,,

Universitetet
i Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELORS THESIS

Study program/specialization: Bachelor of engineering Computer Science	Spring 2023 Open
Author: Ardijan Rexhaj	
Course manager: Erlend Tøssebro Supervisors: Steffen Thorsen, Nils Arne Vårdal	
Bachelors thesis title: Automating and deploying fast light pollution mapping software using VIIRS DNB satellite data and the rust programming language	
Credits: 20	
Keywords: Light pollution, rust programming language, garstang model	Number of Pages: 60 + GitHub repository: Stavanger 15. may 2023

Contents

Glossary

GeoTIFF is a georefrenced tiff file which is commonly used in geospatial analysis and GIS (Geographic Information System) applications. This file format enables embedding geographic metadata directly into the image file, such as map projection, coordinate systems, and datums.

Luminous flux, the amount of light within a volume.

HDF5, commonly used file format which can store many datatypes and be operated upon while on disk.

GDAL, is translator library for raster and vector geospatial data formats.

GIS, geographic information system is software which can display and manipulate geographic data.

WGS84, a global reference system for geospatial information and is the reference system for the Global Positioning System.

Luminous/Radiant source, a point or surface which emits light.

Docker, provides a isolated virtual environment for software to run in.

Docker-compose, configuration language for orchestrating multiple docker containers.

Summary

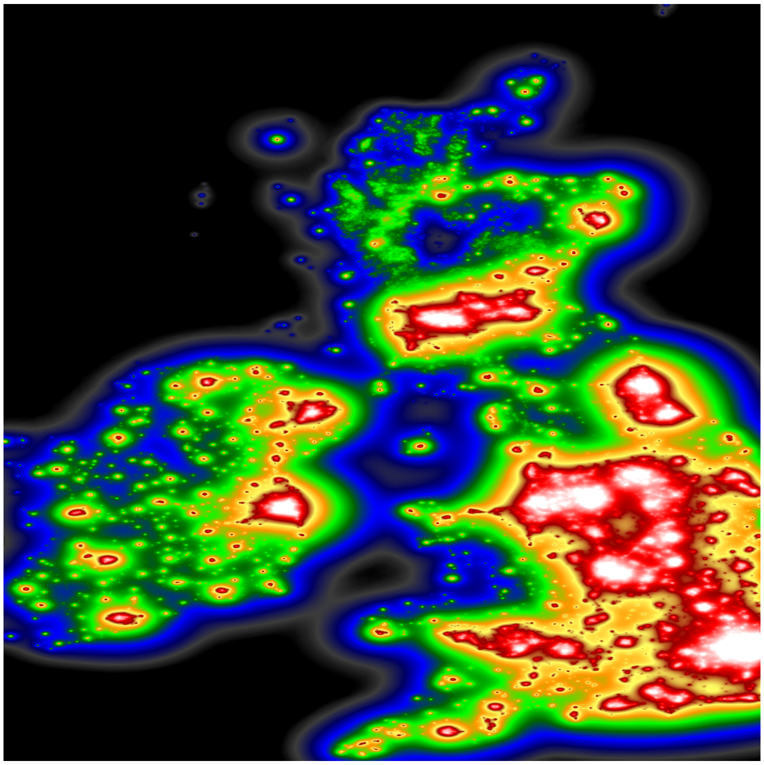

Figure 1: Image of United Kingdom light pollution levels generated in this project.

A fast light pollution map generation software package has been developed, written entirely in Rust, to compute and deliver up-to-date light pollution maps as and when required. The project aims to be open source and available to the public as a fast and accurate light pollution model. Although

the project has not yet reached the intended level of accuracy at the time of writing, it has established a solid foundation.

The project makes use of NASA's black marble VNP46A2 BRDF Corrected nighttime lights product [1]. To facilitate this, VNP46A2 tiles are fetched concurrently through the LAADS DAAC API, converted to the GeoTIFF format, and merged to form a more extensive GeoTIFF that spans the entire globe. While the project currently operates with tiles, transitioning to use the larger GeoTIFF isn't anticipated to pose a significant challenge.

After the download and conversion to GeoTIFF, R.H Garstang's light pollution model is applied on each radiant pixel in the tile. Upon completion, a gradient is applied, resulting in the creation of a PNG image. This image effectively demonstrates the application of the light pollution model.

The project source code is openly available at:

github.com/ardijanr/light_pollution_map

Chapter 1

Introduction

R.H Garstang 1986 [2] ”The astronomical community has become increasingly concerned with the problem of light pollution”. In his paper R.H Garstang described the problem as a concern and 37 years later it has become a issue to such a degree that good observational sites are often remote and hard to find. The goal in this thesis is to produce software that is accurate, automated, and easily deployable either as a software solution or by using the resulting product. In order to achieve this a derivative of the model developed by R. H. Garstang will be used [3], the raw data used to feed the Garstang model is provided by NASA's black marble project [1]. It is required that the processing and production of data happens automatically, this also applies to producing a measure of accuracy which validates predictions to ground measurements.

1.1 About Time And Date AS

Time and Date AS is based in Stavanger, Norway it has around 30 full time employees there with 7 full time developers. Since its inception in 1998 the website evolved through the addition of tools and features. Today its unique collection of openly available tools and data drives over a million unique users to the website every day.

1.2 Motivation

The definition of pollution is a substance which has harmful or poisonous effects on the surrounding environment, and although light is not a substance, its negative effects at nighttime are well documented through numerous articles [4] [5] [6] [7] [8] [9] [10] . An example of how extensive the effects of light pollution can be seen in figure 1.1 the primary objective of this thesis is the mapping of light pollution, and establish a solid foundation for the development of a product which will then be further developed by Time and Date AS in the future.

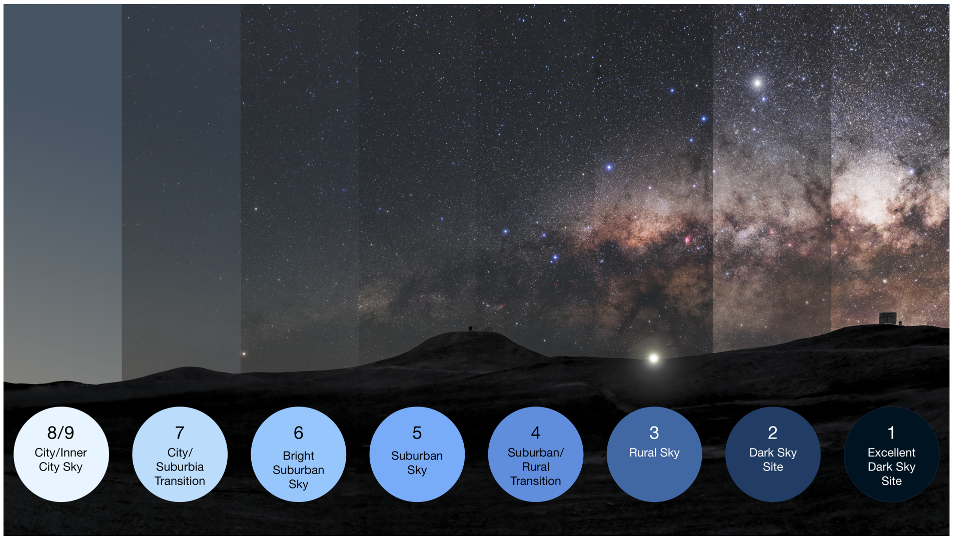

Figure 1.1: Illustration of the Bortle scale (Source: [11])

The ultimate goal will enable future users to see the extent of light pollution, judge its severity and find suitable stargazing locations. This thesis is mainly concerned with the technical aspect of mapping these effects through satellite source data provided by NASA. By applying modern software technologies and development approaches to this problem the goal is to reduce the total computing time and allow for simple containerized deployment.

