MASTER THESIS

GIANNIS ANGELIDIS & DAVID NAGY & EMIL MØLLER RASMUSSEN

Creating a thin client based flood simulation tool based on open source software

June 2015 – version 1.0

ABSTRACT

Short summary of the contents...

We have seen that computer programming is an art,
because it applies accumulated knowledge to the world,
because it requires skill and ingenuity, and especially
because it produces objects of beauty.

— Donald E. Knuth (Knuth, 1974)

ACKNOWLEDGEMENTS

A big thanks goes out to Thomas Balstrøm for his supervision during the creation of this process. Thank you to the Open Source community, who made this project possible

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

ACRONYMS

DRY	Don't Repeat Yourself
API	Application Programming Interface
UML	Unified Modeling Language

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 has over the last couple of decades 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 a extent, that Open Source GIS software now can provide functionality comparable to what is available from the established commercial products. Using open source technologies as the back-end of your 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 exist as libraries to Python, and as simple standalone applications, some all-inclusive Open Source GIS applications exist, amongst these the most well-known are QGIS and 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. Understanding hydrological movement is complicated, and often requires advanced modelling. Furthermore, deciding on what kind of inputs and outputs that can be provided by the user, will simplify the process extensively. Hiding the process from the user, makes the entire ordeal less complicated and more easily handled.

1.2 PROBLEM STATEMENT

This project is inspired by our own work with flood modelling using ArcGIS in a previous semester of studies. Combining a variety of Open Source technologies, it should be possible to create an application that will enable a user, who is not necessarily proficient with GIS, of performing a flooding analysis. As such our problem statement will be as follows:

- Creation of a thin client based flood simulation and management tool using open source technologies

This problem should enable us to do a ton of work in absolutely no time, motherfucker!.

1.3 DELIMITATION

To base the project on the real world, the variables and constants used within the project will be based on usage within Denmark. There are several reasons for doing this. First of all, this is an area the project group has worked with before, and therefore all are comfortable with. This also creates a practical framework for the project I.e we can create the product based on real-life values and data, instead of making it up as we go along. All of these can be changed at a later state, but we fill it makes sense to create them with a foundation in Denmark

1.4 LICENSE

This project and all of it's items have been created as free and open source, and therefore follow the "GNU Software License".

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 the 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 GRASS' 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 in 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 are capable of. 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 more sophisticated, accurate and faster.

When investigating hydrological conditions, DEM (Digital Elevation Models) are used. Because of this, the results are dependent on the quality of the model being used. Most major GIS software have some built-in hydrology tools, capable of being incorporated into various workflows. 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.

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, 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:

- Multiple Flow Direction (MFD)

or

- Single Flow Direction (SFD)

Both of these algorithms are based on a so-called Moore-Neighborhood. This neighborhood involves 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 then 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

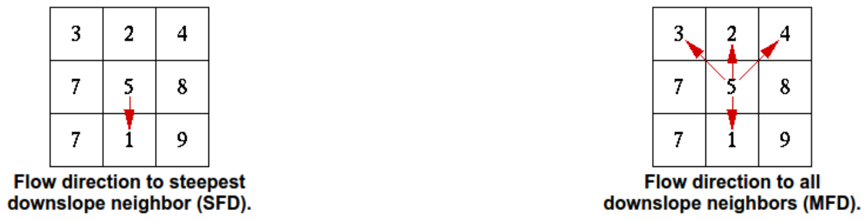

3 2 4
7 5 8
7 1 9
Flow direction to steepest downslope neighbor (SFD).
3 2 4
7 5 8
7 1 9
Flow direction to all downslope neighbors (MFD).

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

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 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 assign a Flow Direction on the filled terrain, thereby the flow direction will not be ruined 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 the flow running through the 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 the an output of stream lines.

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.

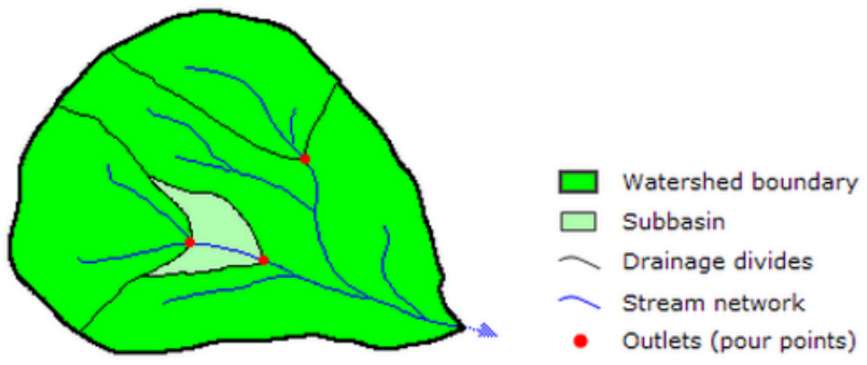


Figure 2: Figure showing important concepts when working with watersheds (ESRI, b).

