MASTER THESIS

IOANNIS ANGELIDIS, DAVID NAGY AND EMIL MØLLER RASMUSSEN

Floodtool 2.0: An online open source application for flood modelling

AALBORG UNIVERSITY
STUDENT REPORT

Title: FloodTool 2.0
Subtitle: An online open source application for flood modelling
Theme: Master thesis
Project period: February, 2015 – June, 2015
Students:

Ioannis Angelidis (20132531)

David Nagy (20140706)

Emil M. Rasmussen (20130954)

Supervisor: Thomas Balstrøm, Professor, Aalborg University Copenhagen

Synopsis:
Geographical Information Systems have become widespread during the past decades. This evolution has also encouraged the development of open source GIS software suites. The goal of this project is to demonstrate a way to combine various open source technologies in order to create an online application that offers the user the possibility to model a flood caused by sea water stowage.

We will begin by presenting the theory that is the foundation of this project. Then we will demonstrate in detail the way we have chosen to combine open source GIS with various programming languages, in order to create an online application. We will then present the outcome of that effort from a developer's point of view and what we think they will experience while using our application. We finish this report by discussing the choices we have made in order to achieve the goals we have set beforehand, but also what we think are the strengths and the weaknesses of such endeavor.

Number of copies: 2
Number of pages: 83
Completed: June 10, 2015

ACKNOWLEDGEMENTS

A big thanks goes out to Thomas Balstrøm for his supervision during the creation of this project.

Thank you to the Open Source community, who made this project possible.

PREFACE

This project is created by Ioannis Angelidis, David Nagy and Emil Møller Rasmussen as part of the Geoinformatics programme Master's thesis. The work presented here was inspired by a former project completed during our 2nd semester. This work can be found in the project library of Aalborg University of Copenhagen under the title :

- Flood Mitigation: Flood modeling of sea water stowing - the development of a tool to facilitate disaster management.

Throughout this report, an extensive codebase is created which can be accessed at

- https://github.com/GISEmil/Floodject.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1

INTRODUCTION & PROBLEM STATEMENT

1.1 INTRODUCTION

Geographical Information Systems (GIS) are designed to manage and analyze geographic data. GIS used to be a niche product, but over the last couple of decades they have become more widespread, and a plethora of products have become available. This evolution has also culminated in the development of open sourced GIS software. Because of their open source nature they have evolved to such an extent, that Open Source GIS software now can provide functionalities comparable to what is available from the established commercial products. Using open source technologies as the back-end of a project, makes the cost of acquisition very low, enabling a whole new group of users access to these tools, and analysis methods.

Whilst a lot of GIS functionality exists as libraries to Python, and as simple stand-alone applications, some all-inclusive Open Source GIS applications exist, amongst these one of the most well-known is GRASS. One sector that historically has used GIS, is Hydrology. Performing geographical analysis on the movements, spread and aggregation of water in a landscape, is crucial to understanding the particular phenomena. One of these phenomena is flooding. Flooding can be caused by different events, such as sea water stowage. Understanding hydrological movement, in the event of a flood caused by sea water stowage, is complicated, and often requires advanced modeling. Deciding on what kind of inputs and outputs that can be provided by the user, will simplify the process extensively. Furthermore, relieving the weight of creating the process from the user, makes the task less complicated.

Removing the burden of complexity from the user, could enable a new audience with the capability of performing advanced GIS analysis, such as flood modeling. This can be achieved by creating a thin client based web application using open source technologies.

1.2 BACKGROUND

During the 2nd semester of our Master's studies, we developed a tool in ArcGIS using Model Builder, with the capability of modelling a flood caused by sea water stowage. The tool was based on a simple hydrological approach, focusing more on the geoinformatical aspect

of the problem, since this is our area of expertise. Our experiences with this process lead us to wonder how this could be achieved by using other tools, and technologies. When dealing with the distribution process we stumbled on the problem that the tool could not be used outside of an ArcGIS environment. We came to the conclusion that Open Source software offers the possibility of overcoming this constraint. Furthermore, using such tools would provide valuable experience in different applications of GIS. Basing the application on the same methodology as previously, we would be able to use the experiences already gained, as the foundation of a development for a new approach.

1.3 PROBLEM STATEMENT

Combining a variety of open source technologies, it should be possible to create an application that will enable a user, not proficient with GIS, of performing flood modeling. Through this development process we will document the necessary steps required in order to establish a working proof-of-concept.

As such, our problem statement will be as follows:

- How can we create an online application capable of performing simple flood modeling using open source technologies.
  - How can we combine predetermined tools to create such a model?
  - How can we make the application as easy to use as possible?
  - How can we best document our progress in order to facilitate development of applications using the same technologies.

2 THEORY

2.1 GRASS

GRASS (Geographical Resources Analysis Support System) is an open source GIS desktop application capable of handling spatial data in both vector and raster formats. GRASS adopted a GNU GPL [1] in 1999, which allowed users and developers to have free access to the GRASS source code, resulting in a library of more than 350 freely available modules capable of management, processing, analysis and visualization of geospatial data (Neteler et al., 2012).
Because Open Source GIS provides full access to its internal structure and algorithms, unlike proprietary GIS software, users can learn from existing modules and create their own GIS modules based on the preexisting ones. GRASS libraries can also be accessed through the built-in API (Application Programming Interface). This enables a more efficient integration of new functionalities into the GRASS environment Most GIS applications can be written in Python, making it possible to automate work-flows. As mentioned earlier, GRASS contains around 350 modules, which can be accessed using GRASS' graphic interface. The three main module-groups are based on vector, raster, and imagery analysis (Neteler and Mitasova, 2005).

A grass project is located in GRASS' designated database folder *grassdata* (also known as GISBASE). This is the directory where processed or imported data is stored and, unless otherwise designated, where most of the processing will occur. A project created in GRASS has to have both a LOCATION and a MAPSET. The most important of these is the LOCATION. This is where the critical information about the project, such as the projection of the data, is stored. GRASS does not have reprojecting on-the-fly functionality as ArcGIS or QGIS have. Using the LOCATION folder properly is therefore necessary. The MAPSET is a way of separating different projects, or phases of processes, and a LOCATION can contain several MAPSETS (Neteler and Mitasova, 2005).

GRASS can be operated by a variety of ways. The most commonly used method is by accessing the modules through the GRASS GUI (Graphical User Interface), but it can also be achieved purely through scripting - such as with Python. GRASS is able to handle most of the vector and raster formats which are supported by GDAL (Geospatial Data Abstraction Library), such as GeoTIFF, ArcGRID, ERDAS, USGS

1 General Public License, see http://www.gnu.org

SDTS DEM, etc (Neteler et al., 2012).

The way GRASS handles region and resolution settings differs from most other GIS software. Since different datasets can have different extents, it is possible to set the current processing region allowing the user to run a specific process on a subset of a raster or the location and not necessarily run a process on the entire image. The lack of on-the-fly reprojection makes GRASS less user-friendly than other similar products. Furthermore it does not allow drag and drop import, and most functions must be invoked using the built-in GRASS commandline-like utilities (Neteler and Mitasova, 2005).

2.1.1 *Hydrology in GIS*

The accurate depiction of hydrological movements and their responses to the land cover has been the objective of hydrological scientists for many decades. As advances in IT have progressed, the calculations and algorithms possible have become faster, more sophisticated and accurate (Khatami and Khazaei, 2014). To give an example of the application of hydrology in GRASS, this section will include a description of some of the main hydrology tools from GRASS' libraries. The tools shown will all be based on modelling on DEM's (Digital Elevation Models).

2.1.1.1 *Flow direction*

Flow direction is a core hydrology tool. Flow direction makes it possible to determine which direction water will flow in, when moving through a DEM. The computational algorithm can be created in a wide variety of ways, for instance GRASS' r.terraflow module has two options (Toma et al., 2001a):

- Multiple Flow Direction (MFD)

or

- Single Flow Direction (SFD)

Both of these algorithms are based on a neighborhood involving the eight cells surrounding a specific cell on a raster. The basis of the flow direction is that the water flows to a cell with a lower value than the current one, but it is in this regard that MFD and SFD differ from each other. As can be seen from Figure 1, the SFD method assigns a single flow direction to the lowest downslope neighboring cell, whilst MFD assigns flow direction to all downslope neighboring cells (Neteler and Mitasova, 2005).
Both methods have the criteria, that the flow direction cannot contribute to cells with the same height as the central cell or cells which

Figure 1: Example of two different flow direction methods (Toma et al., 2001a).

have no downslope neighbors (Toma et al., 2001b). Pits or so called depressions or sinks, are areas which are surrounded by higher elevation values. It is also an internal drainage area, although some of the time it is a real natural feature such as a karst, but usually it is an imperfection of the digital elevation model. The GRASS Terraflow module fills the sinks and then assigns a flow direction on the filled terrain, thereby the flow direction will not be altered by a DEM full of depressions (Toma et al., 2001a).

2.1.1.2 *Flow accumulation*

Another important hydrology tool is Flow Accumulation. This tool is capable of calculating a flow running through the terrain, such as the accumulation of water (Toma et al., 2001a). As input, the module needs a raster indicating flow directions.
It should be noted that flow accumulation is highly dependent on the previous described phenomenas and computational methods. For example MFD would provide a significantly different flow accumulation than SFD, as MFD provides more accumulation possibilities. Also, a depressionless DEM will have a different accumulation than one with depressions (Toma et al., 2001a). Analyzing the direction of the flow can give a limited insight about cell behaviors, however flow accumulation allows the investigation of main stream lines, and how they contribute to the stream system, as well as providing an output of stream lines (Neteler and Mitasova, 2005).

2.1.1.3 *Watershed*

A Watershed is an area where all streams end up in a common outlet. This can also be called a basin or a catchment area, however in this paper it will be referred to as a watershed.

A watershed is a relative term, as it depends on the scale at which you look at them - practically all streams have their own watershed. Therefore to determine the watersheds of a DEM, it is necessary to know a variety of factors, such as flow accumulation, flow direction

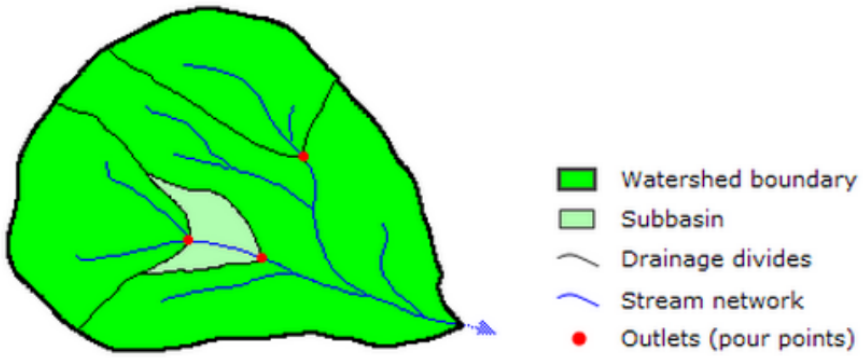


Figure 2: Figure showing important concepts when working with watersheds (ESRI, b).

or slope aspect of the terrain (Echlschlaeger, 2015).

In GRASS there is a watershed algorithm, which considers every aspect of the previously mentioned procedure. The watershed (r.watershed) module has watershed output which is created by the output of the flow direction and the derived flow accumulation output of the DEM. As mentioned earlier, watershed sizes are relative to the scale of the investigation. A threshold value can be set to delineate the minimum size of the resultant watershed, based on the scale of interest (Neteler and Mitasova, 2005) (Echlschlaeger, 2015). Setting this value should be based on the horizontal resolution of the raster being worked on. For instance, setting the value to 1000, on a raster with 10mx10m resolution, would define, that the smallest watershed area has to be larger than 1000*10m*10m = 0,1km2.

2.1.1.4 *Cost Distance*

The Cost Distance analysis is used to calculate the route that will have the least cost when traveling across a given surface (Neteler and Mitasova, 2005). This tool uses a cost surface to determine the weighted shortest path to the nearest source cells or source vector features, and as such does not calculate distance in geographical units, but shows the distance by cost surface units (Awaida and Westervelt, 2013). The cost can include several criteria, factors and weights depending a the specific analysis. For example, the cost surface of a hiking route can be established by taking into account the steepness, the type of the area or even the recommendations of which area should be visited.
Each of the chosen factors should be reclassified in order to put them on a common scale, so that they can be compared in spite of their different nature. The cost distance module of GRASS (r.cost) is based on the previously explained method (Awaida and Westervelt, 2013). In order to get the cost distance output raster, the user needs to provide a cost raster.

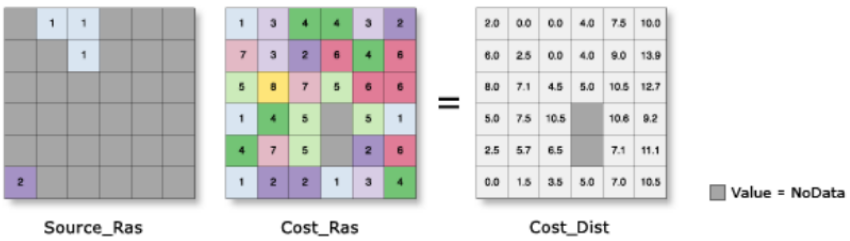


Figure 3: Figure showing how cost distance works (ESRI, a).

2.2 PYTHON

Python is a language that (Lutz, 2013):

- emphasizes code readability.
- is compatible with all major operating systems, and usually comes pre-packaged with these.
- Is completely open source.
- is highly extensible.

Being extensible, means that it does not come prepackaged with all functionality built in, but expects the user to download / add necessary libraries as they are needed. The most commonly used versions of Python are 2.7 and 3.x, which differ from each other. The 2.7 release is a so-called legacy release, which means that it is not the worked-upon version, and will not see major updates. The 3.x is the newest version, and will be updated regularly. When working with older software, and libraries, it is most likely best to use version 2.7 as there could be compatibility issues when using a newer version of Python (Python Software Foundation, 2015).

Standard python syntax looks something like this:

```
import random
randomarray = [1,3,5,7,8]
for element in randomarray:
        print(element)
```

2.3 GRASS

From within GRASS, Python can easily be used to call native functionality. These functions can also be accessed and manipulated by using a variety of scripting languages, without explicitly starting the GRASS software. One of the languages capable of doing this is Python.

(Neteler and Mitasova, 2005)

The functions and modules of GRASS, when used outside of an actual GRASS session, only work when a series of specific environment variables have been set. These are (GRASS Development Team, 2015):

- GISBASE - needs to be set to the top-level directory of the GRASS installation.
- GISRC - needs to contain the absolute path to a file containing settings for GISDBASE, LOCATION and MAPSET
- PATH - needs to include $GISBASE/bin and $GISBASE/scripts.

An example of the syntax of GRASS functions being accessed externally through Python (GRASS Development Team, 2015):

```
import grass.script as gscript
import grass.script.setup as gsetup
gscript.run_command(r.in.gdal,flags=,input=input.tif,output=
    output
```

2.4 WEB TECHNOLOGIES

2.4.1 *HTML*

In 1990, Tim Berners-Lee, a physicist employed at CERN, invented HTML by trying to organize the research documents available to scientists and facilitate their access to them. He is also considered by many the inventor of the Internet by having set the foundations of the web as we know it today (HTML.net, 2015). HTML stands for HyperText Markup Language. To be more specific (HTML.net, 2015),

- Hyper means that HTML does not perform commands in one line and then proceeds to the next one as Python or C++ does.
- Text is self-explanatory
- Markup means that the author can put tags in the text. This means that the text is structured in different sections such as the header, the body etc.
- Language of course means that HTML is a programming language.

The following figure displays an example of a basic HTML code snippet. It will help in understanding the structure and content of an HTML document.

```
<!DOCTYPE html>
<html>
<body>

<h1>My first heading!</h1>
<p>My first paragraph!</p>

</body>
</html>
```

Starting at the top, the DOCTYPE declaration, states what kind of document we are creating. In this case, we have an HTML document (W3Schools, 2015). In the html tags (<html>, </html>), we describe the page we are creating. The body tags (<body>, </body>) contains all the information visible to the user. Finally, the tags <h1> and <p>, set the heading and paragraphs respectively (W3Schools, 2015).

2.4.2 *JavaScript*

JavaScript is a dynamic scripting language developed by Netscape. Brendan Eich, the original developer of JavaScript, created it so that it would be able to support prototype based object construction and that it can work both as an object oriented and procedural language. Its syntax was developed to be similar to C++ and Java in order to minimize its learning curve (Network, 2015). Below, we can see a small example of JavaScript used in an HTML document.

```
<!DOCTYPE html>
<html>
<body>

<h1>My first JavaScript!        </h1>

<script>
document.write("<p>My First Javascript!</p>");
</script>

</body>
</html>
```

In this example, we have a basic HTML document and in the script tags (<script>, </script>) we can insert the JavaScript function that will be executed.

2.5 PYWPS

A Web Processing Service (WPS) is a standard defined by the Open Geospatial Consortium which describes how inputs and outputs (also called requests and responses) for geospatial processing services should

be standardized. WPS Version 1.0 was released in June 2007, and WPS version version 2.0 was approved and released in January 2015 (Open Geospatial Consortium, 2015).
WPS defines how a client can request the way process is executed, and how the output is supposed to be handled. Furthermore, it defines the setup of the interface that enables publishing of geospatial processes, and the user's access to those processes. Through this implementation, it should become easier for people who want to publish custom geospatial functions on the internet, to do it in a similar and organized way (Open Geospatial Consortium, 2015).

One implementation of these standards is PyWPS. This service connects the web browser with a variety of tools installed on a server, such as GRASS GIS, GDAL, PROJ and R. PyWPS does not process the data by it self but it can work with GIS software such as GRASS, enabling the creation of GIS-based analytical web services, based on Python (PyWPS Development Team, 2009). The WPS enables a user to describe a process, execute a process and to get the capabilities of the server, and the instances available. Similar to other OGC Web Services (such as WMS, WFS or WCS), WPS has three basic request types. Namely GetCapabilities, DescribeProcess and Execute (Cepicky and Becchi, 2007).