1.3 Earlier Work

Some of the earliest work on artificial light propagation through the atmosphere dates back to 1973 [12], where P.J. Treanor describes a propagation law for a city or town. In it P.J. Treanor avoids most of the complexities of the problem by simplifying the atmosphere into a homogeneous medium and the city into a single point.

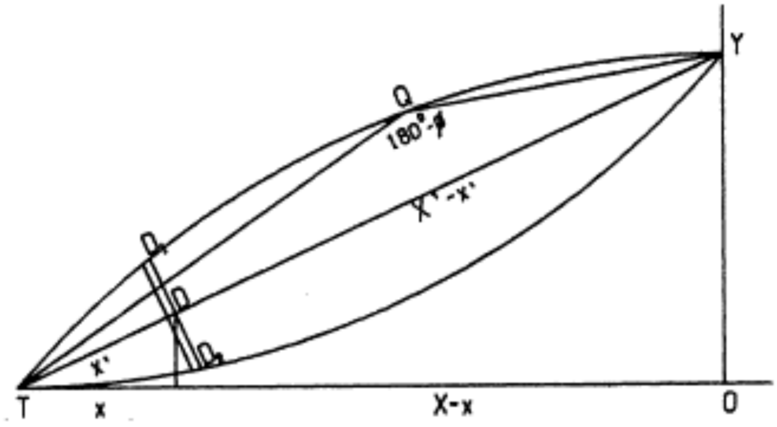

Figure 1.2: Treanor's figure for his light propagation model, city is a point light source and the atmosphere is a uniform medium. (Source: [12])

In his paper [2] from 1986 R.H Garstang then developed a model (figure 1.3) which instead describes the city as "... a circular area of uniform brightness". Compared with P.J. Treanor, R.H. Garstang attempts a more physical approach with his model, his model accounts for the reflectivity of the ground, and molecular and aerosol scattering at different heights in the atmosphere.

R.H Garstang then continued developing his model, in 1989 figure 1.4 R.H. Garstang starts taking into account the curvature of the earth.

In 1991 Garstang again improves the model, it is further expanded upon in order to take into account the ozone layer and as cited from his paper [3] "... a more accurate representation of the atmospheric molecular density variation as a function of height".

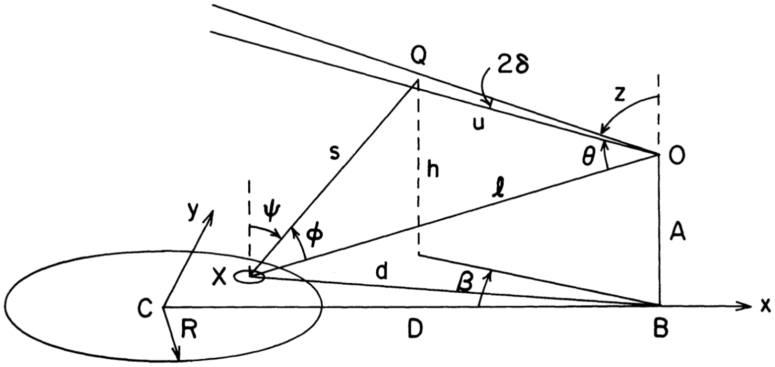

Figure 1.3: Garstangs figure of his 1986 model, the city is represented as a uniform area with center C and radius R. Light is emitted from a point X on the area of the city's defined disk. (Source: [2])

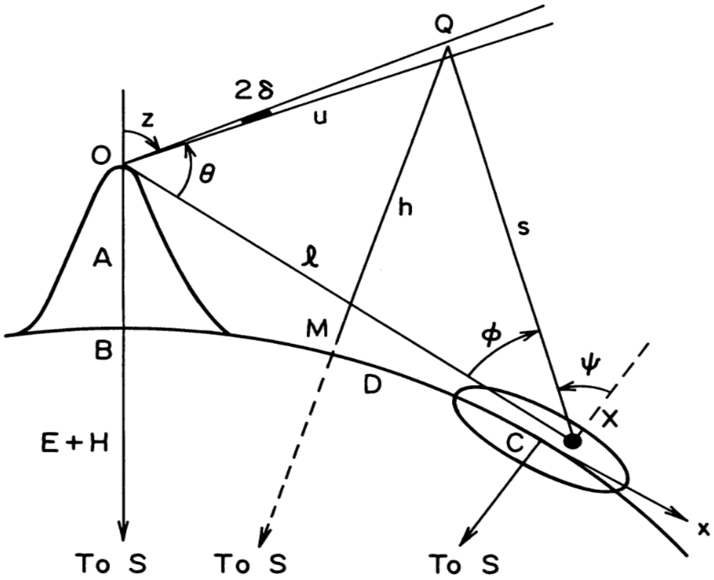

Figure 1.4: Garstangs figure 1. in his 1989 paper. This model is very similar in function to his 1986 model however now it takes into account the curvature of the earth which means the math has changed a lot. (Source: [13])

P. Cinzano, F. Falchi, and C. D. Elvidge in 2001 publish their paper [14] which is the first mapping of light pollution levels covering most of the earth using remote sensing satellite data fig 1.4.

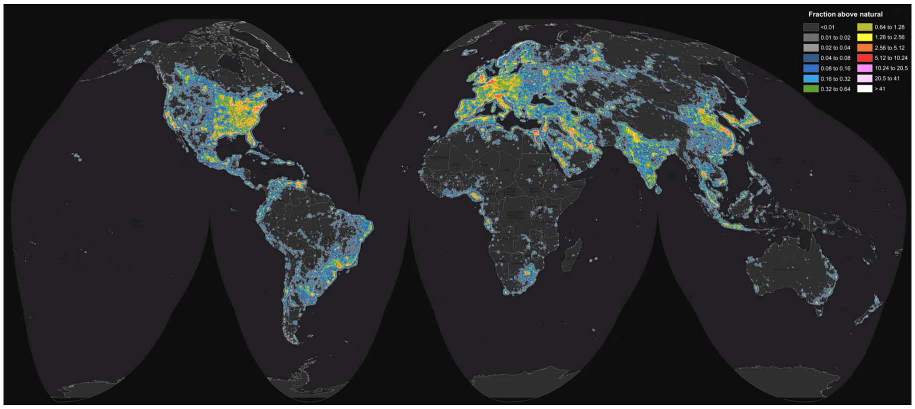

Figure 1.5: P. Cinzano, F. Falchi, and C. D. Elvidge's "the new world atlas of artifical night sky brightness" (Source: [15])

P. Cinzano and F. Falchi continue the work and in 2015 publish [15] (see figure 1.5) a new world atlas using NASA's VIIRS DNB satellite data. Details of their use of R.H Garstang model can be found in their 2011 paper [16], in which the software library LPTRAN is also mentioned. These atlases lay the foundation for a huge amount of derivative work.

Chapter 2

Methods and Materials

2.1 Introduction

The author of this thesis has been unsuccessful in his attempts to contact P. Cinzano and F. Falchi with the aim of receiving a copy of their software library LPTRAN. The LPTRAN software library would have been a good place to start for this thesis. However the excessive computational load of their software is mentioned in their 2015 paper ”The computation of the atlas required and equivalent of 200 days of time on an Intel i5 PC”, thus the author suspects that it would mostly be used as a good reference for model implementation and accuracy verification.

2.2 The light pollution model used

The light pollution model used in this thesis is based on Garstang’s 1989 version which takes into account atmospheric density, molecular and aerosol scattering, height differences between observer and the emission source and the curvature of the earth. This is the model which laid the foundation for P. Cinzano, F. Falchi, and C.D. Elvidge in their 2015 world atlas article.