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 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 the 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 setup 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 cost the least 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 feature, 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 on 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 factors chosen 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

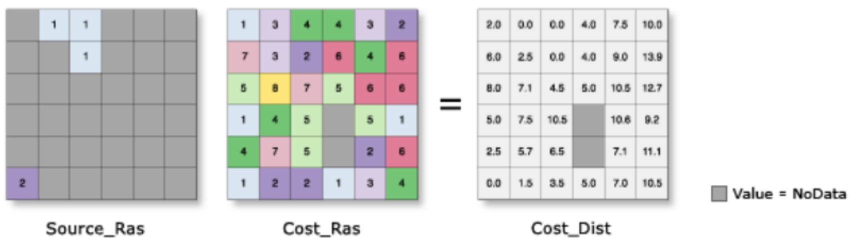


Figure 3: Figure showing how cost distance works (ESRI, a).

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.

2.2 PYTHON

Python is a programming language used extensively within the geospatial world. A lot of GIS packages are either created with Python, or have built-in functionality to inferface with Python. The actual reason for this is generally not known, but it might be because 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 for various reasons. The 2.7 release is a so-called legacy release, which means that it is not the worked-upon version, and wont 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 will likely 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)

Opening, and chaining, functions within GRASS can become tedious, and if wanting to run the same process a lot of times, it can be relevant to setup a workflow with a Python script. 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 (w3shcools). 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 use 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 a 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.

