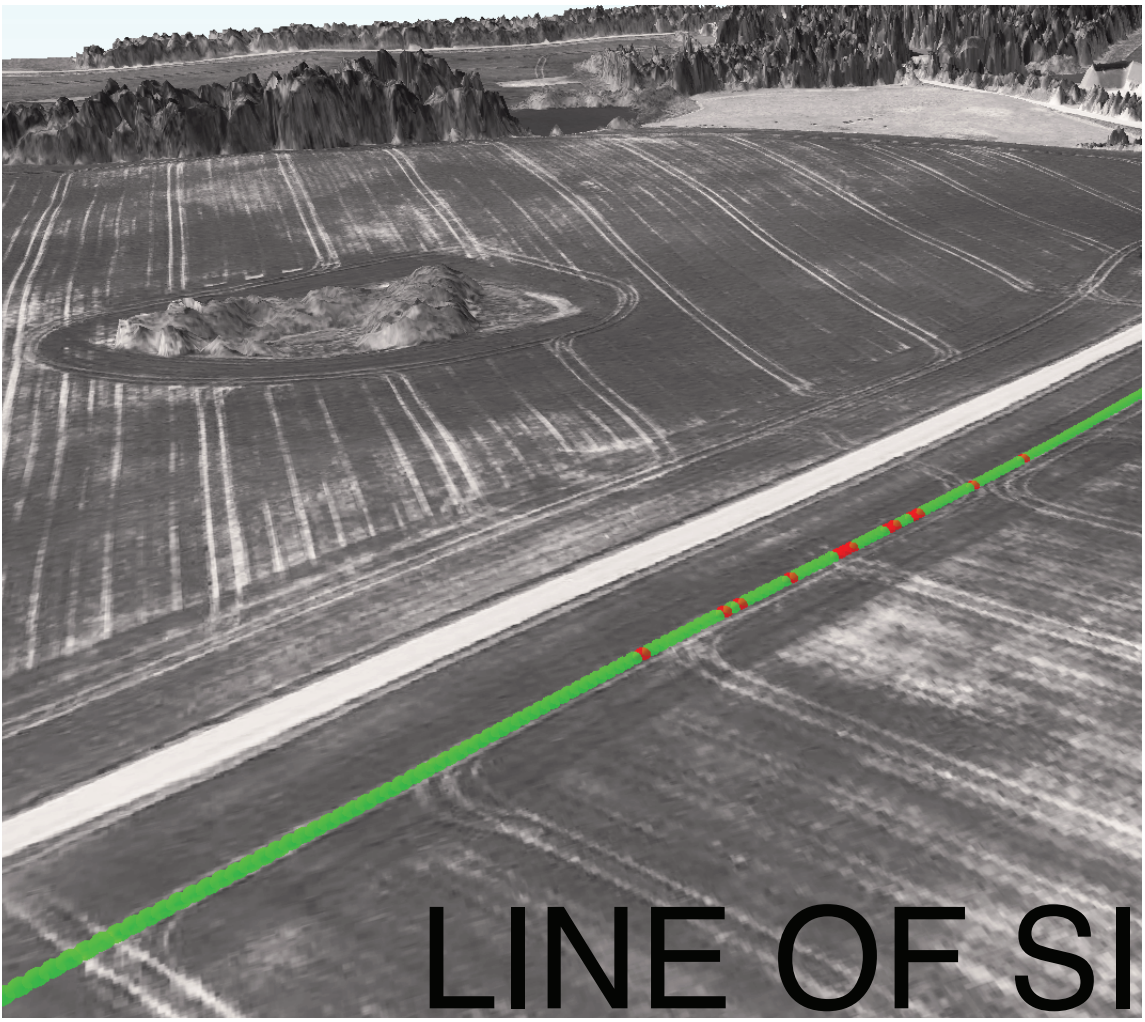

LINE OF SIGHT CALCULATION
AS A SERVICE

MASTER'S THESIS IN GEOINFORMATICS

by JONAS NYGAARD PEDERSEN

SUPERVISORS:
HENNING STEN HANSEN
&
MORTEN WINTHER FUGLSANG

AALBORG UNIVERSITET

STUDENTERRAPPORT

Department of Development and Planning

Course: Master's Thesis

4th Semester

Project period: February 2016 - June 2016

Submission date: 15.06.2016

Line of Sight Calculation as a Service

Jonas Nygaard Pedersen
nr: 20143001

supervised by
Henning Sten Hansen
Morten Winther Fuglsang

Geoinformatics
Aalborg University Copenhagen
A.C. Meyers Vænge 15
2450 Copenhagen SV
Secretary: Trine Kort Lauridsen

Abstract

This thesis explores how visibility can be calculated as a line of sigt between two points and then implementing this solution in a *PostgreSQL* database containing raster data on elevation as a custom *PL/pgSQL* function. This functionality is then published as a service that is easily consumable by web map libraries using the WPS standard. To evaluate the precision and performance of the product an analysis on the result will be performed by comparing it to ortho imagery, an on-site photo, a DEM and a 3D model.

Lastly the potential value of such a tool in a property tax environment is examined as part of a larger modular standardised framework.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API	Application Programming Interface
DEM	Digital elevation model
FOSS	Free and open-source Software
GEOS	Geometry Engine - Open Source
GIS	Geographic Information System
GML	Geography Markup Language
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
KVP	Key Value Pair
OGC	Open Geospatial Consortium
PL/pgSQL	Procedural Language/PostgreSQL
RDBMS	Relational Data Base Management System
REST	Representational State Transfer
SDI	Spatial Data Infrastructure
SKAT	Danish Tax and Customs Authorities
SQL	Structured Query Language
WFS	Web Feature Service
WMS	Web Map Service
WMTS	Web Map Tiling Service
WPS	Web Processing Service

WSDL Web Service Definition Language

W3C World Wide Web Consortium

XML Extensible Markup Language

Chapter 1

Introduction and Problem Statement

1.1 Introduction

Working with Geographic Information Systems (GIS) is mostly a field reserved for professionals within the GIS community. While these professionals can create content that is consumable by laymen or share data and packages optimised for the GIS community's specialised software, it has not been easy for laymen themselves to do spatial analysis. By increasing simplicity the user base of spatial analysis will increase and bring with it new possible scenarios for the use of spatial analysis.

The *Danish Customs and Tax Administration* are experiencing that their methods of valuing housing and property for the use in subsequent tax calculations are at the centre of a heated public debate. Their current methodology is being accused of being subjective and non-repeatable and leading to great inconsistencies in the property and housing tax levels even within similar types of property (Drachmann 2016; Drachmann and J. Hansen 2016; Dengsøe 2015; Bang 2013).

The negative public perception of the *Danish Customs and Tax Administration* and the negative trend in the publicly perceived sense of justice of the valuation results can potentially undermine the collective national contract of equality that supports the welfare model employed in Danish society. Undermining this model can result in greater resistance towards participation in the model and an increased tendency towards tax evasion.

With these issues receiving increased public attention, the *Danish Customs and Tax Administration* face a possible reform of their housing and property valuation methodology, moving from a somewhat subjective process that have resulted in the great debate on its legitimacy, to a an objective process that includes geodata as some of the possible parameters that influence valuation.

The previous government recognised the issues at hand and undertook an expert hearing to formulate a new and more objective methodology for the valuation of housing and property (Skatteministeriet 2013b). The expert hearing suggestions and findings

included the increased use of geodata as an objective valuation parameter and a focus on repeatability of the calculations used (P. E. Jensen et al. 2014).

To implement these findings in a model, it is an obvious short reach to use the geodata available from different state agencies and ministries to construct an objective model that can process the information in a considerate fashion that respects the parameters' influence on real life pricing.

Among these possible geodata parameters is the visibility of amenities such as forest and coast, which are often considered premium localities, or the visibility of features such as motorways and rail-roads, which have a negative connotation. This thesis will focus on the visibility parameter and its possible calculation on the basis of national elevation data and its possible usage in property tax valuation.

One way of creating such a calculation and publish it as a single task spatial analysis package is the *Open Geospatial Consortiums* Web Processing Service standard (WPS), which can be employed to create standardised (the different *OGC* standards are inter connectible) geospatial processing services, where the main burden of data and analysis processing can be located on a remote server.

Beyond offloading the burden of most of the data used for the analysis, the WPS enables a simplified approach to spatial analysis so that the user only will need a browser (or other WPS client, like *QGIS*) to do the analysis.

1.2 Problem Statement

The visibility of the geodata features can have an influence on property valuation, but how should the visibility be calculated, how can such a calculation be exposed in a simple and accessible format so that non-GIS workers can use the service and how can the visibility calculation be incorporated into The *Danish Customs and Tax Administration* valuation workflow. To answer this problem statement the following clarifying questions will be sought answered:

Thesis Questions:

- How can a software stack capable of hosting a remote server running a spatial processing service that is accessible from across a network be implemented?
- How can a line of sight that returns information on visibility be calculated?
- How do the proposed service perform against corporate services?
- What precision does the line of sight calculation yield compared against an on sight investigation?
- How can a line of sight calculation be used in housing and property tax valuation?

Chapter 2

Theory

2.1 Standards and Their Role in the Geospatial Community

The Web Processing Service standard is one of many standards defined by the *Open Geospatial Consortium* (*OGC*). The *OGC* is a non-profit organisation that is committed to creating open standards for the geospatial community to facilitate the easy sharing of data. Along with some proprietary formats like *ESRI* shape files, the *OGC* standards are at the base of much of the data sharing that takes place within the community.

Beyond data sharing within the geospatial community *OGC* standards are prevalent as a method of publishing geospatial data on the web in the form of several standards, of which the most widespread are the *Web Mapping Service*, which is designed to retrieve geo-registered images from a server (OGC 2006), the *Web Map Tile Service* that also retrieves geo-registered images, but which are pre-rendered (OGC 2010b), *Web Coverage Service*, which provides raster coverage data (OGC 2012) and *Web Feature Service*, which serves features and enables querying and editing of these features (OGC 2010a). Lastly of the *OGC*'s web services is the Web Processing Service (WPS), which is used to enable server side execution of a processing algorithm on spatial data (OGC 2008).

These standards, when adopted broadly, are effective in avoiding isolated islands of data, which would else not be easily utilised. Allowing data to be easily utilised avoids the expenses associated with the need to acquire new software or acquiring similar data and facilitates the sharing of data in a way that facilitates the interoperability of data (Steiniger and Hunter 2011).

As the implementation standards from *OGC* are free of royalties they have been picked up by several software packages and web frameworks as the basis for distribution of geospatial data(OGC 2010a; OGC 2010b; OGC 2008; OGC 2006).

This ubiquity makes the *OGC* standards an obvious choice as a basis for creating a spatial data infrastructure (SDI) (Steiniger and Hunter 2011).

2.2 Web Processing Service

Spatial analysis has traditionally, or at least simultaneously with the rise of the digital age, been done using desktop class geographical information systems (GIS)(Matt Duckham and Worboys 2007). The use of these have primarily been in the realm of a small group of GIS specialists, to an extent excluding the layman.

These software packages have been characterised by being expensive excluding even further, the practise of spatial analysis to a small subset of professionals. The advent of Free and Open Source Software (FOSS) has to some extent alleviated this problem, but usability, installation issues and complexity still prevails within the community and the software on which it depends.

With the advent of the web and subsequently web standards the possibilities of integrating hitherto desktop bound spatial analyses in to a web framework has increased (Matt Duckham and Worboys 2007). Development within *JavaScript* mapping libraries and map sources helped along by the standards defined by *OGC* creating standards like Web Mapping Service (WMS) (OGC 2006), Web Mapping Tile Service (WMTS) (OGC 2010b), Web Feature Service (WFS) (OGC 2010a) and the subject of this thesis, the Web Processing Service (WPS).

The line of sight service product, which is one of the goals of this thesis, runs within the *ZOO-Project* WPS framework (ZOO-Project team 2016). Like *PyWPS* (PyWPS 2016), *52°North WPS* (52° North GmbH 2016) and *GeoServer* (Open Source Geospatial Foundation 2016), *ZOO-Project* WPS is a framework that enables spatial processing to be handled as a remote process according to the previously presented *OGC* standard (OGC 2008).

2.2.1 OGC WPS Standard

In the following section the focus will be on the *OGC* WPS standard version 1.0.0 and the *ZOO-Project* WPS framework, which is an implementation of the *OGC* standard (*ZOO-Project* WPS also supports version 2.0.0, but this version will not be used and will not covered any further).

The WPS processes can be designed to call other web services of the *OGC* flavour and thus it can act as a processing engine that can drive an analysis work flow. These processes are made discoverable through a series of commands that can be sent to the server. With the command GetCapabilities returning an overview of the of the different services that are available at the server and also additional information on keywords, title, the name of the provider and physical address.

The available processes each have an unique id, which in turn can be queried through the use of the DescribeProcess command. This returns the meta data defined for the specific process including the needed input(s), format of the input(s) and the format of the output(s). Both the GetCapabilities and the DescribeProcess commands returns a human readable meta data document in the XML format (OGC 2008).

The WPS standard supports the use of a multitude of protocols for communication between user and server, among them the Web Service Description Language (WSDL)

and SOAP (Previously known as Simple Object Access Protocol) (Fu and Sun 2011). These two methods will not be investigated further as they will not be utilised in the subsequent WPS product.

2.2.2 REST

Instead of using the WSDL or SOAP protocol, communication between user and server is done through the Representational State Transfer (REST) protocol using a http GET transfer. Where the WDSL and SOAP protocols wrap requests and responses in a SOAP XML wrapper, REST is designed to transmit information over http without the XML wrapper that SOAP needs. This is done by sending the command as a URL, which can be read by the server. This URL contains the different parameters that the service or process supports in a Key-Value Pair (KVP) format separated by the & symbol (Fu and Sun 2011; OGC 2008).

2.2.3 Key-Value Pairs Used and Data Returned

The following is a theoretic walk-through of the key-value pairs used by the WPS set up as the product of this thesis. As these KVPs are of the nested type, the total depth and subsequent complexity of the explanation is outside the scope of this thesis. Instead I will focus only on the KVP parameters that are absolutely relevant for the end product. For a total description of the *OGC* WPS standard I refer to the official standard from *OGC*[1].

The mandatory GetCapababilites request has to be combined with the service key containing the value of the service type, in this case WPS. If other *OGC* services are present on the server, their GetCapabilities documents can be accessed by specifying their service value (WMS, WMTS, etc.).

Optionally the keys AcceptVersions (filtering processes by which WPS version they support) and language (Specifying the return language of the GetCapabilities document) can be included in the REST payload of the URL if supported by the server.

When receiving the GetCapabilities request from the user, the server returns an XML document containing the mandatory and supported optional sections shown in table 2.1. Each of these sections can contain further nested KVP, e.g. the *Operation Metadata* section provides information on which request type are implemented by the server and the section *Process Offerings* section provide the identity and a brief description of the processes offered (OGC 2008).

Table 2.2 shows the mandatory and optional sections returned by the DescribeProcess command, which can be combined with any process identifier available from the GetCapabilities return. Like the GetCapabilities return each section can contain several nested properties.

The DataInputs section (see table 2.3) contains the identifier of each input used as the key in the REST payload of the URL as well as the number of occurrences of the inputs. The InputFormChoice refers to the data type(s) supported by each input as described in the next section.

[1] `http://www.opengeospatial.org/standards/wps`

Name	Definition	Data type and value	Multiplicity and use
service	Service Identifier	Character String type, not empty, Shall contain WPS	One (mandatory)
version	Specification version for operation	Character String type, not empty, Shall contain 1.0.0	One (mandatory)
update sequence	Service metadata document version, having values that are "increased" whenever any change is made in service metadata document.	Character String type, not empty. Values are selected by each server implementation.	Zero or One (optional)
lang	Language Identifier	Character string type, not empty RFC4646 language code of the human readable text	One (mandatory)
Service Identification	Metadata about this specific server.	The schema of this section shall be the same as for all OWSs, as specified in Subclause 7.4.4 and owsServiceIdentification.xsd of [OGC 06-121r3].	One (mandatory)
Service Provider	Metadata about the organization operating this server.	The schema of this section shall be the same for all OWSs, as specified in Subclause 7.4.5 and owsServiceProvider.xsd of [OGC 06-121r3].	One (mandatory)
Operations Metadata	Metadata about the operations specified by this service and implemented by this server, including the URLs for operation requests.	The basic contents and organization of this section shall be almost the same as for all OWSs, as specified inSubclause 7.4.6 and owsOperationsMetadata.xsd of [OGC 06-121r3], modified as specified in subclause 8.3.2 below.	One (mandatory)
Process Offerings	Unordered list of brief descriptions of the processes offered by the server	ProcessOfferings data structure, see subclause 8.3.3 below.	One (mandatory)
Languages	Languages supported by the server	Languages data structure, see subclause 8.3.4 below.	One (mandatory)
WSDL	Location of a WSDL document describing all operations and processes offered by the server	WSDL data structure, see subclause 8.3.5 below.	Zero or One (optional)

Table 2.1: Mandatory and optional contents of *OGC* standards WPS GetCapabilities document (OGC 2008)

Name	Definition	Data type and values	Multiplicity and use
Identifier	Server unique process identifier	ows:CodeType	One (mandatory)
Title	Process title	Character string type	One (mandatory)
Abstract	Abstract description	Character string type	Zero or one (optional)
Metadata	Metadata on process	ows:Metadata	Zero or more (optional)
Profile		URN type. E.g OGC:WPS:somename	Zero or more (optional)
processVersion	Version of the process	ows:VersionType	One (mandatory)
WSDL	Location of a WSDL document that describes this process	WSDL data structure	Zero or one (optional)
DataInputs	List of the required and optional inputs to this process	DataInputs data	Zero or one (optional)
ProcessOutputs	List of the required and optional outputs from executing this process	ProcessOutputs data	One (mandatory)
storeSupported	Indicates if complex data output(s) from this process can stored by the WPS server as web-accessible resources	Boolean type	Zero or one (optional)
statusSupported	Indicates if Execute operation response can be returned quickly with status information	Boolean type	Zero or one (optional)

Table 2.2: Mandatory and optional contents of *OGC* standards WPS DescribeProcess document. Modified from OGC 2008

Name	Definition	Data type and values	Multiplicity and use
Identifier	Process unique identifier	ows:CodeType	One (mandatory)
Title	Title of process	Character string type	One (mandatory)
Abstract	Abstract describing process	Character string type	Zero or one (optional)
minOccurs	Minimum number of times that values for this parameter are required	non-negative integer type	One (mandatory)
maxOccurs	Maximum number of times that this parameter may be present	positive integer type	One (mandatory)
Metadata	Reference to more metadata about this input	ows:Metadata	Zero or more (optional)
InputForm Choice	Identifies the type of this input, and provides supporting information	InputFormChoice data structure	One (mandatory)

Table 2.3: Mandatory and optional contents of the DataInputs section of the DescribeProcess document. Modified from OGC 2008

2.2.4 OGC WPS Data Types

The *OGC* WPS standard for data inputs supports three general types of data inputs, which can be seen in table 2.4. The product developed in tandem with this thesis only uses the ComplexData and LiteralData types, so only those two will be described in the following.

Name	Definition	Data type and values	Multiplicity and use
ComplexData	Indicates that this input shall be a complex data structure (such as a GML fragment), and provides lists of formats, encodings, and schemas supported	See table XX for Data structure for ComplexData	Zero or one (conditional). Only one of these shall be included
LiteralData	Indicates that this input shall be a simple literal value (such as an integer) that is embedded in the execute request, and describes the possible values	See table XX for Data structure for LiteralData	Zero or one(conditional). Only one of these shall be included
BoundingBoxData	Indicates that this input shall be a BoundingBox data structure that is embedded in execute request, and provides a list of the CRSs supported in these Bounding Boxes	No further description of BoundingBoxData as it is not used further	Zero or one(conditional). Only one of these shall be included

Table 2.4: Possible data types for OGC WPS DataInput and Outputs

ComplexData The ComplexData type is designed to contain data of a more complex nature than the LiteralData described later. ComplexData covers data types such as image types like .png, geospatial data formatted as GML or GeoJSON and all other formats that are supported by the MIME internet protocol (Klensin, T. Hansen, and Freed 2013).