In order to adapt Garstangs model to work with satellite data a few changes must be made, mainly to the luminous source. Garstang envisioned a city with some radius as the emission source, however when implementing Garstang's model each pixel is treated as a square city. One must also remember that the satellite is measuring the upwards emission of an area through the atmosphere. This should be corrected for but due to time constraints such a correction is not implemented, it is however further expanded upon in chapter 4.

2.3 Garstang's model and its mathematical basis

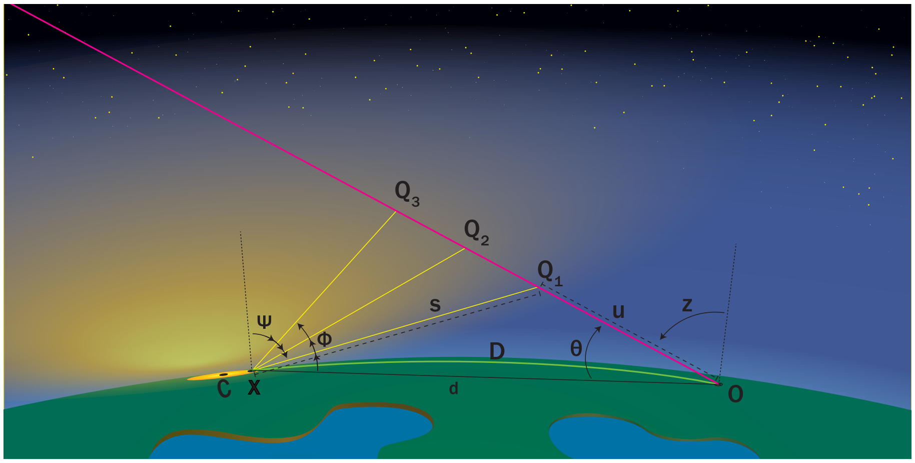

Figure 2.1: Simpler version of Garstangs illustration, depicting a city as a disk and emmiting light which affects the night sky.

The Garstang model is based on a city with center C and radius R which uniformly emits light in an area of πR^2. figure 2.1 light is emitted from a point on the city's disk X, it travels a distance s and reaches the point Q. Some of the light reaching Q is scattered in the direction of observer O, and travels the distance u.

Constants and minor functions which Garstang defines and justifies in his

1989 paper are listed here:

$$
\begin{aligned}
&\text{Constants:}\\
&N_m = 2.55 \cdot 10^{19} \quad \text{Particle density at sea level}\\
&\sigma_R = 4.6 \cdot 10^{-27} \quad \text{Aerosol scattering coefficient}\\
&c = 0.104 \quad \text{Molecular scale height}\\
&K = 0.5 \quad \text{Fraction of atmospheric clarity}\\
&F = 0.15 \quad \text{Fraction of upward emission}\\
&G = 0.15 \quad \text{Fraction of ground reflectivity}\\
&\gamma = \frac{1}{3} \quad \text{Arbitrary value chosen by Garstang}\\
&\\
&\epsilon = \frac{16}{9\pi} \quad \text{Integral compensation factor for aerosol density}\\
&a = 0.657 + 0.059K \quad \text{Scaling factor for aerosol density}\\
&E = 6371 \quad \text{Radius of the earth}\\
&\\
&C = (0,0,0) \quad \text{City center is always at origin}\\
&\\
&S = (0,0,-(E+H)) \quad \text{Center of the earth}
\end{aligned}
\tag{2.1}
$$

$$
\begin{aligned}
&\text{Variables:}\\
&H \quad \text{Height of the city above sea level}\\
&A \quad \text{Height of the observer relative to city}\\
&D \quad \text{Arc distance between observer and city center}\\
&X \quad \text{Emission source somewhere on the surface of the city}\\
&Q \quad \text{A point along u where scattering occurs}\\
&O \quad \text{The point from which the observation is made}\\
&u \quad \text{Center line of observation}\\
&h \quad \text{Height of Q, relative to the city}\\
&s \quad \text{distance between light source X and Q}\\
&\theta \quad \text{angle XOQ}\\
&\phi \quad \text{angle OXQ}\\
&z \quad \text{Zenith angle of u}\\
&\psi \quad \text{Zenith angle of XQ}
\end{aligned}
\tag{2.2}
$$

2.3.1 Sky brightness

$$b = \pi \ N_m \ \sigma_R \ e^{-cH} \times \iint \frac{\int_0^\infty I_{up} \ s^{-2} \ EF_{XQ} \ EF_{QO} \ DS \ AT \ \ du}{\pi R^2} \ dxdy \tag{2.3}$$

The double integral $\iint dxdy$ is the integration over every point on the area of the city, and the integral $\int_0^\infty du$ is the integration along the centerline of observation u. These integrations in simpler terms produce a result which takes into consideration how much light there is at every point Q along the line u, from every source X.

The only difference between here Garstang's original calculation of b and the implementation of it is the factor πR^2 which was changed to $(2R)^2$ such that instead of the city being a disk it is now a square. This is due to the fact that the data is based on square pixels.

2.3.2 Upward intensity

In 1986 Garstang did not have access to satellite data but instead estimated the total luminosity of the city based on its population, which is why LP (Lumen · Population) is used as the total light output of the city. Garstang used this factor in a I_{up} function which describes how much light is emitted upwards.

$$I_{up} = \frac{LP}{2\pi} \ (\ 2G(1-F)cos(\psi) + 0.554 \ F \ \psi^4) \tag{2.4}$$

2.3.3 Extinction functions

When light travels through the atmosphere it is scattered and absorbed Garstang wrote two equations EF_{XQ} and EF_{QO} which describe these phenomena. These functions only differ due to the fact that the observer may be at some height A above the surface relative to the city center as shown in figure 1.4. In his 1989 paper Garstang writes these functions in different forms, however as shown here they can be simplified into a single equation accepting multiple parameters. However another condition is added to the EF_{OQ} function since it is only valid up to the point u_c which is the point along u closest to the source X, it is not shown here but implemented in software.

The overline on variables in function p 2.5 is used to mark function variables, such that they are not confused with similarly named variables previously mentioned in 2.1 and 2.2.

$$p(\overline{c},\overline{u},\overline{\psi},\overline{A}) =$$

$$\overline{c}^{-1}e^{-\overline{cA}}sec(\overline{z})\{1-e^{-\overline{cu}\ cos(\overline{z})}+\frac{\epsilon tan^2(\overline{z})}{2\overline{c}(E+H)}((\overline{c}^2\overline{u}^2cos^2(\overline{z})+2\overline{c}\ cos(\overline{z})+2)e^{-\overline{cu}\ cos(\overline{z})}-2)\}$$

$$\begin{aligned} p_1 &= p(c,u,z,A) \\ p_2 &= p(a,u,z,A) \\ EF_{QO} &= \exp(-N_m\sigma_R e^{-cH}p_1 + 11.778K\ p_2) \\ \\ f_1 &= p(c,s,\psi,0) \\ f_2 &= p(a,s,\psi,0) \\ EF_{XQ} &= \exp(-N_m\sigma_R e^{-cH}f_1 + 11.778K\ f_2) \end{aligned} \tag{2.5}$$

EF_{XQ} is defined by Garstang as EF_{QO} with $A = 0$ and $z = \psi$ which results in a simpler equation. However in software only EF_{QO} is implemented as a function and the values A and ψ and s are given as inputs.

2.3.4 Double Scattering

The luminous flux at the point Q is not only determined by the light which directly arrives from the source X but also from light which is scattered from some point around Q back towards Q, and some of this light will also reach the observer. DS is an equation that describes the amount of double scattering that occurs.

$$DS = 1 + (\gamma f_1 + 11.11K\ f_2) N_m \sigma_R e^{-cH} \tag{2.6}$$

2.3.5 Atmospheric density

In order to accurately approximate how light behaves in the atmosphere Garstang added what in this thesis is named AT. This equation models how scattering behaves as a function of height h and the angle of observation θ.