When requesting data from the server, the URL you send to the server, defines what kind of request you have made. Example strings for the three processes mentioned above:

- http://webaddress/pywps/?service=WPSrequest=GetCapablities
- http://webaddress/pywps/?service=WPSversion=1.0.0request=DescribeProcessidentifier=all
- http://webaddress/pywps/?service=WPSversion=1.0.0request=Executeidentifier=<PROCESS>datainputs=[<INPUT1>=<VALUE>;<INPUT2>=<VALUE>]

When an Execute request has been posted to the WPS, it will start processing on the server, and when it is done outputs will be provided encoded in a standardized XML document. A PyWPS service must contain the following elements (Cepicky and Becchi, 2007):

- A class defining the initation of a WPS Process: class Flooding(WPSProcess):
- A function called *__init__*: def __init__(self):
- A function called *execute*: def execute(self):

The init function contains a variety of settings relevant to the process being executed. Furthermore the inputs and outputs get defined, whilst the execute contains the code that is to be run. The PyWPS syntax for calling GRASS differs somewhat from when interfacing purely with Python. An example of GRASS accessed with PyWPS is:

```
self.cmd(['r.in.gdal','input=%s' % self.rasterin.getValue(),'
    output=%s' % original,'-o'])
```

2.6 FLASK

Flask is a web application framework written in Python. The framework is based on a so-called Web Server Gateway Interface (WSGI) called Werkzeug and a website templating engine called Jinja2. The framework is built to be as simple as possible, comes with a core of the most needed libraries, and expects further functionalities and modules to be imported as third-party libraries (Grinberg, 2014).
Several similar frameworks exist, but they either become very advanced, or very specialized. The Flask framework has a simple setup, and is easy to use. (Grinberg, 2014)
By using Flask it is possible to quickly create a dynamic web environment, by writing it in a combination of Python and HTML. A simple application using Flask looks something like the following (Ronacher, 2014):

```
from flask import Flask
app = Flask(__name__)

@app.route("/")
def hello():
    return "Hello World!"

if __name__ == "__main__":
    app.run()
```

2.7 OPEN SOURCE GIS

The definition of Open Source is clearly set by the Open Source Initiative (OSI), making it possible to clearly define which licenses actually are "Open Source". In addition to this, the OSI provides certification to these licenses to indicate that they follow the open-source principles and comply with the Open Source definition. This definition states the following (Laurent, 2004):

- The license should allow the sale or gifting of the software as a part of software distribution along with programs from several different sources. For such sale no royalty or monetary compensation should be required.

- The source code of the software should be included in the software and it must be distributed freely along with the compiled form. In the case where the source code is not distributed with the software, an easily reached alternative must be provided, at most with a minimum reproduction cost.
- Modifications and derived works must be allowed and distributed as freely as the software itself.
- Modification of the source code can be restricted by the license only in the case that the license allows "patch files" distribution along with the source code for program modification. Software built from modified source code must be allowed to be distributed.

GIS task vs. GIS software	query/select	storage	exploration	create maps	editing	analysis	transformation	creation	conflation
Desktop GIS									
- Viewer	●	●	●	○					
- Editor	●	●	●	●	●		○	●	
- Analyst/ Pro	●	●	●	●	●	●	●	●	●
Remote Sensing Software		●	●	○	●	●	●		
Explorative Data Analysis Tools	●	●	●	●	○	●	●		
Spatial DBMS	●	●				○	●		
Web Map Server	●		●	●	○			○	
Server GIS / WPS Server	●	●		●		●	●		●
WebGIS Client									
- Thin Client	●		●						
- Thick Client	●	●	●	●	●	●		●	
Mobile GIS	●	●	●		●			●	
GIS Libraries		●		●		●	●		●

Figure 4: Figure showing categorization of GIS software (Steininger and Hunter, 2012)

- Works derived from the source code may be required to have a different name or version number.
- Discrimination against persons or groups is not allowed
- All fields of endeavor must be allowed to use the software, unrestricted by its license. In the case where the program is redistributed, the same rights must apply without the execution of an additional license from those parties.
- In the case where the program is part of a specific software distribution, parties to whom the program is redistributed should

have the same rights with those to whom the original program is distributed.

- Restrictions on other software, distributed with the licensed software must not be placed under restrictions.
- Access to the license must not be dependable on any individual technology per interface.

As far as GIS software is concerned, some of the most common desktop applications can be seen from Figure 4. The main reason open software exists is because the US Government, during the 1970's and 1980's implemented changes in the patent laws that allowed software and hardware companies to unbundle software and hardware and the source code for software was under restricted access. For the reasons stated above the Free Software Foundation (FSF) was created (Grassmuck, 2004).

Table 1: GIS software

Desktop GIS Software	Developer
GRASS GIS	Neteler & Mitasova, 2008 Neteler, Bowman, Landa & Metz 2012
QGIS	Hugentobler, 2008
ILWIS / ILWIS Open	Valenzuela, 1988 Hengl, Gruber & Shrestha, 2003
uDig	Ramsey, 2006
SAGA	Olaya 2004, Conrad 2007
OpenJUMP	Steiniger & Michaud, 2009
MapWindow GIS	Ames, Michaelis & Dunsford, 2007
gvSIG	Anguix & Diaz, 2008

Categorization of GIS software (Steininger and Hunter, 2012).

Due to their widespread need, Geographic Information Systems software are also available to the public under the "Open Source" label. These software include a wide variety of open source examples that can be divided based on the functionalities they offer. Table 1 below can provide with some insight on such categorization.

2.8 DIGITAL ELEVATION MODELS

A Digital Elevation Model (DEM) is a collection of three-dimensional points (X, Y, Z) that describe the Earth's ground-elevation. Often stored in raster format, these models hold a very important role in the fields of photogrammetry, remote sensing and mapping. They

also are a valuable component in the process of orthophoto creation along with aerial and satellite imagery. There are many ways someone can create a DEM (Liu, 2008):

1. From surveying and topographic maps
2. Using aerial imagery
3. From LiDAR data
4. Through satellite imagery

DANISH ELEVATION MODELS: The Danish elevation models (DK-DEM) are created using LiDAR point clouds. The latest fully available elevation models and LiDAR point clouds were collected from 2005 to 2007, by two Danish companies (Blominfo and Scankort A/S). Responsible for the quality control of the elevation models was the Danish National Survey (KMS). The DK-DEM is using UTM32N/ETRS89 as horizontal reference system and DVR90 as vertical reference (Rosenkrantz and Frederiksen, 2011). In order to estimate the accuracy both vertical and horizontal, a number of control measurements were performed using sophisticated GPS instruments. As far as the vertical accuracy is concerned, the average Root Mean Square (RMS) error was 5.9cm while the standard error was 3.44cm. As far as the horizontal accuracy is concerned, the process in order to calculate it included using 568 corners of 142 buildings. The results from this process resulted in an average RMS error of 6.7cm (Rosenkrantz and Frederiksen, 2011).

3 METHODOLOGY

The project startup was officially set to be the 2nd of February with an expected turn-in date on the 10th of June. This time-span provided the group with approximately four months to complete the development of this project and the accompanying report.
The four months of official working time were divided into a series of phases, in which we expected various parts of the project to have progressed a certain amount.

As far as the working schedule is concerned, we decided as a group on a fixed set of days each week, where the group would meet and work on the project. The rest of the days, each member of the group would work individually on pre-assigned tasks that would be discussed among the group in the next available meeting. Taking the above into account, we set a weekly schedule that was followed throughout the completion of this project as seen in Table 2.

Monday	Tuesday	Wednesday	Thursday	Friday	Saturday	Sunday
Meet	–	–	Meet	Meet	–	Flex

Table 2: Planned weekly working schedule.

In an effort to maximize productivity and collaboration between group members, we decided to follow a classic software methodology, called SCRUM. This choice was made due to the fact that this project includes significant amount of programming work and most of the members have experience working under this method.
Being a sub-version of the Agile methodology, SCRUM adheres to a specific routine:

- Which tasks have been progressed since yesterday?
- Which tasks will be worked on tomorrow?
- Are there any obstacles preventing tasks from completion?

Always keeping in mind this routine, the three to four meetings scheduled each week were used to discuss the direction and progress of the project. During the meetings, group members could present propositions on how to enhance the project or seek assistance in the case where a task could not be completed.
In addition, other means of connection (Dropbox, Google Drive) were established in order to maintain communication among the group

members on the days that group meetings did not occur. Furthermore, code was kept up-to-date between users, by using the code repository Github. This made it possible for all to work on the code at the same time, without diverging from the main script in general. Developing the software and writing the report did not occur in parallel fashion, but extensive notes have been kept and a log was created in order to document the important issues and queries that occurred during the development phase.

The main issue that occurred during the development phase of the project was that in some cases, code needed to be tested on the server to determine whether it performed without any errors. That fact, crippled the flexibility of the group on the occasions where we needed to test fixes for broken code. To be more specific, each time a fix was implemented, the server needed to be restarted in order to load the new script and test its new version. When restarting the server we had to make sure that no other group member was working on the server side, so a waiting gap existed between fixing and testing the code.

3.1 PREVIOUS PROJECT

This project is inspired by the work done during the second semester of our studies. In that project, we developed an ArcGIS toolbox providing a user with tools enabling them to simulate a flood caused by the stowage of sea water. The aim of the project was to use ArcGIS geoprocessing modules to develop a user friendly model capable of modelling the mentioned phenomena.

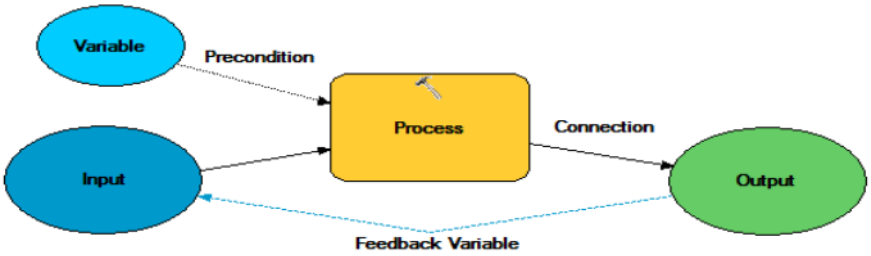


Figure 5: An example model built in ArcGIS modelbuilder.

The toolbox we created, comprised of ArcGIS functionalities with the core being the Cost Distance tool. The development was executed with the built-in Model Builder. Model Builder has the capability of invoking modules and algorithms through an easy-to-use Drag'N'Drop functionality. Although the preliminary model-development

is significantly easier due to the GUI, applying more complex algorithms and connections between models and submodels, can quickly become unmanageable. The developed tool made it possible to simulate water advancing through the landscape, covering the parts of land that were below a certain predefined elevation, and connected to the source of the flood.

After completing that project, we were left wondering how this toolbox could be made available to an even greater amount of people. Another question that arose was how this tool could be made available freely without any of the software limitations that using a proprietary licensed software, like the ArcGIS software suite, provide. In order to pursue such an endeavour, we must make sure that this new product has to offer at least the same functionalities as the one created before. In addition, the way it is developed has to provide us with greater distribution potential in order to ensure widespread availability to any interested organization or person. Keeping these ideas in mind, we reached the aforementioned problem statement and began working on this project.

3.2 WORKFLOW

In order to be in accordance with the goals that we set, before working on the technical aspects of the project, we decided to split the project into several distinct phases. These phases comprise the general technical processes we would have to go through when completing the project. They are created in such a way that they divide our work into parts splitting the work into logical sub-parts. This division will also serve the purpose of making this report easier to read and more understandable. Each phase describes a different process of the project from setting up its foundations to the final, fully working prototype.
The phases will be described as being distinctive from each other. In reality this is not the case. Since this is a product of group work, it made a lot sense to divide the workload among the different members of the group. This would save a significant amount of time since not all different phases required more than one member working on that specific part. That being said, the way the work unfolded, it resulted in overlapping phases. This simultaneous development of parts of the application ensured that constant communication among the group members was a high priority, and also made sure that changes on existing and fundamental parts were not too time consuming. For example, tweaking of the PyWPS in order to connect and enable specific functions as they get developed and deployed to the server, was a task that was worked on throughout the entire project.

Below, is a quick sketch of how the phases were planned to look:

START-UP: This phase involves defining exactly what we wanted to attain with the project.

PHASE 1 - ENVIRONMENT: In this phase we performed the core setup of the project. To be more specific, we decided on which server our data and functionalities will be deployed upon. In addition, we decided on the software and the programming languages we will use.

PHASE 2 - CONVERSION: In this phase we deal with the conversion of the already created model to a model that is compatible with the choices we made on the phase before. In fact, this part does not deal with anything other than that conversion. It is the core of the process because the vital processes of the service are developed here.

PHASE 3 - BACK-END: In this phase, we create the foundation of the web service. To be more specific, it involves the back-end of our service. How it is structured in order to support all the functionalities that will be provided to the user.

PHASE 4 - ADDED FUNCTIONALITIES: All the added functionalities of the application are developed. This is where the extended functionalities of the service is developed, from creating a barrier to finding critical areas.

PHASE 5 - FRONT-END: Here we mainly describe the front-end of the application. This is what the users sees and uses in order to get the results that they need. It includes all interactive parts of the web service that initialize various functions that provide the user with results.

WRITING: As the project would become more and more finished, we will start writing. As mentioned we will keep notes, and write various parts that are important to memorize as the project progresses, but the writing will occur mostly when the project is fully finished. This is done because some aspects of the project might change during the course of the development, and we do not want to have to redo various written parts.

We expected the various phases to take up approximately three weeks but with five weeks set aside for Phase 4 where the functions were developed, a table of the estimated time for each phase is provided in Table 3.

The phases of the project will be explained in detail on the next chapter. The way we will present the development of our work is

Phase	From	To	Duration
Start-up	02/02/15	16/02/2015	14
Phase 1	16/02/2015	09/03/2015	21
Phase 2	02/03/2015	23/03/2015	21
Phase 3	16/03/2015	06/04/2015	21
Phase 4	30/03/2015	04/05/2015	35
Phase 5	27/04/2015	18/05/2015	21
Writing	11/05/2015	03/06/2015	23
Finishing up	03/06/2015	03/10/2015	7

Table 3: Phases and their allotted time

aiming towards a more understandable and logically segmented report. That being said, we think that firstly presenting the completed phases and then documenting the other unsuccessful routes that we tried is, for us, the best way to communicate our work to the reader. To be more specific, the layout of the next section of the report will provide the reader with the process of creating a working prototype in detail. This means that the various tools and techniques that we use will be presented and explained along with arguments on why we think that this is the best available solution at the time of creation. After we are finished with documenting this part, we spend some time presenting the alternatives that we tested before reaching to that solution, if there are any, and provide the reasons and thoughts that lead us to abandon that specific course of action. By separating these two phases of development, we want to make sure that the reader can easily locate what they are looking for, i.e. suggested course of action or courses better avoided.

3.2.1 *The practical foundation*

We decided to base the project, on a well-known location. This would make sure that the project would be grounded in reality, and make it easier to test the project consistently. We decided on using Denmark as our reference country. This was first of all done, because the (previously mentioned) last project was based on working within Denmark as well, and this would give us a logical starting point for our model, and would provide a point of comparison for the results of the new application.

Furthermore, the data standard for Denmark is very high. There are a variety of different DEMs that can be used, and they are created for a variety of purposes, as well as span a broad swathe of different resolutions. Additionally, they are free to use, and can be easily accessed from the Danish Geodata Agencies website Kortforsyningen.

Additionally, the landscape and weather in Denmark do not suffer from being very extreme. What this means is that the landscape as such is relatively uniform and does not experience sudden extreme highs or lows, and the weather is also very consistent, and never experiences such extreme phenomena such as tsunamis.

Some of the highest water levels ever recorded were between 4,3 meters (in Esbjerg) and around 5 meters (in Ribe around 1919). As mentioned, these were extremes, and the historical heights of floodings caused by storm water, have been in the area of 200 centimeters (Kystdirektoratet, 2011).

Towards the year 2050, DMI expects that there will be an increase in the water level between 10 and 50 centimeters in the internal areas of Denmark. (DMI, 2015)

For this reason we decide to model floods in our application based on a water rise of 300 centimeters. The reason for this, is because it will reflect an extreme situation in Denmark, based on the heights provided by the historical records. Furthermore, it will also, to some degree, describe the expected water rise in Denmark.

3.2.2 *Calendar*

Each phase has a weeks overlap with the previous phase. This is done to indicate that as one phase is slowly finishing up, the next one will slowly start up. This also shows how dynamic the creation of a project such as this is. To give an overview of how we were expecting to progress with our project, we sketched the phases above into a so-called Gantt diagram. This gives a nice overview of this overlapping of the phases as mentioned above, and can be seen on Figure 6.

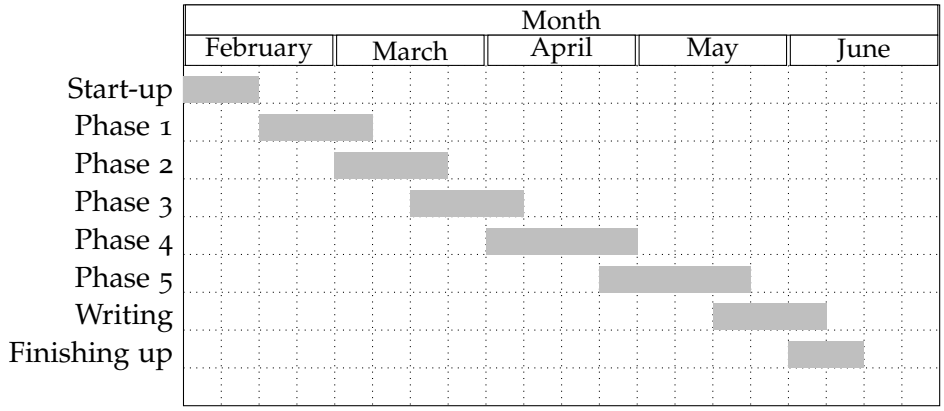


Figure 6: Gantt-chart showing the overlap of phases

4 IMPLEMENTATION

As mentioned previously the work was divided into a series of phases. Before these commenced, we wanted to decide on which tools we were going to use for the application. We knew we wanted to make an online application that would enable the user to model the flooding of a provided area.

To model the flood we decided on using GRASS at it is a powerful Open Source software, with a history of being used in a variety of professional settings such as the American Army Corps of Engineers. Furthermore, the application allows easy access to its core functions through a Python interface. Furthermore, GRASS is natively supported by the WPS called PyWPS. Therefore this seemed like a natural choice for our project. To achieve online functionality, a web server would obviously be needed, and we decdied on using Amazon Elastic Compute Cloud (or EC2) for this purpose. The majority of the development was performed on machines running Ubuntu, so the syntax and explanations will therefore reflect this fact. Being set on the tools we were going to use, we started the actual implementation.