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 Development Team, 2009). 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 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()
```

A website written in Flask consists of a variety of templates that change content depending on where on a site a user is, or wishes to visit. This is done by defining some general templates, and some blocks of content that get swapped out depending on the current view.

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 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 license shall not restrict any party from selling or giving away the software as a component of an aggregate software distribution containing programs from several different sources. The license shall not require a royalty or other fee for such sale".)

- 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 distributes 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.

- Works derived from the source code may be required to have a different name or version number.

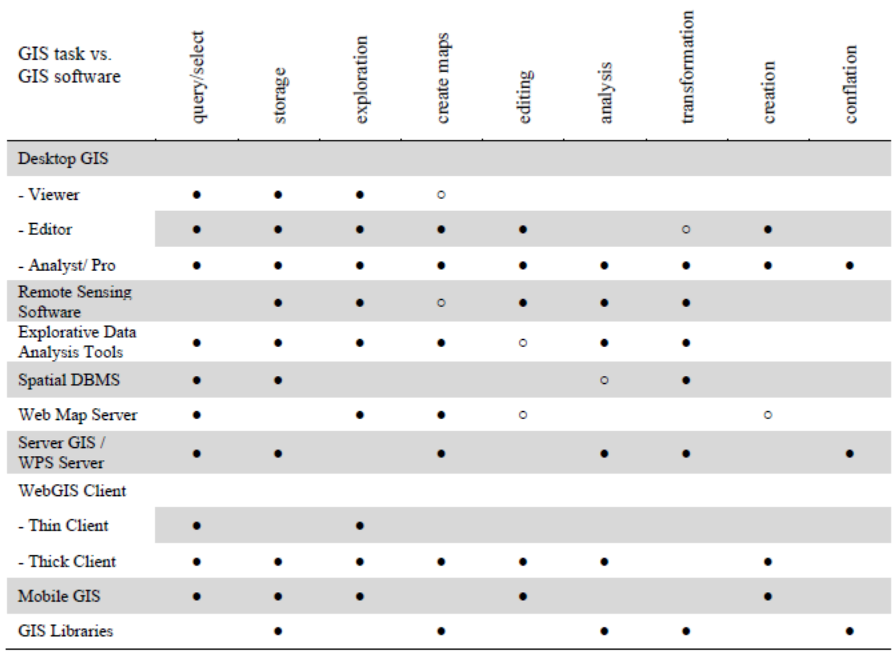
- Discrimination against persons or groups is not allowed

- 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 pr interface



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)

Some of the most common desktop GIS Software can be seen from Figure 4. The main reason open software exists is because the US Government, during the 1970ies and 1980ies 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).

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 the offer. Table 1 below can provide with some insight on such categorization.

Table 1: GIS software

Desktop GIS Software	Develope
GRASS GIS	Neteler & Mitasova, 2008; Neteler, Bowman, Landa & Metz 201
QGIS	Hugentobler, 200
ILWIS / ILWIS Open	Valenzuela, 1988; Hengl, Gruber & Shrestha, 200
uDig	Ramsey, 200
SAGA	Olaya 2004, Conrad 200
OpenJUMP	Steiniger & Michaud, 200
MapWindow GIS	Ames, Michaelis & Dunsford, 200
gvSIG	Anguix & Diaz, 200

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

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 three overarching stages, in which we expected various parts of the project to have progressed a certain amount.

Here there will be a graph of how we used our time.

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 (table XXX).

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

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 to all work on 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

As mentioned previously in this paper, this project is inspired by the work done during the second semester of our studies. In that project, we developed an ArcGIS toolbox which allowed the user 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.

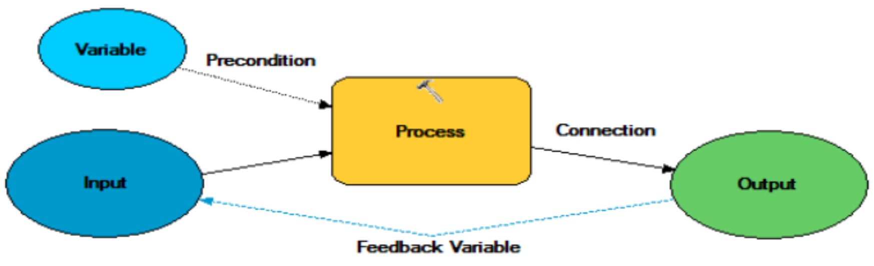


Figure 5: The old model.

The toolbox we created, comprised of ArcGIS functionalities with the core being the Cost Distance tool. The development was executed in 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, in order to apply more complex algorithm and connection between models models can quickly scale out of hand. The final tool made it possible to simulat water advancing through the landscape and covering parts of land that were below a certain predefined elevation, and were 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 endeavor, 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 begun 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 process we would have to go through when completing the project. They are created in such a way that they divide our work into parts that split our work into logical subparts. This division will also serve the purpose of making this report easier to read and more understandable to the reader. Based on this, we divided the project into a series of phases. 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, this 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 to the server setup was a constant

task throughout the entire project.

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

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

PHASE 1: 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. Finally, during this phase we setup the foundations we will base the work that will be performed in the following phases on.

PHASE 2: 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: 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: All the core functionalities of the applications are developed. This is the where the essence of the service is developed, from uploading an elevation model to creating a barrier and simulating the flood.

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

TESTING: This is the final step of the implementation process. It involves experimenting with the various functionalities of the application and monitoring the way they affect the results. It also includes trying to predict how a user might try and use the software, and then seeing how that behavior will affect the process.

WRITING: As the project would become more and more finished, we will start the writing part. As mentioned we will keep notes, and write various parts that are important to memorize down 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 don't want to have to redo various written parts, or we might even forget that they got changed, and forget to include the change in the project.

FINISHING UP: The final step is to make sure everything works, and to put a nice knot on the loose ends, as well as make sure that there is a red thread going through the project. Making sure that there is time to actually go through all the parts of the project is essential when doing a project such as this.

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 aiming in a more understandable and logically segmented report. That being said, we think that firstly presenting the final product that serves the purpose of its creation 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 reason 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 *Founding the project in Denmark*

3.2.2 *Timeline*

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, which would look something like Table 2

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.

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	10/03/2015	7

Table 2: Phases and their allotted time

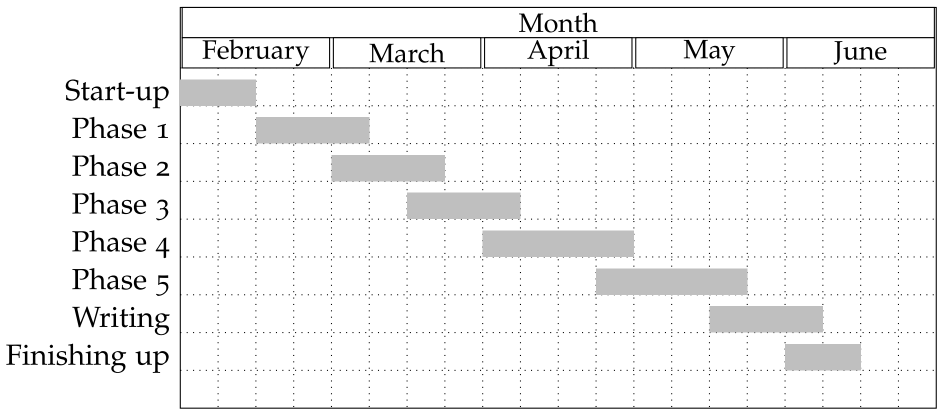


Figure 6: Gantt-chart showing the overlap of phases

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 a 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 being used in a variety of professional settings such as the american army corps of engineers. Furthermore, the application allows easy access to it's core functions through a Python interface. GRASS is natively supported by the WPS called PyWPS. Therefore this seemed like a natural choice for our project. To gain the online application of the project, a web server would obviously be needed, and for this hosting of some sort, and we ended up using the 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. Now that we were set on the tools we were going to use, we were starting the actual implementation, we decided to sketch out how we expected the application would end up looking, and functioning.

IMAGE OF THIS

Now that we have a rough idea of how the project would be working, we could get started on implementing these ideas into an actual solution.

4.1 PHASE 1

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 didn't have to update the server whenever changes were

made to the code.
The syntax for running GRASS through Python and GRASS through the WPS differ, 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 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 this.

Connecting to your server is done using SSL, 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 Linux is, and Ubuntu 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 it's 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. A thorough walkthrough of the installation of PyWPS can be found on our Github, so installation of PyWPS was performed into the folder of our webserver at the following directory:

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

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 bindings 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 then we were possible at first.

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

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.

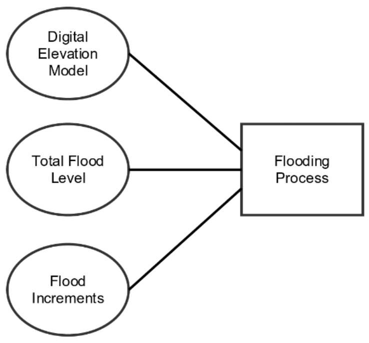


Figure 7: Inputs for the flooding model.

As seen on Figure 7, 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 geographical projection. 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 is mentioned in theory, GRASS uses GDAL for importing several type of raster maps and OGR for importing vector datasets. Taking the advantage of these modules it is possible for the to use any raster, as long as it is project into WGS 84. Another input of the model is the maximum flood level and the water rise increments of 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 flood level 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 that all streams and lakes in an area are filling with water) the model will require that the user interacts further. The user needs to choose from which source he/she would like flood the DEM from. 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 select 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.

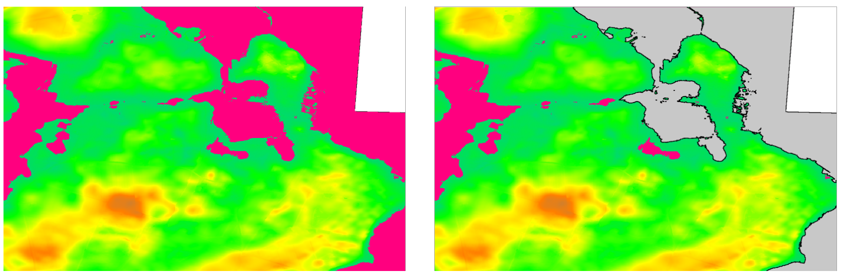

(a) Every cell with a value less than the current flood level.
(b) Cells reachable from the source point.

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

4.3 PHASE 3

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 2 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 scripting (figure XXX).