Originally Garstang did not define AT separately from b 2.3. But in the interest of trying to fit the function b in one page and simplify its explanation it's defined here as AT.

$$AT = (\ e^{-ch}\ 3\frac{1\ +\ cos^2(\theta\ +\ \phi)}{16\pi}\ +\ e^{-ah}\ 11.11K\ f(\theta + \phi)\ ) \tag{2.7}$$

$$\begin{aligned} 0^\circ \leq \theta \leq 10^\circ,&\ \ f(\theta) = 7.0\ e^{-0.2462\ \theta} \\ 10^\circ < \theta \leq 90^\circ,&\ \ f(\theta) = 0.9124\ e^{-0.0.04245\ \theta} \\ 0^\circ < \theta \leq 180^\circ,&\ \ f(\theta) = 0.02 \end{aligned} \tag{2.8}$$

2.4 Calculations using linear algebra

R.H. Garstang's model is a three dimensional system, which is why the author chose to deviate from Garstangs original method of calculation and instead implement the calculations using linear algebra, this allows the mathematical implementation to be more readable as software.

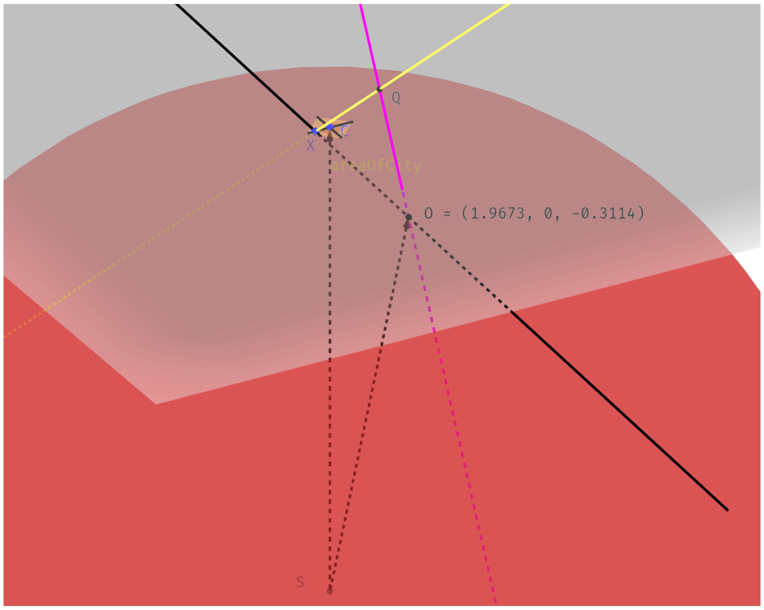

Figure 2.2: A three dimensional interactive model was built in Geogebra, the aim of this was to provide a testing platform to test and verify the math. These geogebra files can be found in the repository under the garstang folder.

2.4.1 A mathematical optimization

Garstang's calculations are based on the length of u however when doing these calculations numerically the integration step length du is very important. If du is constant it causes an issue where the distribution of the points Q_0 to Q_n will be evenly distributed along the line OQ see figure 2.3 but in terms of distance to the point X the points will distribute themselves with higher density furthest from X and the fewest close to X. In practice we are then using more resources to calculate the least important sections of the integral and less resources to calculate the important parts of the integral figure 2.3.

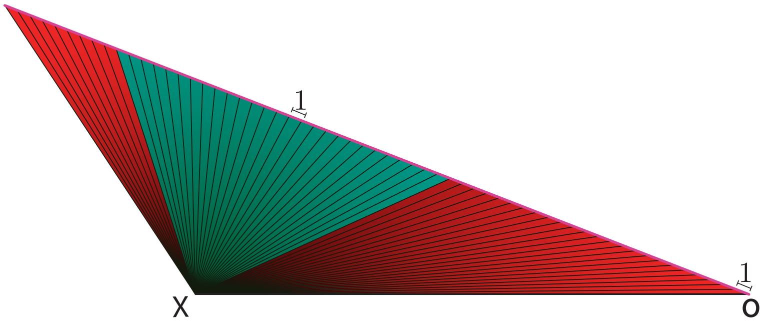

Figure 2.3: Illustration of the distribution of Q if du the integration step is constant. The intersections along the pink line u are where each point Q would be. Notice how the intersections have higher density in the red areas (higher distance to X) than in the green area (closer to X).

The solution used is to calculate du based on the angle ϕ. This has the effect that the points Q are distributed with the highest density close to the light source and fewer far away from the light source figure 2.4.

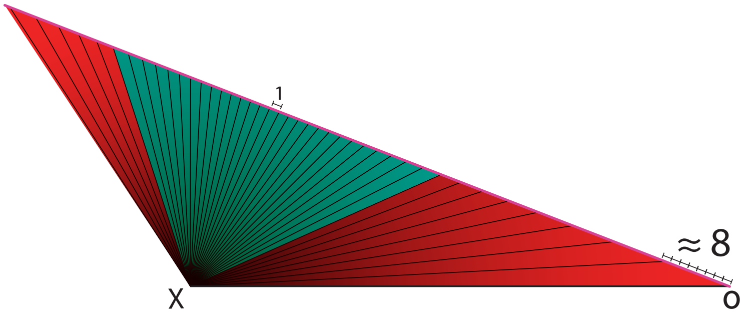

Figure 2.4: Illustration of the distribution of Q if du is variable and based of the angle ϕ. The illustration shows that the intersections between the pink and black lines, which would be the points Q_0 to Q_n along the line u have a higher density the closer they are to X

2.4.2 Performing calculations

This subsection shows how the variables defined in 2.2 are calculated. Some of these variables are set and do not need calculations however they are still variable in the sense that they can be accepted as input.
H and A are in this case set to 0.

D, the arc distance between the city and the observer is given as input. X, the point of emission is calculated through iteration over the area of the city.

Finding the observers position

The Garstang model is uniform around C and therefore O can be placed anywhere as long as the distance is correct, for the sake of simplicity O is forced to be on the xz plane, this means O can be determined using:

$$O = \begin{cases} x = E(\frac{D}{E+H}) \\ y = 0 \\ z = E \cdot cos(\frac{D}{E+H}) - (E + H)) \end{cases} \tag{2.9}$$

The distance between city and observer

Now that the points X and O are determined the distance d between them can be caclulated.

$$d = |\vec{OX}| = \sqrt{XO_x^2 + XO_y^2 + XO_z^2} \tag{2.10}$$

Direction of observation

In order to simplify calculations in the Garstang model and be able to scan the horizon with respect to the observer, a directional vector $\vec{V_{dir_l}}$ is required as input, this directional vector is later normalized to length 1, but it is used to tell where the observer wants to look. The vector $\vec{V_{dir_l}}$ is on a local coordinate system, which is identical to the global one except for the fact that it is relative to the observers horizon. From the observers perspective the city is located towards the $-x$ direction, $0y$, and some height z.