As mentioned in Chapter 3 we used GitHub to store our code. Most of the code discussed in the following part will be very abbreviated. If the reader wants to see how the various functions are written they can be found at the following site:

- https://github.com/GISEmil/Floodject

4.1 PHASE 1 - ENVIRONMENT

This phase involved setting up all the necessary preliminary settings, such as our local and server-side GRASS installations, the needed environment variables, and setting up a server.

SETTING UP A LOCAL SCRIPTING ENVIRONMENT: Instead of working directly on the server, and working with PyWPS we decided to create a local working environment using GRASS with Python, on our own machines. This was done to simplify the workflow and make sure that we did not have to update the server whenever changes were made to the code.
The syntax for running GRASS through Python and GRASS through the WPS differs, so it would have to be changed accordingly when porting it online, but this would allow us to get started immediately.

When working with GRASS, but not explicitly starting a GRASS session, a variety of environment variables have to be set.

- GISBASE: This points to the top level directory of the GRASS installation
- GISDBASE: GRASS' database location, usually initiated as grassdata.
- PATH: The path location of GRASS, usually includes /script/ and /bin/.

These settings are set up in the header of the project, and for our set up the settings look like this:

```
gisbase = '/usr/lib/grass70'
os.environ['GISBASE'] = gisbase
os.environ['PATH'] += os.pathsep + os.path.join(gisbase, '
    extrabin')
os.environ['GISDBASE'] = gisdb
```

When running the python script, it now initiates the GRASS environmental variables that are necessary for importing and using GRASS' functions.

SETTING UP THE SERVER: Before being able to set up PyWPS, it is necessary to set up a server, capable of serving the processing service. As PyWPS was created with Linux in mind, and because the creators even specify that it works better on a Linux-based operating system, we wanted to make sure that the server was running this as well. Today there are a variety of different hosting services available, but for this project it was decided to use Amazon Web Services (AWS).
The main reason for running the server on this service is because several members of the group have had previous experience with launching minor applications on this platform, but also because it is possible to launch a so-called micro instance, which is free for the first year. The free-tier provides a very basic server, with little processing power and hard drive space - but for a proof-of-concept project such as this, it would suffice.
Initiating a server on Amazon's Elastic Compute Cloud (EC2) is easy, and only takes a couple of minutes. As several members of the group were already using the Linux-based operating system Ubuntu we decided to base the server on this.
Connecting to your server is done using SSL (Secure Sockets Layer), and when using Ubuntu, can be performed with the terminal. The only situation where this would not be so, is when having to upload specific files. For this a FTP client software is used.

SETTING UP APACHE: Every instance set up with Amazon, is provided with an IP address which can be used to visit the server using a browser. The Ubuntu image installed on the server did not contain a pre-installed web server, so it was necessary to install and configure this first. Installing software on a machine running Ubuntu is easy, as it uses the package manager aptitude. Using the following terminal command, the web server is installed:

```
sudo apt-get install apache2
```

Now when visiting the IP address provided by Amazon, a standard Apache welcome-page greets the visitor. The web server will serve the files that are placed in the following folder:

```
/var/www/html/
```

So this is where most of our server-side changes would occur during the development of our application.

INSTALLING GRASS: The next step is setting up GRASS on the server. Version 6.4 was installed by using the following command:

```
sudo apt-get install grass
```

GRASS depends on a folder with the name *grassdata* on the computer's home directory. This functions as a database, and where its input and output will be stored. Instead of setting this up from scratch, our local *grassdata* folders were copied straight to the server's home drive at

```
/home/ubuntu/
```

INSTALLING PYWPS: Now that Apache and GRASS have been installed, it was time to install the backbone of our entire project, PyWPS. Installation of PyWPS was performed into the folder of our webserver at the following directory:

```
/var/www/html/pywps/
```

We also provide a thorough walkthrough of the installation of PyWPS on our Github page.

4.1.1 *Problems encountered*

SETTING UP GRASS SCRIPTING TO RUN ON A WINDOWS MACHINE: It turned out that setting up the GRASS environment on a Windows machine, was not very easy. This is because the GDAL and OGR packages depend on a variety of specially created bindings when using them in a Windows environment, so this can cause trouble if you

do not download the right ones.

LACK OF DOCUMENTATION: As the installation instructions of the PyWPS was not very well documented, and because a wide variety of settings have to be set correctly during installation, it was necessary to reinstall the server several times. This was both time consuming and frustrating, but the steps to install the service have now been documented greatly on our Github, so that hopefully someone else can now do it faster.

GRASS64 VS GRASS7: We actually started by installing the newest version of GRASS, GRASS7. It turned out that there were some incompatibility issues when using PyWPS with GRASS7. This meant that we had to downgrade the server-side GRASS package to GRASS64.

4.2 PHASE 2 - CONVERSION

The model of this project is based on our previous project mentioned earlier. Since that project was developed in an ArcGIS environment, each functionality had to be converted to GRASS python modules. Some of the modules are not directly transferable between GRASS and ArcGIS, therefore we needed to find the best fitting functions that would also keep processing time at an acceptable level.

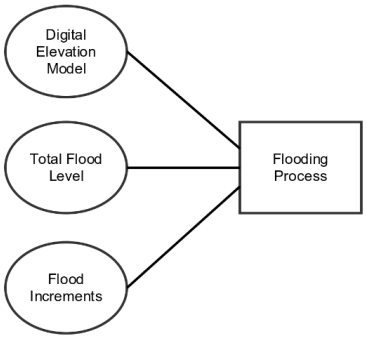
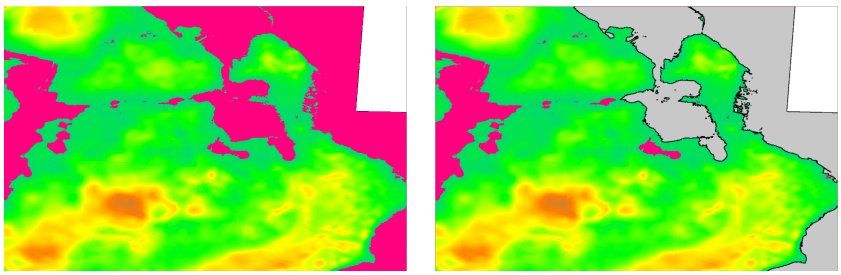


Figure 7: Inputs for the flooding model.

Figure 7 is an attempt at showing the inputs used for the previous project. As can be seen, only one external data source is required - a DEM. According to the LOCATION used within GRASS, every imported DEM has to have a WGS 84 coordinate system. An important factor of the model, is that all web-mapping operations are done on Open Street Map, which uses a slight variant of WGS 84 as well. As mentioned in theory, GRASS uses GDAL for importing several types of raster maps and OGR for importing vector datasets. Taking advan-

tage of these modules makes it possible to use any raster, as long as it is projected into WGS 84. Another input of the model is the maximum flood level and the water rise increments of the flood. These numbers define the amount of loops necessary before the flooding is complete.
To flood the DEM, the model extracts those areas that are below current level of the iterator (and therefore flood). The first iteration will always be the sea level, where the actual level of flood is 0. This extracts every cell with a value of zero.
This method will extract *ALL* areas that have a value of 0, which usually will generate several disconnected clumps of cells, located in different areas of the DEM. As our model has to generate a flood from one specific area (so as not to simulate a flood of all streams and lakes in the area) the model will require that the user interacts further. The user needs to choose from which source they would like to flood the DEM from. Therefore the pre-selected point has a major role in this part of the model, as the flooding will be based on it.
After extracting every cell that is below the actual flood level, the process runs a cost distance analysis using the point as a source. The returned raster shows only one continuous area, for that level of the flood. The output cost raster is then converted to a vector with the value of the flood level. Each iteration extracts the land from the raster which is below the current flood level and outputs the continuous area depicting the actual flood on the land from the preselected source.

The process is illustrated on Figure 8. On Figure 8a, the algorithm identifies every possible cell that is under the current water level, indicated by the pink color. After the r.cost module has finished, the process returns a layer indicating which cells are reachable, as seen on Figure 8b.



(a) Every cell with a value less than the current flood level.

(b) Cells reachable from the source point.

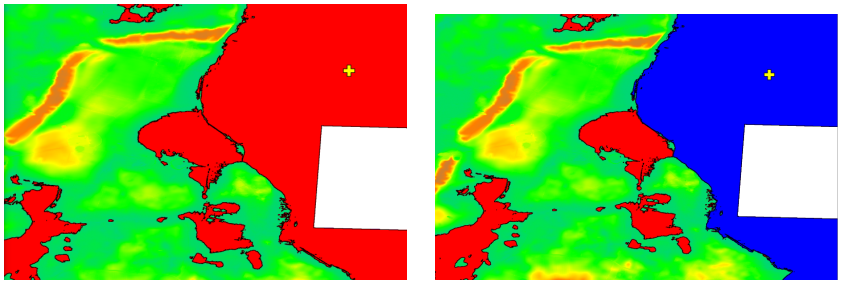
Figure 8: Figures indicating how r.cost is used to identify continuous areas

Every complex module from the earlier project has been taken out, and it has been simplified by using basic geoprocessing tools such as rasterizing, vectorizing, raster calculator and cost analysis. After the

ArcGIS to GRASS conversion was done, additional functionality was created, which will be discussed in Section 4.4.

4.2.1 *Problems encountered*

From the very beginning we were planning on using cost distance as the basis of our flood model, but we developed a different solution for the flooding algorithm, which turned out to be problematic - a problem we will address later in this section. The intent was to substitute the cost distance function with a more simple, and significantly faster method. The procedure entailed spatially selecting only the extent which is connected to the source vector feature. The process is mostly the same, however instead of running the cost distance function, the extracted areas are converted to vector polygons and by using an intersect tool (v.select), only the polygon which is intersected with the pre-selected vector point is selected, this can be seen from Figure 9a The red polygons represent all the areas that can possibly be flooded with a 1.2 meter water level. However these areas are widely spread across the investigated area. The selection is based on the user's pre-selected source, which is the yellow cross in the figure. The GRASS module selects the intersecting flood polygon, which represents only the stage of flood which is connecdted to the source, illustrated by the blue polygon on Figure 9b.



(a) Every cell with a value less than the current flood level.

(b) Cells reachable from the source point.

Figure 9: Figures indicating spatial intersection used to identify continuous areas

COST DISTANCE VS SPATIAL SELECTION Comparing the cost distance and the spatial vector selection methods, the following problems and drawbacks appeared. According to our tests with the two process, we realized that the processing time was significantly greater with the cost distance function, because of its cell by cell calculation. At the end of the development stage we realized that the spatial selection did not behave in the same way as the r.cost functionality does. As is described earlier, the spatial selection selects only the polygon connected with the source. Since the vectorizing is done based on the

cell resolution, if a cell is only diagonally connected to a coherent area, then it will be allocated as a different polygon, as presented on Figure 10.

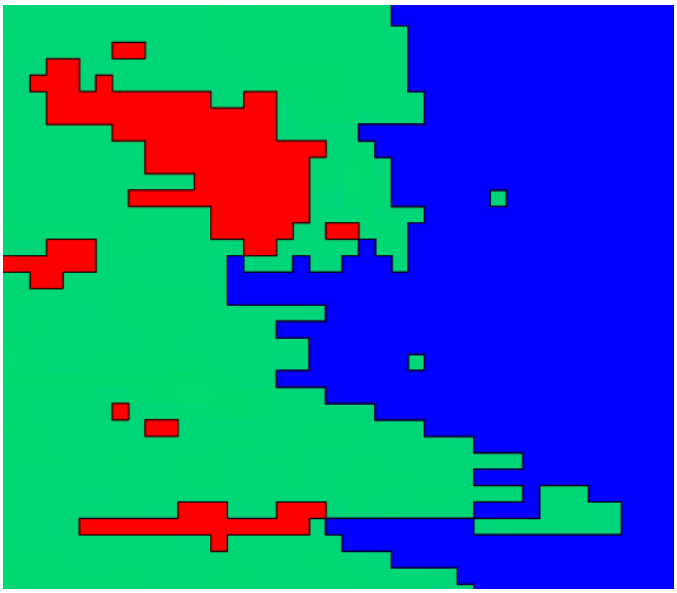


Figure 10: Image illustrating problem with the intersection method.

Therefore every cell, which is diagonally connected to a coherent flood surface will be disregarded, even though they, according to the flow direction and accumulation theory, would be the part of the flood extent. We tested both functions and have confirmed a significant impact on the results of the process. For example, comparing a flood scenario of a 3 meter flood rise with 10 cm increments, it can be seen that at stage 1,20m, the flood extent is significantly different, as seen on Figure 11

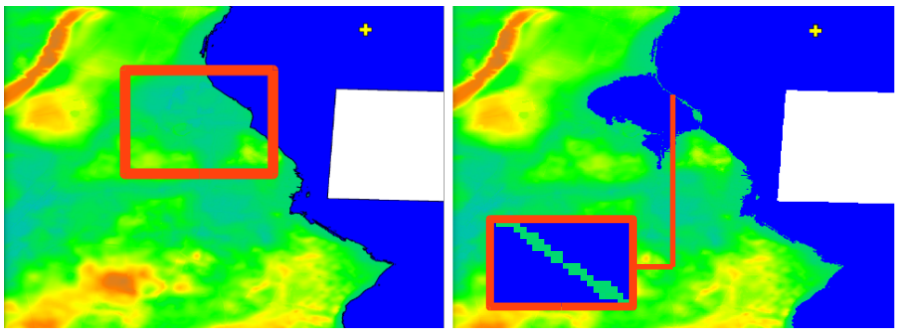


Figure 11: Difference of flood extent, based on model used.

4.3 PHASE 3 - BACK-END

This section will focus on the back-end of our web server. We begin by initializing Flask on our server. It is important to keep in mind

that we earlier installed the Apache web server. After extensive research online, several different sources suggested a universal process on how to install Flask on an Apache server. To go on, before actually installing Flask, we need to initialize mod_wsgi. Mod_wsgi is a tool that specializes in serving python applications from Apache servers. Installing mod_wsgi is quite easy, and only takes a few lines of command line code:

```
sudo apt-get install apache2 apache2-base apache2-mpm-prefork
    apache2-utils libexpat1 -ssl-cert
sudo apt-get install libapache2-mod-wsgi python-pip git
pip install flask
```

Following this installation we need to create an *application.wsgi* file. This is basically a file that contains code that initializes the application and comes with the wsgi file extension. Now that Flask and prerequisites have been set up, it is time to look at what the backbone of the application looks like. At first, it is important to mention that the initial setup of the application will use Flask. To start with, and keeping in mind the way Flask actually works, we need a main Flask script that can initialize the application and connect various functions with its main core. In addition, that main script will also be connected to various html pages depending on where the user wants to navigate in our application. The first action we took was to change the default path of our application in the server. Instead of using */var/www/html*, we started using */var/www/html/FlaskApp/FlaskApp*. This path will be referred to as the root url. After doing that, we create the script that will perform the functionalities of the application. That script is named __init__.py. Using Flask allows us to connect to various html scripts by using only one main script. Depending on the URL of the page the user wants to get to, the corresponding HTML script is called and displayed on the user's screen. For example, if the user visits our homepage, which is set by the *app.route()*, then the *'index.html'* will be initialized and displayed on his screen:

```
@app.route('/')
def hello_world():
        return render_template('index.html')
```

The first action the user must perform in order to begin using the application is to upload a DEM. When the user clicks on the FLOOD button on our starting page, they get redirected to the root url/upload section of the application. The way that section is working is that it expects a file to be posted. When that happens, it saves it to a pre-designated folder on the server. Right after that, we create a copy of the uploaded file and convert it from .tif to .png. The reason behind that conversion is that we need to display the uploaded ele-

vation model on a map for the user to see, which is vital to the next steps of the application. In order to be able to overlay an object on top of a map using the leaflet mapping library, then that object has to be of .png or .jpeg format. All the aforementioned functions are included in the script below.

```
def upload_file():

        if request.method == 'POST':
                file = request.files['datafile']
                if file:
                        filename = secure_filename(file.filename)
                        file.save(os.path.join(app.config['
                            UPLOAD_FOLDER'], filename))
                        src_ds = gdal.Open( os.path.join(app.
                            config['UPLOAD_FOLDER'], filename) )
                        formatimage = "PNG"
                        driver = gdal.GetDriverByName(
                            formatimage )
                        fileName, fileExtension = os.path.
                            splitext(filename)
                        finalloca = '/var/www/html/FlaskApp/
                            FlaskApp/static/images/' + str(
                            fileName) + '.png'
                        dst_ds = driver.CreateCopy( finalloca,
                            src_ds, 0)
```

When the upload process is complete, a function is initialized allowing us to display that image on the map. Since we are using leaflet, in order to be able to display an image, we need to provide the function with the coordinates of the South-West and North-East corners of the image's display boundaries. To acquire this piece of information, we use GDAL. The way we obtain the required coordinates are show below.

```
                        width = ds.RasterXSize
                        height = ds.RasterYSize
                        gt = ds.GetGeoTransform()
                        minx = gt[0]
                        miny = gt[3] + width*gt[4] +height*gt[5]
                        maxx = gt[0] + width*gt[1] + height*gt[2]
                        maxy = gt[3]

                        return redirect(url_for('upload_file',
                            filename=filename,minx=minx,miny=miny
                            ,maxx=maxx,maxy=maxy))
```

The final step we need to take before we are able to show the uploaded image, is to pass coordinates back to the html document that is responsible for showing that image so that a javascript function can

get and display the image properly. That is achieved by the final line of the script on the image above.

4.3.1 *Problems encountered*

Having presented a viable option on how our web-service is structured and what tools we used to achieve successful functionalities, it is time to examine what other alternatives we have explored that did not result in acceptable results.
As presented above, we use Flask to create the back-end of our application. This decision was not our initial one, since none of the group members had any particular experience using this framework. That being said, it normally would seem a rather unorthodox approach to start using a tool that no one is familiar with at the middle of our project development, since that is when we introduced Flask to the project. The truth is that our first option was to use an HTML and JavaScript core and PHP to upload the user provided input to the application. To be more specific, we would have an HTML document that allowed the user to upload a file and use PHP to convert that file from .tiff to .png. Then that PHP script would call a python script that calculated the bounding box coordinates of the image and then pass them on to a second HTML document, to be used by a JavaScript function that overlays the image on the map. The reason we considered using that approach is that we were more accustomed to using PHP to perform specific functions such as manipulating and uploading a file. On the other hand, this approach is clearly much more complicated than the one we finally used. Especially, if we keep in mind on how many different programming languages are included in that approach and how many different scripts we need to connect in order for it to work. In addition, the transition from PHP to python was never achieved up to the point where we decided to change directions.