```
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, as shown below (figure XXX), 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 .tiff to .png. The reason behind that conversion is that we need to display the uploaded elevation 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 leaflet, 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 as a javascript library, 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 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

To expand on the functionality of the application, we have additional functionalities and tools, which could be a tool used for flood or disaster management.

4.4.1 *Identifying critical areas*

4.4.1.1 *Outlet analysis*

During the development of the pour-point analysis, we encountered sine problems. At the beginning, we tried to derive this problem from a watershed analysis, based on the identification of watershed's outlet. Our hypothesis was , that the flood would flow backwards through the flow accumulation and flow stream model to the source cells and when the water enters through an outlet, to a new watershed, than that area would become flooded. Therefore we created a workflow which extracts only the outlet of the watershed. As it 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, has to be at least the ten percent of the DEM. After the workflow run the previously used neighborhood analysis(r.neighbor) with diversity method with we could extract the watershed borders on a raster. Using the terraflow module with single flow direction setup, we created the a flow accumulation raster. This is important to determine the inlet/outlet location. Using the single flow direction method, every outlets will be located on the border of a watershed where the flow accumulation is the maximum. With this approach we though we got can a relatively good result to determine where the flood flow will enter to new area, however the testing of the algorithm showed its failure of our planned purpose.

4.4.1.2 *Pour point analysis*

Only providing the a extent is not enough when wanting to prepare against the impact a disaster. We wanted to provide more information about the flood regarding areas that might be particularly dangerous.

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

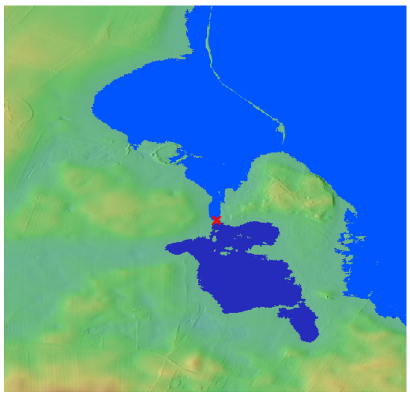


Figure 9: Figure showing the pourpoint we are trying to identify.

Figure 9 shows two consecutive flood extents with 140cm(light blue) and 141 cm (dark blue) water rise. The red area indicated on the figure is 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 stage 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 overcome the situation 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, thereby we can built a decision analysis 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 subracted from each other, 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 to avoid relatively small areas (indicated in grey on Figure 10a, which the pour point analysis would not make a reasonable investigation since their size is negligible compared to those area 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.

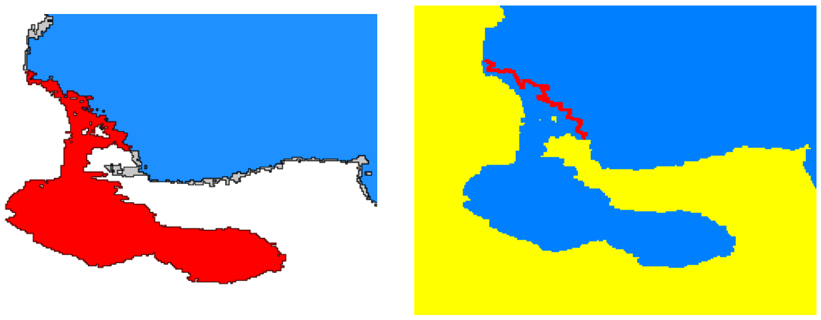

(a) Showing two flood extents. (b) Showing line of pour points.