Name	Definition	Data type and values	Multiplicity and use
MimeType	Format for process input or output	Character String type, not empty	One (mandatory)
Encoding	Encoding for process input or output	URI type	Zero or one (optional)
Schema	Schema document for process input or output	URI type	Zero or one (optional) Include when encoded usingXML schema

Table 2.5: The ComplexData type is described using this notification

The structure of the ComplexData section is shown in table 2.5. Here the MimeType refers to the format which is covered by the MIME internet protocol (Klensin, T. Hansen,

and Freed 2013), the Encoding refers to the character encoding that text input must be in, and Schema refers to XML encoding schema used when the input is encoded as such.

The ComplexData data type can used as both the default format or the supported. This enables a process to accept arbitrary number of formats as inputs or outputs. Lastly the ComplexData type can be assigned a maximumMegabytes value, which limits the size of the input to an integer value representing megabytes.

LiteralData The LiteralData type is designed to contain simple data types like integers, floats, character string, dates and other types represented in the *World Wide Web Consortium*(W3 n.d.) (*W3C*) XML Schema language standards specification (OGC 2008).

Name	Definition	Data type and values	Multiplicity and use
DataType	Data Type of this output (or input)	ows:DataType type	Zero or one (optional)
UOMs	List of units of measure supported for this numerical output (or input)	Units of Measure	Zero or one (optional)
LiteralValues Choices	Identifies type of literal input and provides supporting information	AllowedValuesAnyValue ValuesReference	One (mandatory)
DefaultValue	Default value of this input	CharacterString, not empty	Zero or one (optional)

Table 2.6: The LiteralData type is described using this notification

The data structure of a LiteralData input as described in the WPS standard (OGC 2008) can be seen in table 2.6. The DataType refers to the simple data types represented in the *W3C* XML Schema language standards specification, the UOMs refer to the units of measure supported for the input, the LiteralValues refer to which values in the form of a range, maximum or minimum can be accepted or if any value will work, and DefaltValue refer to an optional default value of the input.

Analogue to the DataInputs section, the Outputs section specifies an Identifier for the result, a title, an optional abstract, a optional metadata value, and an OutputFormChoice, which like the specifications for DataInputs in table 2.4 can contain either ComplexData, LiteralData or BoundingBox data, all with the same nested possible data types available as DataInputs.

Using GetCapabilities and DescribeProcess then returns the information needed to formulate an URL with a REST payload to request an Execute command for the wanted process. The Execute command is the last of the mandatory commands that the WPS should be able to handle (OGC 2008). The Execute command is used to run a specific process from the WPS by the client. Specifying the inputs along with other mandatory and optional parameters in an URL with REST is very similar to the DescribeProcess command parameters listed in table 2.2 (OGC 2008).

As a result of the Execute command, the WPS then returns a ExecuteResponse XML document containing the mandatory parameters along with any specified optionals as shown in table 2.7.

Name	**Definition**	**Data type and values**	**Multiplicity and use**
service	Service Identifier	Shall contain 'WPS'	One (mandatory)
version	Specification version for operation	Version of WPS standard	One (mandatory)
lang	Language identifier	RFC4646 language code of the human readable text	One (mandatory)
statusLocation	location where current ExecuteResponse document is stored	URL type	Zero or one (optional) Include when storeExecuteResponse= TRUE
serviceInstance	GetCapabilities URL of the WPS service which was invoked	URL type	One (mandatory)
Process	Process description	Compressed DescribeProcess KVP	One (mandatory)
Status	Execution status of this process	Created time, Accepted, Started, Paused, Succeeded, and failed	Except for Created time, only one can be present
DataInputs	List of inputs provided to this process execution	DataInputs data structure, see table 2.4	Zero or one (optional) Include if lineage=TRUE
OutputDefinitions	List of definitions of outputs desired from executing this process	OutputDefinitions see table	Zero or one (optional) Include if lineage=TRUE
ProcessOutputs	List of values of outputs from process execution	ProcessOutputs see table 2.4	Zero or one (optional) Include when process execution succeeded

Table 2.7: The ExecuteResponse document is returned to the client containing the mandatory parameters in the table along with any optional specified.

In the following section I will present the *ZOO-Project* framework and outline its functionality, specialities and stand-out features.

2.3 ZOO-Project Framework

The *ZOO-Project* framework is an open source implementation of the previously described *OGC* WPS standard. The implementation relies on three components, the *ZOO-Kernel*, the *ZOO-Services* (these are what in the *OGC* WPS standard is called processes) running on the kernel and the *ZOO API*, which is used for chaining of WPS processes.

2.3.1 Kernel

The *ZOO-Kernel* is a WPS server programmed in the *C* language running as a CGI program on top of a web server like *Apache*. The kernel takes inputs in the request form standardised by the *OGC* WPS standard described in section 2.2.1 as well as returns outputs conforming to the same standard.

The *ZOO-Project* stands out from many other WPS implementations by being a polyglot supporting processes written in multiple programming languages[2] as seen in table 2.8 (Evangelidis et al. 2014).

Language	**DataStructure**
C / C++	maps* M
Java	HashMap
Python	Dictionary
PHP	Array
Perl	Not defined
Ruby	Hash
Fortran	CHARACTER*(1024) M(10,30)
JavaScript	Object or Array

Table 2.8: Programming languages supported by the *ZOO-Kernel* and their respective KVP data structure (Modified from footnote to zoo www).

The *ZOO-Kernel* is configured using a main.cfg configuration file where options for the WPS, such as version, encoding, language, provider data, WPS title and abstract, along with more technical details such as directory paths to temporary file storage (used when storage of an Execute response is requested), the URL to access temporary files, and the server address, which is the URL to the *ZOO-Kernel* instance (Fenoy, Bozon, and Raghavan 2012).

The main section also allows for optional configuration of the kernel to access *MapServer* instance, Cross origin resource sharing, and database connection to responses.

2.3.2 Services

ZOO-Services are what in *OGC* standard terms is coined a process. In all further descriptions I will refer to *OGC* processes as services, when in the context of the *ZOO-Project*.

These services connects through the *ZOO-Kernel*, which handles the parsing of user defined inputs and the return of XML response including the process result data (Fenoy,

[2] `http://zoo-project.org/doc/_build/html/kernel/what.html`

Bozon, and Raghavan 2012).

A service can utilise any of the supported programming languages represented in table 2.8 and consists of a .zcfg *ZOO-Service* configuration file, and the source code of the service in one of the supported languages.

The .zcfg file consist of three different sections;

- A main section, containing values corresponding to table 2.2 as in the *OGC* standard, a serviceType key holding the used to implement the service, and also a serviceProvider key holding the file name of the source code without the extension
- A list of inputs and their metadata, corresponding the *OGC* standard in table 2.3 , which which further holds the optional data type values as described in table 2.4. For ComplexData these will hold parameters seen in table 2.5 and for LiteralData they will hold the parameters as seen in table 2.6
- The output section is analogue to the inputs section and contains the relevant parameters for the chosen data types

The main section contains the identifier of the service in [square brackets] as the very first line, which must also be the name of the function inside the source code. This way the source code can contain the functionality of several services, each with their own .zcfg file associated.

Likewise each input and output also has an identifier cast in square brackets. All the identifiers are used by the *ZOO-Kernel* to construct a KVP data structure in the format corresponding to the services programming language (see table 2.8) with the identifiers as key and the client defined inputs stored as values nested in a key [value] a level below. The output value is assigned to a key in the data structure of the language source code function.

This value is then served back to the client wrapped in a XML, ExecuteResponse, document which contains information on the time of execution, the executed process, location of the *ZOO-Kernel* CGI, execution status and the process outputs, all according to the *OGC* specification for the ExecuteResponse document as seen in table 2.7.

2.4 Data

This section will introduce the data used to calculate the line of sight, how it was used and the output data that is the result.

2.4.1 Raster

The DEM, which is at the base of all analysis performed in the product of this thesis, is a raster acquired from download.kortforsyningen.dk. A raster is a matrix structure, where each cell holds a value, usually used to convey images. In the context of GIS, a raster is usually a georeferenced raster matrix, where the pixels in the raster are defined in a spatial reference system (Matt Duckham and Worboys 2007).

Georeferenced rasters often include more than one band, each representing a specific data type, which overlaps. The Danish Height Model (DHM) is a DEM, which holds just one band containing a height value for each pixel.

The height values are stored as 32 bit floating point numbers, a data type that can hold numbers in the range $10^{\pm 38}$ and 6 to 9 significant decimal places (Higham 1996). This range and precision is also used when doing calculations on the values.

Later the data type will be transformed into 16 bit signed integer that can hold integers in the range $-1x2^{15} = -32,768$ to $2^{15} - 1 = 32,767$ for faster calculations (Higham 1996).

The overall raster can be divided into smaller raster called tiles. This is done to increase the speed of operations involving the rasters, so that a filter can be applied that selects only the relevant tiles (Regina O Obe 2015).

2.4.2 Sampling

From the georeferenced raster a sampling strategy needs to be employed that samples at both a detailed rate to ensure as good a fit as possible, as well as having a service that performs well. As each pixel is a 0.4 by 0.4 meter square, a sample every 0.4 meters is the maximum sampling frequency considered. To ensure the performance of the service, a frequency of one sample per 1 meter is considered 'good enough' detail wise and good for performance.

2.4.3 Line of Sight Using Points

As this service is envisioned as an official housing tax variable, the primary objective is a simple browser based tool to enable the quantification of the view at a specific location and height. A few web based line of sight tools that utilise the DHM exists already all of which are vector end products created by polling an underlying DEM at set intervals, determining the visibility of the specific point and then connecting the points with lines that are supposed to represent an uninterrupted line of sight.[3]

This methodology creates a good looking result of an uninterrupted line, which indicates that the visibility is determined continuously along the line, but as mentioned above, this is not the case. Depending on how often the underlying DEM is sampled, there is a significant potential for height variations between the sampling points.

The solution chosen by *Sweco* and *Septima*, which assumes continuous visibility until the next point on the line is evaluated, is problematic in a taxation context. That is why the service product at the centre of this thesis will not assume anything about the heights and visibility of any locations not sampled and will thus not aggregate point data together into line strings.

[3]Septima documentation: `http://labs.septima.dk/sigt/info.html`. Conversations with developers at *Sweco* confirms that they also poll.

2.5 Digital Elevation Models

To construct the raster which is used for the line of sight calculations (in this case the Danish Height Model), various kinds of original data like contour lines (e.g. from pre-digital maps), photogrammetry, Interferiometric Synthetic Aperture Radar and Light Detection and Ranging (LiDAR) (J. R. Jensen 2007) can be coerced into the raster format.

In the subsequent sections a description of how a DEM is created from initial LiDAR data to final raster is provided. This transformation of LiDAR point cloud to a DEM georeferenced raster is analogue to the creation of the DHM (Geodatastyrelsen 2015a; Geodatastyrelsen 2015b).

2.5.1 Collecting Data

The LiDAR system consists of the following components and excludes the software needed later for interpretation and post-processing of the data

- A laser, used to emit pulses of light
- Scanning mirror optics, a rotating mirror that reflects the emitted light pulse in a perpendicular-angle pattern to the flight direction
- A receiver, which records the return time and intensity of the return signal generated by the emitted pulse
- Differential Global Positioning System (DGPS), a highly accurate GPS to record the exact position of the laser
- A Inertial Measurement Unit (IMU), which is used to record the absolute orientation of the laser for each emitted pulse (J. R. Jensen 2007; Charaniya, Manduchi, and Lodha 2004; Koukoulas and Blackburn 2005)

For a national DEM like the DHM covering a large area, the most viable solution is to mount the LiDAR system on an aircraft to be able to cover the total area extent within the given time-frame of two years (Geodatastyrelsen 2015b).

2.5.2 Generating Point Cloud from Aircraft Mounted LiDAR System

To employ the LiDAR system on an aircraft, various parameters needs to be considered for both the actual sensors and the aircraft. In the following a walk-through of the involved parameters on the final data collection will be presented.

The emitted pulses are used to measure the height of the area that they hit. This is done by measuring the time it takes for a pulse to return to the receiver. The return time relates to the range as shown in equation 2.1

$$t = 2\frac{R}{c} \tag{2.1}$$

where t is the time it takes for the pulse to travel from the laser and back to the receiver, R is the range from the transmitter to the surface and c is the speed of light. From this equation R can be determined as in equation 2.2

$$R = \frac{1}{2}tc \tag{2.2}$$

The area scanned for each emitted pulse is an approximate circle, which is deformed according to the angle between the point on the surface surveyed and the perpendicular line from the surface to the aircraft. The diameter (Fp_{inst}) of this approximate circle can be determined from equation 2.3

$$Fp_{inst} = \frac{h}{\cos^2(\theta_{inst})}\gamma \tag{2.3}$$

where h is the height of laser above the surface, θ_{inst} is the scan angle, and γ is the divergence of the laser beam.

The specifications of the rotating scanning mirror determines which angular coverage perpendicular to the flight path the emitted pulses cover. The metric coverage on the surveyed surface or swath (sw) can be determined by equation 2.4 as

$$sw = 2h \tan\frac{\theta}{2} \tag{2.4}$$

where h is the aircraft's altitude above the ground and θ is the effective angular coverage of the rotating scanning mirror.

The *perpendicular resolution* or space between the pulses when hitting the surface perpendicular to the flight path is determined by the pulses per second frequency (PS), the aircraft's height above the surface h, the instantaneous angular scanning speed in radians per second (α_{inst}), and the instantaneous scan angle (θ_{inst}) as in equation 2.5

$$\textit{perpendicular resolution} = \frac{h}{\cos^2(\theta_{\text{inst}})} \times \frac{\alpha_{inst}}{PS} \tag{2.5}$$

The resolution along the flight path is determined by the ground speed (v) of the aircraft and the time used for one perpendicular line scan (t_{sc}) as in equation 2.6

$$\textit{along resolution} = vt_{sc} \tag{2.6}$$

Together the perpendicular and along path resolutions determines the density of the point cloud that is generated from the system. All the above equations are inspired by the work in (Wehr and Lohr 1999; J. R. Jensen 2007; Koukoulas and Blackburn 2005).

Individual Georeferencing To ensure the accuracy of the range calculation for the emitted pulse, each of them is georeferenced according to the the data from the IMU and the DGPS systems. The IMU determines the roll, pitch and heading of the aircraft, so that the direction of each pulse can be determined with a high degree of accuracy. The orientation information is combined with the the x, y, z location of the aircraft as

determined using DGPS. The DGPS works by having a ground based GPS station, which location has been surveyed to a very high precision, and a GPS mounted on the aircraft. As the ground based GPS station has been surveyed independently from the GPS, the difference between the GPS signal and the surveyed position can be subtracted from the aircraft mounted GPS position. This process increases the precision of the signal to < 5 cm (Wehr and Lohr 1999; J. R. Jensen 2007).

LiDAR Returns Each pulse emitted by the LiDAR system creates a return when it hits the ground and it is not necessarily a 1:1 ration between emitted pulse and returns. When a pulse hits a surface it interacts with the surface on an area sw as described in equation 2.4. If this area is completely flat, only one signal from the pulse will be returned, but if the sw area is a heterogeneous collection of angled surfaces like a tree canopy, the emitted signal will generate multiple return signals. In LiDAR terminology these returns are numbered according to the return timing of the signal, so that the first return (in this example the top of the canopy) is marked as the first return, there can subsequently be several levels of returns, each time another object is hit by the sw area. The last return signal will be the actual terrain.

From these multiple returns signal levels a terrain model can be put together from the 1st and 2end returns and a surface model including all features can be generated from the last returns (J. R. Jensen 2007).

From Point Cloud to Raster Creating a raster surface model from the 1st and 2end returns collected in the point cloud, can be done by interpolating from the points using the x, y, z information that each point contain. Common methods used for the interpolation is the Inverse Distance Weighting (IDW) that assigns a weight to points based on their proximity to the currently evaluated point, natural neighbour that is based on Voronoi polygons and nearest neighbour that simply takes the value of the closest 'real' measured point (Bater and Coops 2009; Guo et al. 2010).

2.6 Trigonometry

The issues of calculating visibility and compensating for the curvature of earth when calculating visibility, can both be solved in the realm of trigonometry and specifically the domain of the right-angled triangle.

Below I will account for the theory behind calculating visibility by an angular relationship and the influence of distance from the line of sight start point on the effect of earth curvature.

2.6.1 Calculating Visibility

Determining visibility for a target point has to account for the height and distance related to the original observation point of all points that are between the original point of observation and the target.

To account for these factors the problem is reduced to a trigonometry issue, where the angle at the vertex created by the hypotenuse and adjacent in a right angled triangle created by difference in observation and target height and distance to point is calculated for each point (see figure 2.1) (Young 2012; Haverkort, Toma, and Zhuang 2009; Fisher 1993).

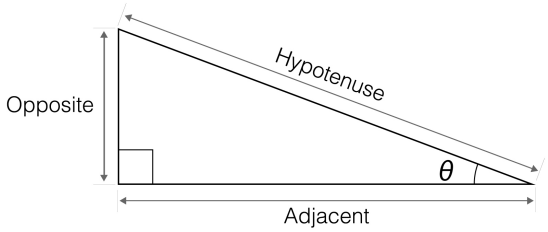

Figure 2.1: Terminology of the right-angled triangle.

To calculate the angle at the vertex, the trigonometry functions describing a right-angled triangle is used. For the specific vertex where hypotenuse and adjacent meet, the tangent to the angle is equal to the relationship between the opposite and adjacent sides in the right-angled triangle as in equation 2.7:

$$\tan\theta = \frac{opposite}{adjacent} \tag{2.7}$$

By using the inverted function arctan on the relationship as in equation 2.8 θ can be determined.

$$\theta = \arctan\left(\frac{opposite}{adjacent}\right) \tag{2.8}$$

As can be seen from figure 2.2 each sampled point along the line of sight (id_1 - id_n) has the angle θ calculated using the trigonometry relationship in equation 2.8.

On the same figure it can also be seen that the θ angle for id_9 is apparently smaller than the θ for id_n, but point id_n is obviously still not visible. This is because θ calculated to points that are below the initial observation point has a negative angle, so while θ gets more obtuse while the target point moves further below the initial observation point, the numeric value gets smaller as the negative number goes higher.

After calculating θ, determining the visibility is a matter of evaluating if any point, which is closer to the original observation point, has a higher angle than the evaluated point, and if this is the case, then the point is not visible as the line of sight is blocked by one or more points (see figure 2.3) (Haverkort, Toma, and Zhuang 2009; Fisher 1993).

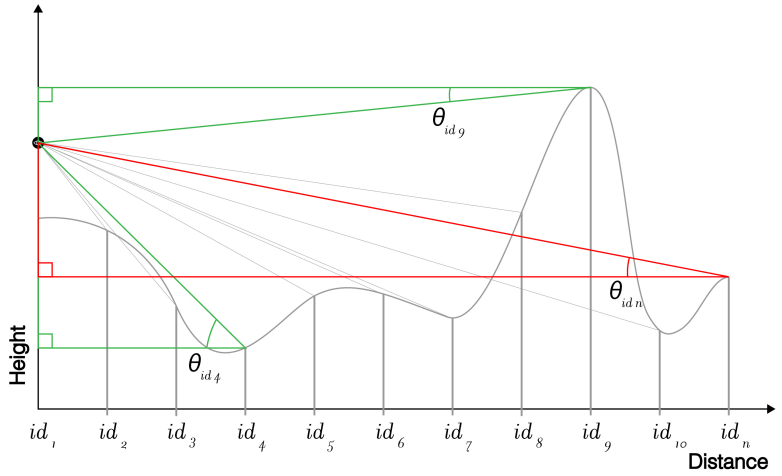

Figure 2.2: Calculating θ at each point.