Rotating the observational direction vector

Before the vector $\vec{V_{dir_l}}$ can be used it must be rotated to fit the global coordinate system. This is done using a rotation matrix R_y which rotates the vector about the y axis.

$$R_y = \begin{bmatrix} cos(Y_{rot}) & 0 & sin(Y_{rot}) \\ 0 & 1 & 0 \\ -sin(Y_{rot}) & 0 & cos(Y_{rot}) \end{bmatrix} \tag{2.11}$$

The angle of rotation for Y_{rot} can simply be calculated using the fact that both C and O are on the same plane. The angle Y_{rot} in radian is given by the arc distance D divided by the radius of the system $E + H$ see figure 1.4.

$$Y_{rot} = \frac{D}{E + H} \tag{2.12}$$

The direction of observation $\vec{V_{dir}}$ in the global coordinate system is then given by:

$$\vec{V_{dir}} = Ry \cdot \vec{V_{dir_l}}. \tag{2.13}$$

Finding θ

It's now possible to calculate the angle θ by finding the angle between the vectors $\vec{OX}$ and $\vec{V_{dir}}$.

$$\theta = cos^{-1}\left(\frac{\vec{OX} \cdot \vec{V_{dir}}}{|\vec{OX}||\vec{V_{dir}}|}\right) \tag{2.14}$$

Finding Q using the law of sines

Although the angle ϕ has not been calculated it will be set once we start integrating. Using the information which has now been calculated or set, a triangle $\triangle XQO$ can be defined, by the distance d and the angles ϕ and θ figure 2.5.

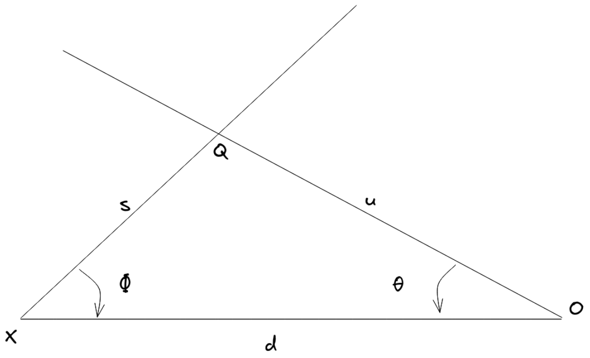

Figure 2.5: Illustration of triangle, only d, ϕ and θ are known.

In fig 2.5 the length u can be found through the law of sines 2.15. This is also the function where the angle ϕ is iterated on, and by doing so the integration step length du becomes variable.

$$u = d \cdot \frac{sin(\phi)}{sin(\pi - \phi - \theta)} \tag{2.15}$$

We can now use this length u to find the point Q, since the length of the direction vector is $|\vec{V_{dir}}| = 1$ the point can simply be calculated through 2.16.

$$Q = O + u\vec{V_{dir}} \tag{2.16}$$

Now that the point Q is known s can be found $s = |\vec{XQ}|$, the same method as in 2.10. And since the integral is calculated by iterating through the angle ϕ the number of iterations needed to achieve an accurate result is drastically reduced.

Determining the start of integration

By using this approach of calculating the integral based of ϕ some of the more complicated parts in Garstangs model are simplified. Such as the point u_0, if the city is below the horizon of the observer its light will not reach the entire line of observation OQ but a section of it starting at u_0. Garstang sets up rules for finding this point and determining if the observer O is above or below the city's tangent plane on a spherical earth. In the implementation used here finding this point is trivial and is simply done by checking if $O_z \geq 0$ and if not find the starting angle ϕ_0 2.17.

$$\phi_0 = \theta = cos^{-1}\left(\frac{\vec{XO} \cdot (1,\vec{0},0)}{|\vec{XO}||(1,\vec{0},0)}|\right) = cos^{-1}\left(\frac{\vec{XO}_x}{|\vec{XO}|}\right) \tag{2.17}$$

Another benefit is that finding the point Q which has the shortest distance s_m to X is simply done by setting the angle $\angle OQX$ in function 2.15 to $\frac{\pi}{2}$ which gives equation 2.18 and calculating s the same way as earlier $s = |\vec{XQ}|$.

$$u = d \cdot \frac{sin(\phi)}{sin(\frac{\pi}{2})} = d \cdot sin(\phi) \tag{2.18}$$

2.5 Satellite data

The common data source for nighttime lights is NASA and NOAA's joint Suomi National Polar-orbiting Partnership satellite named after the pioneering US meteorologist Verner E. Suomi. The Suomi satellite 2.6 and its predecessors JPSS-1,2,3 and 4 are all intended as meteorological satellites.

Figure 2.6: Image of NASA and NOAA's NPP Suomi satellite (Source: [17])

The VIIRS low-light imaging instrument's intended use was to enable the detection of clouds using moonlight instead of sunlight [18]. Although not originally intended, the ability of these satellites to accurately detect surface lighting significantly broadens their potential use, making them suitable for a wide variety of different applications across numerous fields of study.

2.5.1 The orbit

The Suomi NPP satellite and it predecessors observes the Earth's surface twice every 24-hour day, once in daylight and once at night. NPP flies 824 kilometers above the Earth's surface in a sun-synchronous orbit with an orbital period of 101 minutes, circling the planet approximately 14 times a day. NPP transmits its data once per orbit to the ground station in Svalbard, Norway, and continuously to local direct broadcast users.[19] This high availability of data is crucial for meteorological observations but it also benefits surface light observation, since over time a map without cloud coverage can be generated. There are other satellites with similar orbits but not with measurements taken at night.

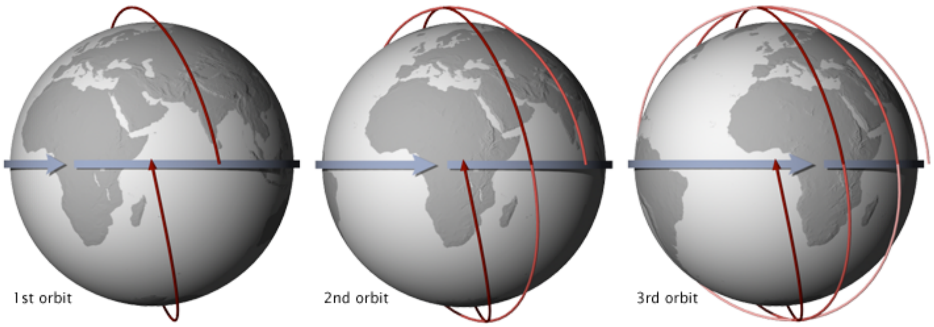

Figure 2.7: Image curtesy of NASA showing a sun synchronous orbit, which is beneficial due to the angle which the sunlight arrives in is near constant (Source: [20])

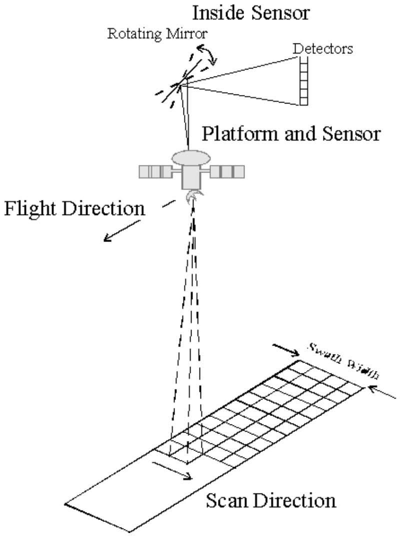

Figure 2.8: Figure showing cross track scanning method employed by VIIRS DNB sensor (Source: [21])

2.5.2 How the VIIRS sensor works

The VIIRS DNB sensor employs a cross track scanning technique with a swath of 3000km, in terms of degrees the swath is $+-56°$, it has a near constant spacial resolution of 742 meters. The large swath is what allows the satellite to cover the entire surface twice a day. The constant pixel size which the sensor produces are processed into a simple equirectangular projection where each pixel is 0.004166666666666666609 degrees latitude and longitude and the datum used by the satellite is WGS84. The value and its rather long representation is in fact significant in its entirety since when moving millions of pixels along the image the error would accumulate if the number was not accurate. The important part to remember here is that tough the sensor measures at near constant distance once projected the pixels sizes are constant in terms of degrees latitude and longitude.