Figure 10: 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) with increments which are over the criteria. This raster has raster values where the flood extent exists and NULL values everywhere else. Afterwards a neighborhood analysis (r.neighbors) is conducted, based on the diversity method which is based on the Moore-Neighborhood. 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 10a a black border can be seen, where the red and blue area touch. The process need to identify automatically only that border which connects the two different flood stage 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 10b. After the identifying the pour-point cells with the above-mentioned 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 th elevetion of the pour-point location and the maximum flood depth based on the actual scenario. This was done by point sampling method of GRASS. (r.what.rast). Sampling method extracts those cell attribute, where the vector points act as a centroid of the cell square.

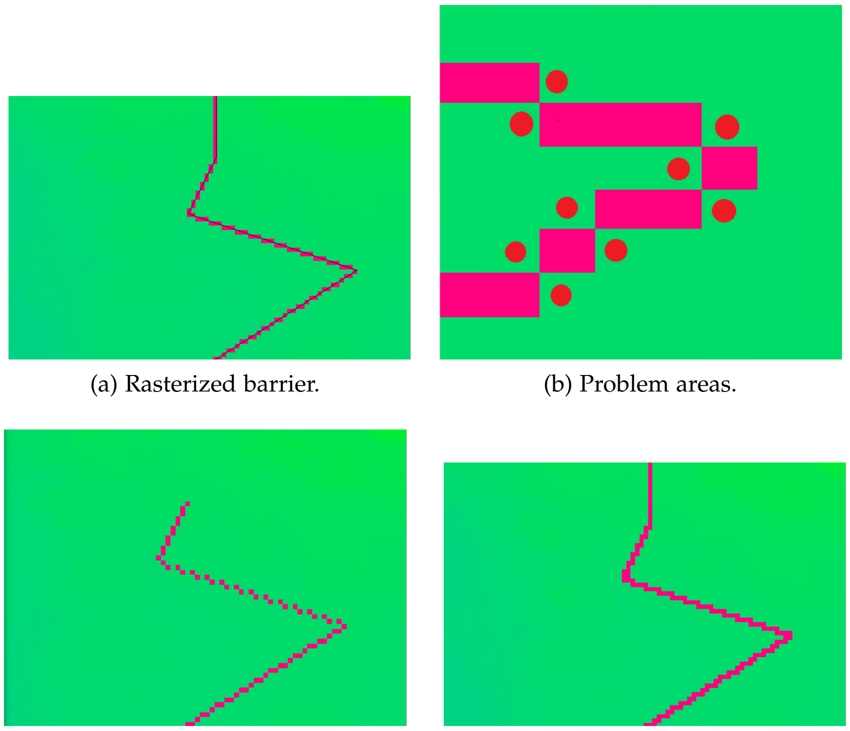
4.4.2 *Barrier placement*

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 advanced simulation can take a lot of time.

The process begins by getting the input from the webpage where the user draws a barrier on a web map. This results into the creation of a GeoJSON file that is then transferred to the python function that creates the does the modelling. Since the barrier is 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 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 11a.

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 11b.

In order to do that, first we extract the edges of the non-vertical, non-horizontal sections of the barrier. In addition, we extend those



(a) Rasterized barrier.

(b) Problem areas.

(c) Generating pixels in the designated areas.

(d) Fixed barrier.

Figure 11: The process of fixing the rasterized barrier.

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

Table 3: Relative cell coordinates to the center cell (0,0)

cells by one cell in the northerly direction. That action is performed by the following part of the script, as seen on Figure 11c.

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 full neighborhoods (e.g. Moore Neighborhood) but specific cells that exist in the neighborhood of the cell mapcalc is actually 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 3.

The result of this edge selection is shown on the figure below (figure XXX).

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 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. Which is done with the following script:

```
gscript.run_command('r.mapcalc', expression='original_b = if(
    isnull(merged),original,original+merged)')
```

This process results in the elevation model shown on Figure 12.

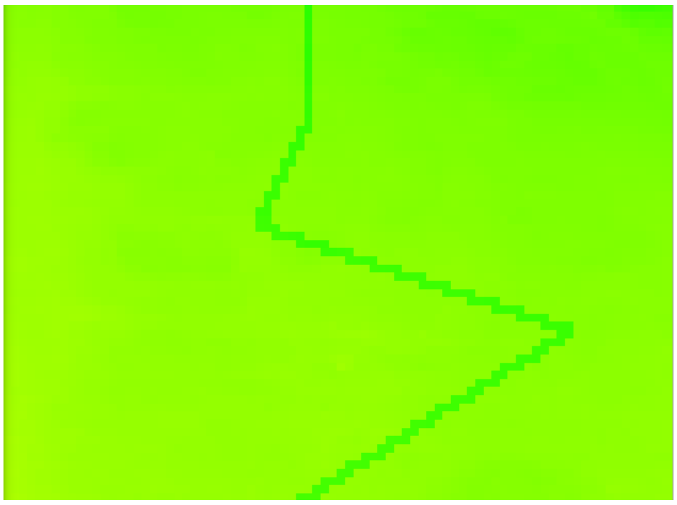


Figure 12: Figure showing important concepts when working with watersheds

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.