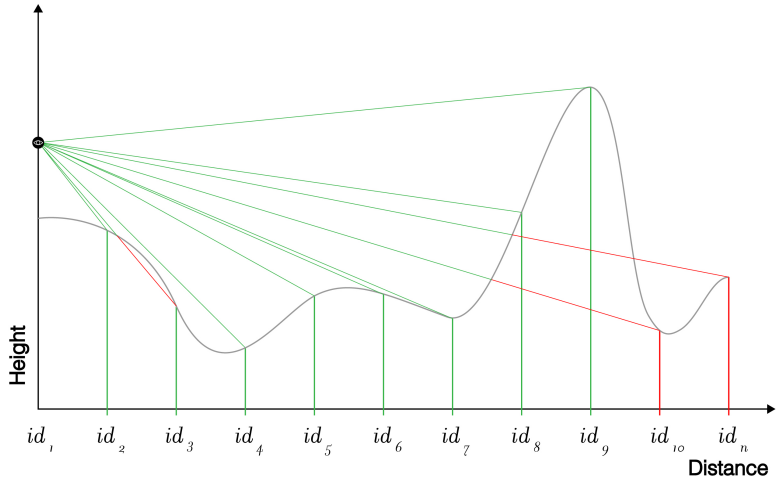

Figure 2.3: Visible (green) and invisible points (red). Inspired by "Extending the Applicability of Viewsheds in Landscape Planning"(Fisher 2006)

2.6.2 Compensation for Earth Curvature

As the line of sight calculation is done across the surface of the earth, the influence of the curvature of the earth surface increases exponentially as a function of distance (if the earth is thought of as completely round any ways). To compensate for the earth curvature when calculating visibility the curvature effect at each point is subtracted from the point height. The influence is visualised in figure 2.4.

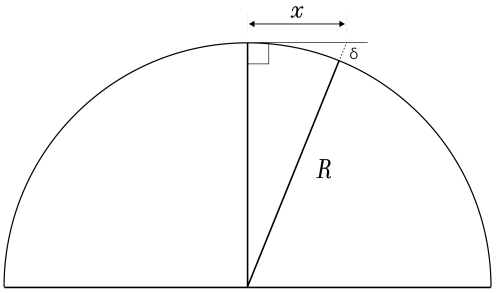

Figure 2.4: Influence of earth curvature calculated using the Pythagorean theorem.

The right-angled triangle is formed by the distance (x), the earth radius (R) and $R + \delta$. The δ can be calculated using the Pythagorean theorem as seen in equation 2.9 (Young 2012; Sullivan 2011).

$$(R + \delta)^2 = R^2 + x^2 \Rightarrow \delta = \sqrt{R^2 + x^2} - R \tag{2.9}$$

The calculated effect is subtracted from the height of all points.

2.7 Languages Used and Evaluated

Programming languages are designed to interact with the underlying hardware and operating system and they do this at different levels of abstraction to the underlying hardware. Low-level languages such as machine language interacts directly with the hardware, and can be incredibly fast at execution time, but be equally incredible slow to write and interpret as a human, also machine language can not easily be transferred to other machines.

Higher-level languages such as *Python*, *Java* and *C* sacrifice some of the speed of execution to enhance readability and writability. The high-level languages relates to the hardware through several abstraction layers, where the shared common is the bare hardware, the operating system and on top of that often a compiler and even a virtual machine. At each step from the initial source code until execution on the hardware, the code gets transformed into code readable by the abstraction layer below (Sebesta 2015).

While an actual functionality walk-through of the different languages introduced in this section is beyond the scope of this thesis, an overview of the general design principles and syntax will be provided along with some of the terms used.

2.7.1 Python - An Interpreted Language

Python is a relatively recent object-oriented interpreted scripting language. As opposed to *Java*, which is a compiled language, the interpreted nature of *Python* means that the source code is interpreted at execution time. In some cases this leads to slower execution compared to compiled languages.

On the other hand, the fact that the source code is not compiled means that for the programmer can iterate on code faster and smother as their is no time consuming compile time (Sebesta 2015; Westra 2013).

Python also features an extension interface, that makes it easy to extend the functionality of the core language. This is done by importing external modules.

In this thesis only few of the *Python* languages functionality will be used. The most important will be presented below.

```
1  import json
2
3  def chili_properties(chilies):
4      print("\n\nChili's and their strength:")
5      for plant, strength in chilies.items():
6          print("\nPlant: %s" % plant)
7          print("Strength: %s" % strength)
8
9
10 chilies = {'Bhut jolokia': '1.58M SHU',
11            'Naga Viper': '1.4M SHU',
12            'Carolina Reaper': '2.2M SHU',}
13
14 chili_properties(chilies)
15
16 how_hot_is_this_chili = "\nThe Bhut jolokia is " + chilies['Bhut jolokia'] + " strong!"
17
18 print(how_hot_is_this_chili)
19
20 print("\nJSON formated chilies\n" + json.dumps(chilies, sort_keys=True, indent=4,
   ↪  separators=(',', ': ')))
```

Listing 2.1: A module imported, a *Python* function defined and dictionary defined and used.

In the thesis product *Python* will act as a mean of communication between the *ZOO-Project* framework and the *PostgreSQL* database. This involves importing modules for non-core functionality, the use of dictionaries to hold and input data to the *ZOO-Kernel* and the format of the *Python* code. Listing 2.1 is used in the following to illustrate these pythonic concepts.

Custom Function When creating a *Python* WPS process the *ZOO-Kernel* expects that the source code contains a custom function with the same name as the the as the serviceProvider parameter in the corresponding .zcfg file. A function is a block of code that is easy to reuse when called again and as such facilitates the reuse of code (the "Don't Repeat Yourself" principle) (Herman 2013).

A function is prefixed in the header by the reserved keyword def, then follows the function name and a parameter list in parenthesis. The naming and number of parameters in the function has to correspond to the inputs used in the body of the function. In listing 2.1 the function chili_properties is defined, which takes exactly one argument as parameter, the variable chilies. The function header is ended with a :.

Below the header, the body is indented two spaces and the chilies parameter is the only parameter referenced.

Dictionaries The dictionary type is a simplified database structure, that associates any data with a look-up value, called a key. The dictionary syntax uses {} to encapsulate the structure, inside the keys are quoted in '' and are on the left of the :, and the value is on the right (see listing 2.1, where the chilies variable holds the dictionary referenced in the function) (Herman 2013).

Specific data contained under a key is accessed by the name of the variable holding the dictionary followed by the key in angle-brackets as in line 16 of listing 2.1.

The dictionary type is the pythonic way to define KVPs.

Modules In *Python* modules are used to extend the functionality (adding extra functions) of the core language. On line 1 in listing 2.1 a module is imported using the syntax import module_name. This module then provides the function json.dumps, which is used to transform the chilies dictionary into a JSON formatted string.

2.7.2 SQL - A Language to Query Structured Data

As opposed to the two other programming languages described, Structured Query Language (SQL), is used to express programs using symbolic logic to derive a result. Whereas the two other languages could be described as procedural, the SQL language is declarative as in the programmer specifies the parameters that the result must comply to. When using such a program the user asks a question, which the program attempts to answer by regarding the previously mentioned parameters as a set of rules and facts (Sebesta 2015).

As such the program does not exactly state how the result is to be computed but instead describe the form of the result. It is the job of the RDBMS's query planner to decide to decide the most efficient plan to obtain the requested result.

```
1 SELECT email FROM users WHERE name = 'Billy';
```

Listing 2.2: Basic SQL syntax.

users		
id	**email**	**name**
1	metus.In@Donecest.org	Troy
2	montes.nascetur@lacusQuisque.com	Billy
3	nisi.nibh@dui.com	Billy
4	mus@pellentesquemassalobortis.com	Tove
5	commodo@gravida.ca	Rune

Table 2.9: A table, users, containing the names and emails of users.

To query a table relation like the one in table 2.9, a set of specific parameters and rules can be set up that a result would have to fulfil, e.g show all email addresses for users named Billy would, in SQL syntax[4], be written as in listing 2.2

where SELECT email denotes the column, FROM users the table, and the filter WHERE name = 'Billy' denotes rows that fulfil the condition that the value under name must be 'Billy'. The output of this query is shown in table 2.10. Note that information on the name associated with the email address was not requested, so only the email value is returned.

email
montes.nascetur@lacusQuisque.com
nisi.nibh@dui.com

Table 2.10: Result of of query in listing 2.2.

Queries can also be expanded to create new relations that include more than one table by joining them together on a common key. Table 2.11 contains publications that the users from table 2.9 subscribes to. The common key between these two table is the column id from the user table and the column user_id from the subscription table (Date 2015).

To utilise this common key when querying the data a join between the two table relations can be made. In listing 2.3 an INNER JOIN that combines the two tables based on equal values in the id and user_id columns is created.

The INNER JOIN is used to return a new relations table consisting of all rows that share the common key, in this case the condition u.id = s.user_id.

The INNER JOIN is just one of many join types used to query table relations. LEFT JOIN returns all rows from the left table in the query and the rows from the right that match the common key. RIGHT JOIN is the mirror join of the LEFT JOIN. The FULL OUTER JOIN returns all rows from both table relations, but only combine rows where there is a matching key (Beaulieu 2009).

[4]In the following *PostgreSQL* specific SQL is used, which conforms very close to the ISO/IEC 9075 "Database Language SQL" standard (Dar et al. 2015)

subscription		
id	**user_id**	**magazine**
1	2	Luctus Ut Pellentesque Magazine
2	2	Porttitor Interdum News
3	1	Non Enim Commodo Limited
4	2	Eu Accumsan Daily
5	3	Cras Lorem Lorem Update

Table 2.11: Magazine subscriptions.

```
SELECT name, email, magazine FROM users u INNER JOIN subscription s ON  u.id =
↪  s.user_id;
```

Listing 2.3: INNER JOIN returning rows that have matching id = user_id.

Extending the traditional capabilities of the *PostgreSQL* RDBMS, spatial data can be be stored and queried using the *PostGIS* extension. Analogue to the above queries, spatial data can be queried in a similar manner to traditional table relations.

Querying the relationship between a line string and a raster to return only raster cells that are intersected by the line string can be done with an INNER JOIN in a similar fashion as the example in listing 2.3. If the raster is table raster and the raster cells are rows, rast and the linestring it table line and the geometry is row geom (respectively grid and red line in figure 2.5), then returning the raster cells that are intersected by the line is done using the SQL in listing 2.4.

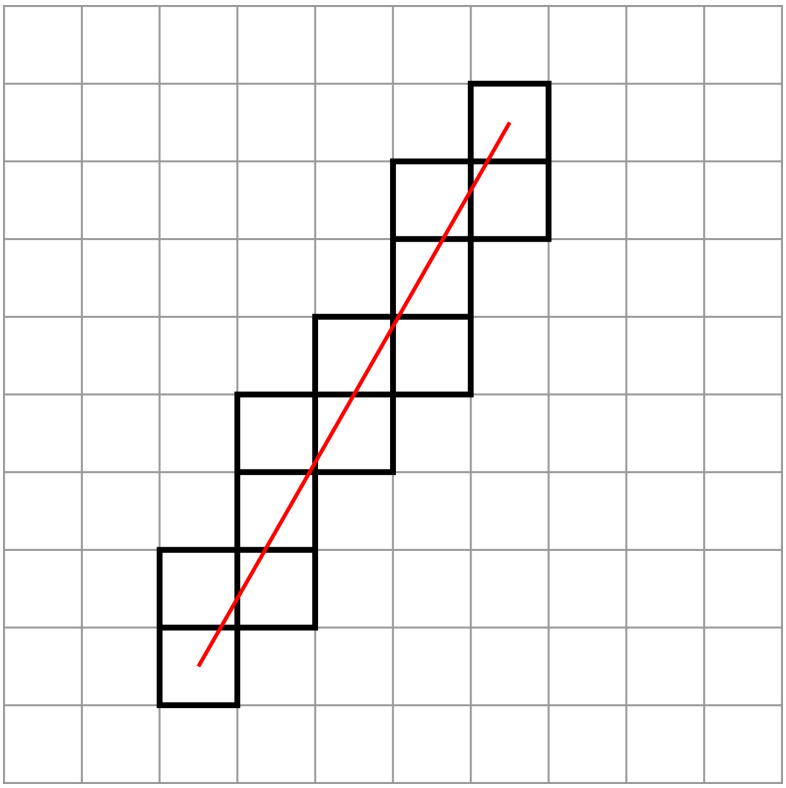

Figure 2.5: Spatial relationship between line and raster.

```
SELECT rast FROM rasters r INNER JOIN line l ON  ST_Intersects(r.rast, l.geom);
```

Listing 2.4: INNER JOIN returning raster cells that are intersected by LineString.

The result of this query is the raster cells or rows rast marked in figure 2.5 with the thick outline.

2.7.3 PL/pgSQL

As the standard *PostgreSQL* SQL is not a procedural but a declarative language the traditional flow control statements LOOP, WHILE, FOR and FOREACH are not available. Instead of including these directly in the SQL, *PostgreSQL* includes a procedural programming language, *PL/pgSQL*, that, in practice, extend the declarative SQL language with a procedural part[5] (Dar et al. 2015).

```
1  CREATE OR REPLACE FUNCTION public.for_loop_through_query(n integer DEFAULT 10)
2   RETURNS void
3   LANGUAGE plpgsql
4  AS $function$
5  DECLARE
6     rec RECORD;
7  BEGIN
8     FOR rec in SELECT name
9         FROM users
10        ORDER BY name
11        LIMIT n
12    LOOP
13   CASE WHEN rec.name = 'Billy' THEN RAISE NOTICE '%', 'BAD CUSTOMER!' ;
14        ELSE RAISE NOTICE '%', rec.name;
15   END CASE;
16    END LOOP;
17 END;
18 $function$
```

Listing 2.5: An example of a custom *PL/pgSQL* function that declares a function parameter, initiates a LOOP and uses a CASE WHEN conditional statement.

In listing 2.5 the *PL/pgSQL* functionality used in creating the thesis product is demonstrated on the tables previously used to demonstrate the SQL syntax in section 2.7.2, table 2.9. Starting with creating the function and defining the input parameters on the first line. A name for_loop_through_query is defined and the input is aliased as n declared as type integer. As the return of the function is not at table or a record, the return type is set to VOID in line 2.

In line 3, the specific PL language is declared, here *PL/pgSQL*. The actual function body is the defined between the $function$ at line 4 and 18.

In the function body, the variable rec, is declared as a type RECORD before the SQL evaluation begins. The RECORD type is a pseudo-type, that just indicates that it is a unspecified row type (Dar et al. 2015). The SQL evaluation starts at BEGIN and ends with END;.

A FOR LOOP is initialised on line 8 that loops through the names in the users table (see table 2.9). A limit on the number of records returned is declared in line 11, where the n alias is substituted with the user input when initiating the function.

On line 13 a CASE WHEN statement is initialised that when meeting the criteria that a loop record matches the string 'Billy' THEN it raises a notice BAD CUSTOMER!

[5]Other procedural language extensions are available for *PostgresSQL*, like *PL/Python* (based on *Python*), *PL/R* (based on *R*) and many more

indicating some very bad experiences with users named Billy. To handle all other CASEs the ELSE keyword is employed in this case raising a notice with the name of the user.

The CASE WHEN statement is ended using END CASE on line 15 and the loop is terminated with END LOOP on line 16. When using the function as in SELECT for_loop_through_query(5);, where 5 is input as the parameter value (5 substitutes n in the function) the resulting output is shown in listing 2.6.

```
NOTICE:  BAD CUSTOMER!
NOTICE:  BAD CUSTOMER!
NOTICE:  Dan
NOTICE:  Ditte
NOTICE:  Poul
```

Listing 2.6: The output of for_loop_through_query(5) (5 as the input).

2.8 Data Structure

The data, which underlies all output of the WPS, is stored in a *PostgreSQL* database, which is queried by the user to determine visibility. The input data provided by the user is in the GML format and the output from the WPS process is provided in the GeoJSON format. Below I will describe the characteristics of the different systems.

2.8.1 Relational Database Management System

Storage and Accountability The relational database management system model is designed to reliably store a large amount of data in a centralised server and at the same time accommodate large number of concurrent users accessing said data.

The main development point for the RDBMDS model has been focused on efficiency and integrity - especially integrity (Sumathi and Esakkirajan 2007; Stones and Matthew 2006). This integrity permeates all aspects of how data in the database can be accessed and modified.

The data can be read, inserted, updated and deleted as the privileges of the user allow. These user roles can be defined in a very granular fashion based on individual users or as groups sharing privileges.

By incorporating a user based access to the database it is possible to create a trail of actions for each of these users (Dar et al. 2015). Registering information on the event (like a SELECT query), time stamp and user.This would be especially useful in an environment where auditing is critical to support a legally binding system like the Danish housing and property tax.

This functionality will not be incorporated into the WPS to be created along with the thesis, but it will be brought up as part of the discussion on using the line of sight WPS in a housing and property tax environment.

The database system is deleted, updated and queried through the use of the SQL language presented in section 2.7.2.

2.8.2 Indexes

One of the advantages of using a RDBMS like *PostgreSQL* is the possibility of employing indices on the stored data, which can have a profound influence on the speed of a query or other statement.

The data stored in the database is unstructured and as such any query will have to look trough all records before returning a record set. this situation can be optimised with the use of indices. An index can be created on one or more columns in a table and builds an ordered structure, which for large tables are faster to query. Many different index types exist, below I will describe just two, the B-tree index and the GiST index which is used to index geometries in the *PostGIS* enabled database.

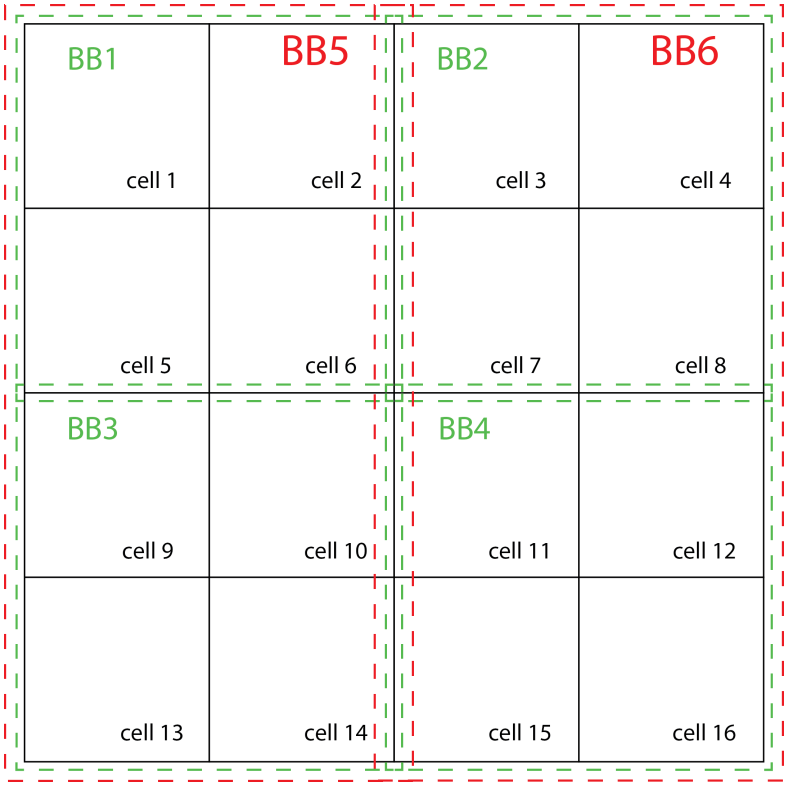

Figure 2.6: Grouping of raster cells based on closeness create a *GiST* index (figure inspired by Westra 2013).

A B-tree (Balanced tree) index is organised like a tree with branches where each subsequent branch from the stem will hold fewer and fewer elements, until only one indexable element is left. This last branch then hold the location of the records that have the specified value in the indexed column. The B-tree index is not used any further in the thesis product, but is included as it is the default index in many RDBMS (Beaulieu 2009).

In many ways the GiST index, which is used to index spatially related geometries, works similarly to the previously described B-tree index. The GiST index uses the minimum bounding rectangle around each geometry and then groups geometries together by closeness as shown in figure 2.6. Here the individual raster cells are each registered by a bounding box at the lowest level, and then they are grouped together on iterative higher levels by closeness. The level above the individual cells is the green bounding boxes BB1-BB4 followed by the next level - the red bounding boxes BB5-BB6 (Westra 2013; Regina O Obe 2015; Marquez 2015).

This enables using the GiST index similarly to the B-tree, where the query on geometries moves through the nested branches of the index until it reaches the last bounding box to encapsulate the requested geometries avoiding a costly sequential scan where each row (containing a raster value) must be examined.