What we managed by using Flask instead, was to exclude the use of PHP and thus reduce the complexity of the script by a great deal. Firstly because we do not have to worry about creating extra connections with various other scripts. In fact Flask, and the way it is designed to operate, simplified our development considerably. It allowed us to have a central script of python that inherently interconnected with all our HTML scripts and supported other python functions at the same time.

4.4 PHASE 4 - ADDED FUNCTIONALITIES

POUR POINT ANALYSIS: Only providing the extent is not enough when wanting to prepare against the impact of a disaster. We wanted

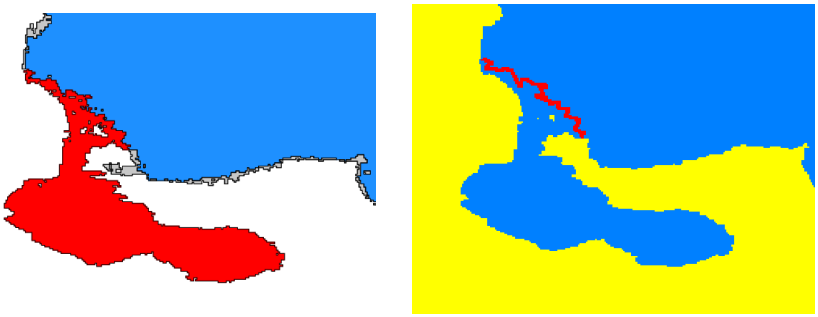
to provide more information about the flooded areas that might be in danger. After monitoring the spread of the flood using our model, we realized there were areas where a relatively large area got flooded immediately. This can occur when the flood steps over a low elevation point on a natural barrier. This is what we call a pour-point. One of our major challenges was how to identify this behaviour and to locate the actual position of these critical points automatically. These pour point phenomena are very dependent how the process is run; which increment the water is rising in; which output (raster or vector) we are working with. Firstly, we realized that in order to give an accurate flooding, the increments would have to be 1cm. By using this increment, the flood can be accurate enough to identify major changes between consecutive flood levels.

Figure 12: Figure showing the pourpoint we are trying to identify.

Figure 12 shows two consecutive flood extents with 140cm (light blue) and 141 cm (dark blue) water rise. On Figure 12 the red cross indicates what we needed to locate. Although it might seem to be a fairly easy procedure, we have to add that the large blue area on the image is not the only area that has increased in size between the two stages of flooding. Along the border of 140 cm (light blue) flood level, there are a relatively small areas or cells which are flooded at the 141 cm. In order to avoid small or irrelevant pour point identification the following workflow was created.

As described in phase 2, after each flood extent the output is stored in both raster and vector format. In order to identify pour points at

each level, we implemented a condition which checks the size difference between each consecutive stage. We added an area field for each flood extent polygon, enabling us to make a decision based on the flooded area. If the difference between two consecutive flood stages is larger than one percent of the investigated DEM, then the pour-point analysis workflow is triggered. The two investigated flood extents are subtracted from each other, and a selection based on its area field is conducted. By performing this subtraction, a lot of polygons are created. For each polygon the process calculates an area field and selects only those that are larger than 1000 square meters. This criteria is set to avoid relatively small areas (indicated in grey on Figure 13a), which the pour point analysis would not make a reasonable investigation of since their size is negligible compared to those areas which have a larger impact on the advancing of the flood (indicated by the red color). After the exclusion, only those areas which have a larger extent than the previously mentioned criteria are left.



(a) Showing two flood extents. (b) Showing line of pour points.

Figure 13: Showing the concepts of finding the pourpoints.

The remaining polygon(s) are converted back to raster and merged with the previous raster extent of the flood, thereby a new raster map represents the previous flood (blue) and the actual flood (red). This raster has values where the flood extent exists and NULL values everywhere else. Afterwards a neighborhood analysis (r.neighbors) is conducted, based on the diversity method. This method is applied in order to find the location, where the raster changes values, since that place is where the flood enters into the new area. In Figure 13a a black border can be seen, where the red and blue area touch. The process needs to automatically identify only that border which connects the two different flood stages in the critical location - i.e the pour points. The diversity method can be used to identify only the border cells as it seen in Figure 13b. After identifying the pour-point cells with the aforementioned analysis, the algorithm then extracts those cells and converts them to vector points.

To have the pour-points as vector points is partly because we can then easily store information about the necessary height of a barrier at this place, and so that they easily can be displayed and manipulated on our web map following the analysis procedure. For the better understanding and providing essential information of the critical area, we decided to add the elevation of the pour-point location and the maximum flood depth based on the actual scenario. This is done by giving the raster generated by the diversity function, not the value of the diversity, but the value of the maximum water flooding the landscape, minus the elevation of the DEM at that point. The function that does this is provided below:

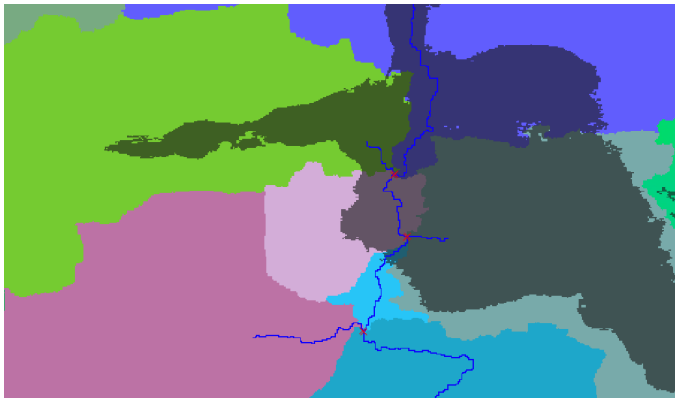
```
self.cmd(['r.mapcalc', '%s=if(%s == 2, 3 - %s, null())' % (
    pourpoint, 'diversity', original)])
```

When the raster is converted to vector, these values will automatically become the attribute table values of the vector point.

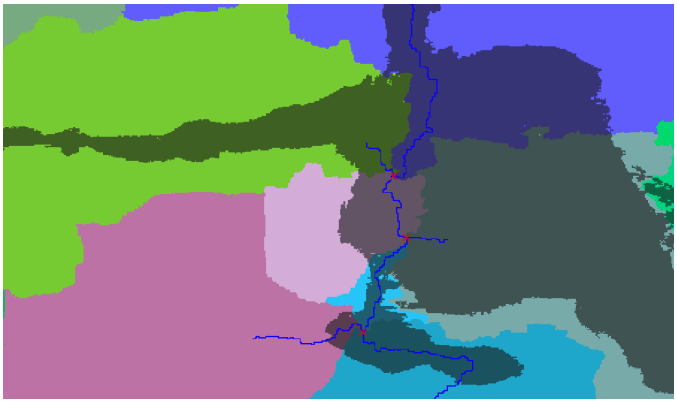
OUTLET ANALYSIS: During the development of the pour-point analysis, we have encountered different problems and theoretical interferences. At the beginning, we tried to derive the pour point analysis from a watershed analysis, based on the identification of watershed outlets. Our hypothesis was, that the flood flows backwards through the flow accumulation and flow stream model from the source-cells and when the water enters to a new watershed, then that area gets flooded. Therefore we created a workflow which extracts only the outlet of the watershed. As presented in the theory part, we used the watershed (r.watershed) and flow accumulation module (r.terraflow) in order to determine the inlet-outlet location of the watershed. We tried to establish an algorithm for relative watershed determination based on the investigated DEM. We set up the minimum threshold size of a watershed - this has to be at least ten percent of the DEM. After the workflow finished running the previously mentioned neighborhood analysis (r.neighbor) using the diversity method, we could extract the watershed borders on a raster. Using the terraflow module with single flow direction setup, we created a flow accumulation raster. This is important to determine the inlet/outlet location. Using the single flow direction method, every outlet will be located on the border of a watershed where the flow accumulation is the maximum. With this approach we thought we could get a relatively accurate result to determine where the flood will flow into a new area.

During the testing of this function we wanted to justify the importance of the watershed inlet/outlet by predicting the sudden flood advancing. As can be seen in Figure 14, the flood is following backwards the flow stream of the watershed and entering to the adjacent watersheds. However we realized after investigating the flood at stage

170 cm and 180 cm, that the flood only partly occupies the watersheds (Figure 14). This is because each watershed has different morphologies, therefore in a particular floodstage the watershed will not be completaly flooded. Also we realized, that several watersheds can be flooded partially in order to compose together a relatively continuous flood extent.



(a) Showing the placement of outlets before spillover.



(b) Showing the placement of outlets after spillover.

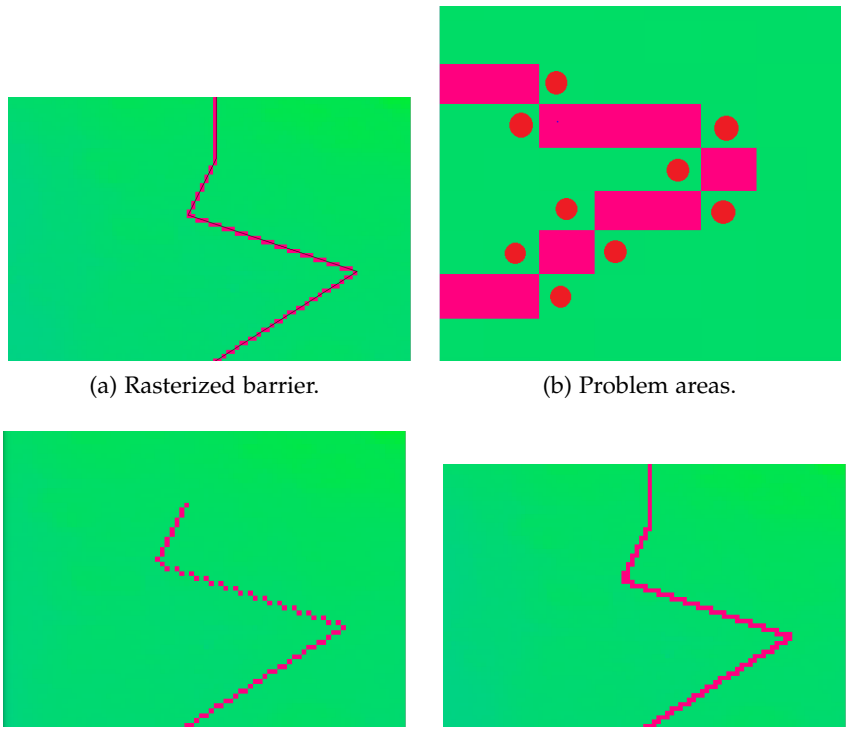
Figure 14: Showing the relationship between watershed outlets and the flooding of water.

Figure 14 shows a relatively large take over of the flood. Since this flood extent is advancing regardless of watershed position, the outlet (red cross') have no meaning for the further investigation of predicting pour-point positions.

BARRIER PLACEMENT: An integral part of the application is allowing the user to implement barriers that will mitigate the effects of the flood. The user will be able to determine whether an area can be secured by the implementation of obstacles in the landscape or how the flood can be affected by installing a barrier in a given position.
The general operation of having the barrier placement process available to the user starts after the flood simulation has been performed and completed. This is because the user has to have an overview of

the overall situation of the flood and the location of the critical areas in the landscape, before starting placing barriers in random areas and in an arbitrary fashion. We believe that this would ensure that the entire process of finding a suitable solution to a flood situation is fast since performing the simulation can take a lot of time.

The idea behind the barrier creation is to get the input from the webpage where the user will draw a barrier on a web map. This results in the creation of a GeoJSON file that is then transferred to the general workflow. Since the barrier would be in GeoJSON format, we need to convert it to raster in order to hardcode it into the elevation model. Normally, converting the barrier to raster and then adding it to the elevation model would be enough. Unfortunately, we stumbled upon a quite unexpected problem. The sections of the barrier that are not completely vertical or horizontal are rasterized in an erroneous fashion. To be more specific, in these sections the cells are not connected with a shared side of a cell. Instead, they are connected by sharing an edge. The figure below demonstrates the problem when the barrier is converted to raster, as can be seen in Figure 15a.



(a) Rasterized barrier.

(b) Problem areas.

(c) Generating pixels in the designated areas.

(d) Fixed barrier.

Figure 15: The process of fixing the rasterized barrier.

The problem depicted above is quite unexpected especially if we take note of that fact that the conversion ignores several cells that

intersect with the barrier line and instead keeps only cells that are connected diagonally with each other. That leads to gaps on the barriers that would allow water to flow through them. In order to deal with that problem we have developed the following solution.
Practically, we need to assign the pre-designated value of the barrier height to one of the cells per diagonal intersection to create a continuous raster of the barrier, as indicated on Figure 15b.

In order to do that, first we extract all non-vertically and non-horizontally connected edges of the cells. In addition, we extend those cells by one cell in the northerly direction. That action is performed by the following part of the script:

```
#extract the edges of the barrier when its direction change
gscript.run_command('r.mapcalc', expression='edges_1_1 = if(
    barr_raster[-1,-1]==barr_raster, barr_raster, null())')

#extend the extracted edges to the north by 100%
gscript.run_command('r.mapcalc', expression='edges_1_1_ext =
    edges_1_1[1,0]')

gscript.run_command('r.mapcalc', expression='edges_11 = if(
    barr_raster[-1,1]==barr_raster, barr_raster, null())')

gscript.run_command('r.mapcalc', expression='edges_11_ext =
    edges_11[1,0]')

gscript.run_command('r.mapcalc', expression='edges1_1 = if(
    barr_raster[1,-1]==barr_raster, barr_raster, null())')

gscript.run_command('r.mapcalc', expression='edges1_1_ext =
    edges1_1[1,0]')

gscript.run_command('r.mapcalc', expression='edges11 = if(
    barr_raster[1,1]==barr_raster, barr_raster, null())')

gscript.run_command('r.mapcalc', expression='edges11_ext =
    edges11[1,0]')
```

As can be noted in the script above, we are using neighborhoods in order to extract the necessary cells. In fact, we are not using eight-cell neighborhoods but specific cells that exist in the neighborhood of the cell which mapcalc is currently querying. In order to query the correct cells, we are required to know their relative coordinates to the center cell that is queried. These relatives coordinates are displayed on Table 4.

The result of this edge selection is shown on the Figure 15c

Since we have the edges of the barrier, we go on by expanding each one by one cell, as mentioned above, and merge the resulting raster

-1,-1	-1,0	-1,1
0,-1	0,0	0,1
1,-1	1,0	1,1

Table 4: Relative cell coordinates to the center cell (0,0)

with the initial barrier raster. That is done with the following script:

```
gscript.run_command('r.patch', inputs=['edges_1_1_ext','
    edges_11_ext','barr_raster'],output='merged)
```

The final step in order to complete the barrier correction and deployment to the landscape is to hardcode it to the elevation model.

This process results in the elevation model shown on Figure 16.

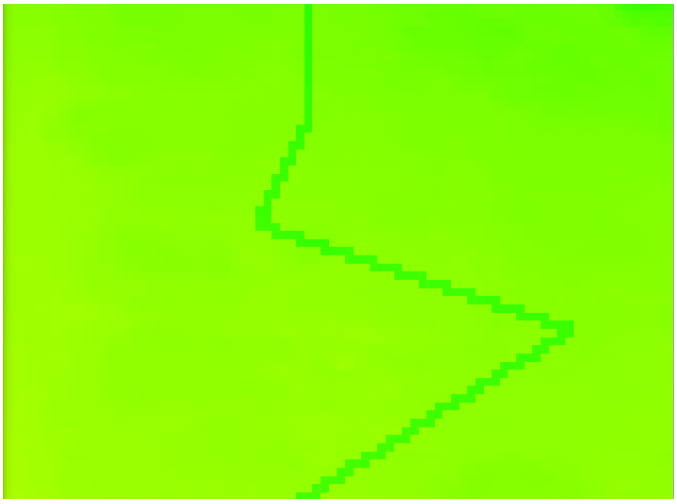


Figure 16: Showing the barrier hardcoded into the DEM.

By observing the result above, someone might think that it is difficult to observe the actual barrier and distinguish it from the elevation model. Unfortunately, this is the result of merging the two rasters together. It is logical that the results are not visually distinctive since they are in fact one raster represented by a sliding color scale.

4.4.1 *Problems encountered*

The main issue with the process of creating the barrier and performing the necessary correction was the necessity of actually performing the correction of it. That means that normally, the conversion of a linear vector file to raster should not behave the way we described above. We assumed that GIS software suites, such as QGIS and GRASS as

well as ArcGIS, address issues regarding the connectivity of cells in identical fashion. But unfortunately that was not the case. In fact, ArcGIS considers cells that share an edge (diagonal connection) as connecting cells and will include them in a vector to raster conversion. On the other hand our experience was that GRASS could not do this. This lead us to dedicate a considerable amount of time to creating a solution.

Another issue that we faced was the lack of documentation on the mapcalc function that we used in this process. Despite following the existing documentation closely, we ran into problems that should not have existed, in our opinion. Initially, the proposed solution to the barrier problem consisted of a much more compact script that performed a universal query on all the cells of the barrier raster in order to cover its gaps. Unfortunately, for reasons yet undiscovered, that solution did not produce any results. The script we used can be seen below at its final state before we decided to develop the solution that performed as expected.

```
corr_barrier = if(isnull(barr_raster), if(barr_raster[0,-1] == 3
    && (barr_raster[-1,0] == 3 || barr_raster[1,0] == 3) ,
    barr_raster, null()) || if(barr_raster[0,1] == 3 &&  (
    barr_raster[-1,0] == 3 || barr_raster[1,0] == 3), barr_raster
    , null()), barr_raster)
```

4.5 PHASE 5 - FRONT-END

This phase involves setting up the necessary website relevant elements. This means both HTML, Javascript, web mapping and the integration of PyWPS.

PYWPS: As the the main functionality of our script was developed, we began porting the script to PyWPS syntax.

As mentioned in the theory part, every PyWPS script must include these parts:

- __init__

and

- execute.