The VIIRS day night band sensor's spectral response measures between the wavelengths 0.5 to 0.9 μm. This puts the VIIRS DNB in the near infrared spectrum and although this is the best source of publicly available nighttime spectral data it is not necessarily an accurate representation of actual light emission from an electrical source. This can be compensated for however due to time limitation it is not implemented here but discussed in the calibration chapter 3.

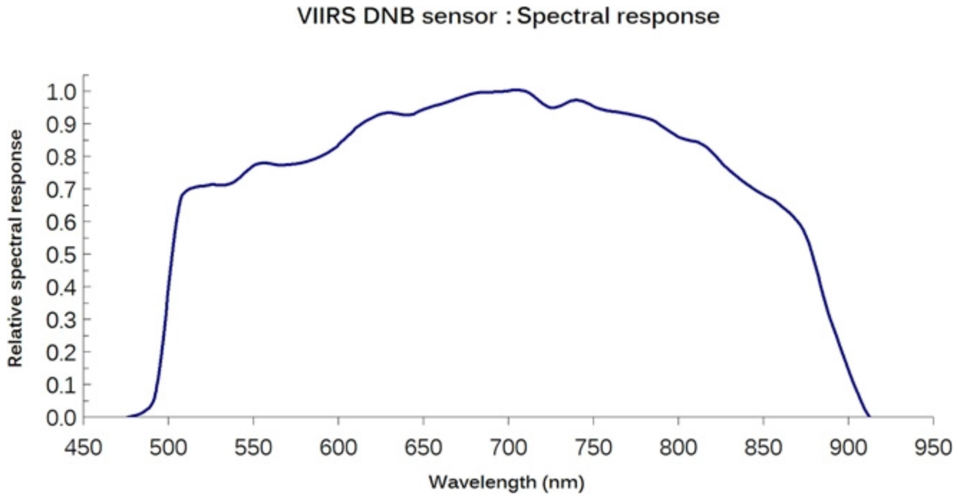

Figure 2.9: Figure showing VIIRS DNB sensor spectral response, this shows how sensitive the sensor is to certain wavelengths. (source: [15])

2.5.3 Data availability

NASA makes the VIIRS DNB night time dataset public in what it calls the ”Black Marble Nighttime Lights Product Suite”. The relevant dataset is called VNP46A2, this is a product containing preprocessed data which dates from present day all the way back to 2012 and has a temporal resolution of one day. The dataset is updated every 24h.

VNP46A2 is preprocessed such that pixels containing clouds are removed, atmospheric conditions are compensated for, seasonal error correction is applied, and it is moonlight BRDF-corrected [22]. Even with this preprocessing the data contains some noise at the lowest values and there is at this moment no good way of filtering out auroras.

However since the data comes in several temporal resolutions the decision was made to use the VNP46A2 at a temporal resolution of a day since one can find individual clear nights where aurora is not visible. It was unfortunately not possible to aggregate and process several such clear nights in order to generate a more accurate dataset in time for this thesis deadline, although such a dataset should be used for these computations.

Chapter 3

Software design and implementation

3.1 Programming language

Given the extensive data set and high computational requirements, it was clear from the beginning that the project needed to be written in a language capable of fast execution. This ruled out some of the more modern, high-level programming languages like C# and Python.

The viable alternatives left were Rust and C. Of these two, the author possessed the most experience with Rust. Furthermore, Time And Date is currently transitioning their codebase to Rust. This transition further affirmed the decision to develop the software using the Rust Programming Language.

3.1.1 The benefits of Rust

As a programming language, Rust is unique in its safety features including memory safety, type safety, null safety, and thread safety. These safeties are attributed to Rust's capability to enforce these guarantees at compile time, made possible by its stringent adherence to rules governing ownership and borrowing memory. While it is possible for developers to bypass these safety measures, they are an intrinsic aspect of the programming language, making any such circumventions obvious. This transparency in Rust's error handling and memory management encourages developers to pay extra attention to these areas, this helps prevent unforeseen behavior by assuring that there is no undefined behavior.

Rust is fast, and depending on implementation and compiler optimizations it can accelerate performance by a large extent. Rust also boasts a comprehensive standard library that reveals low-level, architecture-specific instructions. A prime example of this is the atomic 64-bit unsigned integer type, which allows multiple threads to read and write to the same memory section simultaneously, eliminating mutation locks. As a result, it removes the overhead typically associated with synchronous operations.

3.2 Deployment

Even though this project didn't reach its final form, decisions about software deployment were made early on. The primary requirements centered on ease of deployment, with a focus on creating a set of containerized services capable of operating independently. In Rust, this was accomplished by creating a Rust virtual workspace. Essentially, this is a folder housing several interconnected Rust projects. The advantage here is the proximity of the code, allowing each underlying project to import code from the others with relative ease.

In the end two services and one library were created:

- Sattelite data retrieval service
- Map generation service
- Light pollution model library

All of these components reside in the virtual workspace, which also serves as the root directory of the repository (link). This project leverages a

Continuous Deployment (CD) workflow on GitHub, utilizing Docker and Docker-compose. This approach allows for the building of service container images, which can then be conveniently downloaded using GitHub packages.

The reason for using GitHub packages is that they are easy to maintain, updated automatically and can easily be retrieved and replaced when deploying, in addition they allow for fine grained access control.

Source code 3.1: Multi stage build using Dockerfile, this Docker container is responsible for building the binaries

```
FROM docker.io/fedora:37

RUN dnf update -y

RUN dnf install -y Rust cargo gdal gdal-devel hdf5 hdf5-devel …
    pkg-config openssl-devel

WORKDIR /lp

COPY . .

RUN cargo build --release --bin map_generation

RUN cargo build --release --bin sat_dl
```

The project construction involves several steps. The initial phase involves creating a common build container capable of generating the binaries for later deployment. In the subsequent stage, these binaries are extracted and copied into a clean environment where only runtime dependencies are installed. This approach facilitates the use of smaller, more efficient images.

Source code 3.2: Second stage for satellite data downloader, this is where the image which is ready for deployment is built

```
FROM docker.io/fedora:37

RUN dnf update -y

RUN dnf install -y gdal gdal-devel hdf5 hdf5-devel pkg-config …
    openssl-devel

COPY --from=build_image ./lp/target/release/sat_dl /sat_dl/

WORKDIR /sat_dl
```

For the map generation service there are fewer dependencies which makes for a smaller image, once the binary is copied in it can be run with no additional requirements.

Source code 3.3: Second stage for the map generator, which creates an image with the binary copied from the build stage, this image does not require any additional dependecies.

```
FROM docker.io/fedora:37

COPY --from=build_image ./lp/target/release/map_generation …
    /map_generation/

WORKDIR /map_generation
```

The entire process is orchestrated by a compose.yml file. This is where Docker-compose constructs the build image and the two services. It then mounts a shared folder so that the map generator can utilize the data downloaded by the satellite downloading software.

Source code 3.4: Example compose file for deployment

```
version: "3"
services:
  build_image:
    build:
      context: ./
      dockerfile: Containerfile
    image: build_image

  sat_dl:
    build:
      context: .
      dockerfile: dl_service.dockerfile
    volumes:
      - ./archive://archive:Z
    depends_on:
      - build_image

  map_generation:
    build:
      context: .
      dockerfile: map_gen.dockerfile
    volumes:
      - ./archive:/archive:Z
    depends_on:
      - build_image
```

For deployment a rather simple compose file can be used, however since the project did not reach a deployable state a demo was created to illustrate the concept. In this example 3.5 the compose file downloads the leaflet demo image from GitHub and runs it. The demo dockerfile resides in the leaflet_demo folder in the repository (link), it is built and deployed using a GitHub workflow which runs after pull request merges into main.