2.8.3 Common Table Expression

The Common Table Expression (CTE) is a SQL syntax used to define a temporary table, which is only available in the query in which it is included. These CTEs can be chained and as such can be used to filter and alter data sent through the chain. Listing 2.7 shows a short example where the CTEs name is defined with the WITH keyword and inside the parathesis the structure of the CTE is defined.

```
1 WITH dummy_cte AS
2 (
3 SELECT part_type, price * 0.25 AS price_inc_vat
4 FROM volvo_parts
5 )
6
7 SELECT part_type, price_inc_vat FROM dummy_cte;
```

Listing 2.7: CTE syntax.

The CTE is ended with a) and now the SELECT statement can be used to query the CTE as a normal table.

2.8.4 GeoJSON

GeoJSON is used as the output format for the WPS process and is a standard based on the JSON format, used to represent geographic features as points, lines and polygon. The GeoJSON object consists of name-value pairs, where some of the names are required. "type" has to exist and have a value of either "Point", "MultiPoint", "LineString", "MultiLineString", "Polygon", "MultiPolygon", "GeometryCollection", "Feature", or "FeatureCollection". If the object contains a type "FeatureCollection", as is the case for all GeoJSON object output from the WPS, the GeoJSON is a feature collection and must then contain "name" features, which contains an array of GeoJSON features (Richardson and Amundsen 2013; Butler, Gillies, and Schaub 2016). Outputting to GeoJSON has been chosen as many standard *JavaScript* mapping libraries supports GeoJSON.

2.8.5 GML

The GML format is used to handle the inputs of the start and endpoint for the line of sight calculation. GML is based on XML and is used to represent geographic features like points, lines and polygons. The standard was created by the *OGC* to facilitate the transport of geospatial data over the internet (OGC 2002; Regina O Obe 2015) .

GML exists in different formats with 2.1.2 and 3.1.1 being the commonly used (Regina O Obe 2015). For the line of sight process input, GML 2.1.2 was chosen, as this format is often supported in WFS systems, possibly easing the interconnection of the line of sight service with other *OGC* services (Regina O Obe 2015).

The GML standard is able to handle geometries represented in the 'Simple Features Specification' (OGC 1999).

2.9 Housing Tax

Currently the *Danish Tax and Customs Authorities* (*SKAT*) base their housing and property tax on a valuation of the housing or property, which takes its value from an average based on a not specifically defined average of the market value of 'comparable properties' traded in a not defined area near the evaluated property. This average is then adjusted to the date of the valuation by a projection that incorporates market trading trends (Skatteministeriet 2013a; P. E. Jensen et al. 2014). For each valuation case, *SKAT* can employ a valuation method that they deem effective.

The subjective nature of which properties and housing to include in the average is also the subject of much debate and seem to the fluctuate among property that for a layman might seem very similar (Drachmann 2014).

As such there is no market independent valuation, which can be problematic especially when valuing housing or property that is atypical and not traded regularly, such that a reasonable average can be obtained.

These challenges to the housing and property tax has been recognised by the previous government, which initiated a expert hearing to explore the possible structure of new and more objective valuation of property. One of the main results from this hearing was a new focus on geodata to be included in the development of an automated valuation model, with an emphasis on repeatability of calculations (P. E. Jensen et al. 2014).

In this thesis I will explore the possible value of a line of sight model as a parameter in a new housing and property tax model.

Chapter 3

Methodology

The following chapter will deal with the practical implementation of the thesis project. Detailing the process from installing the required frameworks to coding and setting up the actual product.

3.1 Installing ZOO-Project WPS and Dependencies

The line of sight service product, which is one of the goals of this thesis, runs within the *ZOO-Project* framework (Fenoy, Bozon, and Raghavan 2012; ZOO-Project team 2016). Like *PyWPS* (PyWPS 2016), *52°North WPS* (52° North GmbH 2016) and *GeoServer* (Open Source Geospatial Foundation 2016), *ZOO-Project* WPS is a framework that enables spatial processing to be handled as a service.

The framework can be installed on a variety of operating systems, including several flavours of *Linux*, *Windows* and *OS X* but in the following I will describe the installation process for *Ubuntu Server 14.04*, which is a command line only server edition of the popular *Ubuntu Linux* operating system. *Ubuntu* was chosen as its popularity entails a large amount of available software packages and a healthy community that offers help and advice on issues (Haines 2015; Tsai et al. 2016). Version 14.04 is a long term support version released in April 2014 with support until 2019[1].

The server edition of Ubuntu was chosen to avoid the unnecessary system overhead of a graphical user interface and to be able to easily remote deploy the system.

3.1.1 Dependencies

The *ZOO-Project* WPS is dependent on several open source software packages to be able to perform its job. Below I will introduce the most important packages, their use and how to install them.

[1] Official *Canonical* support periods: `http://www.ubuntu.com/info/release-end-of-life`

flex Flex is a tool which recognises lexical patterns in text. When finding defined patterns, flex executes an associated piece of *C* code. For the *ZOO-Project* framework, this is used when parsing .zcfg and .cfg metadata files[2]. On Ubuntu Server the package is installed using the the apt−get package manager with the command sudo apt−get install flex. sudo is used to run the command with superuser rights, which is necessary to obtain the needed privileges to write and execute within the shell. All following commands associated with the *apt-get* package manager are assumed to be prefixed with sudo.

Bison Bison is a general purpose parser generator and is used in conjunction with flex to parse the metadata .zcfg and .cfg files associated with the service (Fenoy, Bozon, and Raghavan 2012). The Bison package was installed using the apt−get install bison command.

FastCGI *FastCGI* or *Fast Common Gateway* Interface is used along with the *Apache* http-server for the client to access server-based executables, in this case the *ZOO-Kernel* through the zoo_loader.cgi. The FastCGI package was installed using the apt−get install libfcgi −dev command.

Libxml2 *Libxml2* is the XML C parser and tool kit used to create and parse the metalanguage[3]. In the *ZOO-Project* Kernel, it is used to format outputs as XML. The XML C package was installed using the apt−get install libxml2 libxml2−dev command.

cURL *cURL* is needed when transferring large and complex data as inputs or outputs in the *ZOO-Project* framework. The *cURL* package is installed with the apt−get install curl command.

OpenSSL The *OpenSSL* package is a tool kit that enables the use of *Transport Layer Security* (TLS) and the *Secure Sockets Layer* protocols. In effect it enables communication through the https standard, which is a uniform resource identifier scheme with an encryption layer that provides authentication security between client and server. The package is installed with the command apt−get install openssl command.

GNU Autoconf *Autoconf* is used to produce configure scripts for the building and installation of software on the system. This is done by testing the target system's already installed components against the dependencies of the source code and configuring the makefile as to correspond to the target system. This way the software in question can be installed with as little user interaction as possible. The *GNU Autoconf* package is installed with the command apt−get install autoconf.

[2] *ZOO-Project* SVN `http://zoo-project.org/trac/browser/trunk`
[3] Official page: `http://xmlsoft.org`

Apache2 Web Server *Apache* is the web server that serves the content from server to client. The main process associated with *Apache* is the httpd daemon, which handles requests from users and serves content back. All settings and permissions are set up in the associated plain text .conf files, where permissions for the WPS .cgi files can also be set. The *Apache* web is installed using the apt−get install apache2 command.

Python The python−dev package adds header files, a static library and especially important for *ZOO-Project* development tools for building Python modules, extending the Python interpreter or embedding Python in applications. python−dev is installed using the command apt−get install python−dev

Spidermonkey JavaScript *Spidermonkey* is the *Mozilla Foundations* JavaScript engine. The JavaScript header files and library is used when compiling the *ZOO-Project* kernel to enable the use of the JavaScript language for creating and linking services in the WPS framework. The JavaScript engine is installed using the command apt−get install libmozjs185.

Build-essential *build-essential* is a meta package that contains a list of dependency packages to be installed, which are needed for compiling programs from source code. In the case of the *ZOO-Project* WPS the source code is written in the *C* language[4]. To compile the *ZOO-Project* source code the contents of the *build-essential* meta package is needed. Of notable importance are the *gcc* (*Gnu C Compiler*) compiler package, the *libc-dev, libc6-dev* packages containing an embedded version of the *GNU C* library with development libraries and header files, and the *make* package that is used in conjunction with a Makefile, which is a script that controls the compiling of the source code, and the previously mentioned *gcc* compiler.

3.1.2 Geospatial Data Abstraction Library

Geospatial Data Abstraction Library (*GDAL*) is a library that provides spatial processing algorithms, translation between geospatial file formats and as such act as the standard link between much open source geospatial software. For both the *ZOO-Project* WPS, and *PostGIS* I need to install the developer files which are used when incorporating the *GDAL* into other software (GDAL Development Team 2016).

To install the *GDAL* library I will first need to employ the command add−apt−repository, to add a specialised GIS repository from the *UbuntuGIS* team. *UbuntuGIS* employs two branches for publishing GIS related packages - the stable and the unstable branch (The UbuntuGIS Team 2016). These two branches offer different versions of many of the same GIS related packages, and normally one would assume that the unstable branch should only be used for testing new features with a high risk of bugs and instability, whereas the stable branch should offer stable and tested packages. In the case of these two branches the stable branch unfortunately is several years old and looks to be basically

[4] `http://zoo-project.org/docs/kernel/what.html`

unmaintained, resulting in outdated packages that are not compatible with many other current dependencies. The unstable branch on the other hand gets regular maintenance and updates, leaving it in effect as the only viable repository for GIS packages.

To add the *UbuntuGIS Unstable* repository the command add−apt−repository ppa:ubuntugis/ubuntugis− is executed and afterwards apt−get update to include the new repository. After the new repositories has been linked, installation of the *GDAL* developer library is done using the command apt−get install libgdal1 −dev.

3.1.3 GEOS

Geometry Engine - Open Source (*GEOS*) is a *Open Source Geospatial Foundation* project[5] that provides the spatial geometry functions on which the *PostGIS* extension relies on when executing functions involving intersects, length, distance and others, and defines the data types Point, MultiPoint, LinesString, MultiLineString, Polygon, MultiPolygon and GeometryCollection (which themselves are based of the *OGC Simple Features Specifications(OGC 1999)*). The *GEOS* library is installed using the command apt−get install libgeos −dev.

3.1.4 PostgreSQL with PostGIS

Choosing, which versions of the *PostgreSQL* and *PostGIS* packages to install depends on the feature set needed to complete the task at hand and balance between stability and speed.

Older maintained versions of the software might provide best stability at the cost that fewer features are present and speed is not optimised. The opposite is the case for the newest versions which offer the maximum number of features and optimised speed at the cost of stability.

Lastly the different types and versions of the operating system can also limit the availability of software.

PostgreSQL version (9.5) offers general performance enhancements, but also more specific enhancements that benefits working in the spatial realm like the improvements made to *GiST* indices (The PostgreSQL Global Development Group 2016a) (see section 2.8.2 for the theory behind *GiST* indices).

For the *PostGIS* extension, the just released version 2.2 added new functions, which was considered possibly beneficial as parts of the final product like in-database raster processing and new spatial clustering functions and also a general code optimisation, which should add to the speed of spatial calculations (The PostgreSQL Global Development Group 2016b).

As this thesis is partially about the performance of a RDBMS both *PostgreSQL* and *PostGIS* was installed in the latest stable release version at the time of the thesis start.

The actual installation of the software was done from *The PostgreSQL Global Development Group (PGDG)*'s repository, which was added to the Ubuntu Server package system along with the repositories public encryption key using the command in listing 3.1.

[5] `https://trac.osgeo.org/geos/`

```
sudo sh -c 'echo "deb http://apt.postgresql.org/pub/repos/apt trusty-pgdg main" >>
 ↪  /etc/apt/sources.list';
sudo apt-get install wget ca-certificates;
wget --quiet -O - https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key add
 ↪  -;
```

Listing 3.1: Adding PGDG repository and public encryption key.

After connecting to the repository it is necessary to update the contents of the package system before installing both *PostGIS 2.2* and *PostgreSQL 9.5* using the commands in listing 3.2.

```
sudo apt-get update;
sudo apt-get install postgresql-9.5-postgis-2.2 postgresql-contrib-9.5;
```

Listing 3.2: Updating repository and installing the database system.

In listing 3.2 I also install the postgresql−contrib−9.5 package, which supplies some basic administrative tools through the *adminpack extension*. With the *PostgreSQL* package installed, the next step is to add the adminpack extension to the standard postgres database using the postgres user using the command CREATE EXTENSION adminpack; from within any *PostgreSQL* client or the psql bin. The postgres user is only meant for accessing the database software, so next I'll create a user that only has the minimum of privileges for an associated database and nothing else. sudo -u postgres psql

The new user with the proper rights and an associated database is created using the commands CREATE ROLE zoo_wps WITH login; CREATE DATABASE zoo_wps WITH OWNER zoo_ Now a new database accessible only by the postgres superuser and the zoo_wps user has been created. If ever another service or program needs database access, a separate user and database would be created with the steps above, to enable a database system where each user has as few privileges as possible.

To enable the *PostGIS* extension I run the command CREATE EXTENSION postgis;. This completes the setup of the *PostgreSQL*/*PostGIS* framework. A few utilities to load raster data into the database are still needed though.

3.1.5 Installing the ZOO-Project Package

After installing all dependencies, installation of the *ZOO-Project* WPS framework was done by using svn[6] to download the latest version into a directory zoo−project using the command svn checkout http://svn.zoo−project.org/svn/trunk zoo−project.

Provided with the download of the *ZOO-Project* is *cgic*[7], a *C* library for creating CGI-based programs. In this case it is used as the basis for the zoo_loader.cgi executable which handles input and output to the kernel.

[6]svn (*subversion*) is used as a revision control system by *ZOO-Project* `http://zoo-project.org/docs/contribute/dev.html`

[7]https://www.boutell.com/cgic/#whatis

The *cgic* library is implemented by navigating to the directory zoo−project/thirds/cgic206/ and running the make command to compile the library locate its contents at its proper locations.

The *ZOO-Kernel* is installed by navigating to the directory ../../ zoo−project/zoo−kernel/, creating a configure script by running the command autoconf (see paragraph on *GNU Autoconf*), running the configure script using the command ./configure −−with−js=yes −−with−python=yes −− will create the Makefile required to compile the *ZOO-Kernel*. The arguments after ./configure are included to hook into *JavaScript* and *GEOS*, which are both optional inclusions.

The created Makefile has to be edited though as one of the pointers to dependencies within the script points to a /usr/lib64 directory whereas /usr/lib is the correct directory path.

Lastly *ZOO-Project* is compiled by running the Makefile using the command make and the result is placed in the proper locations by executing sudo make install.

3.2 Selection and Preparation of Data

The line of sight calculation which is covered in this thesis, relies on raster data on elevation and the processing of these data in *PostgreSQL* with the spatial functions provided by the *PostGIS* extension.

Selecting a subset of the total Danish dataset on elevation is needed as the resources to store all them is not available for this student.

As the raster data type is suboptimal for the purpose intended here, the data type will be transformed before importing it in to the database. Below I will present the selected data, its transformation and the import method.

3.2.1 Selection of Area for Raster Coverage

As the physical data size of a download and subsequent storage of a total coverage *DHM* is very large and would require an investment in hardware beyond the scope of this thesis (the *DHM* covering all of Denmark consists of 628, 10 by 10 kilometre raster products, each the size of approximately 2 gigabytes, equalling around 1.3 terabytes) I have chosen to focus on just one of the 10 by 10 kilometre cells of which the *DHM* is divided into. The 10 by 10 kilometre cell that was chosen covers an area around Hillerød, Birkerød and Lillerød north of Copenhagen and has the extent 710000, 6190000 : 720000, 6200000 in the ETRS89 UTM 32N projection. When downloaded and unpacked from the delivered zip file the size of the content is 2.5 gigabyte.

3.2.2 Preparation of Data

Getting several gigabytes of raster into the *PostgreSQL*/*PostGIS* database requires an automated process to be practically possible. For this purpose there are several small utilities installed along with the *GDAL* and *PostGIS* package, which I will use for the preparation and import of data into the database.

The *PostGIS* package comes with a small utility or bin called *raster2pgsql* that can export raster files in all *GDAL* supported file formats into a specified database, schema and table. The *raster2pgsql* can also, at execution time, tile the raster data into even smaller tiles as to improve performance when querying the raster layer (Regina O Obe 2015)

gdalinfo gdalinfo is part of the *GDAL* package and is used to extract the metadata on *GDAL* supported raster files. The metadata for the selected raster data will be used to optimise the sql code described later, to specify bounding box values for the service, determine optimal tile size and determining the data type of the raster.

When using the command gdalinfo DSM_1km_6190_710.tif the output seen in listing 3.3 shows that the pixel size is 0.4 as the heigh of the raster in meters divided by the height in pixels ($\frac{1000\ meters}{2500\ pixels} = 0.4$) and that the projection is ETRS89 UTM 32N (Rosenkranz 2014). The data is of type FLOAT32.

```
Driver: GTiff/GeoTIFF
Files: /Users/jonaspedersen/Desktop/rasters/raster/DSM_1km_6190_710.tif
Size is 2500, 2500
Coordinate System is:
PROJCS["ETRS89 / UTM zone 32N",
    GEOGCS["ETRS89",
        DATUM["European_Terrestrial_Reference_System_1989",
            SPHEROID["GRS 1980",6378137,298.2572221010002,
                AUTHORITY["EPSG","7019"]],
            TOWGS84[0,0,0,0,0,0,0],
            AUTHORITY["EPSG","6258"]],
        PRIMEM["Greenwich",0],
        UNIT["degree",0.0174532925199433],
        AUTHORITY["EPSG","4258"]],
    PROJECTION["Transverse_Mercator"],
    PARAMETER["latitude_of_origin",0],
    PARAMETER["central_meridian",9],
    PARAMETER["scale_factor",0.9996],
    PARAMETER["false_easting",500000],
    PARAMETER["false_northing",0],
    UNIT["metre",1,
        AUTHORITY["EPSG","9001"]],
    AUTHORITY["EPSG","25832"]]
Origin = (710000.000000000000000,6191000.000000000000000)
Pixel Size = (0.400000000000000,-0.400000000000000)
Metadata:
  AREA_OR_POINT=Area
Image Structure Metadata:
  COMPRESSION=DEFLATE
  INTERLEAVE=BAND
Corner Coordinates:
Upper Left  (  710000.000, 6191000.000) ( 12d21' 7.79"E, 55d49' 7.77"N)
Lower Left  (  710000.000, 6190000.000) ( 12d21' 5.01"E, 55d48'35.48"N)
Upper Right (  711000.000, 6191000.000) ( 12d22' 5.15"E, 55d49' 6.20"N)
Lower Right (  711000.000, 6190000.000) ( 12d22' 2.36"E, 55d48'33.91"N)
Center      (  710500.000, 6190500.000) ( 12d21'35.08"E, 55d48'50.84"N)
Band 1 Block=256x256 Type=Float32, ColorInterp=Gray
  NoData Value=-9999
```

Listing 3.3: Output of the gdalinfo command.

Raster Data Type Transformation As seen in the previous section, the values contained in the raster are of the type Float32, which is used internally in the *GDAL* package to denominate a 32 bit floating point number. As shown in the section 2.4.1, the data type Float32 is an expensive type, which is far too precise for the purposes of this thesis and associated products. To save space and speed up processing I will convert the GeoTIFF files before importing them into the database.

When converting the raster from a type float in meters to a integer type, I also want to convert the data to centimetres by multiplying by 100 so that it conforms well to the new format. This should be reasonable for representing the Danish landscape heights, which lie within a range of approximately -700 to around 17,000 centimetres (see theory section 2.4.1 on data types).

The conversion is done using the bin gdal_calc.py, installed using apt−get install python−gdal, wrapped in a bash script to iterate over all the .tif files downloaded from *download.kortforsyningen.dk* as is shown in listing 3.4.

```
#!/bin/bash
for i in *.tif
do
gdal_calc.py -A $i --calc="A*100" --type=Int16 --outfile=dhm_new_type/"new_$i"
done
```

Listing 3.4: bash script to convert raster data type.