We set up the required settings to be:

- identifier = "flooding",
- title = "Flooding",
- abstract = "This process is used to flood a DEM with water",

- version = "1.0",
- grassLocation = "/home/ubuntu/grassdata/WGS_1984",
- statusSupported = True,
- storeSupported = True

The most un-intuitive of these settings are the *grassLocation*, *statusSupported* and *storeSupported*. We set *grassLocation* to our predefined database folder. This is done in order to make sure that the uploaded data will work within a coordinate system, and that we can do the proper calculations on it. When setting *statusSupported* to True, it means that the process can run asynchronously. And finally, using *storeSupported* means that it is possible to deploy the results to the server, for later usage. After these have been set, we define the needed inputs. As mentioned previously, we want the user to provide a DEM and a point indicating where the flooding should occur from. Therefore we must inform the program that we will be expecting an image and a GeoJSON file.

```
self.rasterin=self.addComplexInput(identifier='rasterin',
                    title="input image",
                    formats = [{'mimeType': 'image/tiff'}, {'
                        mimeType':
                                        'image/geotiff'},
                           {'mimeType': 'application/geotiff'},
                                                {'mimeType': '
                        application/x-           geotiff'}])

self.vectorin = self.addComplexInput(identifier="vectorin",
                      title="Input point",
                      formats = [{"mimeType":"text/json","encoding
                          ":"utf-
                                        8","schema":None}])
```

After having defined the inputs. It is also necessary to define the outputs. We want to provide the user with a raster of the flooding and the extent of the flooding as a vector file. Furthermore, after the process has completed, we want to provide an in-browser image of how the actual flooding looks. Because very few browsers natively support the display of TIFF images, we will output a JPEG as well:

```
self.outputImage0=self.addComplexOutput(identifier="output0",
    title="output image", asReference=True)

self.outputImage1=self.addComplexOutput(identifier="output1",
    title="output image", asReference=True)

self.outputVector=self.addComplexOutput(identifier="output2",
    title="output vector", asReference=True)
```

By setting *asReference* to True, the output will not be delivered straight back to the user, but will be provided as a link to the location in which it is stored on the server. When returning images with PyWPS, they get delivered as Base64 images, which means that they are returned as a long string of text. This can increase the loading times significantly, and can for very large images also freeze the browser handling them. Furthermore it has the advantage, that if a user should lose connection, the images can still be recovered.

Now that the initialization settings have been defined, the execute method can be set up. In general the code looks very similar to the functions created earlier, so they will not be explained here further (and they can be found on the previously mentioned GitHub page). After creating the service, they are placed in the processes folder on the server. Furthermore the __init__.py file is updated to now include the uploaded file. After having set up the script in PyWPS, and transferring them to the server, they are ready to be called. This can be done through the web browser. In general it is possible to call either a GetCapabilities, Execute or DescribeProcess. For this project it will only be necessary to Execute a process, as this project is not meant to be accessed outside of our web application interface. They way the WPS service is requested from the server is as follows:

- http://52.17.144.192/cgi-bin/pywps.cgi?request=execute&service=WPS&version=1.0.0&identifier=flooding&datainputs=[rasterin=<LINK TO DEM>;vectorin=<GEOJSON AS STRING>]

The server will process the request, and return a XML document with links to where the outputs of the process are stored.

SETTING UP THE DESIGN: Using the power of Flask as much as possible, we wanted to make the website have as small of a footprint as possible. The application would consist of a welcome page with a brief introduction which would link the user to the actual application, which would be followed by a multi-layered page, that would guide the user through the process of flooding. Using the free Twitter Bootstrap CSS theme, it is possible to quickly create some working HTML pages.

As the front page mostly contains information about the projects capabilities, this part will mostly focus on the second page.

To guide the user through the process of flooding, it was decided to split the entire process into three steps:

- Step 1: Upload DEM
- Step 2: Select Source
- Step 3: Get Results

The three steps consist of various HTML parts that are hidden from the start, and as the user clicks through the steps, the next one is shown and the previous one is hidden. This is done using the following jQuery commands:

```
$.(#step1).hide();

$.(#step2).show();
```

At most of the stages, asynchronous requests or operations will be performed. As they can take some time, we want to give the user some feedback that the page is loading. This is done by creating an overlay that takes up the entire screen, and includes a loading icon. The icon is hidden, but gets shown when a request is performed, in the same manner as above. To describe how the outlined stages work we will go through them below.

STEP 1 - UPLOAD DEM: In this step the user uploads a DEM. Most of the functionality here is dependant on the functionality of Flask. So the only thing necessary to add here, is a HTML form with the capability of letting the user browse his computer for a DEM, and then clicking a button to upload it.

STEP 2 SELECT SOURCE: In this step we want the user to select from where the flooding will occur.

This entails showing the uploaded image on a webmap, making the user able to place a marker on the map, and then sending this information to the server.

When the upload has completed, the page refreshes. This means that the page clears all set variables. This is an issue when wanting to stay on the same page, but to load a different step. As Flask adds a variety of variables to the URL of the page, a workaround for this problem is to check to see if the URL contains one of the variables assigned by the Flask upload script:

```
url = window.location.href;

if (url.indexOf("filename") > -1); {
        STEP 2
```

As mentioned, we want the DEM to be uploaded to a webmap. To do this, a copy of the uploaded DEM is created as a PNG, which then can be used to overlay on the map. The PNG cannot contain embedded coordinates. Using Flask and GDAL, the corners of the uploaded TIF are extracted, and get returned to the URL of the webpage. The format of this is something like:

- http://52.17.144.192/upload?minx=COORD&miny=COORD&maxx=COORD&maxy=COORD&filename=FILENAME

To extract the relevant data the following JavaScript code is executed:

```
var findcorners = url.replace("http://52.17.144.192/upload?");
var findcorners2 = findcorners.split("&")
var filename = findcorners2[4].split("=")
var coordinatesforuse = [];
var expendable;
for (var i = 0; i < findcorners2.length; i++) {
    expendable = findcorners2[i].split("=")
    coordinatesforuse[i] = expendable[1];}
These are then stored into variables with a readable format:
var south = coordinatesforuse[0];
var west = coordinatesforuse[1];
var north = coordinatesforuse[2];
var east = coordinatesforuse[3];
```

Lastly they are prepared for insertion as the bounding box for the uploaded image, into the leaflet map:

```
imageBounds = L.latLngBounds([
      [west, south], // Southwest
      [east, north] // Northeast
]);
```

As the uploaded TIF is converted into a PNG, but not returned to the URL, we are manually going to setup the URL where the PNG version of the uploaded image can be found:

```
var imageuse = filename[1];
var imageaspng = imageuse.replace(/\.[^/.]+$/, ".png")
var variable = 'static/images/' + imageaspng;
var imageUrl = variable;
```

Now that all of this is done, we can initiate our map. Add the uploaded PNG as an overlay to our MapBox image, and define the marker that will be used.

```
// Initiate map "around" the uploaded DEM
var map = L.mapbox.map('map', 'piratosthegreat.i7aeacaf').
    fitBounds(
imageBounds);
// Add the DEM to the map
var overlay = L.imageOverlay(variable, imageBounds).addTo(map);
// Define marker
var marker = L.marker([16.436673, 38.322712], {
                icon: L.mapbox.marker.icon({
                'marker-color': '#f86767'
    })
});
```

To make it easier to find the ocean underneath the overlaid image, a transparency slider is added. This slider is provided by MapBox and can be freely found in their documentation.

The final thing is to add the onClick functionality of adding a point when clicking on the map, and to be able to send it to the server as a GeoJSON. This was done as follows:

```
map.on('click', function(e) {
            marker.setLatLng(e.latlng).addTo(map);

            markercoords = marker.getLatLng()

            geoj = [{
                "type": "FeatureCollection",
                "features": [{
                    "type": "Feature",
                    "properties": {},
                    "geometry": {
                        "type": "Point",
                        "coordinates": [markercoords.lng,
                          markercoords.lat
                        ]
                    }
                }]
            }];
        });
```

We decided to add the possiblity for the user to choose between two different types of flooding at this stage. Either to do a simple flooding simulation, or a more advanced flood. As soon as the relevant button is clicked, an asynchronous request is sent to the relevant PyWPS service.

STEP 3 GET RESULTS: Because of the response document from the PyWPS service being the same for either call, the structure and functionality of the asynchronous request performed will be (mostly) the same for both requests. Therefore only one request will be described here. The buttons used to send the request have been marked with either an advancedflood or simpleflood ID. In this instance we will look at the advanced flood async request. We have stored the relevant PyWPS URL in a variable in our script, and we have replaced relevant input with the data uploaded by the user.

```
var preconpath = 'http://52.17.144.192/cgi-bin/pywps.cgi?request=
    execute&service=WPS&version=1.0.0&identifier=flooding&
    datainputs=[rasterin=http://52.17.144.192/static/images/' +
    imageuse + ';vectorin=' + JSON.stringify(geoj[0]) + ']';
```

The request we will perform is a standard jQuery async request to the PyWPS by using *$.get()*. This is encased in a $.when and a *$.done*function as follows: *$.when($.get()).done();*

This sets the script up in such a way that, when the async request is done, it launches another script. The $.when and async request code looks like the following:

```
$.when(
$.get(xmlPath, function(xml) {
        $("#overlay").hide();
        $(".step2").hide();
        $(".step3").show();
// Find the elements in the response XMl that contain our data
        $("wps\\:ProcessOutputs", xml).find("wps\\:Reference")
         .each(function(i, value) {
                  var imagepath = value.getAttribute("href");
                   var imagepathbroken = imagepath.split("/");
                  outputimagearray.push(imagepathbroken[5])
});
}, 'xml')
)
```

What this does is to get the XML document returned from PyWPS, traverse it and for each result, store it in a variable called outputimagearray. When this is done it takes the outputs stored in the mentioned array, and adds them to the website as an overlay to a map, and with possibility of downloading them.

```
.done( function() {
              $(".loading").append('<div class="map" id="map1"
                  style="height:          500px; width: 100%;">'
                  )
              var map = L.mapbox.map('map1', 'piratosthegreat.
                  i7aeacaf')
              .fitBounds(imageBounds);

              var outputimagesimple = 'static/outputs/' +
                  outputimagearray[1];

              var overlay = L.imageOverlay(outputimagesimple,
                                  imageBounds).addTo(map
                  );

              overlay.setOpacity(1.0);

             var featureLayer = L.mapbox.featureLayer()
             .loadURL('/static/outputs/' + outputimagearray[0])
             .addTo(map);

              $(".loading").append('<form method="get" action="
                  http://52.17.144.192/static/outputs/' +
                  outputimagearray[2] + '"><button class="btn
```

```
                btn-success btn-lg btn-block" type="submit">
                Download data !</button></form>')
            });
});
```

4.5.1 *Problems encountered*

CURRENTLY DOES NOT WORK ON CHROME: The way the URL's are formed, with variables added to them, do not work well with Chrome. This is an issue that can probably easily be fixed, but during this process there was no time to discover how to solve this issue.

IT IS NOT POSSIBLE TO OVERLAY A TIF: It is not possible to overlay a TIF onto the Webmap, and as such we had to also create a copy of the uploaded file, as a PNG. This does not take a long time, but makes it more tricky to make the entire process work the way we intend it to.

AT THE MOMENT FLOODING CAN ONLY OCCUR FROM ONE PLACE: The way the GeoJSON creation is set up, and the way that the marker gets added to the map, the flooding can only occur from one spot. This should be relatively easily remedied, but at the time of creation, it was deemed satisfactory that the flood could only occur from one location.

5 ANALYSIS

5.1 INCLUDED AND OMITTED FUNCTIONALITIES

On this section we will present which functions that have been created, are actually included in the website service and which are not due to various reasons. In the previous section, we describe the various phases of development of this project in detail. Despite the fact that all of the phases mentioned above have been fully developed, some functionalities have been unfortunately not included in the website application. This is mainly due to the limited amount of time at our disposal but also due to the amount of time required to include these functions in the application.

To begin with, we will briefly present what the application is capable of doing when used. When the user starts using the application, they upload an elevation model. Right after that, the application redirects them to a map with an overlaid image of the model they have uploaded. When that happens the user has to choose a point from where the flood will start. After doing that, they choose which type of flood they want to perform. The application then performs the chosen one and presents the results.

Now that we have stated what functions the application can perform, we will take a look on which functions have *not* been included. To begin with, the first function concerns the barrier placement. On phase 4, we discuss in great extent the development of adding a barrier as a capability of the service. We also present in depth the problems we faced during the development of that functionality. In fact, this sub-process is fully developed and fully functional. Unfortunately, we have not managed to include it on our web-application so far. The reason is that we simply did not have the time to create the interface and connect the script to the server in order to allow the user to use that function to its full extent.

To go on, another function that is fully developed, tested and ready to use is the flood modeling with cost distance. The application is currently using the method spatial intersection. We have mentioned the reasons why we think that the cost distance modeling might be a better solution to simulating the flood. On the other hand, the cost distance method takes considerably more time to complete the entire process, especially if the user decides to use the advanced flood

option. The changes that are needed in order to change from one method to the other are minimal, so if decided that the cost distance method is considerably better than the one currently in use, the switch to that method is fairly easy to make.
Finally, the final function that we created but unfortunately was not included in the application was the outlet point identification. This process was created early in the development phases in order to locate points in the landscape where a bottleneck is created when the flood advances. Unfortunately, that approach did not perform as expected and the points that were identified did not contribute to the mitigation of the flood. For that reason, we decided that such process would not add to the overall efficiency of the application and as a result was discarded from the project.

5.2 ARCHITECTURE

When the user visits the page, they have to provide a DEM, which has to fulfil the following criteria:

- Be a DEM
- Have tiff image extension
- Have WGS84 projection encoded

When this DEM is uploaded, and the source cell is selected, a request is sent to the server, by using PyWPS. PyWPS acts as a middleman, accepting the inputs from the user, initiating GRASS, and returning the results provided by GRASS encoded in a XML document. These results are then returned to the user through the browser, which are fetched and parsed using a combination of jQuery, JavaScript and HTML.

This overall workflow has been put into a diagram as can be seen on Figure 17

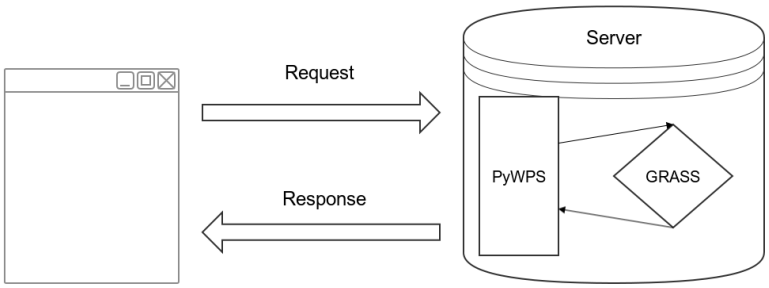


Figure 17: The overall working of the application

It should be noted however that, as mentioned in Chapter 4, we described the development of the functions implemented into the PyWPS service. As it turned out we created two different flooding models. One based on cost distance analysis, and one based on spatial selection. As mentioned in the problems encountered part of that section (Section 4.2.1), we started out by developing the spatial selection method. As there was not time to fully implement the cost distance analysis method into the application, it is currently deployed to use the spatial intersection method.

5.3 INSTALLATION STRUCTURE

Now that the application has been implemented, it is relevant to analyze how the entire installation is structured on the server. As mentioned in Chapter 4, a variety of software packages have been installed on the server, these being:

- PyWPS
- Flask
- Apache2
- GRASS

DEBUGGING: When the server has been installed, it runs on its own, and not much else has to be manipulated with, from then on. When the process fails for some reason, it can be relevant to troubleshoot the problem by reading the error log. Both Apache and PyWPS have error logs that can provide information about the situation, which can help to fix the error. The logs for both of these critical components can be found in the directories shown on Figure 18:

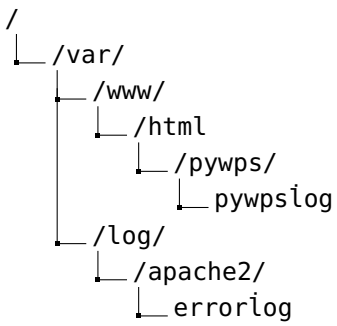


Figure 18: Location of the error logs.

PYWPS INSTALLATION: When wanting to change or modify some of the functionality of PyWPS, it is necessary to access the server and perform the changes there. The setup is displayed in Figure 19.

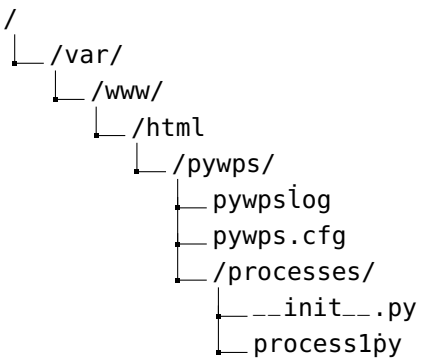


Figure 19: Structure of the PyWPS installation.

When adding or removing processes, it has to be done in the processes folder. The process is added (or removed) from the drive, and the __init__.py file is modified to reflect the change that has occurred in the drive.

A variety of serverside settings can be managed from the pywps.cfg file, for instance the maximum allowed filesize of a DEM used as input, or the location of PyWPS outputs.

FLASK INSTALLATION: Flask is installed in the directories shown on Figure 20.

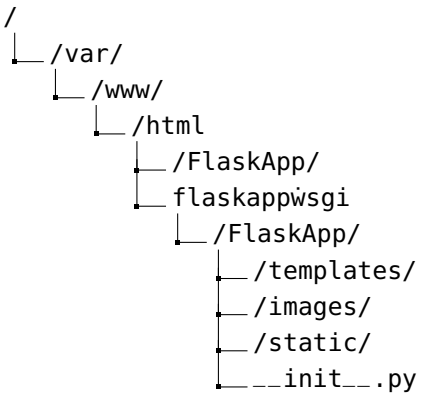


Figure 20: Installation structure of Flask

It is within the __init__.py file that most changes mentioned in the Implementation section have been performed. The templates folder contains the templates used to display HTML. The static folder contains a variety of static content, such as the jQuery and CSS libraries. A functionality of Flask is that content on the server that is not directly served to the user, is not meant to be accessible. But if it is placed within the static folder, and the URL to the content is known, it can be accessed from outside. The outputs of the PyWPS service are therefore directed to a subdirectory of the static folder, so that we

can easily access the data through the website front-end. The images folder is used to store the DEM the user uploads and works with, through the entire process.

GRASS INSTALLATION: The GRASS installation is fairly simple (and the structure can be seen in Figure 21), and after the initial setup is not touched even when adding new functionalities. The *grassdata* folder normally contains the various locations and mapsets that are used by the user. In our case the folder only contains one LOCATION, using the WGS84 geographical reference system. This will be used by the PyWPS process to work with the data in question. The .grassrc6 folder contains some settings that are needed for GRASS to function properly.

