://www.unidata.ucar.edu/software/netcdf/docs/netcdf_introduction.html.

[19] Unidata. *NetCDF File-structure and performance*. URL: http://www.unidata.ucar.edu/software/netcdf/docs/file_structure_and_performance.html.

[20] *XDMF Main Page*. URL: http://www.xdmf.org/index.php/Main_Page.

[21] *XDMF Reading Example*. URL: http://www.xdmf.org/index.php/Read_Xdmf.

[22] *XDMF Model and Format*. URL: http://www.xdmf.org/index.php/XDMF_Model_and_Format.

[23] A. Breuer, A. Heinecke, S. Rettenberger, M. Bader, A.-A. Gabriel, and C. Pelties. "Sustained Petascale Performance of Seismic Simulations with SeisSol on SuperMUC." In: *Supercomputing - 29th International Conference, ISC 2014*. Ed. by J. Kunkel, T. T. Ludwig, and H. Meuer. Vol. 8488. Lecture Notes in Computer Science. PRACE ISC Award 2014. Heidelberg: Springer, June 2014, pp. 1–18.

[24] S. Wenk, C. Pelties, H. Igel, and M. Kaser. "Regional wave propagation using the discontinuous Galerkin method." In: *Solid Earth* 4.Issue 1 (2013), pp. 43–57.

[25] M. Kaese, M. Dumbser, J. de la Puente, V. Hermann, and C. Castro. "Overview of the High-Order ADER-DG Method for Numerical Seismology." In: *CIG/SPICE/IRIS/USAF WORKSHOP*. JACKSON, NH, 2007.

[26] A. Breuer, A. Heinecke, and M. Bader. "Petascale Local Time Stepping for the ADER-DG Finite Element Method." In: *2016 IEEE International Parallel and Distributed Processing Symposium*. June 2016, pp. 854–863.

[27] A. Heinecke, A. Breuer, S. Rettenberger, M. Bader, A.-A. Gabriel, C. Pelties, A. Bode, W. Barth, X.-K. Liao, K. Vaidyanathan, M. Smelyanskiy, and P. Dubey. "Petascale High Order Dynamic Rupture Earthquake Simulations on Heterogeneous Supercomputers." In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis SC14*. Gordon Bell Finalist. IEEE. New Orleans, LA, USA: IEEE, Nov. 2014, pp. 3–14. ISBN: 9781479954995.

[28] S. Rettenberger and M. Bader. "Optimizing Large Scale I/O for Petascale Seismic Simulations on Unstructured Meshes." In: *2015 IEEE International Conference on Cluster Computing (CLUSTER)*. Chicago, IL: IEEE Xplore, Sept. 2015, pp. 314–317.

[29] O. Meister. "Sierpinski Curves for Parallel Adaptive Mesh Refinement in Finite Element and Finite Volume Methods." Dissertation. München: Institut für Informatik, Technische Universität München, Dec. 2016.

[30] O. Meister and M. Bader. "2D adaptivity for 3D problems: Parallel SPE10 reservoir simulation on dynamically adaptive prism grids." In: *Journal of Computational Science* 9 (May 2015). Special Issue ICCS 2015, pp. 101–106. ISSN: 1877-7503.

[31] P. Samfass. "A Non-Hydrostatic Shallow Water Model on Triangular Meshes in sam(oa)^2." Studienarbeit/SEP/IDP. Institut für Informatik, Technische Universität München, Apr. 2015.

[32] R. Schaller. "Parallelization of a Non-Hydrostatic Shallow Water Model in sam(oa)^2." Studienarbeit/SEP/IDP. Institut für Informatik, Technische Universität München, Oct. 2015.

[33] C. Uphoff, S. Rettenberger, T. Ulrich, A. Heinecke, and S. Wollherr. *SeisSol wiki for ASAGI*. URL: `https://github.com/SeisSol/SeisSol/wiki/ASAGI`.

[34] S. Rettenberger. *HOWTO presentation for SeisSol+ASAGI*. URL: `http://www.seissol.org/sites/default/files/asagi.pdf`.

[35] S. Rettenberger, O. Meister, M. Bader, and A.-A. Gabriel. "ASAGI - A Parallel Server for Adaptive Geoinformation." In: *Proceedings of the Exascale Applications and Software Conference 2016 (EASC '16)*. ACM, Sept. 2016, 2:1–2:9.

[36] *Usage instructions for ASAGI*. URL: `http://tum-i5.github.io/ASAGI/usage.xhtml`.

[37] British-Oceanographic-Data-Centre. *General Bathymetric Chart of the Oceans*. URL: `http://www.gebco.net/`.

[38] *Area of a Triangle by formula*. URL: `http://www.mathopenref.com/coordtrianglearea.html`.

[39] *Haversine formula*. URL: `https://rosettacode.org/wiki/Haversine_formula`.

[40] C. Veness. *Calculations on the earths sphere*. URL: `http://www.movable-type.co.uk/scripts/latlong.html`.

[41] F. Warmerdam. *Cartographic Projections Library*. URL: `https://github.com/OSGeo/proj.4`.

[42] B. Molkenthin. *2D Clipping with the Sutherland-Hodgman-Algorithm*. URL: http://www.sunshine2k.de/coding/java/SutherlandHodgman/SutherlandHodgman.html.

[43] *Sutherland-Hodgman algorithm*. URL: https://en.wikipedia.org/wiki/Sutherland-Hodgman_algorithm.

[44] R. Gaul. *Understanding Sutherland-Hodgman Clipping for Physics Engines*. URL: https://gamedevelopment.tutsplus.com/tutorials/understanding-sutherland-hodgman-clipping-for-physics-engines--gamedev-11917.

[45] Leibniz-Supercomputing-Centre. *Munich Centre of Advanced Computing - MAC Cluster*. URL: http://www.mac.tum.de/wiki/index.php/MAC_Cluster.

[46] P. Shearer and R. Buergmann. "Lessons Learned from the 2004 Sumatra-Andaman Megathrust Rupture." In: *Annual Review of Earth and Planetary Sciences* 38.1 (2010), pp. 103–131. DOI: 10.1146/annurev-earth-040809-152537. eprint: https://doi.org/10.1146/annurev-earth-040809-152537.

[47] Physical-Oceanography-Distributed-Active-Archive-Center. *Jason-1 GDR SSHA version E NetCDF Geodetic*. URL: https://podaac.jpl.nasa.gov/dataset/JASON-1_L2_OST_GPR_E_GEODETIC?ids=&values=&search=Jason-1%20ssha.

[48] OPeNDAP. *OPeNDAP Server Dataset Access Form*. URL: https://podaac-opendap.jpl.nasa.gov/opendap/allData/jason1/L2/gdr_netcdf_e/c109/JA1_GPN_2PeP109_129_20041226_022717_20041226_032328.nc.html.

*every website was last visited on 28.08.2017