The −A option specifies the input file, which in this case is an iteration over all the .tif files in the directory using the variable constructor $ with i as defined in the loop. As the A iterator changes along with the i iterator the A used within the −−calc is being calculated for each .tif as a multiplication of 100. With the −−type flag and Int16 value I specify that the output raster values should be of the type 16 bit signed integer instead of the original 32 bit floating point. Lastly I specify a directory for all the files and that the filename should be the same as the $i iterator prefixed with new_ and be put in the directory dhm_new_type using the −−outfile flag.

The resulting new .tif files have a size of 1.1 gigabyte instead of the original 2.5 gigabyte.

3.2.3 Importing Data into PostgreSQL

Loading the new .tif files with the 16 bit signed integer data type into the database is done with the *raster2pgsql* combined with the *psql* tool. As command line tools in *Linux* can be chained together I will combine the output of the *raster2pgsql* with the *psql* tool with the special pipe command (|). The *raster2pgsql* is part the *GDAL* package and the *psql* tool is a command line *PosgreSQL* client. As the other tools previously used, the functionality of the tools are defined by the specific flags used. Below I will explain these flags used.

raster2pgsql When using the *raster2pgsql* command to export file system based .tif files into the database environment I use the flags shown in listing 3.5. To establish a way of referencing back to the original .tif files the flag −F adds a column in the target table with the name from_file specified with the −n flag. The −I flag ensures that a *GiST* type spatial index is created on the raster column of the target table. The −c flag creates a table in the database schema that acts as target for the export. The −s flag combined with the value 25832 sets the rasters projection to ETRS89 UTM 32N. The −C enables a set of constraints on the target raster table. These constraints are used in the raster_columns, which is a database wide registration of rasters and their associated metadata.

```
raster2pgsql -F -n from_file -I -Y -c -s 25832 -C -t 125x125 *.tif
rasters.dhm04 | psql -d jonaspedersen
```

Listing 3.5: Flags used when exporting .tif files into the database using *raster2pgsql* and *psql*.

If new raster data were to be appended to this dataset, it would need to adhere to these constraints. Alternatively the constraints can be dropped and then re-added after appending. The −t flag combined with the value 125x125 creates tiles from the input .tif file that are 125 by 125 pixels. The small size compared to the original files 2500 by 2500 pixels extent is very practical when doing analysis on the raster later as it enables the query planner to only select a very small subset of data for analysis. Finally the −Y flag enables the use of the SQL COPY instead of the INSERT command. The COPY command is much faster for bulk loading of large datasets, such as the ones being dealt with in this thesis, as it is optimised to have less overhead when loading large datasets. The disadvantage is that, unlike INSERT, the COPY command insert all rows as a single statement, so in case of an error the entire load has to be redone.

psql The execution of the raster2pgsql command in conjunction with the specific flags described above returns a very large sql output that when run from any client, would create the raster table shown in table 3.1. To avoid saving the output of the raster2pgsql command to a very large and very impracticable .sql file I pipe the output into another *Linux* command. As seen in listing 3.5 I append the raster2pgsql with the symbol | and the new commands. Using this operator, I direct the output of the raster2pgsql command into the psql command. The psql command is used in conjunction with the −d flag, which is followed by the name of the target database resulting in the output being run a sql code directly in the *psql* client creating the table in the target database as seen in table 3.1.

3.3 Working in PostgreSQL/PostGIS Environment

After setting up a working environment that is able to handle the development of the stated service within the *PostgreSQL*, *PostGIS* and *Python* framework, the next step is

dhm04		
rid	**rast**	**from_file**
1	01000001009A9...	new_DSM_1km_6190_710.tif
2	01000001009A9...	new_DSM_1km_6190_710.tif
3	01000001009A9...	new_DSM_1km_6190_710.tif
4	01000001009A9...	new_DSM_1km_6190_710.tif
5	01000001009A9...	new_DSM_1km_6190_710.tif

Table 3.1: *PostgreSQL* table structure for imported raster data. The binary blobs in the rast column have been truncated to fit.

to code the service that is to run in the WPS.

The service consists of roughly 3 interacting parts of code:

- The *PostgreSQL* sql code. This is the code that interacts directly with the *DHM* raster and extracts the relevant height values based on the user input.
- The *Python* code, which enables user input to be transferred from the URL user input into the *PostgreSQL* database
- The *ZOO-Project* ZCFG service configuration file, which defines the data types for both inputs and outputs

Beyond the three main bullet points that defines this WPS I will look at the performance of the *PostgreSQL* database and examine the specific methodologies used when writing the main *PostgreSQL* sql code and how *Python* connects to the database.

The chapter will conclude with the details behind getting the line of sight service up and running to be accessible using REST payloads through an URL, an evaluation of the returned line of sight result and finally a performance evaluation of the service created compared to *Septima's* and *Sweco's* own line of sight application.

3.3.1 Creating Indexes to Optimise Search Speed

As noted earlier in the theory section on indexes (page 28), one of the advantages of the *PostgreSQL*/*PostGIS* database is the possibility to use indices to speed up the selection of data. Different index types covers different data scenarios and in this specific scenario, trying to speed up querying a raster table, a *GiST* index will be appropriate (see section 2.8.2). As covered in the theory section, the GiST index works by indexing on the minimum bounding box for each geometry, and then grouping these by closeness. As a raster table contains no geometries, the *GiST* index is instead created based on the output of the *PostGIS* function ST_ConvexHull, which creates a small (bounding box) geometry around each raster pixel. To create the index the sql code in listing 3.6 is run (see figure 2.6 on page 28 for reference).

Having an *GiST* index on the raster column of our table will enable a faster execution of the sql code that will be presented in section 3.4.2 on page 42.

```
CREATE INDEX dhm04_ST_ConvexHull_idx
  ON rasters.dhm04
  USING gist
  (ST_ConvexHull(rast));
```

Listing 3.6: SQL code to create the index dhm04_ST_ConvexHull_idx on the rast column.

3.4 Creating Line of Sight PL/pgSQL function

Having imported the *DHM* raster data and created an index to speed up queries on the data, the next step is to create the actual code that will extract the relevant values from the raster and transform them into binary values representing either a visible or invisible point as seen from the user defined elevation above the start point.

3.4.1 Establishing a Line String Representing the LoS

As this line of sight service is based on a line that starts at the observation point and ends at the target of interest, the first step is to construct a line between these two points as seen in listing 3.7. In line 9 a variable is declared that contains the result of the function ST_MakeLine. The two inputs are represented by the $1 (the start point) and $2 (the target point). These inputs are defined by the user when calling the function. The specifics of how variables are defined and used will be presented in section 3.4.8 on page 49.

Encapsulating the ST_MakeLine is the function ST_SetSRID, which sets the SRID of the line to 25832 (ETRS89 UTM 32 N), which is the same as the underlying DEM.

Adding a Measure Element The vector line which was created in the previous paragraph defines the initial area of interest where it intersects the underlying DEM, but as the function described in this section needs a more specific dimension to poll the DEM, the line will be split into points.

This is a two step process where, first a measure element is added to the line using the ST_AddMeasure function, which takes as inputs the line geometry, a start position and an end position. In listing 3.7 line 16 the measure element is added to the previously created line from the start of the line to the end, defined by using the st_length function to retrieve the length of the line geometry.

The second step as can be seen on line 22 in listing 3.8 is to use the ST_LocateAlong function to create a geometry collection that contains a point for each step of i. i is defined by the user as the resolution in the final function and is a range of numbers generated using the generate_series function, which in this case is the numbers between 0 and the length of the line in increments of i. For the default value of 1 (this is only defined later in the .zcfg file for the WPS process) the st_locatealong function will collect a point for each meter of the line length.

```
8  BEGIN
9  line := ST_SetSRID(ST_MakeLine($1,$2),25832);
10
11
12 RETURN QUERY
13
14 WITH measured_line AS
15 (
16 SELECT ST_AddMeasure(line,0,ST_Length(line)) AS geom,
17        generate_series(0, ST_Length(line)::INT,$4::INT) AS i
18 ),
```

Listing 3.7: Create a line between start point and target and add measure element.

```
20 points_from_mline AS
21 (
22 SELECT (ST_Dump(ST_LocateAlong(measured_line.geom,i))).geom AS geom
23 FROM   measured_line
24 ),
```

Listing 3.8: Using st_locatealong and st_dump to get points at specified intervals along the line.

As the output of the st_locatealong function is a multipoint collection and because geometry collections are difficult to work with in the later functions, the output is encapsulated in the ST_Dump (also line 22 in listing 3.8) function, which expands the geometry collection to its individual parts. The result being many rows each containing a point instead of one row containing a multipoint collection.

3.4.2 Using PostGIS Functions to Access Raster Values

After the linestring representing the line of sight was dissolved into points, the next step is to extract the values of the underlying DEM where the points, extracted in the previous section, intersects the DEM. The CTE height seen in listing 3.9 shows the process of extracting the raster values using the function ST_Value. On line 29 in listing 3.9 the raster column of the DEM and point geometry created in section 3.4.1 is used as input in the ST_Value function. The ST_Value function then extracts the specific raster value where the point geometry intersects the raster. This value is then associated with the point geometry as the column cm.

To engage the index on the raster table and subsequently reduce the rows which need to be evaluated for the ST_Value it is prudent to use a WHERE clause as filter on the query. Here the function ST_Intersects (line 33 in listing 3.9) is used as the modifier, with the points and raster rows as input. This clause combined with the tile size determined at import (see section 3.2.3) reduces the amount of data to be evaluated to only the raster rows that are intersected by the points, greatly reducing the data that has to be evaluated (see section 3.4.2, page 42).

In the height CTE on line 30 a column id containing a sequential integer for each

```
26 height AS
27 (
28 SELECT geom,
29        ST_Value(dhm04.rast,points_from_mline.geom) AS cm,
30        row_number() over ()                        AS id
31 FROM   rasters.dhm04,
32        points_from_mline
33 WHERE  ST_Intersects(points_from_mline.geom,dhm04.rast)
34 ),
```

Listing 3.9: Extracting values from the raster and assigning id to points based on its position counted from the observation point.

point is created using the window function row_number () over (). The numbering is related to each points position from the original observation point such that the start point will be assigned the lowest id and the observation target (end point) will get the highest number.

3.4.3 Using PostGIS Functions to Calculate Earth Curvature Influence

The height_incl_start_curve CTE shown in listing 3.10 serves two purposes:

- Adjusting height to include earth curvature
- Setting the observation height relative to the actual height

As described in section 2.6.2 on page 20 the influence of the earth curvature on the points is a function of distance from the original observation point. To include the effect of the curvature, the extracted height value in the cm column from the height CTE needs to be modified. This is done in the next CTE height_incl_start_curve by passing each value through the function shown in equation 3.1:

$$height\ adjusted = height - \left(\frac{distance\ to\ point^2}{2 * earth\ radius}\right) \tag{3.1}$$

Reading listing 3.10 inside out from innermost to outermost parenthesis starting at line 50 to 452, the point geometry with the lowest id is selected. As the id is assigned based on the position of the point relative to start and end of the line, it is known that the point associated with the minimum id is also the first point on the line.

Using the ST_Distance function on line 48 with the start point just found by the association to the minimum id and point in the currently evaluated row, results in the distance to the current point being calculated.

As the projection of the point table is in ETRS89 UTM 32 N, the units of measurement is meters and as such the distance needs to be multiplied by 100 to achieve the same units as the DEM which is centimetres. This result is then, using the power function, raised to the power of 2, which is then divided by 2 and multiplied with the earth radius in centimetres. This calculation of the effect of the earth curvature on each point is then subtracted from the original height column cm.

```
36 height_incl_start_curve AS
37 (
38 SELECT geom,
39        id,
40        degrees_id,
41        CASE
42           WHEN
43             id = (SELECT min(id)
44                   FROM height)
45           THEN cm+$3
46           ELSE
47             SELECT cm -
48               ((power((ST_Distance(
49               (
50                SELECT geom
51                FROM   height
52                WHERE  id IN (SELECT min(id) FROM height)
53                ),geom)*100),2 ))
54               /
55               (2*637100000))
56           END AS cm
57 FROM   height
58 ),
```

Listing 3.10: Including the effect of earth curvature on elevation.

To factor in a user defined observation height a conditional expression is used. The CASE WHEN expression[8] that is initiated on line 41 in listing 3.10 is only fulfilled for the point having the minimum id (this will always be the observation point). When the expression is true the THEN statement is activated with the result that the height of the observation (start) point is summed together with what the user has input in the function (represented by the variable $3 in line 45). Being able to set a custom observation height enables the line of sight to be calculated correctly later in the function.

3.4.4 Adding a Z Value to the Points

In the CTE points_3d seen in listing 3.11 the ST_MakePoint function is used to add the cm value as a Z value to the already existing X and Y coordinates of the points. The points with their new Z value are then used in the CTE pitch_at_point to calculate the angle at the vertex where the adjacent and hypotenuse meet (see figure 2.1 on page 18 for the terminology of the right-angled triangle).

3.4.5 Using PostgreSQL Built-in Trigonometric Functions to Calculate Angles to Each Point

Evaluating the visibility of each point relies on the ability of representing visibility as a function of both distance and height of the point evaluated relative to the start point. As described in section 2.6 on page 17, the angle created between the line representing the

[8] http://www.postgresql.org/docs/current/static/functions-conditional.html

```
60 points_3d AS
61 (
62 SELECT ST_SetSRID( ST_MakePoint(ST_x(geom),ST_y(geom), cm ),25832) AS geom,
63        id
64 FROM   height_incl_start_curve
65 ),
```

Listing 3.11: Adding a Z dimension to the points.

point of observation and the observation target, and the line representing the observation point and observation point nadir incorporates both height and distance and expresses the visibility of the current point when relating this angle to all previous points on the line, so that if any previous point has a higher degree angle, the point is not visible (see figure 2.2 on page 19).

To calculate the angle for each point extracted from the line the equation 3.2 is used:

$$tanx = \left(\frac{(observation\ height - height\ at\ evaluated\ point)}{distance\ from\ observation\ point\ to\ evaluatedpoint} \right) \tag{3.2}$$

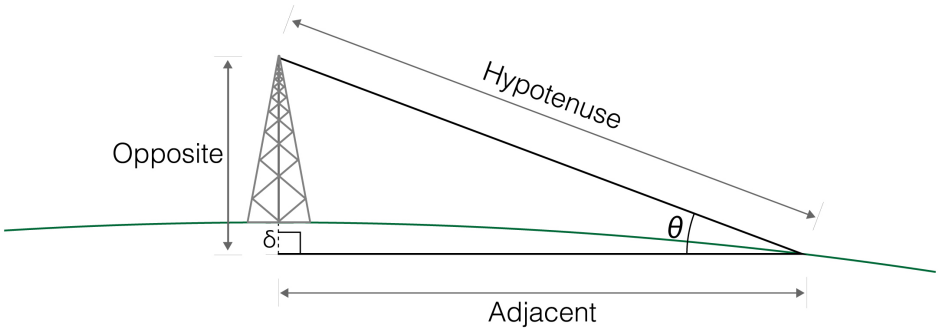

Figure 3.1: The relationship between distance and height to point from the start point including the influence of earth curvature.

To implement equation 3.2 in *PostgreSQL* sql the observation point needs to be selected. In listing 3.12, line 72 to 78, this point is selected using a WHERE clause to limit the selection to only points having the minimum id from a sub-select that selects only the minimum id. Using the ST_Z function the height at the observation point and evaluated point is extracted. The two are then subtracted to retrieve the *opposite* (see figure 3.1) in the triangle (line 70).

```
67  pitch_at_point AS
68  (
69  SELECT degrees(atan(
70          ((ST_Z(points_3d.geom) - ST_Z(
71            (
72             SELECT geom
73             FROM   points_3d
74             WHERE  id IN
75                    (
76                     SELECT min(id)
77                     FROM   points_3d
78                    )
79            )))/100)
80
81            /
82
83            nullif( ST_Distance(points_3d.geom,
84            (
85             SELECT geom
86             FROM   points_3d
87             WHERE  id IN
88                    (
89                     SELECT min(id)
90                     FROM   points_3d
91                    )
92            )), 0 )
93                     )
94                )
95     AS degrees,
96        geom,
97        id
98  FROM  points_3d
99  ),
```

Listing 3.12: Calculating the angle for each point.

Similarly the distance to the evaluated point, used as the *adjacent* (see figure 3.1) in the calculation, a sub-select is used to find the minimum id, which is also the observation point, and use the ST_Distance function to calculate the distance to the evaluated point (see line line 83 to 94). Figure 3.1 shows the relationship between the distance and height. The opposite is then divided with the adjacent, but as the the distance (adjacent) is calculated once with the value for observation point as input to both parameters, resulting in a division by zero error, the function NULLIF is used in line 83 to replace the zero value with NULL to eliminate this error.
Lastly, in line 69, the angle is calculated using the function atan, which returns the inverse tangent in radians and the function degrees which converts this to degrees.

3.4.6 PostgreSQL Window Function

Having calculated the viewing angle from the observation point to all evaluated points in the pitch_at_point CTE, the next step to determine if a point is visible is to evaluate, for each point, if any other point, which is closer to the observation point, has a higher viewing angle. If any point is found with a higher viewing angle, the currently evaluated

point is not visible (see figure 2.2 on page 19).
As calculating the distance again from the observation point to evaluated point is a computational expensive process, I instead opt to base the evaluation of closeness on the column id, which contains an integer representing the points position starting from the observation point (see section 3.4.2 on page 42).
To evaluate the above conditions I employ a window function[9] that for each point evaluates all previous points by id to find the maximum viewing angle in this pool and set the column visible to false if the maximum angle supersedes the currently evaluated point.

```
101 winfunc AS
102 (
103 SELECT id,
104        geom,
105        degrees,
106        visible
107 FROM   (
108         SELECT id,
109                geom,
110                degrees,
111                CASE
112                   WHEN degrees <= max(degrees) over (ORDER BY id ROWS BETWEEN unbounded
                       ↪  preceding AND 1 preceding)
113                   THEN FALSE
114                   ELSE TRUE
115                END AS visible
116         FROM   pitch_at_point
117       ) data
118 )
```

Listing 3.13: Window function to establish if angles calculated in previous CTE are lower than any angles closer to observation point.

In listing 3.13 starting at line 111 a conditional CASE WHEN expression is established to handle the output of the window function and set a boolean TRUE or FALSE value for the visible column. The window function starting at line 112 establishes a dynamic data frame from within which the maximum angle is determined (max(degrees)). The data frame is initialised by the over () keyword and is defined by ordering the id column and then establishing the extent of the frame with the keywords unbounded preceding, which is all rows before the current row until the first, and 1 preceding, which is the the row just before the currently evaluated (see figure 3.2 for a visual depiction of the evaluated extent using the previously mentioned keywords).
Each time the window is evaluated the maximum angle is found using the aggregate function max() and if the current row has a lower angle it is not visible, which is determined with the <= operator.

[9] http://www.postgresql.org/docs/current/static/sql-expressions.html#SYNTAX-WINDOW-FUNCTIONS

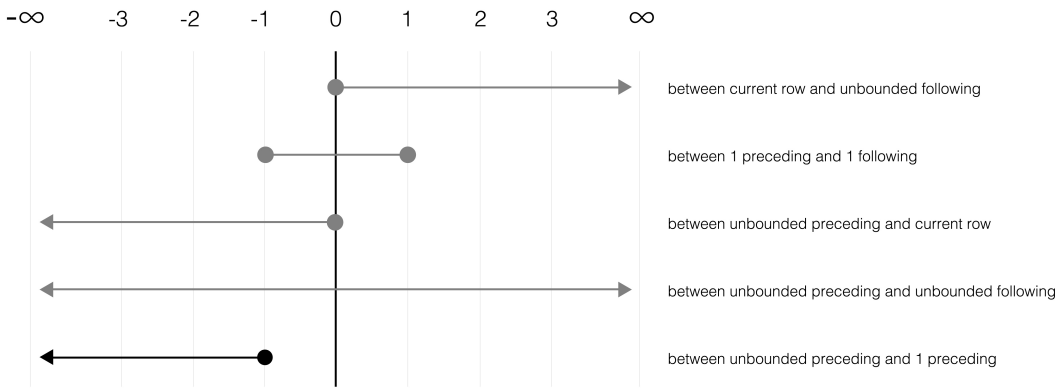

Figure 3.2: evaluated window keywords. Between unbounded preceding and 1 preceding highlighted.