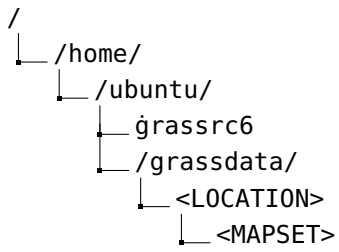


Figure 21: Installation folder structure of GRASS

5.4 USER EXPERIENCE

In this part of the report we will present how our web service works from the point of view of the user. What we will do, is try to demonstrate as detailed as possible the actual results of the phases we described on the previous chapter. We will also include a step-by-step guideline on how the service should be used and what actions are necessary for the user to follow in order to get the best results possible out of this process. But before we go in depth of the structure of the website, we will take a look at the overview of the setup, as seen on Figure 22

We begin by presenting the first page the user will see when they visit our website Figure 23. The url that leads to the homepage of the application is `http://52.17.144.192/`. The first page is then displayed, this provides information about the website such as its name and a basic description of what services it offers (Figure 24a.

Lower on the same page, we provide the user with detailed information about the application; The reason we have created it; who can use it; its most important advantages and features Figure 24b

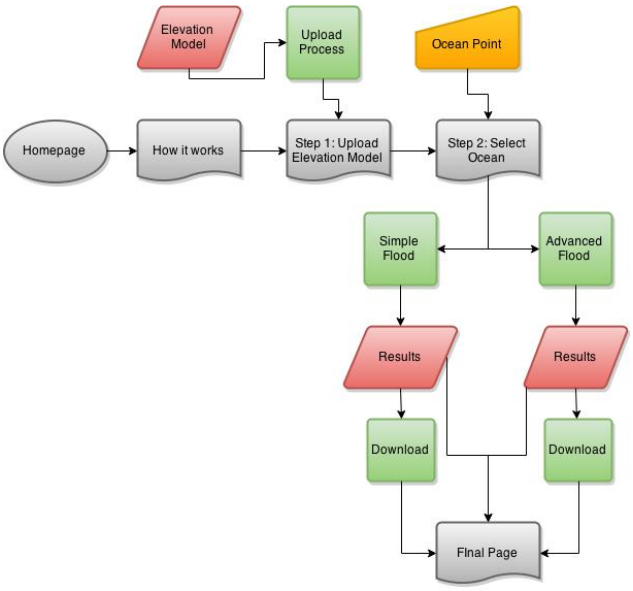
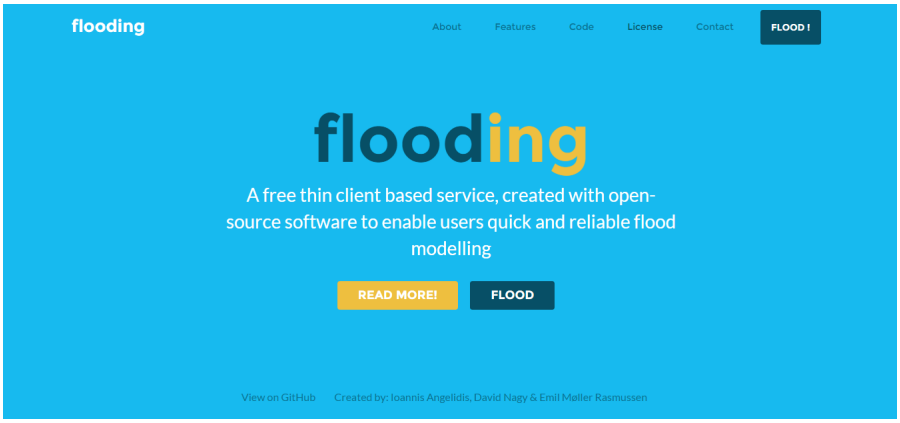
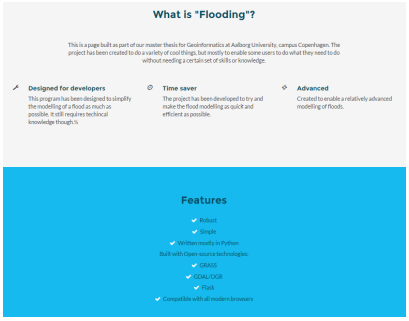


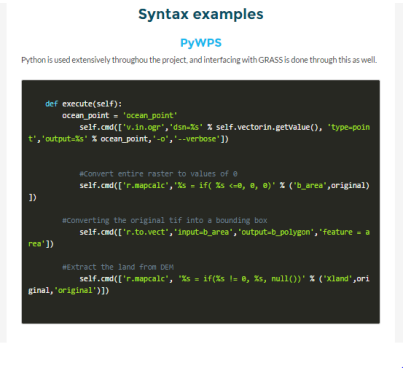
Figure 22: Overall workflow of the application



(a) Banner.



(b) Banner

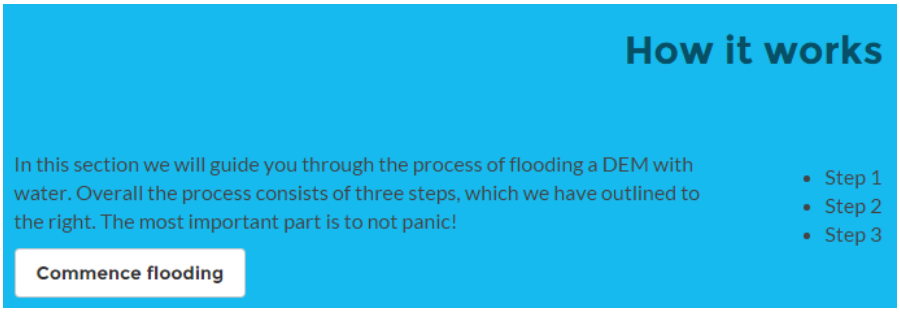


(c) Middle.

Figure 23: Frontpage of the web application, tentatively dubbed *flooding*.

On the final part of the page, we provide the user with small examples of the programming languages, frameworks and other tools we used on order to create this service (Figure 23c)

In order to proceed with the actual simulation the option FLOOD on the first part of the page has to be chosen. That will create a new tab on the browser that leads to the second part of the website (Figure 24). This part is a guided process of the flood simulation that follows. By clicking on commence flooding option, the initial part of the flood commences.



(a) Page describing the coming process.

(b) Uploading a DEM.

Figure 24: Accessing the possibility of uploading a DEM.

The process starts with requesting the user for the necessary input in order for the application to run (Figure 24b). As mentioned in previous parts of the report, that necessary input is an elevation model.

Once the user chooses an elevation model from his local hard drive and then uploads it, the second step is initialized (Figure 25).

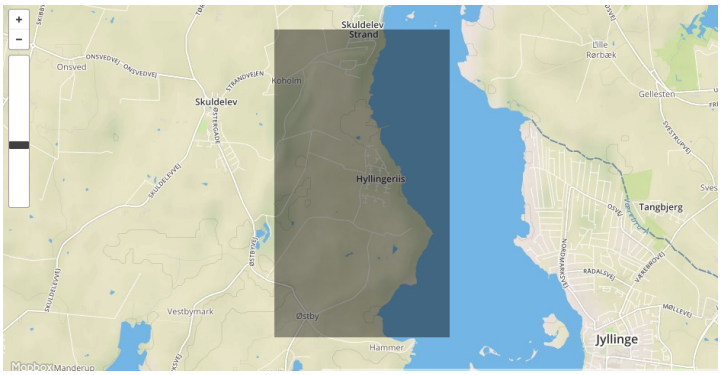


Figure 25: Step 2 display uploaded elevation model and insert point to designate origin of flood

Afterwards the user is required to provide it with the source of the flood. By simply clicking on the map, that point is created. Finally, the user then has to decide which type of process they wish to perform.

The simple flood option will instantly fill the elevation model with a predefined level of water and display the results on a map on the third step (Figure 26).

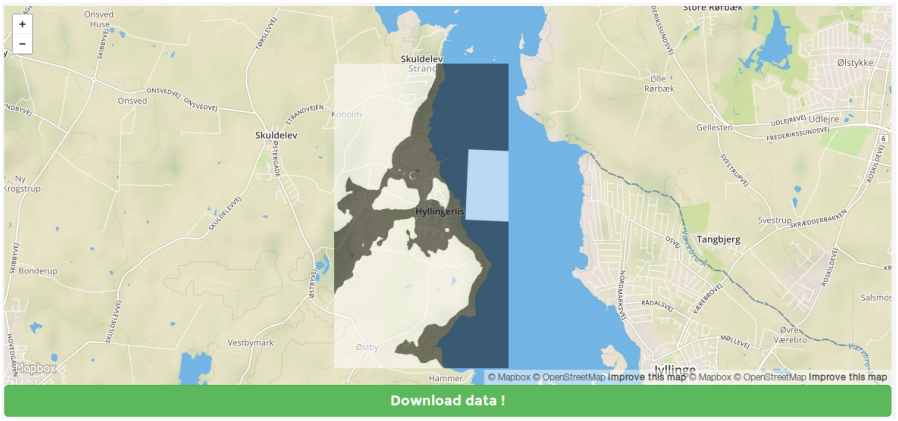


Figure 26: Results of the simple flood option

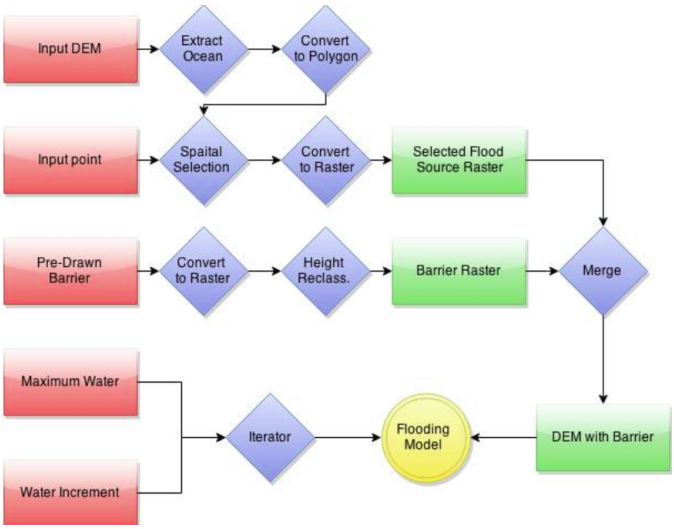
The advanced flood will return the same flood extent, with the pour points returned as vector points.

5.5 OLD VS NEW MODEL

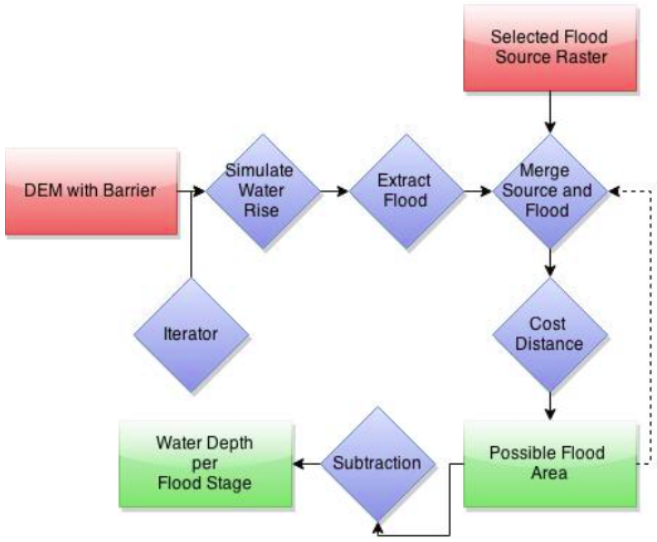
As discussed in Section 4.2, we have tried to simplify the model as much as possible, using only simple geoprocessing tools and map calculator. However here we would like to mention, since we used only ArcMap Model Builder in the old version, the coding part behind that model will not be taken into this comparison.

THE OLD WORKFLOW: The old model had been set up with two consecutive parts. An input part of the model, that specified the required input for the further modelling, such as input point for source selection and maximum flood level. As seen in Figure 27, the input part of the model, consists of a series of different modules. ArcGIS ModelBuilder is useful when constructing and executing simple workflows, and also has the functionality enabling it to export the created model to python script, however the python script conversion will disregard most of the environment setting and iteration functions. Therefore it is only useful as a preliminary decision making tool for developing a model.

Our ArcGIS model, however it was functioning perfectly, it was slightly harder to track down the errors, when the process executed with a lot of iteration. Also, the whole workflow was separated into two parts because you cannot iterate a part of a model, which means that another process needs to be involved in workflow, that part needs to incorporated into a separate model. Because of this, input variables for the flood modeling were defined in the first part of the workflow,



(a) Input part of the old model.



(b) Flooding iteration of the old model.

Figure 27: ArcGIS modelbuilder workflow of old model.

and were passed a second model containing the flooding modelling part.

THE NEW MODEL As described in Chapter 4 we simplified the model as much as possible, by taking out most of the unnecessary processes. During the development we have created a series of extensions for our flood model, however only the pour point analysis has been deployed to the server, due to the lack of time. As can be seen in Figure 28, the workflow consists of the barrier process, however it has not been included in the deployed script, but it has been developed and is working correctly.

If we are comparing the two workflows (old and new), it is clearly visible how much the simplification has impacted on the process. The iteration process is clearer and more straight forward. Also here we can mention, that our ArcGIS model was a very good model, however it is not suitable for distribution, since we do not have the right to publish the modules on an online service, as it can cause copyright issues.

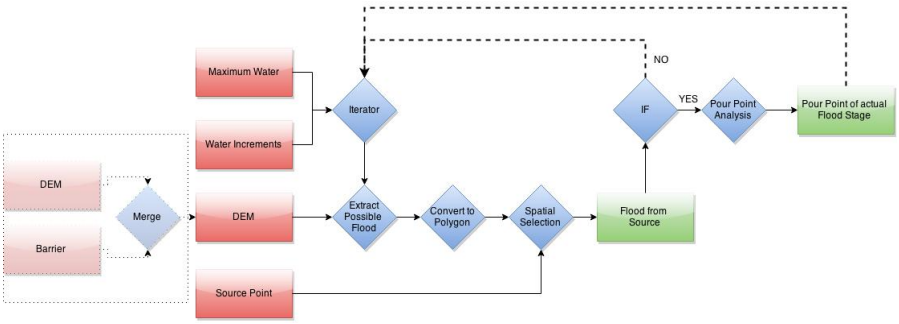


Figure 28: The new GRASS python model.

NEW MODEL VS RESULTS FROM COWI: As in our last project, we have compared our flood extent results with COWI data, who have created a delineation based on the flood aftermath of the storm Bodil in 2013. COWI calculated that the flood extent will appear as shown on Figure 29. This stage appears at level 165 cm, and it can be seen that the water accessed the residential area through from the North and furthermore a channel formed from the West, which can lead the accumulated water further.

Figure 29: COWI flood extent.

As described in Phase 2, we have created an approach for flood extent delineation with spatial selection and cost distance analysis.

(a) Cost distance model flooding. (b) Spatial selection model flooding.

Figure 30: Output of the GRASS modelling approaches.

Comparing our flood stages with the COWI flood, we realized that our calculated extent is the same, however the above presented extent appears at stage 174 cm with our Cost Distance model Figure 30a. The model with spatial selection (Figure 30b) could reach this stage at 175 cm, therefore we can state that the deviation between the two models is relatively low.

5.6 LIMITATIONS

In this section, we will discuss the existing limitations the users might face while using our application. These limitations are either placed by us, to ensure that our application works at an optimal level or exist purely because of the features and the parts that consist it. Regardless of the origins of these limitations, we think that it is our responsibility to let the user know about them and notify them before using the application, so that we can save them time while using it.

EXTENSIONS AND FILENAME: The first limitation we set is during the uploading of the input by the user. In an effort to apply some sort of control on what the user is capable of uploading to our server, we made sure that only certain file types are allowed for upload. The reason behind this specific action is that we do not want to allow irrelevant files on our server. Mainly because we want to make sure that the memory of the server is reserved for files that are actually compatible with the application but also install a first wall of security against malicious intents. That being said, the user can only upload *.tif* files. This file type check is performed in two steps. First, we declare the types that are allowed for upload.

```
ALLOWED_EXTENSIONS = set(['tif'])
```

Then we create a function that checks the extension of a file-to-be uploaded and if it is included in the ALLOWED EXTENSIONS, basically a tiff file, it returns a true value:

```
def allowed_file(filename):
        return '.' in filename and \
                        filename.rsplit('.',1)[1] in
                            ALLOWED_EXTENSIONS
```

Finally, we set a condition that checks the result of the function mentioned above and if it is true, then it proceeds with uploading the file, otherwise it will reload the upload page without any results

```
#check if the extension is allowed
if allowed_file(file.filename):
```

Having said that, the main reason we decided to use tiff as the only allowed extension the uploaded file can have is because tiff files contain necessary spatial information of the file. To be more precise, important information about the geographical reference system, the projection in use and the coordinates of the file's bounding box are all contained within tiff file. This is the main reason we set this file extension as the only one allowed and do not include for example ASCII files that need an additional file that holds some of the important geographical reference system information.

Another limitation we have implemented concerns the name of the file. This is because we want to make sure that our server is secure from malicious files that may be able to affect our server in any way through their filename.

NODATA VALUE: Since we have established what kind of limitations exist based on the file that the user wishes to upload, we will now examine how the file itself can obstruct the user from using our application properly. One obstacle that can result in a critical failure of the application exists when the user selects a point to designate water in the uploaded file. To be more specific, when the user uploads a file, that file is converted to *.png* in order to overlay it on the map and display it. That affects the way nodata cells are represented. Normally, when examining a *.tif* file, the cells that have null values are easily distinguished On the other hand, *.png* formatted files cannot display null valued cells as shown on Figure 31.

Since this is a case when displaying a *.png*, the user must be very careful when choosing the origin of the flood because if the coordinates of the point are within an area of null values, then the application will produce an error.