4.4.3 *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 suits, such as QGIS and GRASS as well as ArcGIS, address issues as 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 raster to vector conversion. On the other hand, QGIS and GRASS does not. This lead us to dedicate a considerable amount of time in 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. 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

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

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

The output cannot be defined dynamically.

As mentioned in the theory part, every PyWPS must include

- __init__

and

- execute.

The various settings we set up 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, 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 this has 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's 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 the 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 reduce the loading times significantly, and can for very large images also freeze the computer 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 reday 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 webapplication interface. They way the WPS service is requested from the server, I.e a flooding service with the identifier flooding, is called like follows:

- http://52.17.144.192/cgi-bin/pywps.cgi?request=executeservice=WPSversion=1.0.0identifier=floodingdatainputs=[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. IMAGE OF FRONTPAGE IMAGE OF UPLOAD 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 Ocean
- 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 Ocean: 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 is created as a JPEG, which then can be used to overlay on the map. The JPEG cannot contain embedded coordinates. Using Flask, 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=COORDminy=COORDmaxx=COORDmaxy=COORDfilename=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 quick 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. So therefore only one request will be described here. The buttons used to send the request have been marked with either a advancedflood or simpleflood ID. In this instance we wil 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 asynch request is done, it launches some other code. 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 above 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 not 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 INSTALLATION STRUCTURE

Now that the application has been implemented, it is relevant to analyze how the entire installation structure.

A variety of softare packages have been installed on the server, these being PyWPS, Flask, Apache2 and GRASS. Debugging: When the server has been installed, it runs on it's own, and not much else has to be manipulated with from then on. When the process fail 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 following directories:

```
/
└─/var/
   ├─/www/
   │  └─/html
   │     └─/pywps/
   │        └─pywpslog
   └─/log/
      └─/apache2/
         └─errorlog
```

Accessing the 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 here. The setup is displayed in Figure XXX.

```
/
└─/var/
   └─/www/
      └─/html
         └─/pywps/
            ├─pywpslog
            ├─pywps.cfg
            └─/processes/
               ├─__init__.py
               └─process1py
```

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 is installed in the directories shown on Figure XXX.

```
/
└─/var/
   └─/www/
      └─/html
         ├─/FlaskApp/
         └─flaskapp.wsgi
            └─/FlaskApp/
               ├─/templates/
               ├─/images/
               ├─/static/
               └─__init__.py
```

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 isn't directly served by the user, isn't 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.

```
/
└─/home/
   └─/ubuntu/
      ├─.grassrc6
      └─/grassdata/
         └─<LOCATION>
            └─<MAPSET>
```

The GRASS installation is fairly simple, 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, projected to WGS84. 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.

5.2 ARCHITECTURE

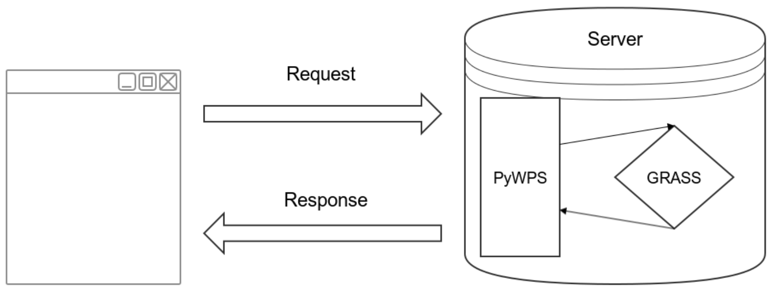


Figure 13: The architecture of the software

5.3 USER EXPERIENCE

In this part of the report we will present how our web service works for 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 **??**