3.4.7 Converting PostgreSQL table to GeoJSON output

Although outputting the visible points to a new *PostgreSQL* table or view would be the most rational choice for performance, subsequent storage and querybility, the potential for connectability with other frameworks like *JavaScript* suggests a different output route to be chosen. As the priority to easily incorporate the result into *JavaScript* based web map systems like *Sweco*'s *Spatial Suite* is high, the output chosen is the *GeoJSON* format (see section 2.8.4 page 29).
To convert the traditional table structure to GeoJSON, *PostgreSQL* contains a few special functions to deal with the *json* format. The goal of the code shown in listing 3.14 is to convert the table structure into *json* attribute/value pairs. This is done in sql by using SELECT statements to format the data properly into a *GeoJSON* feature collection (see section 2.8.4 page 29) and finally formatting it as a proper *json* object[10].
The row_to_json() function takes the column name as the attribute and the column content as the value, the function is used first in the innermost parenthesis on line 128 in listing 3.14. Here the column visible containing a boolean value is selected as a nested attribute/value pair under the properties attribute.
At the same level as the properties attribute, the geometry attribute is added by using the function ST_AsGeoJSON to transform the *PostGIS* geometry into a *GeoJSON* compatible object. This object is then cast into a json object using the :: operator. Still on the same level of the *json* structure, the standard *GeoJSON* attribute denoting the object type is declared by using SELECT on a fixed text string 'Feature' as the value and 'TYPE' as the key.

[10]The *PostGIS* to GeoJSON code is inspired by Regina Obe of *BostonGIS* `http://www.postgresonline.com/journal/archives/267-Creating-GeoJSON-Feature-Collections-with-JSON-and-PostGIS-functions.html`

```
121 SELECT row_to_json(fc) AS feature
122 FROM   (
123         SELECT 'FeatureCollection' AS TYPE,
124         array_to_json(array_agg(f)) AS features
125         FROM   (
126                 SELECT 'Feature' AS TYPE ,
127                  ST_AsGeoJSON(lg.geom)::json AS geometry,
128                  row_to_json((SELECT l FROM ( SELECT visible) AS l
129                              )) AS properties
130                 FROM   winfunc
131                 AS lg)
132         AS f)
133 AS fc;
```

Listing 3.14: Converting table output to *GeoJSON*.

Moving one level up in the *GeoJSON* structure and one parenthesis out in the sql, on line 123 in listing 3.14 I again declare a standard *GeoJSON* attribute denoting (again using SELECT on text string) the enclosing object TYPE as a 'FeatureCollection'. By the GeoJSON standards definition (section 2.8.4 on page 29) the 'FeatureCollection' value is designed to nest one or more 'Feature''s (as declared in the above paragraph).
To enable the nesting of several 'Feature' objects under the 'FeatureCollection' value I employ the array_agg in line 124, which concatenates the input values into an array structure. In this case all previously generated points and their properties, as constructed above, goes from a table structure to array. This array is then converted with the array_to_json aggregate function into a *json* structure.
Finally in the outermost parenthesis on line 121 the function row_to_json is used to convert the single row that is the output of the previous process into a *json* structure.
In the next section I will wrap the sql code in a function which will simplify the later *Python* script and keep most of the complexity inside the database.

3.4.8 Custom Functions in PostgreSQL

PostgreSQL allows the creation of custom functions, which can be written in multitude of languages, like *C*, *Perl*, *Python*, *Java*, *R* and also the built-in *Procedural Language PL/pgSQL* (see section 2.7.3 page 26).
With these languages one can achieve things that are not possible with raw sql code, like FOR LOOPS (which will not be used) and the declaration of variables, constants and function parameters, which I will use when wrapping the code described in the previous sections into a *PL/pgSQL* function.
On line 1 in listing 3.15 the function name is declared together with the schema where it is physically placed. In the parenthesis immediately following, the input parameters and their data type is declared, such that the function expects a start and end point, both of the geometry type (startpoint and endpoint), a resolution of type integer (the parameter res) and lastly a height of type double precision (the parameter height).
Also in the function definition header, the output of the function is defined with the

```
1  CREATE OR REPLACE FUNCTION PUBLIC.los( IN startpoint geometry,
2                                         IN endpoint geometry,
3                                         IN res INTEGER,
4                                         IN height DOUBLE PRECISION) returns
                                          ↪  TABLE(feature json) AS $body$
5  DECLARE
6  line geometry;
7
8  BEGIN
9  line := ST_SetSRID(ST_MakeLine($1,$2),25832);
10
11
12 RETURN QUERY
```

Listing 3.15: Start of custom *PL/pgSQL* function.

RETURNS keyword. In this case a TABLE is returned with the column feature of data type json.
After establishing the name, inputs and outputs of the function, the body of the function starts after the $body$ string. First, in the declaration section of the function, a variable that will hold the line is declared as a geometry type. After the declaration section, the keyword BEGIN indicates the beginning of the actual processing of values.
The input parameters are called from inside the function using the $n nomenclature, where n is equal to the position of the parameter in the definition. On line 9 in listing 3.15 the parameters startpoint and endpoint are called and as they are respectively the first and second parameter defined, they are called using $1 and $2.

```
135   END;
136   $body$ LANGUAGE plpgsql immutable strict cost 100 ROWS 1000;
137   ALTER FUNCTION PUBLIC.los(geometry,
138                         geometry,
139                         INTEGER,
140                         DOUBLE PRECISION) owner TO jonaspedersen;
```

Listing 3.16: End of custom *PL/pgSQL* function.

In listing 3.16, line 135, END, the antithesis to the BEGIN keyword appears and indicates the end of the processing section. On line 136 the language, plpgsql (*PL/pgSQL*), is defined and the functions volatility is classified. As this function will not modify the database and will return the same result given the same arguments, it can be classified as IMMUTABLE. Selecting the strictest volatility level will allow the *PostgreSQL* query planner to better optimise or even cache the query plan of the function.

3.5 Defining the Metadata for Inputs and Outputs

The *ZOO-Project* WPS framework defines a service and enables its discovery by the metadata provided from a ZOO configuration file (Fenoy, Bozon, and Raghavan 2012). Below I will describe how the inputs and outputs are defined in the configuration file.

3.5.1 The ZOO Configuration File (.zcfg)

To enable the discovery of services by the *ZOO-Kernel*, a ZOO configuration file (.zcfg) needs to be defined and created. The content of the file is accessed when the DescribeProcess and GetCapabilities requests are sent to the kernel. Here the .zcfg file contents are parsed by *flex* and *Bison* (see section 3.1.1) to create the *XML* formatted output.

```
1 [ExtractlineofsightGML]
2 Title = Return LoS (points).
3 Abstract = Calculate Line of Sight from a start- and end-point and a height.
4 processVersion = 1
5 storeSupported = true
6 statusSupported = true
7 serviceProvider = lineofsightGML
8 serviceType = Python
```

Listing 3.17: Meta data on Line of Sight service.

The top eight lines of the .zcfg file, shown in listing 3.17, defines the basic parameters for the service and how it is capable of interacting with the rest of the *ZOO-Project* framework.
In line 1 the identifier of the service, which is used when requesting the DescribeProcess and Execute commands is declared, in this case defined as ExtractlineofsightGML. The Title and Abstract parameters on line 2 and 3 are returned for both the DescribeProcess and Execute commands and are used for descriptive purposes.
Line 4 is an arbitrary version number assigned to this instance of the line of sight algorithm in this case 1. Line 5 enables or disables the ability of the service to store the result of a process on the server as an accessible resource and line 6 enables or disables messages on the current status of a process (for long running processes, the it would return a message on the service still being processed).
On line 7 and 8 the serviceProvider and the serviceType is declared where the provider is the file name of the file containing the actual code and the type is the coding language used, in this case the file is named lineofsightGML.py and the language is *Python*.
Occupying the greater part of the .zcfg document is the definition of the data inputs and outputs. Lines 10 through 49 in listing 3.18 defines the inputs that the service expects when running an Execute command.
These inputs consists of the three parameters; start point, end point (endpoint) and height (height). The start point parameter is defined with an identifier startpoint on line 10 and on line 13 and 14 the minimum and maximum number of start points are defined. As we are working with a line for which the line of sight will be calculated, only one point can serve as the start point input.
The input data type for the start point is defined on line 15 though 21, where the data type is defined as ComplexData (see section 2.2.4 on page 9 for details on *OGC* WPS data types). For the start point parameter, the complex data container specifies the GML data type in version 2.1.2 from line 29 through 31.

```
10  [startpoint]
11   Title = Input startpoint
12   Abstract = Input startpoint in GML format.
13   minOccurs = 1
14   maxOccurs = 1
15    <ComplexData>
16      <Default>
17        mimeType = text/xml
18        encoding = UTF-8
19        schema = http://schemas.opengis.net/gml/2.1.2/feature.xsd
20      </Default>
21    </ComplexData>
```

Listing 3.18: Defining the startpoint parameter.

Completely analogues to the start point parameter, the end point parameter is specified from line 22 through 33 in listing 3.19.

```
22  [endpoint]
23   Title = Input endpoint
24   Abstract = Input endpoint in GML format.
25   minOccurs = 1
26   maxOccurs = 1
27    <ComplexData>
28       <Default>
29        mimeType = text/xml
30        encoding = UTF-8
31        schema = http://schemas.opengis.net/gml/2.1.2/feature.xsd
32      </Default>
33    </ComplexData>
```

Listing 3.19: Defining the endpoint parameter.

The last input of the service is the height parameter, which is defined in listing 3.20 on line 34 through 45. as opposed to the start and end point parameters, the heigh parameter can occur either 0 or 1 time as a default value of 200 centimetres is defined (see listing 3.20 line 37-38 and line 41-44). The container LiteralData using integers as data type is used for the height parameter as the precision of the underlying DEM being used is capped at centimetres (see section 3.2.2).
Listing 3.21 shows the specification of the line of sight process output. Like the start and end point parameters, the output data is a ComplexData container, which in this case is set to the *json* format (line 51 through 56).
When accessing the data from the *Python* code later, the label [result] defined in line 48 will also be the the key in a *key/value* pair, where the output *json* data will be the value. The next section will deal with how the connection is made between the *ZOO-Project* WPS and the *PostgreSQL* database using the supported *Python* language.

```
34    [height]
35     Title = Input height
36     Abstract = Input height in centimeters.
37     minOccurs = 0
38     maxOccurs = 1
39      <LiteralData>
40        DataType = int
41      <Default>
42        value = 200
43        uom = centimeters
44      </Default>
45     </LiteralData>
```

Listing 3.20: Defining the height parameter.

```
47  <DataOutputs>
48    [result]
49     Title = geojson points
50     Abstract = Line of Sight output in GeoJSON format.
51       <ComplexData>
52        <Default>
53          mimeType = application/json
54          encoding = UTF-8
55        </Default>
56      </ComplexData>
57  </DataOutputs>
```

Listing 3.21: Defining the result data output.

3.6 Python Scripting

To comply with the interaction methodology of the *ZOO-Project* WPS framework a *Python* function that fulfils a specific set of requirements has to be created that reference back to the .zcfg input and output definitions. In section 2.7.1 on page 21 I describe the *Python* terminology used below.
Accessing the data model from the .zcfg requires importing the zoo *Python* module (line 3 in listing 3.22), using the module allows us to access the data using the previously mentioned *key/value* pairs in the form of a *Python* dictionary (see section 2.7.1 on page 22).
The input parameters and result output are handled through a small piece of *Python* code that handles the inputs and passes them to the previously defined sql function (see section 3.4). The *Python* code also handles the output from the *PostgreSQL* database so that it is parse-able by the *ZOO-Kernel*. Listing 3.22 shows the limited amount of *Python* code, which will be described below.

3.6.1 Connecting to DB with Psycopg2

Accessing the database from within *Python* requires an interface driver, which is provided using the *Psycopg2* module that is imported on line 2 in listing 3.22. Using the function

```
1  import geojson
2  import psycopg2
3  import zoo
4
5  #DATABASE CONNECTIONS
6  conn = psycopg2.connect("dbname=jonaspedersen user=jonaspedersen host=localhost
   ↪  password=logger")
7  cur = conn.cursor()
8
9  def ExtractlineofsightGML(conf,inputs,outputs):
10   cur.execute("""select los(ST_GeomFromGML(%(start)s),ST_GeomFromGML(%(end)s),%(z)s) as
     ↪  json;"""
11   , {'start':inputs["startpoint"]["value"],'end':inputs["endpoint"]["value"],
12   'z':inputs["height"]["value"]})
13   outputgeojson = geojson.dumps(cur.fetchall())
14   outputs["result"]["value"] = outputgeojson
15   conn.close()
16   return zoo.SERVICE_SUCCEEDED
```

Listing 3.22: Wrapping the *PostgreSQL* function in a thin *Python* wrapper.

connect from the connections class, a connection is made to the database on line 6 using the credentials set up when originally initialising the database (see section 3.1.4 on page 34).
Within the connection object a cursor is initialised on line 7. The cursor class method execute is then used on line 10-11 to execute the sql code in the database, substituting the input parameters with the values from the *ZOO-Kernel* dictionary.
Assigning the user generated values to *Psycopg2* parameter place holders is done using the syntax 'place holder name': dictionary name["input identifier "]["value"] to define the place holder name and access the specific input value.
Like accessing the user defined inputs is a case of requesting the value of a specific key of the *ZOO-Kernel* generated dictionary, accessing the output of the function is completely analogues but instead of retrieving a input value, the result of the sql query is assigned as a value in the *ZOO-Kernel* dictionary.
As seen in line 13 of listing 3.22, the *ZOO-Kernel* generated dictionary holding the output is called outputs and the return result is assigned to the key "value" nested under the "result" key.

3.7 Connecting the Dots and Running the Web Processing Service

Previously the methodology section has dealt with the description and creation of the different pieces that the *ZOO-Service* consists of and relies on. Putting the puzzle together is then a matter of moving the files into the right folders.
The zoo_loader.cgi, which is the CGI program that connects to the *ZOO-Kernel* and the associated configuration file main.cfg along with the lineofsightGML.py and its associated configuration file ExtractlineofsightGML.zcfg, are all moved into the directory

/usr/lib/cgi−bin, where all CGI programs that are executed through the *Apache* server also resides.
Now the line of sight service is accessible from the browser when calling the zoo_loader.cgi in combination with the identifier, and either the parameter inputs and an Execute request or combined with the DescribeProcess request.
The results returned when calling the WPS and the individual process is discussed in the next chapter.

Chapter 4

Results and Discussion

In the following chapter a discussion on; the structure and capabilities of the line of sight WPS produced, the output of the line of sight WPS and the usability of the line of sight service as a housing tax variable will be initiated.

4.1 WPS output

In chapter 3 I described how to set up a WPS in the *ZOO-Project* framework and create a line of sight process that can be called, executed and return results through the service. Below I will describe how this functionality is presented to the user.
The WPS implemented here reacts to the three commands described in section 2.3:

- GetCapabilities
- DescribeProcess
- Execute

Figure 4.1 shows the overall physical structure of the WPS, its dependencies and how the user input flows to from the client to the WPS using the *REST* framework as described in section 2.2.2 on page 6 for each of the three commands. The figure can be used for reference in the description of the framework below.
In listing 4.1 a GetCapabilities request URL with a a REST KVP payload is shown. This command is sent to the WPS when a description of the capabilities of the WPS server is requested by the client.

```
http://45.32.186.126/cgi-bin/zoo_loader.cgi?
 ↪  Service=WPS&Request=GetCapabilities&Version=1.0.0
```

Listing 4.1: GetCapabilities command sent to the WPS.

As described in section 2.2.1 (2.2.3) a mandatory request return is required of the WPS framework to a GetCapabilities request. This request will return the overall information

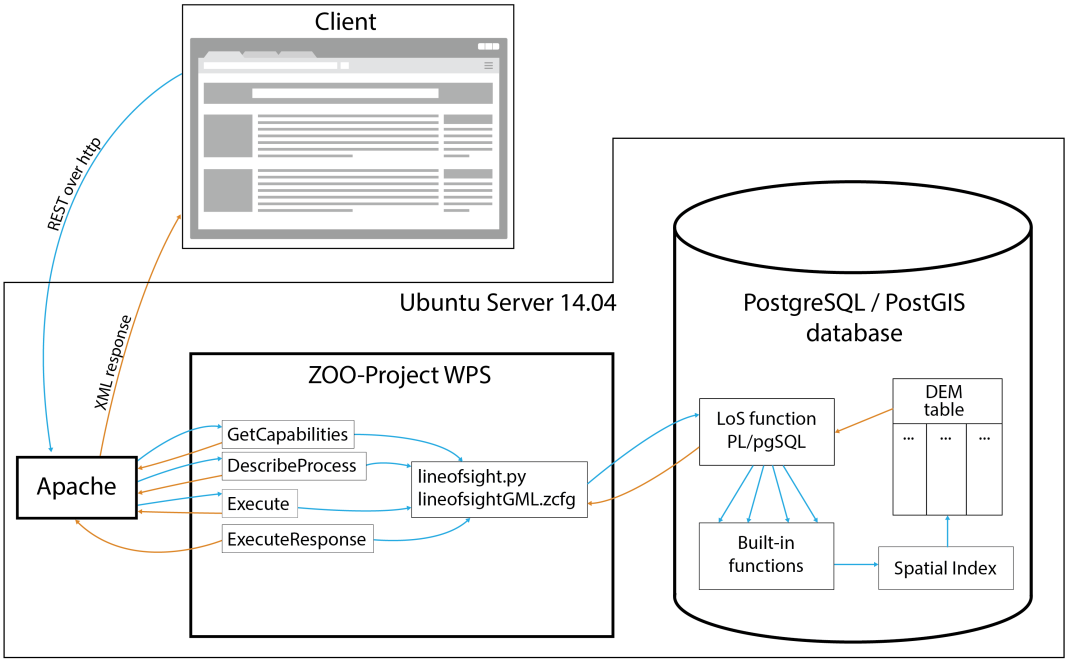

Figure 4.1: The physical structure and interdependencies of the WPS process.

on the WPS parsed from the main.cfg file, including Title, Abstract, ServiceProvider and all supported Operations (GetCapabilities, DescribeProcess and Execute) and the Identifier , Title and Abstract parsed from the .zcgf file from each available process. All parameters are parsed and returned as a well formed XML response.
In listing 4.2 a DescribeProcess request is shown for the line of sight process (Identifier =ExtractlineofsightGML), which is sent to the WPS using the same method as the GetCapabilities.

```
http://45.32.186.126/cgi-bin/zoo_loader.cgi? Service=WPS&Request=DescribeProcess&
 ↪  Identifier=ExtractlineofsightGML&version=1.0.0
```

Listing 4.2: DescribeProcess command sent to the WPS.

When receiving the DescribeProcess for the line of sight process, the WPS parses the information in the ExtractlineofsightGML.zcfg file and sends a XML document back the client containing the information.
Just like the DescribeProcess and GetCapabilities, the Execute command delivers its payload using a REST formatted KVP over http to the WPS server. A URL containing the payload can be seen in listing 4.3, which beyond the KVP's that are included in the GetCapabilities and DescribeProcess also contains an example of input parameters under

the key DataInputs. Here each input is represented with their individual Identifiers and are separated by ;. Both the startpoint and endpoint parameters are input as GML points (see section 2.8.5), the height is input as integers in centimetres (see section 3.2.2, where the data type and units are transformed), and the sampling resolution is also input as an integer.

```
http://45.32.186.126/cgi-bin/zoo_loader.cgi? Service=WPS&Request=Execute&
 ↪  Identifier=ExtractlineofsightGML& DataInputs=startpoint=<gml:Point>
 ↪  <gml:coordinates>711330,6197923</gml:coordinates></gml:Point>;
 ↪  endpoint=<gml:Point>
 ↪  <gml:coordinates>711370,6198970</gml:coordinates></gml:Point>;
 ↪  height=400;resolution=1&Version=1.0.0&
```

Listing 4.3: Execute command sent to the WPS.