Source code 3.5: Example of deployment compose file which downloads the images from github packages

```
version: ”3”
services:
  leaflet_demo:
    image: ghcr.io/ardijanr/light_pollution_map:pr-6
    ports:
      - 127.0.0.1:3000:3000
```

3.3 Data retrieval

3.3.1 Design

Working with satellite data often presents challenges, primarily because different satellite products or missions have distinct methods of data storage. These differences aren’t confined to the data format; they also extend to how each measurement is separated in time. For instance, some products structure their data in folders divided by year/month/day/hour/minute, while others incorporate this information into the filename and place all the files in one directory, expecting users to predict the filename and query for it while not knowing whether it exists. This absence of standardization complicates the development of a generalized implementation, particularly when data from multiple satellites or products needs to be acquired.

For this project, only a single data source was needed: NASA’s ’Black Marble’ product, known as VNP46A2. There are several ways to retrieve

VNP46A2. NASA's own guides recommend using the terminal program 'curl' for data downloads. While a terminal program could manage a handful of manual downloads, handling multiple simultaneous downloads would necessitate a shell script or interfacing with the shell through Rust. Instead, this project leveraged the API exposed by NASA's LAADS DAAC.

When downloading such large datasets a few key requirements become apparent.

- Download verification
- File corruption detection
- Automatically retry missing or corrupted files
- Directory synchronization

Besides downloading the data, the download service also needs to ensure that the data is available in a more accessible format that can be easily shared between services and seamlessly read and parsed at each stage. For this reason it was decided that the service would download the data in its original format, and upon completion, convert it into a GeoTIFF format for further processing.

3.3.2 Implementation

The download service works with dates as input. given two arguments the start and end dates in the format "d.m.yyyy" it will then start downloading the data between those time intervals. If the end date is not given it assumes you gave it a from date and it will download from the given date to the current date.

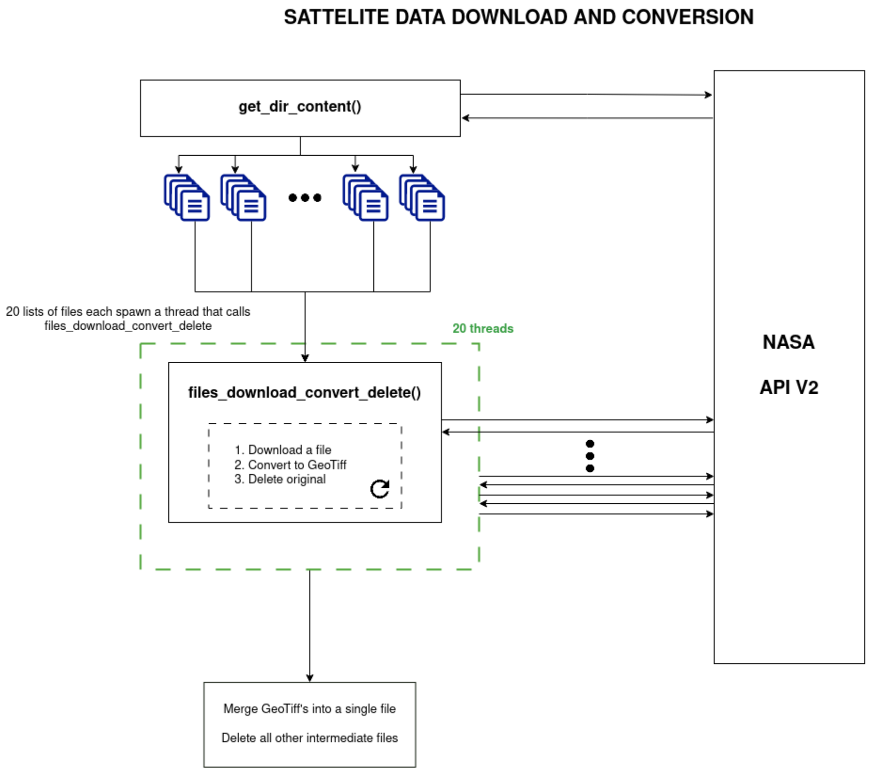
SATTELITE DATA DOWNLOAD AND CONVERSION
get_dir_content()
20 lists of files each spawn a thread that calls files_download_convert_delete
20 threads
files_download_convert_delete()
1. Download a file
2. Convert to GeoTiff
3. Delete original
NASA
API V2
Merge GeoTiff's into a single file
Delete all other intermediate files

Figure 3.1: Data download service design diagram

In source code 3.6 the .expect() statement panics if the value is not of type Some<String> this means that the parser failed to parse the user input, and aborting the program in this case is fine since we want to allow the user to correct their input. The code performs a series of checks to validate the input. This function will always return a tuple containing the corresponding start and end dates. Since Rust cannot return None or Null if this function executes successfully and the program does not exit you are guaranteed that the output is of the denoted type.

Source code 3.6: Argument parsing

```
fn parse_arguments(arguments: Vec<String>)->(NaiveDate,NaiveDate){

    if arguments.len()<2{
        println!(”Missing argument starting date d.m.y”);
        exit(1)
    } else if arguments.len()>3{
        println!(”Incorrect arguments, supply start and end date example: 1.1.2012 1.1.2020”);
        exit(1);
    }

    let start_date = NaiveDate::parse_from_str(&arguments[1], ”%d.%m.%Y”).expect(”Unable ...
         to parse start date”);
    let end_date;

    if let Some(val) = arguments.get(2){
        end_date = NaiveDate::parse_from_str(&val, ”%d.%m.%Y”).expect(”Unable to parse ...
             start date”);
    } else {
        end_date = chrono::offset::Utc::now().date_naive();
    }


    if start_date>end_date{
        println!(”Incorrect arguments, start date is after end date”);
        exit(1);
    }

    (start_date,end_date)
}
```

Each Rust program begins with a main function, however as seen in the code reference 3.7, the main function features a macro attribute on the first line, invoking an external crate known as Tokio.

Tokio is an asynchronous runtime library for Rust, facilitating the development of asynchronous functions. These functions can be identified by the ’async’ keyword, as shown on line 2. The inclusion of Tokio in this scenario is motivated by its capability to streamline the implementation of parallelization wherever required.

Source code 3.7: Argument parsing

```
 1  #[tokio::main(flavor = ”multi_thread”, worker_threads = 10)]
 2  async fn main() {
 3      dotenv::dotenv().expect(”Missing .env file!”);
 4      env::var(”TOKEN”).expect(”Missing environment variable TOKEN”);
 5
 6      //Parse command line arguments to get the start and end dates
 7      let (mut current_date,end_date) = parse_arguments(env::args().collect::<Vec<String>>());
 8
 9      let client = Client::new();
10
11      while current_date<end_date{
12          println!(”Downloading date: {current_date}”);
13          let _ = dl_date_and_convert(current_date.year() as u32, current_date.ordinal(), ...
                 client.clone()).await;
14
15          current_date += Duration::days(1);
16      }
17  }
```

The main function's operations are simple: it first checks for a .env file in the directory, which is where the API token needed by NASA's API is kept. This is followed by parsing command line arguments, executed by invoking the parse_arguments function shown in 3.6.

Assuming the program doesn't terminate by this point, a while loop will start downloading data for each day within the specified start and end dates. The progress is displayed by indicating the current day being downloaded, as seen on line 12 in 3.7. At line 13 in 3.7, a function is called to download a single day's worth of data. The inputs provided to this function are the year and the day of the year, both in integer form. It's important to note that the day of the year ranges from 1 to 366, accommodating leap years. This is reflects NASA's directory structure, which follows the format "VNP64A2/<dataset-name>/<yy>/<ddd>/".

The source code responsible for downloading a day's worth of data initially checks if the data already exists. This is done by searching for a merged data file, as shown on line 12 in 3.8. The existence of this merged file indicates whether the data for that day has been successfully downloaded and merged at an earlier time.