CELL VALUES FOR WATER: One of the core prerequisites our application that needs to be fulfilled is the designation of the water cells. Apart from the point that the user must provide to the application, the file that the user wants to investigate has to have cells that represent the sea. What we mean by that is that the representation of

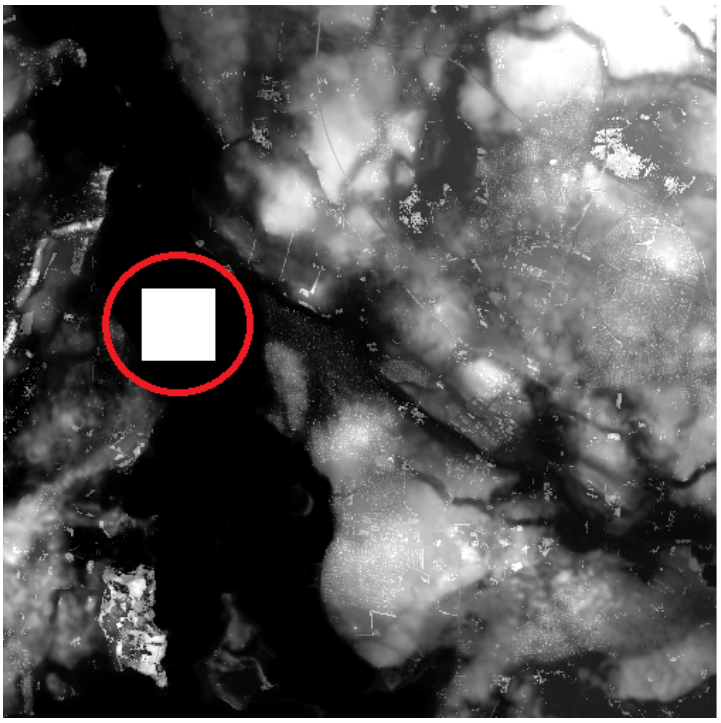


Figure 31: Image showing NoData values in tiff image.

the sea is not consistent across every elevation model worldwide. For example, the Danish DEM represents sea and water in general with zero values. Other elevation models do the same with negative values. Finally, some also do not use any kind of representation of the water resulting in them being shown as null cells for example NASA's Shuttle Radar Topography Mission (SRTM). As far as the negative values are concerned, it does not present as a problem because the way we have developed the application, all cells that are equal or lower than zero are considered as water. In the case where water is not represented, then the application will not work at all, in a similar manner with the nodata box described above.

5.7 AUDIENCE

Throughout this report we have mentioned in detail the characteristics of the application: its requirements to function properly, the decisions that need to be made in order for it to produce reliable results and the problematic parts of it that require attention in order to avoid erroneous results. Taking all of the above into consideration, we have concluded on a group of people that not only will find this application useful, but will also know how to use it to its full extent and potential.

Surely, the product of our work is not meant to be operated by everyone, even though that would be quite desirable. There is a limit on

which people can operate our product in an efficient manner and that fact comes from the various components that were needed in order to create such application. A lot of considerations that we have made are so in order to ensure that our application is available to the widest possible audience. Unfortunately, there are elements that for better or worse, limit that audience.

To begin with, we believe that the setup and interface of the application is as straightforward as possible. That ensure that it is easy to use and understand by anyone. As the user goes on in the application that changes. When we require the user to upload an image, then he/she needs to understand that the application performs at its fullest when provided with an elevation model that uses the WGS84 geographical reference system. That fact alone, requires some amount of knowledge in the fields of geography, surveying or cartography.

Another limiting factor is the actual input of the application. As presented above, the elevation model is the most important input the user has to provide. After the upload the user will then designate the source of the flood. As presented above, that approach requires great attention because an erroneous selection of the flood source can result in failure of the process. This entire process of designating the flood and making sure that the source point is within the extent of the raster requires a certain acquired experience with working with rasters.
Finally, another deciding factor that sets limitations on who can use this application is the hydrologic aspect of it. This aspect concerns the evolution of the flood event, its spread across the landscape as well as the management of the event. To be more specific, we believe that in order to use this tool at its maximum extent and produce reliable results and create efficient solutions, the user would probably need to have some experience with hydrology. That is because the tool uses a number of terms from that field in order to produce the results it does and understanding these terms and how they affect the results of the process is important to design viable solutions.

6 DISCUSSION

During the creation of the application, a variety of issues arose which were deemed fit to be covered in any of the previous chapters. These issues will therefore be addressed in this chapter.

6.1 DIGITAL ELEVATION MODELS

Reading through this report, it is clear that one of the most important elements of the tool created is the elevation model the user is providing. All the processes developed are performed on the elevation model and highly depend on it. That being said, we must underline that the quality of the results produced and their accuracy and reliability are greatly influenced by the accuracy and reliability of the elevation model. This means that if the user decides to use and upload an elevation model that is of poor resolution or accuracy, then the results that will be produced are also going to be of poor quality.

As far as the resolution is concerned, we believe that there has to be a limit on the cell size of the uploaded elevation model. Even though using a large cell sized elevation model e.g. 30m cell size, might speed up the total process considerably, we cannot expect the results of that process to be reliable enough in order to result into successful decision making. Of course we consider that the cell size should be relevant to the area that the user wants to check. For example, it would make sense if a 30m resolution elevation model was used on a nationwide simulation. But in the case of a small area of a few square kilometers then that resolution is considered poor. The reason we believe that, is because all of the core functions we use in the application are highly dependent on the resolution. To be more specific, the better the surface of an area is described by the elevation model, the better the results we can expect from the application.
The resolution also affects the mitigation of the flood. The barrier functionality we have created, (but not included) hardcodes the barrier into the elevation model. As a result, the barrier line inherits the cell size from the elevation model. In reality that means that if the elevation model has a cell size of 10 meters, then the barrier the user designs will have a width of 10 meters as well. That of course is not realistic and might affect the progress of the flood in an erroneous manner. For the reasons stated above we believe that the resolution of the elevation model is critical to the precision and accuracy of the results produced. On the other hand, we have no control on what the

user uploads as far as resolution is concerned. For that reason, we believe that the user should be wary of using a low resolution elevation model. In addition, we stress the fact that this application performs better when a small testing area is uploaded with the best possible resolution available. Finally, we think that areas with great elevation fluctuations should not be used in this application. That is because in order to actually produce results that actually would be realistic, a very high resolution elevation model that represents the surface in the best way possible would be necessary. Because of this, we believe that a DEM with a 10m resolution is the minimum requirement. Furthermore, we think that the application is best suited for areas with mild contours.

Mititigating some of these issues regarding DEM could be done by serving our own tileset of elevations, for instance by using GeoServer. By doing this we would ameliorate some of the issues mentioned above. This could be incorporated into the general workflow of our project by having a webmap showing these tiles at the "Upload DEM" step of the website. The user would be able to draw a square, indicating which area of the served tiles would be of interest. This selection would send a request to GeoServer which would return the DEM clipped to the user specified area. It would still be possible for a user to upload the DEM manually, but the standard option would be to select from our provided tiles. This would enable us to easily enforce our hypothesized minimum requirement of 10m resolution. Furthermore, doing this would bypass the stage at which the highest chance of error could occur; the user providing a DEM as tiff; the tiff having hardcoded WGS84 projection.

6.2 OPEN SOURCE

The entire project was based on the usage of open source software, and as such we experienced the full extent of the advantages and disadvantages of working with these softwares.

Unless the particular software is widely adopted, and has a lot of contributors, the support when using the software can be lacking to say the least. One of the areas where this is observed is in the case of documentation. If unexpected behaviour occurs, it can be hard to track down what the issue is. For this reason sites like StackExchange are vital but sometimes even these do not help you to the extent you need. This means that you have to search through developer mailinglists for an answer which might not even exist.

When using Open Source software, it cannot necessarily be expected that all the required functionality exists within the software used. As such it can be necessary to combine them in various ways

even software that has not been created for the purpose of working together which adds another layer where possible issues can occur.

Furthermore, a lot of software is created using Open Source Software instead of on Windows machines, and as such they can be more difficult to install, and may act more erratically on these platforms.

6.3 GRASS

Using this software, when used to using other GIS products, is not as intuitive as for instance QGIS is. This means that it has a certain amount of know-how if wanted to be used properly. So to mitigate this rather steep learning curve for others, we created this application.

GRASS has the advantage of having been developed a long time ago, and has a dedicated user group that works on it and makes sure that it is available and functioning properly.

GRASS does not support projecting on-the-fly, and as such has a different way of working with projections. Working with different coordinate reference systems in general does not work as it does in other softwares. Here the data has to be imported into a folder with their current projection hardcoded, and then pulled into a folder with the required CRS defined.

We decided early on in this project that, to accommodate the usage with webmaps, the DEM would have to be referenced to WGS 84, for it to work properly with our application. Enabling a more dynamic approach to working with images in other coordinate reference systems has been created, but we have not had the time to implement it or test it - properly into our application.

When creating a project in GRASS it is necessary to first set up a LOCATION then afterwards set up MAPSETS. This is not very intuitive when someone is used to working with software such as QGIS or ArcGIS.

INSTALLATION: Installing GRASS was not hard, on either Linux or Windows, but when accessing the functionality externally, it turned out to be a challenge.

MODULES: There are numerous modules that are officially hosted by GRASS, but there is also the possibility of adding external functions. These are not officially hosted by GRASS, and therefore not quality-tested yet, but in total around 350 modules can be accessed through the software. This means that when creating an application such as ours, the potential for expanding it is huge. On the other hand, it can quickly become hard to have an overview, and when using the non-standard functions one has to be wary.

Sometimes the functionality of modules used in other softwares, do not work the way it is expected to. An example from our pro-

cess of this, is the rasterization of the barrier lines. This behavior can be problematic, as it makes the transitioning from one software to another unexpected, and can be a hindrance in the adoption, or the spread of the software.

USING GRASS WITHOUT STARTING IT EXPLICITLY: By accessing the functions of GRASS through scripting, we were hoping to easily be able to create the necessary workflow, so that the functions could be created and tested early. It was quickly discovered that doing this would not be as easy as hoped for.

Making this work on Windows turned out to be an even bigger issue, especially seeing as some binaries would have to be installed, to get it to work properly through the command line. Making it work on Ubuntu was significantly easier, but setting up the previously mentioned environmental variables was tricky on both systems.

Furthermore, using the actual functionality was also a bit different from the way it is called from within the actual GRASS GUI. Having to call gscript.run_command() for every function was another step, where an error could occur. To give a practical example of this, when setting the flags that can be activated for most GRASS functions, it was not immediately clear if they were set in a similar manner as they would be through command line, or if a setting had to be specified. As it turned out a special flags setting had to be specified. These options were not immediately obvious, and could take several hours of research to actually find the answer.

All of these issues meant that it actually took several weeks before we had set up the proper environment, and were completely comfortable with the process of accessing the GRASS functionality from outside of the software.

SEEING THE RESULTS: When wanting to actually validate the results created by our script, we stumbled upon the problem of the interface. When used to working with a streamlined Drag-N-Drop interface, the GRASS interface leaves a lot to be desired.

6.4 PYWPS

We use PyWPS to initiate and run the relevant processes for GRASS through the browser.

INSTALLATION: When installing PyWPS, settings have to be changed and read/write access has to be managed for a variety of relevant folders and files. If one of these settings does not get set up properly, the software might not work at all. This can be an issue, if you do not have a lot of experience setting up such an environment, it will add to the complexity and a simple unfamiliar operation might re-

sult in abandoning the usage of PyWPS. The pywps installation was straightforward, but there are still some files that have to be added manually. Furthermore, if someone does not want to use the predefined locations for the files, they have to know where to change a certain reference, or else the software will not function.

DEBUGGING: Furthermore, it could be relevant to know where error messages get displayed, as it can be hard to troubleshoot the issue otherwise. For instance, some errors get logged in the Apache error log, whilst others get sent to the manually set up pywps log. Knowing where these are, how to access them, and what errors can be expected to find from both of these, is critical if the set up of an advanced processing service is to be successful.

CONVERSION: The syntax for working with GRASS through PyWPS or through a Python script are different. This creates the issue of having to rewrite parts of the local script, to make it work on the server. As mentioned in the previous part about this, some options could be hard to figure out how to specify for instance setting the relevant flags for a command.

DOCUMENTATION: PyWPS suffers from some very basic and not all-too-helpful documentation. This makes it hard to troubleshoot issues

EXECUTING: When requesting a WPS service from the server, a URL is used. Setting the various relevant parameters here is critical, as just a minor error will make the request fail.

RESPONSE: When getting data returned from the process, an XML document is returned. Data can either be set to be returned as a reference or as the actual data. In the beginning the data was set to be returned as the actual data, this means that an image gets sent back in Base64 format. This might be okay for small images, but for large tiffs, this can actually crash the browser trying to read the document. The function to set it as a reference is not obvious from the beginning, but it is essential for working with projects such as this.

COMPATIBILITY: A critical issue that was encountered with using PyWPS was the fact that it did not support manipulating vectors when using GRASS version 7.x. This could have proven fatal to the entire project, as it was not stated anywhere, but was luckily mitigated by downgrading to GRASS version 6.x. Luckily this downgrade did not prove to have any influence on the rest of the project's workflow, but could have been a critical issue if some of the functions we were dependent on did not work, or did not exist, in the 6.x version of

GRASS. Some functions have different names depending on the version of GRASS being used, and this also meant that we had to make sure that the version, or the inputs, were named the same. An example of this is the fact that inputs in grass 7.x for a lot of functions are called input, whilst in GRASS 6.x they are called dsn. These are some of the small changes that make the transition between these two versions unpredictable.

OGC STANDARD: The PyWPS version used in this software lives up to most of the specifications of the OGC standard, but is actually missing a functionality that provides the user with a URL from which they can check how far along the process is. This is obviously not a critical issue, but would have been informative when the user starts the advanced flooding, as it takes more than twenty minutes to complete. The team behind the PyWPS version used in this project are actually working on a new version, in which this functionality will be enabled.

DEPLOYMENT: In general, the testing of new functionality turned out to be time consuming. As the advanced flood model takes more than twenty minutes to run, it would mean that any time something got changed, the process would have to be re-uploaded, and the service reinitialized. This meant that testing minor changes could take more than an hour per expected change.

DYNAMIC OUTPUT PROBLEMS: As the data outputs are predefined, it is necessary to have a good idea of what you want to have exported. What this means is that when exporting the pour points, a function has to be created that merges all of the pour points instead of just assigning them as output as they get created. This means that it is essential to have a good overview of how the created process works.

6.5 WEBSITE INTEGRATION

Having been using Python for a significant part of the project, it was hard to adjust to the different syntax required by JavaScript and jQuery used on the website.

The asynchronous request is an essential part of the project, but if someone does not have a lot of experience with setting up such requests, it can be hard to know exactly how to make the script, wait until the request is fully done, until it goes on to manipulate the results.

Furthermore, parsing the XML response document and finding the nodes containing the data from the document, turned out to be harder than expected.

Using Flask made the setup significantly easier as the upload and extraction of coordinates from the uploaded DEM is handled easily by calling on GDAL functionality through this. Setting up templates and swapping content based on the user's behaviour is a good way of creating easy to set-up and fully functional websites.

In order to offer the users with a more professional approach, we have considered in creating a feature of user accounts. The idea behind that comes from the fact that the application might have to deal with multiple users using it simultaneously. We have already stated that multiple uploads with the same name is an issue the application has, but using user accounts will change that. By creating folders based on a pre-designated user id, we can store each uploaded file to the respective folder and as a result organize the structure of the server in a more efficient manner.

6.6 EXTRA FUNCTIONALITIES

Several other modules have been created, but have not been implemented into the functionality of the application yet. The reason for this is because the overall set-up of inserting these functions, requires a lot of time making sure all the elements actually work together. Towards the end of this project, time was limited, thus preventing us from making sure the elements worked together properly.

The creation of new functionality has become significantly easier for all the members of the group, after having worked on the application for close to four months. Obviously, as mentioned above, it still takes time to make sure it works 100 percent, but the general implementation time has gone down from days to merely hours.

COST DISTANCE AND SPATIAL SELECTION: One of the major issues we stumbled upon in this project was whether to use either Cost Distance Analysis or the Spatial Selection method for flooding the landscape. The problem here is that the results from the two processes differ significantly from each other, and it could have real life consequences if they were to be used in the real world. On the one hand, the intersection method is significantly faster than the Cost Distance method, but the Cost Distance method what we have tested. It could be said that they differ in the way that they estimate the water in an area. The Cost Distance is more pessimistic in its approach, flooding larger tracts of land whilst the Spatial Selection method is a more optimistic approach, which floods less amounts of land.

Probably the different methods should be incorporated into an even larger workflow, where there are a significantly larger amount of methods to choose from, which might look some thing like: SIMPLE FLOODS: Quick(Spatial Selection) and slower(Cost Distance) and ADVANCED FLOOD: Quick(Spatial Selection) and slower(Cost Distance).

But in the end, it all depends on a significantly more thorough testing of the two methods.

As mentioned previously in this report, the outlet points we started to try and identify, were not very useful, as they gave information that would be obvious from looking at the map. This illustrates the issue of starting to develop functionality that is expected to be helpful in a certain way, but turns out it is not. This was a necessary path to take, as it pointed us towards the actual solution of our pour point analysis method.

6.7 REAL-LIFE

Even though the project as such was not created with the intent of being applicable in real-world situations, and more of a way of exploring how advanced geographic analysis could be made easily available, there are some obvious situations and usages it could be found to have in real life.

People working in Disaster Management might find good use for the software. The application can be used in three of the four stages of a disaster, as defined by the disaster cycle.

During the prevention/mitigation phase the possibility of using the application is apparent. When there is good time it could be argued that more in-depth modelling should be used, but if this is not possible for the relevant agency either because of a lack-of-money, or a lack of expertise, this application would provide adequate functionality.

During the preparedness phase, the application's advanced functionality could be used to prepare the local disaster management agents on which areas they have to be especially wary of.

During the response is where the application would fit the best. Being able to quickly remodel the spread of the water, dependant on the currently projected water level and area affected, would provide the involved agents with a semi-dynamic and up-to-date response background.

By adding functionality, such as the calculation of expected amounts of water in the critical areas, or the area in general, would provide an excellent way for this project to become relevant in the recovery phase as well.

Adding more functionality, dependant on the situation needing a response, would provide disaster managers with a tool that could work fast and increase the likelihood of saving both human life and property, through the expeditious management of the response to a disaster.

6.8 METHODOLOGY

The overall schedule turned out to be appropriate for the various modules. Having defined each part as having a weeks overlap with the previous phase, turned out to be accurate.

Founding the project in Denmark was a good step, as having to make sure the program would work anywhere in the world would have been complicated. The program will actually work anywhere, as long as the DEM is projected into WGS84 and is uploaded as a TIF.

Using it in areas, such as coastal areas, with high escarpment are problematic as the maximum flood level is hardcoded to be 300 centimetres. To successfully use it other areas, more research has to done, in order to apply the appropriate flood heights for the project.