When receiving an Execute command The WPS framework interacts with the *PostgreSQL/PostGIS* database through the lineofsightGML.py processing service programmed in *Python*, and the associated configuration script stored in a .zcfg file.
The *Python* code calls the database end executes a SQL statement containing the custom *PL/pgsql* function. This function in turn queries the contents of the table containing the raster.
The custom function los() described in section 3.4 takes as input two point geometries asides from the height and resolution. The *PostGIS* compliant geometries are created from the clients GML input by the embedding the function ST_GeomFromGML function, which converts GML to *PostGIS* compliant geometries into the SELECT statement used in the *Python* process called from the WPS. A line string is created from the endpoint and startpoint and this is compared to the index on the raster to retrieve only raster tiles that intersect the line.
The custom function then evaluates the visibility along the line, by calling build-in functions and operators and lastly return a record containing a GeoJSON FeatureCollection (see section 2.8.4).
This GeoJSON object is returned to the *Python* code, where it is inserted as a value under the dictionary key ["los"] (see listing 3.22), to be parsed by the WPS into XML
After the *REST* payload has been sent from client to server and the processing has been done, the XML is returned to the client by the WPS via the *Apache* web server.
The output can now be consumed by the client in any way desired, e.g. as input into a *Leaflet* or *Open Layers* web map.

4.2 Evaluating Line of Sight Result

In the following section an evaluation of a result of a line of sight calculation will be presented using:

- orthoimagery
- The original DEM

- A photo from the perspective of the original viewpoint
- A 3D model

The line of sight calculation was done from the point 712422, 6192236 (start point) where 294 cm was added to the surface height and to the point 712189, 6192692 (end point) both points specified in EPSG 25832 coordinates. The sampling of the underlying DEM was done at 1 meter intervals.

4.2.1 Evaluate Line of Sight Result Against Orthoimagery

To evaluate the credibility of the line of sight calculation, ortho imagery is a valuable comparison reference. The ability of the human mind to interpret a true colour ortho photo to estimate heights and the relationship between these heights to establish which features are visible from where is an immediate function of the human minds ability to contextualise information on the typical height and extent of feature types. This can be exemplified by the interpretation one would make of a flat field intersected by a line of trees. Here the human mind would interpret that standing on one side of the trees, one would not be able to see the field on the other side, as the height of the trees block the view of the much lower field on the other side.
The product *ortofoto sommer 2012* WMTS from *kortforsyningen.dk* was chosen as reference among other ortho imagery products as it had the least amount of shadows and an acceptable contrast level.

Figure 4.2: The line of sight result overlaid orthoimagery.

From the ortho imagery displayed in figure 4.2 it can be observed that the start point is just inside a small collection of trees with a cultivated field in front. The field is characterised by sets of furrows that lie perpendicular to the line of sight along with a set of furrows that run parallel to the line of sight (see figure 4.6 containing a photo taken from the start point towards the end point for clarification).
The line of sight then reaches a few trees and afterwards a collection of what appears to be farm house structures. This is followed by a small cluster of trees followed by what seems to be a small garden and then a small forest. Finally the forest is intersected by what seems to be a grass patch and ending in the forest.
The visibility on the line of sight as returned using the WPS shows visible sample points as green and non-visible as red. The field just in front of the start point has a high number of visible points and not until meeting the farm structure in the form of a hedge is visibility hampered considerably. Inside farm a small tree is visible followed by the front of the first farm structure roof and the second farm roof ridge followed by a few points of visibility of the small cluster of trees just after the farm. lastly a single point of the larger forest structure after the farm is visible.
It should be noted that the ortho imagery (figure 4.2) and the on-site photo (figure 4.6) both show a structure orthogonal to the first farm structure with a white gable facing towards the start point. This structure is not represented in the DEM, which introduces an error into the visibility calculation.

The structures on the line of sight after the low lying field in front of the start point that are marked visible are primarily features that are associated with a somewhat higher elevation than neutral terrain.

4.2.2 Evaluate Line of Sight Result Against Original DEM

The DEM raster lies beneath all calculations of visibility in this thesis. In figure 4.3 the line of sight result is shown overlaid the DEM raster to be able to evaluate the influence of elevation to the line of sight calculation.

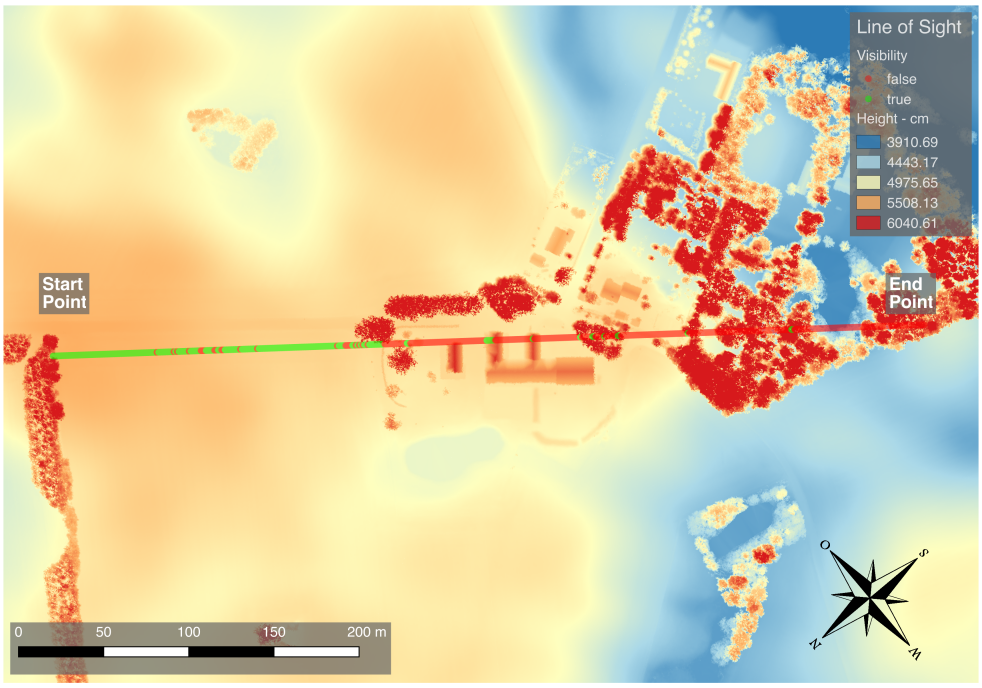

Figure 4.3: The line of sight result overlaid the original DEM.

The raster is coloured according to pixel value, with deep blue as the lowest values and bright red as the highest (3910 - 6040 cm). Excluding feature hight (trees, buildings and hedges) the start point of the line of sight calculation is located at a high point in the terrain, with the terrain sloping downward in the sight direction.
The location of the start point combined with the added 294 centimetres of height at the start point explains the high visibility of the field at the beginning of the line of sight. Is is not immediately obvious why there are some points along the line that are not visible, but this could be the result of very local sudden slope changes downward.

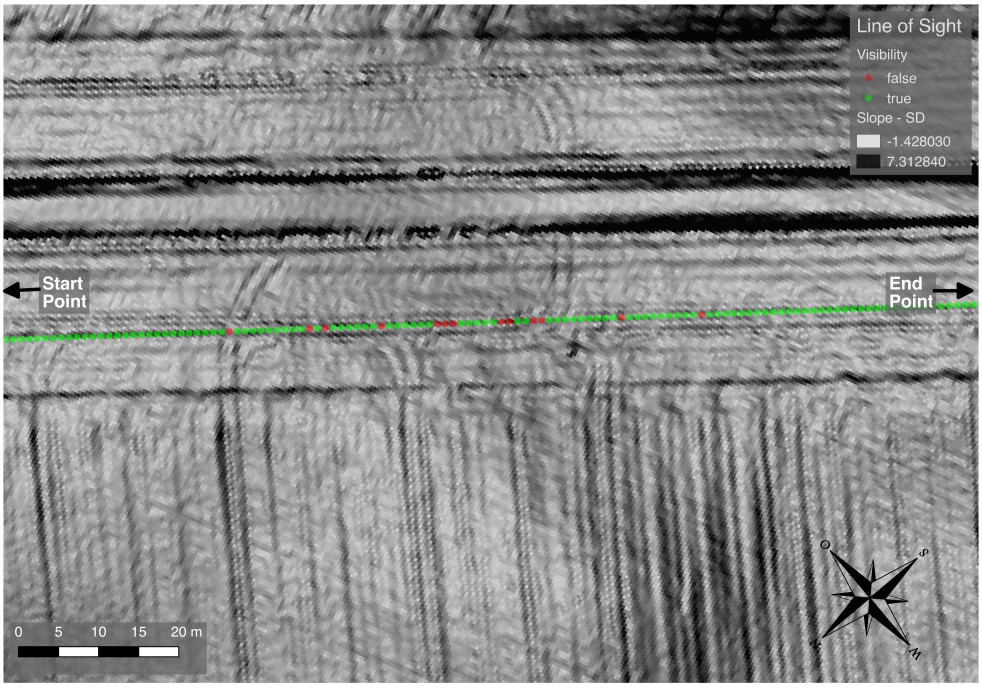

Figure 4.4: Slope calculation based on the DEM.

This theory is supported by looking at a very local slope calculation based on the DEM (figure 4.4). Here it can be seen that invisible points lies mostly just after a furrow which induces a much steeper slope than normally.
The line of sight meets the farm structure at a small hedge like structure, which effectively stops visibility of the surface until a tree is visible a few meters further along.
The remaining points of visibility are very much explained in section 4.2.1 on orthoimagery.

4.2.3 Evaluate Line of Sight Result Against 3D Model and On-Sight Photography

Below an on-site photo from the vantage point of the line of sight start point will be compared to a 3D view from the same perspective with the intent of referencing the two images as to determine the credibility of the line of sight calculation by overlaying it on the 3D on-site perspective.

Figure 4.5: Measuring the height of photo lense at the line of sight start point

Although the on-site photo was sought taken from the exact position and height of the start point the lack of a dedicated GPS system introduces some uncertainty. Instead the location was estimated from maps onto the physical landscape to a certainty of approximately 2 meters and the height above the surface was measured with a measuring

stick (see figure 4.5) and was also the subject of some human introduced measurement error.
The software used to create the 3D model of the DEM overlaid with the orthoimagery[1], did not allow for fine adjustment of the perspective and no numeric input of parameters like height. This results in some uncertainty in the exact location and height of the 3D view.

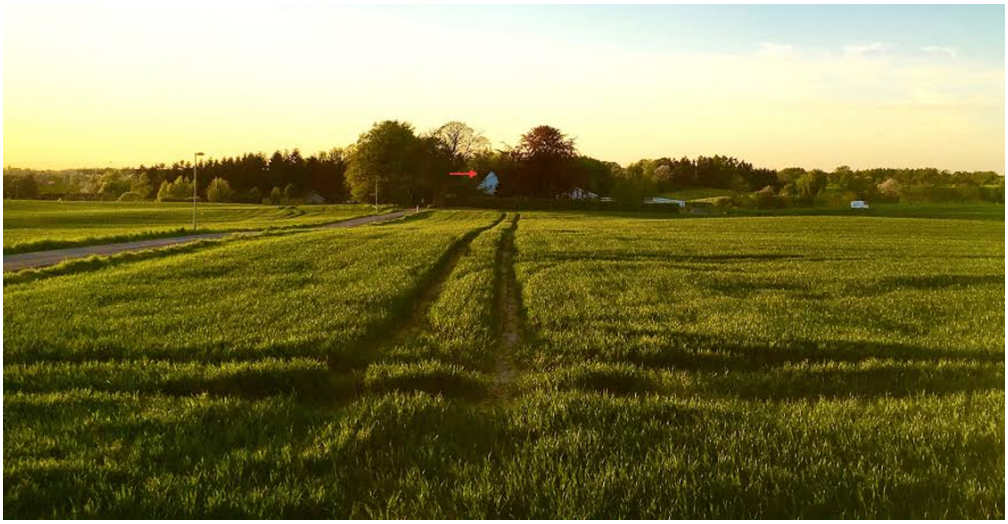

Figure 4.6: Photo taken at the start point of the line of sight calculation

The on-site photo shown in figure 4.6 has its strength at shorter distances before the view hits the complexity of multiple horizons layered after each other. It clearly shows the directionality of the furrows, the two large trees that the line of sight goes between (as a result of the season, one of them has very few leaves), the low hedge that delineates the farm house structures from the field and a white gable of one of the structures. Also visible, although not very clear, is a roof with its ridge marked with a red arrow in.
All features that are further away are hard to distinguish from each other, but seems to form a horizon dominated by treetops, likely formed by the small cluster of threes after the farm structures and the following forest as presented in section 4.2.1.
The equivalent 3D view without (figure 4.7) and with (figure 4.8) the line of sight overlaid, do not show the white gable (see section **??** for discussion on the presence of this structure in ortho imagery and DEM). But like the on-sight photo it shows the roof of one of the farm structures, with the ridge again marked by a red arrow.
This is followed by a complex cluster of trees analogue to the arrangement in the on-site photo (figure 4.6). Although the effect of curvature at distances around 600 meters such as in this case are small, the effect is that the further away from the start point the higher points will be relative to the value used in the line of sight calculation.

[1] Qgis2threejs

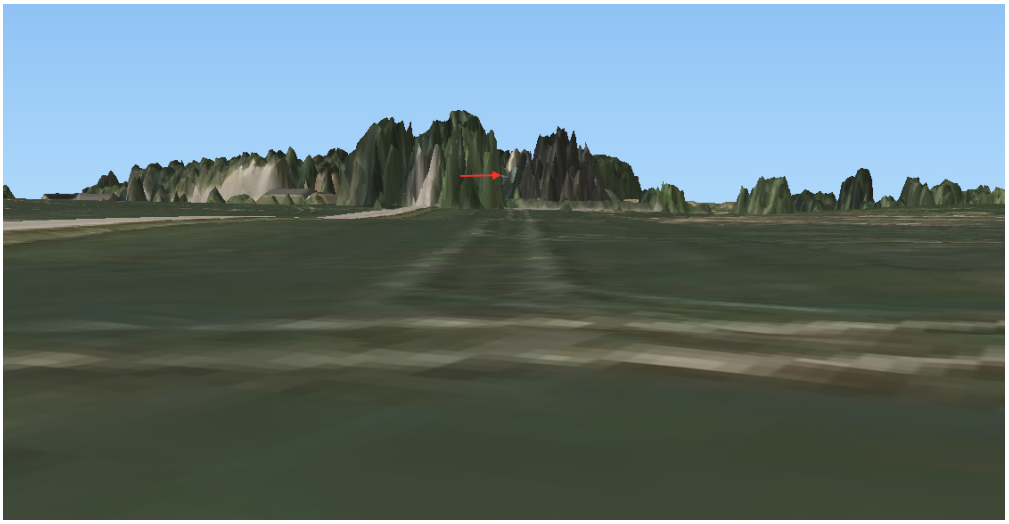

Figure 4.7: 3D View at the start point looking towards end point

Between the two representations there is clear feature to feature equivalence although the differences in resolution and extent of the 3D scene, results in some clear differences in the presentation. The similarity is close enough though that an overlay of the line of sight result can yield an indication of the success rate of the calculation, which is interpreted as being very reliable.
In figure 4.8 the 3D perspective emulating the perspective of the line of sight the is shown overlaid with the line of sight result. There are only green points visible bar for one red point. This red point lies at the top of a tree trunk and could possibly be a result of a human error in adjusting the view or a result of the missing correction for earth curvature effect in the 3D view. That only points classified as green are showing at the perspective of the start point is a strong circumstantial evidence that the calculation creates a reliable result.
When superimposing the 3D view that includes the line of sight calculation shown in figure 4.8 onto the original photo taken on the site shown in figure 4.6 it can be seen that the two close horizons line up with good precision (see figure 4.9) (The far of horizon is not included in the 3D view as only a 3D scene for the bounding box of the line of sight calculation is generated).
Looking a the points representing the line of sight they again correlate very precisely to the photo taken on the site reinforcing the line of sight calculation result.

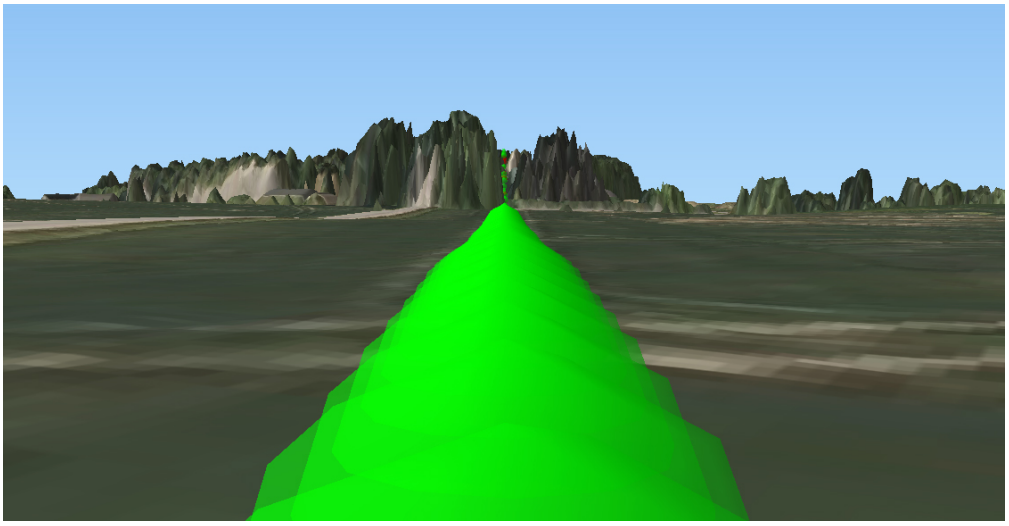

Figure 4.8: 3D view at the start point overlaid with line of sight result

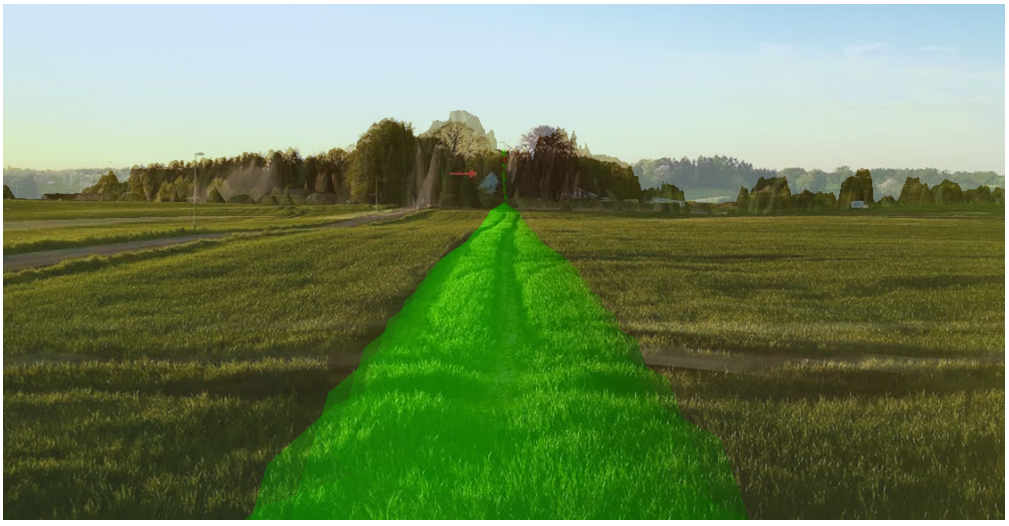

Figure 4.9: Original photo from the point of view of the line of sight calculation start point, superimposed onto the 3D view including the line of sight result.

4.3 Performance Evaluation

As stated in the problem statement in section 1.2, a comparison of the performance of the line of sight *PL/pgSQL* process running in the *ZOO-Project* framework[2] against the equivalent WPS's from *Sweco* and *Septima*[3] was an objective of the thesis.
Below I will compare the speed of execution for the three related services by timing the period from sending the Execute request until the XML response is returned. The timing was done within the *Safari* browser using the *Safari Web Inspector*, which allows to time individual network events when accessing a web site.
This means that it is not the timing of when the result is drawn on a map (as this is not yet implemented in the *ZOO-Project* WPS), but the timing of when the raw data is returned. This methodology should alleviate effect of the overhead of the graphical user interface of the two other interfaces.
For consistency all timings were done from a computer with ethernet connection at *Sweco* and all measurements was done three times, with the average being used for the evaluation.