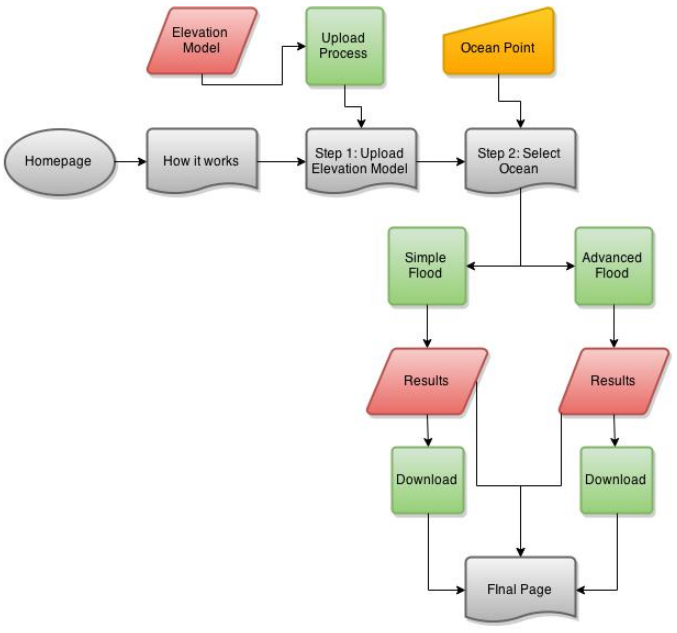
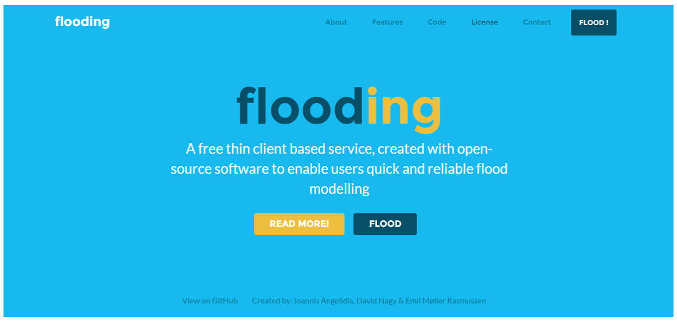


Figure 14: Overall workflow of the application

We begin by presenting the first page the user will see when they visit our website. The url that leads to the homepage of the application is http://52.17.144.192/. The first page is then displayed that provides information about the website such as its name and a basic description of what services it offers (Figure 15

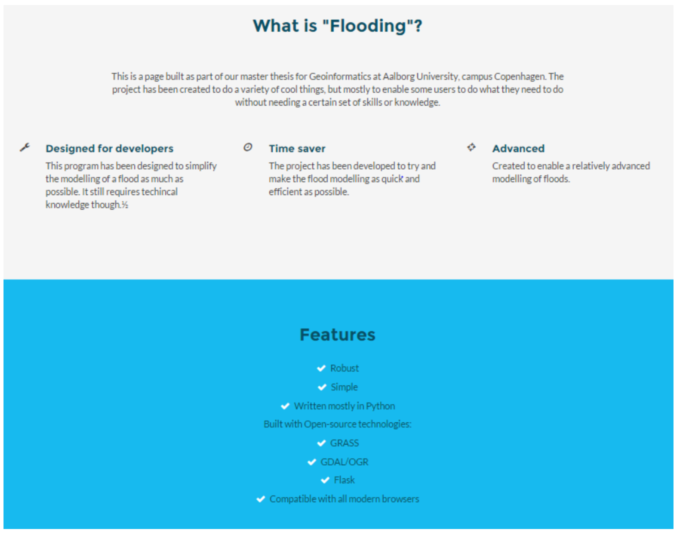


A free thin client based service, created with open-source software to enable users quick and reliable flood modelling

Created by: Ioannis Angelidis, David Nagy & Emil Møller Rasmussen

Figure 15: Homepage of the application

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 and its most important advantages and features Figure 16



What is "Flooding"?

This is a page built as part of our master thesis for Geoinformatics at Aalborg University, campus Copenhagen. The project has been created to do a variety of cool things, but mostly to enable some users to do what they need to do without needing a certain set of skills or knowledge.

Designed for developers

This program has been designed to simplify the modelling of a flood as much as possible. It still requires techincal knowledge though.½

Time saver

The project has been developed to try and make the flood modelling as quick and efficient as possible.

Advanced

Created to enable a relatively advanced modelling of floods.

Features

- ✓ Robust
- ✓ Simple
- ✓ Written mostly in Python

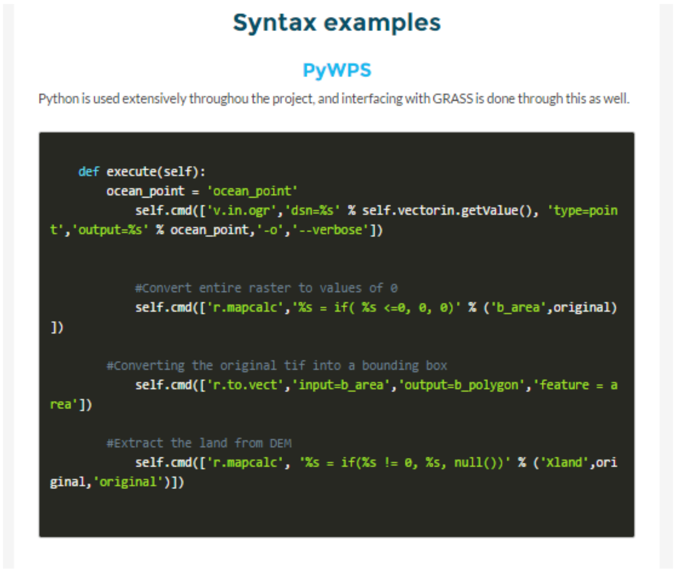
Built with Open-source technologies:

- ✓ GRASS
- ✓ GDAL/OGR
- ✓ Flask
- ✓ Compatible with all modern browsers

Figure 16: Details and characteristics of the application

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 17

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 (**??**). This part is a guided process of the flood simulation that follows. By clicking on the Commence flooding option, the initial part of the flood commences.