Using a code repository such as GitHub is a good way of making sure, that all the group members are up to date. Furthermore, it creates an easy way to restore the code to an earlier stage, if for some reason it has become corrupted by some unknown error, and it also keeps track of who has performed which changes. Additionally, having continuously updated the code throughout the process, and for instance noting the procedure involved in installing PyWPS, we have created, a usable manual in the installation of the service for others, provide working examples and a working application skeleton that can easily be accessed, downloaded and changed for other purposes.

6.9 USAGE ON DIFFERENT COMPUTERS

As the development computers used for the creation of this software, have not been bought with the original intent of using or developing a specific piece of software on them, it can create some compatibility issues. For various reasons a library might not be able to be installed on one computer, but work perfectly fine on the other one. An example of this is the installation of GDAL and OGR on a windows machine. One of the team members was running windows, and the GDAL/OGR installation, if wanted to be used with Python, is dependant on some packages, that have to be installed separately on this platform. This turned out to be very problematic, and for a long time it made the script unusable on this computer.

Furthermore, the scripts expect that all the relevant packages are installed into the same relative paths, or else it will not work when trying it on other machines.

6.10 SERVER

Setting up an Amazon server is very easy, and can be done with a few clicks. The advantages of using Amazon is the fact that you can

expect a high amount of uptime and support if needed. Furthermore, the capability of scaling the server hardware based on the current needs, or expected loads, is useful for a CPU intensive application such as this. This means that in periods where a high server load is expected, the server power can be turned up.

6.11 POUR POINTS

The flooding area that defines how the pour points get distributed in the landscape is currently not set up to be dynamic, but has a hard-coded value indicating what size of area we consider to be critical. The actual size of a critical area should be calculated based on the size and resolution of the used DEM. Actually the capability for doing this has been developed, but when porting it to the web it turned out that this functionality had some issues in PyWPS, and there was no time to troubleshoot it properly.. The way it has been set-up now, it can theoretically be possible to produce any pour points. The functionality has not been tested to see what happens in such situation, but there is a high likelihood that the process would break down and that there would not be any output to the user. It should be noted however that this situation is highly unlikely, and would require very specific conditions from the DEM. The only situation where it could plausibly occur is a situation where the DEM contains a very narrow corridor, with a very steep upwards going slope.

6.12 SWOT

6.12.1 *Strengths*

OPEN SOURCE: Before starting the development of the project, we decided as a group to focus on creating an application using exclusively open source software. The idea behind that decision was that we wanted to create a tool that could be easily available to a great number of people without having to worry about copyright issues. In fact, the idea is that we want to create an application that can help in preventing natural disasters and thus affect a lot of people, and at the same time be freely distributable to any interested party without any limitations. We strongly believe that having achieved that is a great advantage to the application and the overall work we have put in this project. Anyone who is interested in performing flood modeling in any given area around the world can easily access the webpage we have created and execute the application.

EXTENSIBLE: When we started developing the application and decided on the tools we would use, we spent a considerable amount of time trying to familiarize ourselves with the tools we would ul-

timately use. Once we reached a satisfying level of familiarity we discovered that it is actually not quite difficult to expand on the functions that we have created or planned to create. That fact allows us to greatly expand the capabilities of the application and optimize the ones already existing. Keeping in mind that creating functions that already exist in the application took us very little time to complete towards the end of the project, one can conclude that extending these functions and creating new ones will be quite less time consuming than originally expected.

EASY-TO-USE: As far as the usability and the user-friendliness of the application, we believe that we created it with the notion of keeping it as simple as possible. We started developing the service with that idea in mind and we believe that we have achieved it. Firstly, the application needs minimal input to run. Surely, someone needs to have a specific type of elevation model and use the correct CRS which adds a certain complexity to the use, but other than that, the user is not required to perform any other technical manipulation. This fact is quite important as it encourages the user to utilize this application without forcing them to go through multiple demanding tasks in order to achieve what they were aiming for.

SCALABLE: A major strength of this project is the fact that it is scalable. By using the Amazon EC2 server architecture, we have been able to, relatively, easily create a proof-of-concept application deployed on the web. If this application would ever be deployed for production purposes, scaling the server to meet increased usage would be very easy. The scalability can furthermore be dependant on an as-needed basis, meaning that it arbitrarily can be scaled up or down, by the click of a button from the Amazon EC2 dashboard.

SECURE: In any software development project, security is always an important issue that needs addressing. In this case, apart from the Secure Filename restriction that is a function built-in the Flask module, we have not taken any further action towards making this application more secure. Despite that fact, we believe that the security provided from using an Amazon server is more than sufficient. The purpose of the service also strengthens this belief as we think that the information we are handling is not important enough that we should dedicate more time developing that specific aspect of the application.

6.12.2 *Weaknesses*

BROWSER DEPENDENT: Towards the end of this project, we performed testing on the application in order to observe how it performs under different circumstances. Through that process we noted that

the service's functionality is highly dependent on the browser the user runs it through. To be more specific, in the case where the user uses the application through Chrome browser, then the application produces an error when uploading the elevation model. This is a limitation that can be solved but in the time frame of this project, we decided it was not of critical importance to allocate time in order to solve it.

TECHNICALLY DEMANDING: Even though the application has been created to be as usable as possible, it is still not set up in a way that would enable the people not used to work with GIS to use it. As mentioned, it is still necessary to use a TIF with the WGS 84 coordinate reference system. Furthermore, the fact that the user has to provide the DEM makes it even harder for a layman to get started with the software. These facts mean that a certain amount of technical knowledge is needed to successfully run the application.

PROOF-OF-CONCEPT: The application has been created as a proof-of-concept. As such the functionality has not been fully developed, and all situations have not been tested for. This is a weakness, as the application only works in some predetermined, and specific situations that we the developers know about. Because of this, feedback to the user when stumbling on an error, is not provided, and they are left with a page, and no feedback as to why the application stopped working. This means that there can be a significant amount of unknown behaviour from the website, that will have to be thoroughly tested before it could be officially distributed.

OPEN SOURCE: At the first stages of the development of the project we spent a great deal of our available time trying to set up the various tools we decided to use. In addition, while trying to determine what is the best course of action in order to address various development problems, we came to realize that documentation of the open source modules we used was quite lacking. This is surely something that was not predicted beforehand. Since tools such as GRASS are created to be used freely and not to turn a profit to its creators, it seems reasonable to expect that the documentation of the tools offered is not extensive. That is a significant problem when trying to develop new functionalities or expand existing ones. It not only increases the amount of work someone needs to dedicate in order to develop a function but at the same time it is frustrating when trying to understand why a certain function does not work and not being able to find the reason or a solution from the developers or the community of that software.

6.12.3 *Opportunities*

DIFFERENT SCENARIO SIMULATION: The algorithm, which is deployed on the amazon server is supported by PyWPS , this can be substitutable with other processes, which could be supported by GIS geoprocessing. This means, on our server we can deploy different GIS analysis processes, regardless their nature and their purpose.

EXTERNAL DEVELOPING: By making sure that all the code and script are freely available to any interested party, we want to think that we encourage external feedback on the way we implemented the various aspects of our project. Asides from that, we hope that this process and its elements might reach out to other GI developers that might be willing to take our work further or even use it to develop another project. We are welcoming such opportunities and hope that we might receive feedback, advice or propositions from other specialists of the field, in order to expand, optimize and improve this current application or observe how parts of it can be implemented on other projects.

6.12.4 *Threats*

LACK OF SUPPORT BREAKING FUNCTIONALITY: When using a multitude of different software in conjunction with each other - where some are not necessarily created to be interoperable - an update of one of the parts can accidentally cause the disruption of the functionality of some other part in the software chain. This is a major threat to our software, as troubleshooting it could be impossible, which would make the application inoperable. In the development of our application, we had a situation where this situation occurred. As mentioned in the report, we had issues with using GRASS 7 vector functions through PyWPS. Luckily we solved the issue by downgrading to GRASS 6, but this could have had critical implications for our application if it was not fixed.

7

PERSPECTIVES

In this part of the report we will present what we think might be good additions to the application, that can be on a short term. These additions will offer a more all-around flood modelling service. Additionally this section will contain items of technical and non-technical nature, but overall they include expansion to the application we have described in the sections before.

BARRIER: These functionalities include first and foremost the design of a barrier in the landscape. We presented beforehand the work we dedicated on that aspect of the application. What is missing from this extension is its connection to the core script. In addition, our plan includes the ability to define the height of the barrier. We believe that this extension can be easily added since the greatest part of the work needed is already completed. By adding the barrier feature, we will be considerably closer to a fully converted and working version of the older project but also offer to the user a more completed approach to the disaster management aspect of this project.

USER INPUT: The deployed functionalities have a hard coded maximum flood level input, which is set to 3 meters based on the danish flood history. However, for better usability, we would like to allow the user to interactively change which flood level extent they would like to simulate. That way we ensure our application is expanded to include the flood event the user wants to simulate. This would give a certain flexibility to the service and enable the user to adjust the service to their needs.

MULTI-CRITERIA DECISION ANALYSIS: At the beginning of this project we have set up our goals, regarding what this project can be implemented for. Initially we wanted to create a useful application for disaster management purposes, including more thorough decision making features. For instance, after the flood modeling, the user could get some output based on existing infrastructure, population density as well as the land use of the investigated area. In order to show this information, it can be done by Multi Criteria Decision Analysis(MCDA). MCDA can be a very valuable tool that can be applied for complex decision making based upon different flood protection scenarios. Setting up weights for the previously mentioned features, can be critical for flood management purposes. Likewise, establishing danger zones or high importance zones based on the area's im-

portance, it helps users to think, re-think, adjust, test, and decide on a final scenario, which can mitigate the impact of a natural disaster. Using this feature, the users can benefit from highlighting the critical areas. To give an example, we have created some possible output, which an MCDA function could provide to the user.

However to develop and deploy such functionality, it requires more user input, such as census data, network of infrastructure or land use maps. This input can also be invoked from different online services, such as GeoDanmark. Of course it would need further adjustment and programming in order to work with our already existing service, but the time limitation of this project could not allow us to start this implementation.

PROJECTIONS: As mentioned previously, we have created the functionality that will enable GRASS to reproject any DEM into a new coordinate system, thus eliminating the geographical reference system limitation. To create an entirely dynamic application, with the capability of handling any DEM, will require a lot of work on the Flask back-end, to make sure that the DEM contains a projection and it gets reprojected to the right type.

The problem with using WGS84 is that data precision is lost when using very localized data. For this reason it would be critical to support the importing of any coordinate reference system, if this software would ever be launched for production.

USER ACCOUNTS: In order to offer the users with a more professional approach, we have considered in creating a feature of user accounts. The idea behind that comes from the fact that the application might have to deal with multiple users using it simultaneously. We have already stated that multiple uploads with the same name is an issue the application has, but using user accounts will change that. By creating folders based on a pre-designated user id, we can store each uploaded file to the respective folder and as a result organize the structure of the server in a more efficient manner.

SERVICE UPGRADE: To begin with, the most realistic and easy to achieve goal is expanding the capabilities of our server. This mainly focuses on being able to handle larger input data in order to provide the user with more accurate results. This also includes increasing the processing power of the server, thus making the application even faster especially when large elevation models need processing. Promotion Another goal is to make this application available to the public. Even though anyone can access the service even now, we would like to optimize and polish the overall outlook of it. Then making it known to the public that such application is available for free use

would be a logical step in the direction that the group desires to take this project to.

PROFESSIONAL FEEDBACK: From the beginning of the project, we have been discussing that it would be of great value for the project, if we could have feedback from other people on what do they think about the application. Quite often, especially when spending a great amount of time developing the application, we tend to make decisions or deal with issues with a very inflexible point of view. This "tunnel vision" might keep the group from discovering obvious limitations or faults while using the service. For that reason, we strongly believe that it would be essential to have other people using our application and providing us with information about the experiences they had through using it. That way, we will limit the drawbacks and locate faulty development of the application. The idea is to find as many people as possible, coming from different backgrounds, so that their viewpoint would be as diverse as possible. We would then examine their feedback and based on that, decide which changes we need to make in order to enhance the application.

EXTERNAL DEVELOPING: By making sure that all the code and script are freely available to any interested party, we want to think that we encourage external feedback on the way we implemented the various aspects of our project. Asides from that, we hope that this process and its elements might reach out to other GI developers that might be willing to take our work further or even use it to develop another project. We are welcoming such opportunities and hope that we might receive feedback, advice or propositions from other specialists of the field, in order to expand, optimize and improve this current application or observe how parts of it can be implemented on other projects.

WIDER APPLICABILITY: Now that the general shell of the project has been created, it would actually be relatively simple to replace the functionality with other geographic analysis. This would mean that the flooding is an arbitrary choice, and the shell quickly could be reused by us, or others, to create services that could perform analysis of various kinds.

Obviously, the learning-curve with other applications or methods could be an issue, as this has had a very specific focus. As mentioned it would be easy to apply this method to other phenomena, with the learning curve of understanding the actual problem being the issue, and not learning the technology.

8 CONCLUSION

The purpose of this project was to create an online application capable of performing simple flood modeling using open source technologies. In addition to that, we wanted to provide any interested parties with thorough documentation of the technologies used, how they are combined and how they interact with each other. We believe that we have achieved these tasks.

Working with open source technologies, has been an interesting experience for the development group, as they have never used open source technologies to such a degree as done during this project. During the process we encountered a series of obstacles which we either overcame or worked around, but surely this could be expected when basing an entire project on such software.

- In spite of this, we have successfully combined PyWPS, GRASS, Flask and web development languages, into a fully operating application. Even though some changes have been incorporated in the tools we are using, we think that the tools successfully fulfill the expected requirements.
- By simplifying the interface and the inputs required from the user, we believe that the application has a layer of complexity removed, that normally would discourage the usage of such an analysis.
- Having thoroughly recorded the development phase, through the extensive implementation chapter, and also using the code repository GitHub, we believe that we have successfully created usable documentation. Keeping the open source aspect in mind, this documentation is freely available to any interested party. All in all, we have fulfilled the goals specified in our problem statement, and created an application that fulfills our initial ambitions. We have thoroughly documented our ideas for future development and as well as problems encountered, in the hopes that it can benefit other GI professionals, and they can learn from our experiences.

BIBLIOGRAPHY

Antony Awaida and James Westervelt. r.cost. http://grass.osgeo.org/grass64/manuals/r.cost.html, 2013. Accessed: 2015-05-05.

Jachym Cepicky and Lorenzo Becchi. Geospatial processing via internet on remote servers. *OSGeo Journal*, 1:1 – 4, 2007.

DMI. Stormfloder i fremtidens klima. http://www.dmi.dk/laer-om/temaer/hav/fremtidens-vandstand/stormfloder-i-fremtidens-klima/, 2015. Accessed: 2015-05-05.

Charles Echlschlaeger. r.watershed. http://grass.osgeo.org/grass71/manuals/r.watershed.html, 2015. Accessed: 2015-05-05.

ESRI. Cost distance (spatial analyst). http://help.arcgis.com/en/arcgisdesktop/10.0/help/index.html#//009z00000018000000.htm, a. Accessed: 2015-05-05.

ESRI. How watershed works. http://resources.arcgis.com/en/help/main/10.1/index.html#//009z00000068000000, b. Accessed: 2015-05-05.

GRASS Development Team. Working with grass without starting it explicitly. http://grasswiki.osgeo.org/wiki/Working_with_GRASS_without_starting_it_explicitly, 2015. Accessed: 2015-05-05.

V. Grassmuck. Freie software zwischen privat- und gemeineigentum. *Budenszentrale für Bildung*, 2004.

Miguel Grinberg. *Flask Web Development*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st edition, 2014.

HTML.net. Lesson 2: What is html? http://html.net/tutorials/html/lesson2.php, 2015. Accessed: 2015-05-05.

Sina Khatami and Bahram Khazaei. Benefits of gis application in hydrological modelling: A brief summary. *VATTEN - Journal of Water Management and Research*, 70:41–49, 2014.

Donald E. Knuth. Computer Programming as an Art. *Communications of the ACM*, 17(12):667–673, December 1974.

Kystdirektoratet. Forslag til udpegning af risikoområder på baggrund af en foreløbig vurdering af oversvømmelsesrisikoen fra havet, fjorde eller andre dele af søterritoriet. Technical report, Kystdirektoratet, 2011.

Andrew M. St. Laurent. *Understanding Open Source and Free Software Licensing*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 1st edition, 2004.

X Liu. Airborne lidar for dem generation: Some critical issues. *Progress in Physical Geography*, pages 31–49, 2008.

Mark Lutz. *Python*. O'Reilly, 1005 Gravenstein Highway North, Sebastopol, CA 95472, 5th edition, 2013.

Markus Neteler and Helena Mitasova. *Open Source GIS: A GRASS GIS Approach*. Springer Science + Business Media Inc., Boston, MA, USA, 2nd edition, 2005.

Markus Neteler, M. Hamish Bowman, Martin Landa, and Markus Metz. Grass gis: A multi-purpose open source gis. *Environmental Modelling Software*, 31:124 – 130, 2012.

Mozilla Developer Network. What is javascript? https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Introduction#What_is_JavaScript.3F, 2015. Accessed: 2015-05-05.

Open Geospatial Consortium. Ogc® wps 2.0 interface standard. http://www.opengeospatial.org/standards/wps, 2015. Accessed: 2015-05-05.

Python Software Foundation. Should i use python 2 or python 3 for my development activity? http://wiki.python.org/moin/Python2orPython3, 2015. Accessed: 2015-05-05.

PyWPS Development Team. Python web processing service. http://pywps.wald.intevation.org, 2009. Accessed: 2015-05-05.

Armin Ronacher. Flask. http://flask.pocoo.org/, 2014. Accessed: 2015-05-05.

Birgitte Rosenkrantz and Poul Frederiksen. Quality assessment of the danish elevation model (dk-dem). Technical report, National Survey and Cadastre - Denmark, 2011.

Stefan Steininger and Andrew J. S. Hunter. 2012 free and open source gis software map - a guide to facilitate research, development and adoption. *Computer Environment and Urban Systems*, 2012.

Laura Toma, Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, Jeffrey Scott Vitter, Patrick N. Halpin, and Dean Urban. Flow computation on massive grid. *Proc. ACM Symposium on Advances in Geographic Information Systems*, 2001a.

Laura Toma, Rajiv Wickremesinghe, Lars Arge, Jeffrey S. Chase, Jeffrey Scott Vitter, Patrick N. Halpin, and Dean Urban. Flow computation on massive grid terrains. *GeoInformatica, International Journal on Advances of Computer Science for Geographic Information Systems*, 2001b.

W3Schools. Html introduction. http://www.w3schools.com/html/html_intro.asp, 2015. Accessed: 2015-05-05.