4.3.1 How do the Three Solutions Compare in Speed of Execution?

The three line of sight services all perform similarly at the 500 meter distance returning a result within what is perceptible by humans. Measuring the time shows that the *Sweco* service is clearly the fastest at this distance, returning a result in 240 ms where the *Septima* measures at 485 ms and the *ZOO-Project* service is the slowest at 525 ms (see table 4.1).

Line of sight, meters	**500**	**2000**	**5000**
ZOO-Project WPS	525	603	566
Sweco	240	637	1470
Septima	485	1325	2900
PL/pgSQL function	347	1300	3100
Size of ZOO responce	69 KB	271 KB	540 KB

Table 4.1: The response time in milliseconds for each service at different distances queried.

When the distance of the line of sight goes up to 2000 meters, the time to return a result increases for all services, with the *ZOO-Project* service returning a result in 603 ms, *Sweco* in 637 ms and *Septima* in 1325 ms.
Lastly the 5000 meters distance for the line of sight shows that the *ZOO-Project* service returns a result in 566 ms, the *Sweco* service in 1470 ms and the *Septima* service in 2900 ms.

[2]GetCapabilities of *ZOO-Project* WPS `http://45.32.186.126/cgi-bin/zoo_loader.cgi?Service=WPS&Request=GetCapabilities&Version=1.0.0`

[3]`http://labs.septima.dk/`

While both the *Sweco* and *Septima* service scale in a linear and predictable manner along with the distance calculated, the *ZOO-Project* service however seems to hover around 500-600 ms no matter what distance is calculated.

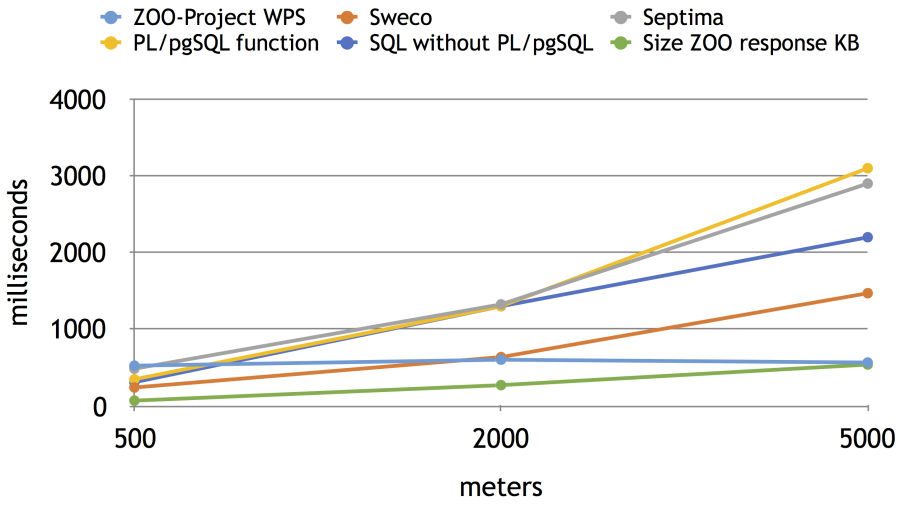

Figure 4.10: Chart showing the relative speed of each line of sight service at different distances.

This discrepancy can be best observed in in figure 4.10 where the *Sweco* (orange) and *Septima* (gray) service scale in a similar way, whereas the *ZOO-Project* service (light blue) is not influenced by the greater distance and subsequently larger size of the return payload (green).
Because of these discrepancies between the *ZOO-Project* service, the size of the payload and the other services the results of the *ZOO-Project* service is somewhat untrustworthy. To investigate this further a test of the *PL/pgSQL* function running in the database was included.
The results of this additional test show that when running the *PL/pgSQL* function, the results scale close to linearly (yellow line in figure4.10) resulting in the slowest measurement result and very close to the result for the *Septima* service.
The explanation to these discrepancies are not clear, but can be the result of how the response result is measured in the *Safari Web Inspector*.
Although the results measured for the *ZOO-Project* service were somewhat uncredible, the result of the *PL/pgSQL* function can give an indication of the speed of the WPS, showing that the performance lies close to the *Septima* service, but is the slowest.

4.4 Usability of Line of Sight as a Parameter in Housing Tax

After creating the line of sight service, establishing its functionality and precision I will now turn to the possible utility of a line of sight WPS process in a housing tax valuation model.
As noted in the section 2.9 on page 30 a possible new and more objective model for housing and property tax could very well end up using geo data to standardise the valuation model. With parameters like distance to coast line, rail-roads, motorways, forest and several others influencing the valuation, incorporating the visibility of these objects would make for an even more realistic model.
There could be different implementation strategies for incorporating a line of sight to the geographical features. If using the WPS created for this thesis, inputs should be provided for the startpoint, endpoint and the height above the startpoint. These inputs could be provided from a separate WPS, which should return the geometry the address in question as startpoint, locate the nearest point on the nearest feature of interest as endpoint and the maximum allowed building height from the areas local plan as height.
This solution would require a separate WPS accessing data from the Danish Address Register (data served from AWS[4]), access to geographical data on the features in question, most of which are available from the *GeoDanmark* dataset (Fællesoffentligt Geografisk Administrationsgrundlag 2014), and finally access to a register of local plans from the Danish Business Authority[5].
By pulling together data from all these independent sources an automated model can be developed as shown in figure 4.11.
The model shown in figure 4.11 shows how the line of sight WPS would be included in a functional workflow to return visibility of geodata features. In this case the user would input just an address and a geodata feature type to determine visibility to.
The address would be georeferenced using data from *Danmarks Adresseregister* returning a geometry. This geometry would be sent to the line of sight WPS as startpoint input. It would also be sent to the *GeoDanmark* WPS where the nearest feature of the requested feature type would be located and the point on the feature with the shortest distance to the address would be returned to the line of sight WPS as the endpoint input.
And lastly to a *PlanSystemDK* WPS where the maximum allowed building height for the address would be retrieved from the local plan that contain the address.
The robustness of using the visibility of just one point would be questionable. To counter this, instead of establishing the line of sight to just one point, the features (often polygons and line strings) could be subdivided into smaller line strings, from which the nearest could be located and subsequently reduced to points at a defined interval. Establishing line of sights to each of the points would result in a more realistic valuation.
By implementing the line of sight calculation as a module in a wider modular strategy of exposing processing abilities through separate WPS systems, the usability of a line of

[4] `http://aws.dk`
[5] `https://erhvervsstyrelsen.dk/plansystemdk`

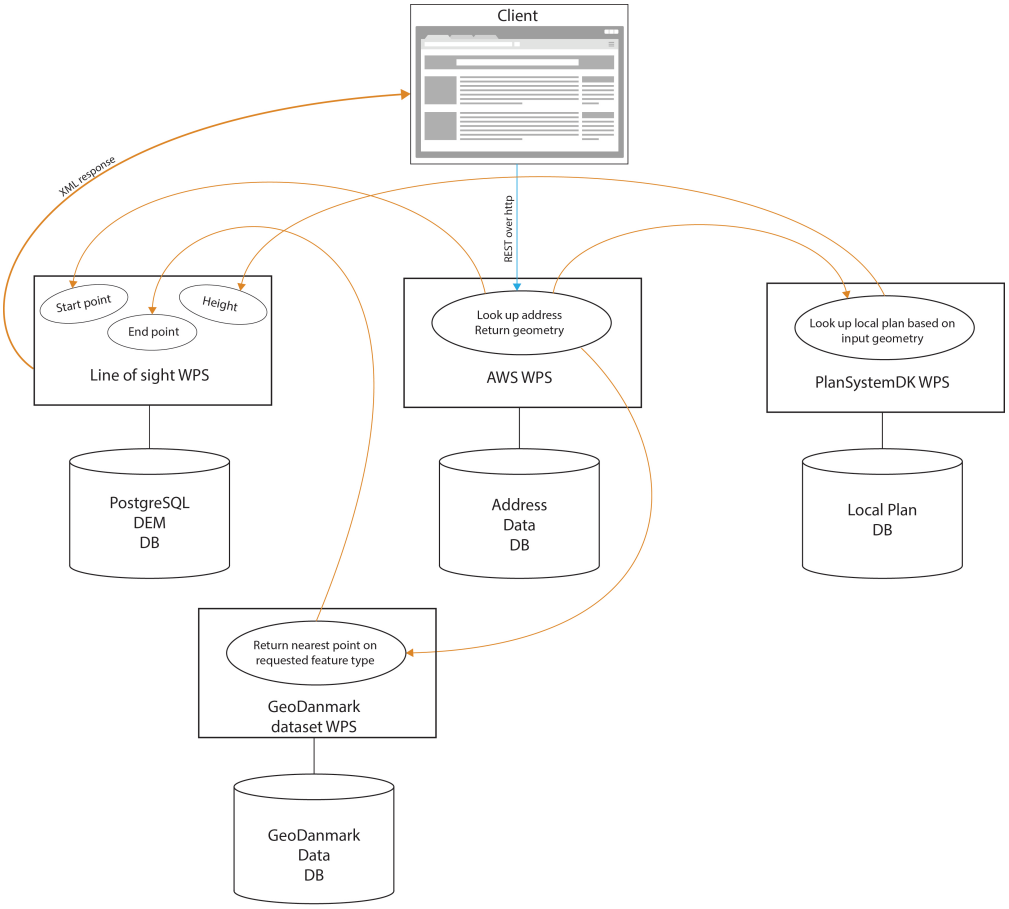

Figure 4.11: Example of interconnected WPS to create a repeatable workflow for visibility calculations from addresses to geodata features.

sight WPS is greatly increased.
The modular approach, if responsibility was delegated along with the modules, would also encourage independent development and maintenance to create individually optimised modules. The different versions of the processing modules would be reachable by using the introduced in section 2.3 on page 11, such that the repeatability for each calculation would be preserved.
Creating the actual geospatial calculations within the database would allow both; a simple way of logging all actions by user (see section 2.8.1 on page 27) to acknowledge the expert hearings focus on accountability (P. E. Jensen et al. 2014) and an effective way of storing results.

Chapter 5

Conclusion

The implementation of a software stack capable of hosting a spatial analysis service was done using the *ZOO-Project* WPS framework. The WPS framework handled the communication to the actual analysis module using a REST framework. The analysis itself was done within a *PostgreSQL/PostGIS* environment and output to GeoJSON format. The result was then returned to the user encapsulated in a well formed XML document.
The line of sight calculation was done by determining the angle created between the line representing the start point and endpoint and the line stretching from the surface directly below the start point and the endpoint, and finally relating this angle to the similarly calculated angles for all other sampled points between the start point and currently evaluated point.
To compensate for the curvature of the earth, the effect was subtracted from the elevations sampled from the DEM at calculation time.
When comparing the performance of the line of sight WPS to both *Sweco* and *Septima* solutions a muddy picture emerged, possibly because of problems with the measurement methodology. The WPS created for this thesis came out as the fastest of all services, when determining the line of sight for longer distances, even returning a response for both 2000 and 5000 meters, which took approximately the same time.
This result went very much against all other evidence and a test closer within the database showed measured times close to the *Septima* solution.
Evaluating the precision of the line of sight calculation, comparing the output to both on-sight photography, ortho imagery, the DEM data, slope calculations and a 3D model showed a high degree of correlation between the calculation and real life, establishing the line of sight calculation as a credible service.
The line of sight as a service was developed with the *Danish Customs and Tax Administration* in mind, to be used when valuing property. As shown in section 4.4 the line of sight WPS, as it is a standards based service, can be implemented in a larger network of WPS's to form an objective and repeatable workflow that, with very few and simple user inputs, can return a line of sight to base the visibility parameter on in the valuation process.

Bibliography

52° North GmbH (2016). *52° North - Initiative for Geospatial Open Source Software GmbH*. `http://52north.org`. Accessed: 2016-04-05.

Bang, Mads (2013). "Her er kritikken af ejendomsvurderingerne". In: *Altinget*.

Bater, Christopher W and Nicholas C Coops (2009). "Evaluating error associated with lidar-derived DEM interpolation". In: *Computers and Geosciences* 35.2, pp. 289–300.

Beaulieu, Alan (2009). *Learning SQL*. "O'Reilly Media, Inc."

Butler, H, S Gillies, and T Schaub (2016). *The GeoJSON Format*. Internet Engineering Task Force.

Charaniya, Amin P, Roberto Manduchi, and Suresh K Lodha (2004). "Supervised Parametric Classification of Aerial LiDAR Data". In: *CVPRW '04: Proceedings of the 2004 Conference on Computer Vision and Pattern Recognition Workshop (CVPRW'04) Volume 3 - Volume 03*. University of California, Santa Cruz. IEEE Computer Society.

Dar, Usama et al. (2015). "PostgreSQL Server Programming - Second Edition". In: pp. 1–508.

Date, C J (2015). *SQL and Relational Theory*. How to Write Accurate SQL Code. "O'Reilly Media, Inc."

Dengsøe, Povl (2015). "Grundskyld får hammeren: Boligejerne stoler ikke på Skats vurdering". In: *Berlingske Tidende*.

Drachmann, Hans (2014). "Skattefolk har opfundet metode, som giver vidt forskellige ejendomsvurderinger". In: *Politiken*.

— (2016). "Skat afviser at ændre grundskyld på trods af underkendelse". In: *Politiken*.

Drachmann, Hans and John Hansen (2016). "Skat gør stadig forskel på naboers ejendomsskat". In: *Politiken*.

Evangelidis, Konstantinos et al. (2014). "Geospatial services in the Cloud". In: *Computers and Geosciences* 63.C, pp. 116–122.

Fællesoffentligt Geografisk Administrationsgrundlag (2014). *Specifikation FOT 5.1 forenklet udgave*.

Fenoy, Gérald, Nicolas Bozon, and Venkatesh Raghavan (2012). "ZOO-Project: the open WPS platform". In: *Applied Geomatics* 5.1, pp. 19–24.

Fisher, Peter F (1993). "Algorithm and implementation uncertainty in viewshed analysis". In: *International journal of geographical information systems* 7.4, pp. 331–347.

— (2006). "Extending the Applicability of Viewsheds in Landscape Planning". In: pp. 1–6.

Fu, Pinde and Jiulin Sun (2011). *Web GIS*. Principles and Applications. Esri Press.

GDAL Development Team (2016). *GDAL - Geospatial Data Abstraction Library, Version 1.11.2*. Open Source Geospatial Foundation. URL: %E2%80%8Bhttp://www.gdal.org.

Geodatastyrelsen (2015a). *Danmarks Højdemodel, DHM/Overflade*. Tech. rep. Geodatastyrelsen.

— (2015b). *Danmarks Højdemodel, DHM/Punktsky*. Tech. rep. Geodatastyrelsen.

Guo, Qinghua et al. (2010). "Effects of Topographic Variability and Lidar Sampling Density on Several DEM Interpolation Methods". In: *Photogrammetric Engineering & Remote Sensing* 76.6, pp. 701–712.

Haines, Nathan (2015). *Beginning Ubuntu for Windows and Mac Users*. Apress.

Haverkort, Herman, Laura Toma, and Yi Zhuang (2009). "Computing visibility on terrains in external memory". In: *Journal of Experimental Algorithmics (JEA)* 13, p. 5.

Herman, Ted (2013). *A Functional Start to Computing with Python*. CRC Press.

Higham, Nicholas J (1996). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics.

Jensen, John R (2007). *Remote Sensing of the Environment: Pearson New International Edition*. An Earth Resource Perspective. Pearson Higher Ed.

Jensen, Peter Engberg et al. (2014). "**Forbedring af ejendomsvurderingen** ". In: pp. 1–248.

Klensin, J, T Hansen, and N Freed (2013). "Media Type Specifications and Registration Procedures". In:

Koukoulas, S and G A Blackburn (2005). "Mapping individual tree location, height and species in broadleaved deciduous forest using airborne LIDAR and multi-spectral remotely sensed data". In: *International Journal of Remote Sensing* 26.3, pp. 431–455.

Marquez, Angel (2015). *Postgis Essentials*. Packt Publishing.

Matt Duckham, Michael F Goodchild and Michael F Worboys (2007). "FOUNDATIONS OF GEOGRAPHIC INFORMATION SCIENCE". In: pp. 1–252.

OGC (1999). ***OpenGISSimple Features Specification For SQL***. Open GIS Consortium.

— (2002). *Open GIS Geography Markup Language (GML) Implementation Specification Version 2.1.2*. Open Geospatial Consortium.

— (2006). *OpenGIS® Web Map Server Implementation Specification*. Open Geospatial Consortium.

— (2008). *OpenGIS® Web Processing Service*. 1.0.0. Open Geospatial Consortium.

— (2010a). *OpenGIS Web Feature Service 2.0 Interface Standard*. Open Geospatial Consortium.

— (2010b). *OpenGIS® Web Map Tile Service Implementation Standard*. Open Geospatial Consortium.

— (2012). *OGC® WCS 2.0 Interface Standard- Core*. Open Geospatial Consortium.

Open Source Geospatial Foundation (2016). *GeoServer WPS*. http://docs.geoserver.org/2.8.x/en/user/extensions/wps/index.html. Accessed: 2016-04-05.

PyWPS (2016). *PyWPS Project*. http://pywps.org. Accessed: 2016-04-05.

Regina O Obe, Leo S Hsu (2015). "PostGIS in Action". In: pp. 1–602.

Richardson, Leonard and Mike Amundsen (2013). *RESTful Web APIs*. O'Reilly Media.

Rosenkranz, Brigitte Christine (2014). "Nye data til Danmarks Højdemodel ". In: pp. 1–2.
Sebesta, Robert W (2015). *Concepts of Programming Languages, Global Edition.*
Skatteministeriet (2013a). "Bekendtgørelse af lov om vurdering af landets faste ejendomme". In: pp. 1–12.
— (2013b). *Danskerne skal have et nyt og bedre ejendomsvurderingssystem.* URL: http://www.skm.dk/aktuelt/presse/pressemeddelelser/2013/oktober/danskerne-skal-have-et-nyt-og-bedre-ejendomsvurderingssystem.
Steiniger, Stefan and Andrew J S Hunter (2011). *Free and Open Source GIS Software for Building a Spatial Data Infrastructure.* Springer Berlin Heidelberg. Berlin, Heidelberg.
Stones, R and N Matthew (2006). *Beginning databases with PostgreSQL: from novice to professional.*
Sullivan, Michael (2011). *Algebra & Trigonometry.* Enhanced with Graphing Utilities. Addison-Wesley Longman.
Sumathi, S and S Esakkirajan (2007). *Fundamentals of Relational Database Management Systems.* Vol. 47. Studies in Computational Intelligence. Berlin, Heidelberg: Springer Science & Business Media.
The PostgreSQL Global Development Group (2016a). *Feature Matrix.* http://www.postgresql.org/about/featurematrix/. Accessed: 2016-04-17.
— (2016b). *PostGIS 2.2.0 Release.* http://postgis.net/2015/10/07/postgis-2.2.0/. Accessed: 2016-04-17.
The UbuntuGIS Team (2016). *UbuntuGIS branches.* https://launchpad.net/~ubuntugis. Accessed: 2016-04-11.
Tsai, Chia-Che et al. (2016). "A study of modern Linux API usage and compatibility". In: *the Eleventh European Conference.* New York, New York, USA: ACM Press, pp. 1–16.
W3. *Primitive Datatypes.* URL: http://www.w3.org/TR/xmlschema-2/#built-in-primitive-datatypes.
Wehr, Aloysius and Uwe Lohr (1999). "Airborne laser scanning—an introduction and overview". In: *ISPRS Journal of Photogrammetry and Remote Sensing* 54.2, pp. 68–82.
Westra, Erik (2013). *Python Geospatial Development, Second Edition.* Packt Publishing Ltd.
Young, Cynthia Y (2012). "Algebra and Trigonometry". In: pp. 1–1441.
ZOO-Project team (2016). *ZOO-Project WPS.* http://www.zoo-project.org. Accessed: 2016-04-05.