Syntax examples

PyWPS

Python is used extensively throughou the project, and interfacing with GRASS is done through this as well.

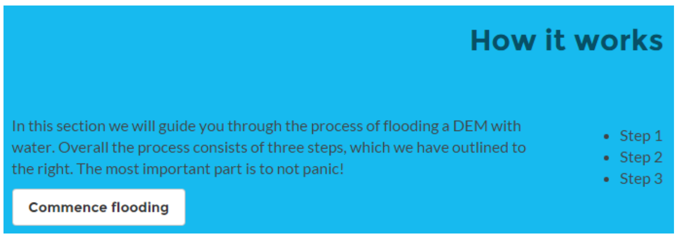
```
    def execute(self):
        ocean_point = 'ocean_point'
            self.cmd(['v.in.ogr','dsn=%s' % self.vectorin.getValue(), 'type=poin
t','output=%s' % ocean_point,'-o','--verbose'])


            #Convert entire raster to values of 0
            self.cmd(['r.mapcalc','%s = if( %s <=0, 0, 0)' % ('b_area',original)
])

        #Converting the original tif into a bounding box
            self.cmd(['r.to.vect','input=b_area','output=b_polygon','feature = a
rea'])

        #Extract the land from DEM
            self.cmd(['r.mapcalc', '%s = if(%s != 0, %s, null())' % ('Xland',ori
ginal,'original')])
```

Figure 17: Examples of scripts in use



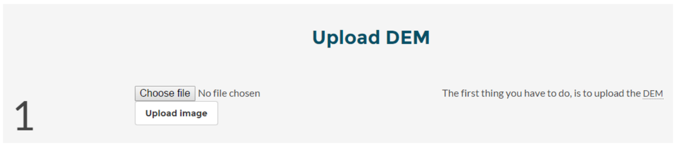
How it works

In this section we will guide you through the process of flooding a DEM with water. Overall the process consists of three steps, which we have outlined to the right. The most important part is to not panic!

- Step 1
- Step 2
- Step 3

Figure 18: Introduction to flooding simulation

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



Upload DEM

1

The first thing you have to do, is to upload the DEM

Figure 19: Step 1 upload Digital Elevation Model

Once the user chooses an elevation model from his local hard drive and then uploads it, the second step is initialized (Figure 20).

the application the user is called to provide it with the ocean. In other words, where he wants the flood to originate from. By simply clicking on the map, that point is created. Finally, the user then has to decide which type of process he/she wishes 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 21).

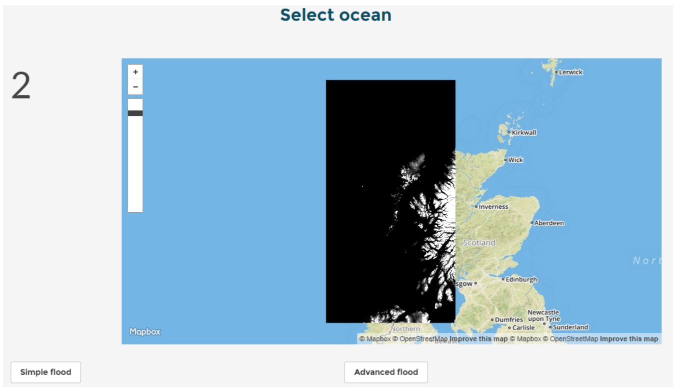


Figure 20: Step 2 display uploaded elevation model and insert point to designate origin of flood

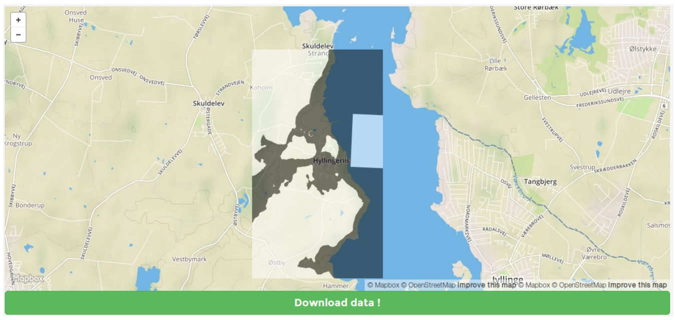


Figure 21: Results of the simple flood option

6 DISCUSSION

Part I

APPENDIX

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.

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.

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

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

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.

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.

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

`http://code.google.com/p/classicthesis/`

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

`http://postcards.miede.de/`

DECLARATION

Put your declaration here.

Copenhagen, June 2015

Giannis Angelidis,David
Nagy and Emil Møller
Rasmussen June 1, 2015