MASTER THESIS

EMIL MØLLER RASMUSSEN, IOANNIS ANGELIDIS AND DAVID NAGY

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.

— **?** [**?**]

ACKNOWLEDGEMENTS

Put your acknowledgements here.

Many thanks to everybody who already sent me a postcard!

Regarding the typography and other help, many thanks go to Marco Kuhlmann, Philipp Lehman, Lothar Schlesier, Jim Young, Lorenzo Pantieri and Enrico Gregorio[1], Jörg Sommer, Joachim Köstler, Daniel Gottschlag, Denis Aydin, Paride Legovini, Steffen Prochnow, Nicolas Repp, Hinrich Harms, Roland Winkler, and the whole LaTeX-community for support, ideas and some great software.

Regarding LyX: The LyX port was intially done by *Nicholas Mariette* in March 2009 and continued by *Ivo Pletikosić* in 2011. Thank you very much for your work and the contributions to the original style.

1 Members of GuIT (Gruppo Italiano Utilizzatori di TeX e LaTeX)

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

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 SDTS DEM, etc.

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.

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 XXX, the SFD method assigns a single flow direction to the lowest downslope neighboring cell, whilst MFD assigns flow direction to all downslope neighboring cell.

Figure 1: Example of two different flow direction methods

Both method has 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. 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 (Osmar et al., 2013), 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.

2.1.1.2 *Flow accumulation*

Another important hydrology tool is Flow Accumulation. This tool is capable of calculating the stream conditions of the terrain, such as the accumulation of water. 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. 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 watershed.

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

A watershed is a relative term, as it depends on the scale at which you look at them practically all streams have their own watershed.

Therefore to determine the watersheds of a DEM, it is necessary to know a variety of factors, such as flow accumulation, flow direction or slope aspect of the terrain.
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. 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. 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. 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 based on the previously explained method. In order to get the cost distance output raster, the user needs to provide a cost raster.

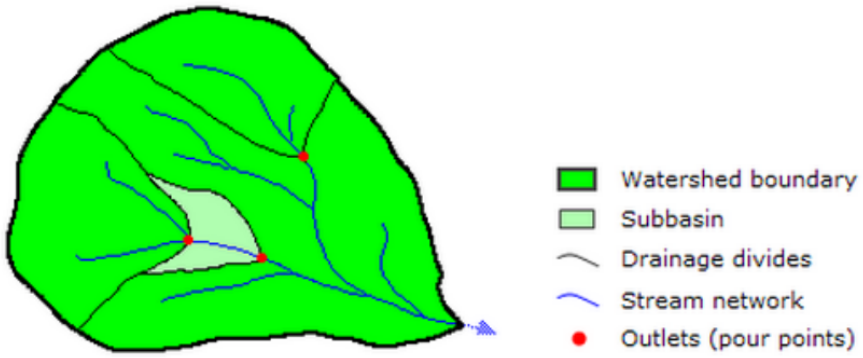


Figure 2: Figure showing important concepts when working with watersheds

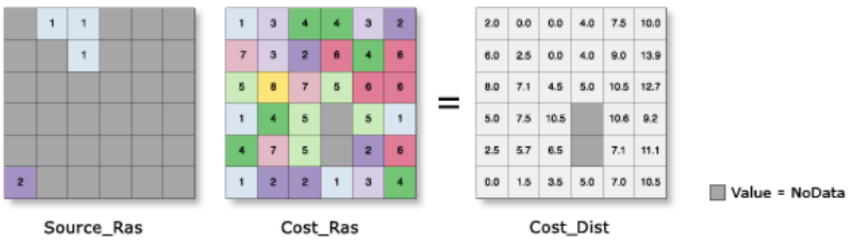


Figure 3: Figure showing how cost distance works

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:

- 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. (https://wiki.python.org/moin/Python2orPython3) Standard python syntax looks something like this:

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

2.3 GRASS

GRASS functions can be accessed and manipulated by using Python. Python can easily be used within the main GRASS shell, but it is

also possible to create Python scripts that can call GRASS functionality without explicitly starting the GRASS software. 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

- 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. (DOCUMENTATION)

An example of the syntax of GRASS functions being accessed externally through Python:

```
1 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 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. http://www.opengeospatial.org/standards/wps
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 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. 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. 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.orequest=DescribeProcessidentifier=all
- http://webaddress/pywps/?service=WPSversion=1.0.orequest=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:

- A class defining the initation of a WPS Process: class Flooding(WPSProcess):

- A function called $_{i}n_{i}t_{:}d_{e}f_{\ i}n_{i}t_{(}s_{e}l_{f}):A_{f}u_{n}c_{t}i_{o}n_{c}a_{l}l_{e}d_{e}x_{e}c_{u}t_{e}:d_{e}f_{e}x_{e}c_{u}t_{e}(s_{e}l_{f}):$

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'])
```

2.5 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, and only includes the strict necessities- further functionality is expected to be import from third-party libraries.
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. https://scholar.google.com/scholar?hl=frq=flask+microframeworkbtnG=lr=
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:

```
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.

2.6 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:

- 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 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. The table below can provide with some insight on such categorization (table XXX).

Some of the most common desktop GIS Software are the following (table XXX).

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

Table 1: GIS software

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

Categorization of GIS software. (Steiniger & 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 in Chapter 1, 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 stowage of sea water in the event of a storm. The toolbox we created, comprised of ArcGIS functionalities with the core being the Cost Distance tool. This tool made possible the simulation of water advancing through the landscape and covering parts of land that were below a certain predefined elevation and were directly connected to the source of the flood.
Through the completion of that project, we were left wondering how this toolbox could be made available to an even greater amount of people. Also, another question that turned up was how we can make this tool available freely without any of the software limitations that using a proprietary licensed software, like the ArcGIS software suite, provides.
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.

3.2.1 *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.

3.2.2 *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.

3.2.3 *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.

3.2.4 *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.

3.2.5 *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.

3.2.6 *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.

3.2.7 *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.

3.2.8 *Finish 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.

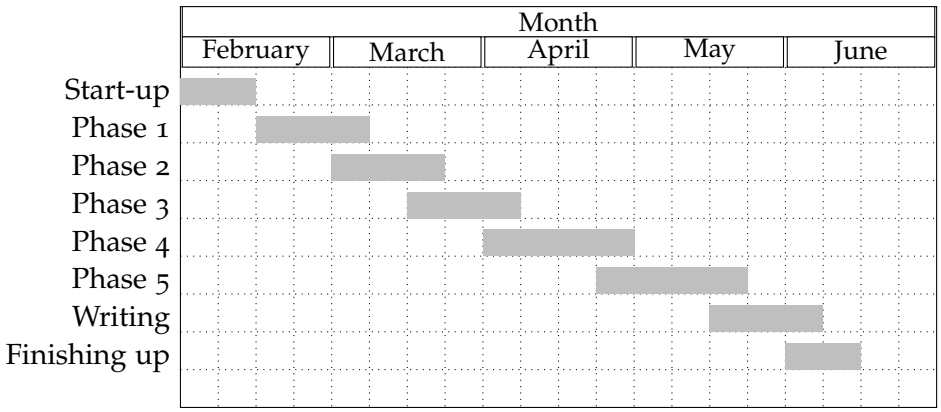
3.2.9 *Timeline*

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 the following table:

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

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:



4 IMPLEMENTATION

As mentioned previously the work was divided into a series of phases. It is important to note that the majority of the development was performed on machines running Ubuntu. The syntax and explanations will therefore reflect this fact.

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:

```
1 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:

```
1 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.

IMAGE

As seen on Figure XXX, 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 XXX. On the left image of the figure, 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 the image to the right.

IMAGE

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 Phase 5.

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 us-

ing */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 upload-

ing 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 *Barrier placement*

An integral part of the application is allowing the user to implement barriers that will mitigate the effects of the flood to a given area. That way, 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.

4.4.2 *Pour point*

4.4.3 *Outlet analysis*

5 ANALYSIS

6 DISCUSSION

Part I

APPENDIX

A

APPENDIX TEST

Aliquam lectus. Vivamus leo. Quisque ornare tellus ullamcorper nulla. Mauris porttitor pharetra tortor. Sed fringilla justo sed mauris. Mauris tellus. Sed non leo. Nullam elementum, magna in cursus sodales, augue est scelerisque sapien, venenatis congue nulla arcu et pede. Ut suscipit enim vel sapien. Donec congue. Maecenas urna mi, suscipit in, placerat ut, vestibulum ut, massa. Fusce ultrices nulla et nisl.

Etiam ac leo a risus tristique nonummy. Donec dignissim tincidunt nulla. Vestibulum rhoncus molestie odio. Sed lobortis, justo et pretium lobortis, mauris turpis condimentum augue, nec ultricies nibh arcu pretium enim. Nunc purus neque, placerat id, imperdiet sed, pellentesque nec, nisl. Vestibulum imperdiet neque non sem accumsan laoreet. In hac habitasse platea dictumst. Etiam condimentum facilisis libero. Suspendisse in elit quis nisl aliquam dapibus. Pellentesque auctor sapien. Sed egestas sapien nec lectus. Pellentesque vel dui vel neque bibendum viverra. Aliquam porttitor nisl nec pede. Proin mattis libero vel turpis. Donec rutrum mauris et libero. Proin euismod porta felis. Nam lobortis, metus quis elementum commodo, nunc lectus elementum mauris, eget vulputate ligula tellus eu neque. Vivamus eu dolor.

A.1 APPENDIX SECTION TEST

Nulla in ipsum. Praesent eros nulla, congue vitae, euismod ut, commodo a, wisi. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Aenean nonummy magna non leo. Sed felis erat, ullamcorper in, dictum non, ultricies ut, lectus. Proin vel arcu a odio lobortis euismod. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Proin ut est. Aliquam odio. Pellentesque massa turpis, cursus eu, euismod nec, tempor congue, nulla. Duis viverra gravida mauris. Cras tincidunt. Curabitur eros ligula, varius ut, pulvinar in, cursus faucibus, augue.

More dummy text

Nulla mattis luctus nulla. Duis commodo velit at leo. Aliquam vulputate magna et leo. Nam vestibulum ullamcorper leo. Vestibulum condimentum rutrum mauris. Donec id mauris. Morbi molestie justo et pede. Vivamus eget turpis sed nisl cursus tempor. Curabitur mollis sapien condimentum nunc. In wisi nisl, malesuada at, dignissim sit amet, lobortis in, odio. Aenean consequat arcu a ante. Pellentesque porta elit sit amet orci. Etiam at turpis nec elit ultricies imperdiet. Nulla facilisi. In hac habitasse platea dictumst. Suspendisse

LABITUR BONORUM PRI NO	QUE VISTA	HUMAN
fastidii ea ius	germano	demonstratea
suscipit instructior	titulo	personas
quaestio philosophia	facto	demonstrated

Table 2: Autem usu id.

Listing 1: A floating example

```
for i:=maxint to 0 do
begin
{ do nothing }
end;
```

viverra aliquam risus. Nullam pede justo, molestie nonummy, scelerisque eu, facilisis vel, arcu.

A.2 ANOTHER APPENDIX SECTION TEST

Curabitur tellus magna, porttitor a, commodo a, commodo in, tortor. Donec interdum. Praesent scelerisque. Maecenas posuere sodales odio. Vivamus metus lacus, varius quis, imperdiet quis, rhoncus a, turpis. Etiam ligula arcu, elementum a, venenatis quis, sollicitudin sed, metus. Donec nunc pede, tincidunt in, venenatis vitae, faucibus vel, nibh. Pellentesque wisi. Nullam malesuada. Morbi ut tellus ut pede tincidunt porta. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Etiam congue neque id dolor.

Donec et nisl at wisi luctus bibendum. Nam interdum tellus ac libero. Sed sem justo, laoreet vitae, fringilla at, adipiscing ut, nibh. Maecenas non sem quis tortor eleifend fermentum. Etiam id tortor ac mauris porta vulputate. Integer porta neque vitae massa. Maecenas tempus libero a libero posuere dictum. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Aenean quis mauris sed elit commodo placerat. Class aptent taciti sociosqu ad litora torquent per conubia nostra, per inceptos hymenaeos. Vivamus rhoncus tincidunt libero. Etiam elementum pretium justo. Vivamus est. Morbi a tellus eget pede tristique commodo. Nulla nisl. Vestibulum sed nisl eu sapien cursus rutrum.

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

Emil Møller Rasmussen,
Ioannis Angelidis and David
Nagy, May 30, 2015