In the absence of the merged data file, the execution proceeds to line 36. Here, the program retrieves the directory content specific to that day. After the API responds with JSON list, each value in the list is iterated over and converted into a FileEntry type. The program then collects the valid entries into a vector.

Source code 3.8: Argument parsing

```
1   //Downloads a certain date with data and deletes intermediate files afterwards.
2   pub async fn dl_date_and_convert(
3       year: u32,
4       day: u32,
5       client: Client,
6   ) -> Result<(), Box<dyn std::error::Error + Send + Sync>> {
7       let date_path = format!("{year}/{day}");
8       let download_dir = format!("{}/{}", download_dir(), date_path);
9       let url = format!("{}/{}", PRODUCT_URL, date_path);
10
11      //Check if folder allready contains the merged file
12      if let Ok(content) = std::fs::read_dir(&download_dir) {
13          let contains_merged_data = content
14              //This will return a single empty Some() if the the merged_data file exists
15              //Filter map will remove any None values
16              .filter_map(|x| -> Option<()> {
17                  if x.ok()?
18                          .file_name()
```

```
                        .into_string()
                        .ok()?
                        .contains(”merged_data_”) {
                            return Some(());
                    }
                    None
            })
            // Once collected into a vector we can check its length to know if the file exists
            .collect::<Vec<()>>().len() > 0;

        if contains_merged_data {
            println!(”Found merged file in {} skipping!”, download_dir);
            return Ok(());
        }
    }

    // Get the remote directory content and create file entries from them.
    let files = get_dir_content(url, client.clone())
        .await?
        .iter()
        .filter_map(|f| -> Option<FileEntry> {
            if !f.name.ends_with(”.h5”) {
                return None;
            }

            Some(FileEntry {
                local_path: download_dir.clone(),
                name: f
                    .name
                    .trim_end_matches(”.h5”)
                    .split(”.”)
                    .collect::<Vec<&str>>()
                    .join(”_”),
                download_link: f.downloadsLink.clone(),
            })
        })
        .collect::<Vec<FileEntry>>();

    let mut set = JoinSet::new();

    let _: Vec<_> = files
        .chunks(files.len() / PD)
        .map(|chunk| {
            set.spawn(files_download_convert_delete(
                chunk.to_owned(),
                client.to_owned(),
            ))
        })
        .collect();

    //Wait for downloads and GeoTIFF generation to complete
    while let Some(_) = set.join_next().await {}

    // Validate that all files are downloaded if not display whats missing
    let _ = files.iter().inspect(|f| {
        if !std::path::Path::new(&f.tif_path()).exists() {
            println!(”{} is missing!”, f.tif_path());
        }
    });

    // Merge GeoTIFF into one file
    merge_GeoTIFFs(download_dir, format!(”merged_data_{year}_{day}”)).await?;

    Ok(())
}
```

In the source code 3.8, on line 60, the vector containing file entries is divided into equally sized chunks, the number of chuncks and later threads depend on the PD (Parallel Downloads) variable. Each chunk is then launched as a Tokio task which assumes ownership of the respective chunk and a copy of the HTTP client. These tasks return a join handle which is awaited at line 70, this makes the main thread wait for the tasks to finish.

Line 73 in 3.8 is a simple sanity check to verify that the files have been downloaded correctly and that no files are missing. If a problem is detected, the program will output what's missing but will assume the file is be corrupt in terms of conversion to GeoTIFF or cannot be downloaded. The final stage involves merging these downloaded tiles into a single file that contains the data for the entire Earth on that specific day.

3.3.3 HDF5 to GeoTIFF conversion

The conversion of data from hdf5 format to GeoTIFF presented a challenge. The difficulty stemmed primarily from a lack of detailed documentation. NASA does offer a Python script which makes use of GDAL which is a geographical information system (GIS) software. However using Python for this conversion process seemed excessive for this project and would introduce a considerable amount of additional dependencies.

Therefore, rather than leveraging Python, GDAL was utilized directly via a shell command, which can be seen on line 23 in 3.9. But to successfully generate a GeoTIFF file with the correct spatial metadata, the metadata of the HDF5 file needed to be parsed. This metadata holds essential information about the placement of tile bounds - without this, proper georeferencing of the tile is impossible.

Source code 3.9: HDF5 file handling, the file is read and checked its metadata is parsed in order to create a georefrenced TIFF

```
1  //Converts hdf5 dataset to GeoTIFF using gdal
2  pub async fn hdf5_file_to_geotif(
3      file: FileEntry,
4  ) -> Result<FileEntry, Box<dyn std::error::Error + Send + Sync>> {
5      let hdf_file = hdf5::File::open(&file.hdf5_path())?;
6
7      //Query the HDF file for information inside the group
8      let data = hdf_file.group(”HDFEOS INFORMATION”)?;
9
10      //Query the HDF file for information inside the group
11      let data_string = data
12          .dataset(”/HDFEOS INFORMATION/StructMetadata.0”)?
13          .read_scalar::<FixedAscii<32000>>()?;
14
15      let bounds = parse_metadata(&data_string).ok_or(”Unable to parse text”)?;
16
17      let data_path = format!(
18          r#”HDF5:”{}”:{}”#,
19          file.hdf5_path(),
20          hdf5_internal_data_path()
21      );
22
23      let translate = Command::new(”gdal_translate”)
24          .args([
25              ”-a_srs”,
26              ”EPSG:4326”,
27              ”-a_ullr”,
28              &bounds.west.to_string(),
29              &bounds.north.to_string(),
30              &bounds.east.to_string(),
31              &bounds.south.to_string(),
32              &data_path,
33              &file.tif_path(),
34          ])
35          .output()?;
36
37      if translate.status.code().ok_or(”No error code”)? != 0 {
38          return Err(”Unable build GeoTIFF from h5 file”.into());
39      }
40
41      Ok(file)
42  }
```

The relevant metadata in the hdf5 file was stored as ASCII string with a fixed length of 3200 bytes (as seen on line 11 3.9). In order to parse this metadata string a nom parser was written. Nom is a framework for building parsers in Rust, the core idea behind nom is to write small functions which individually parse a section of text and return the remainder. These tiny parsers can then be chained together and become a very powerful tool.

The function ’skip_and_find_values()’ in this case consumes the text until it identifies a match with the input text. After this match, it invokes the second parser, ’parse_coord()’, on the remaining text. The role of ’parse_coord()’ is to eliminate delimiters and attempt to find a match for the format (<float>,<float>), where it specifically looks for the delimiters and two values it can convert into a float with a comma separating the two. If a match is found, the function returns the value, having already converted it to a floating-point number.

As indicated by the comments on lines 18 and 19, the coordinates are given in millions, necessitating division by 10^6 to obtain the correct values.

Highlighting the point that can often lead to a bit of head-scratching when dealing with satellite data records is the puzzling naming of the coordinate corner points as ”UpperLeftPointMtrs” and ”LowerRightMtrs”. This naming caused considerable confusion during the software development process, as highlighted by the comment on line 16 in 3.10.

Source code 3.10: Metadata parser of the HDF5 files using Rust nom.

```
1   fn parse_metadata(input: &str) -> Option<TileBounds> {
2       fn skip_and_find_values<'a>(input: &'a str, match_text: &'a str) -> IResult<&'a str, ...
             (f32, f32)> {
3           preceded(tuple((take_until(match_text), tag(match_text))), parse_coord)(input)
4       }
5
6       fn parse_coord(i: &str) -> IResult<&str, (f32, f32)> {
7           let (out, (first, _, second)) =
8               delimited(tag(”(”), tuple((float, tag(”,”), float)), tag(”)”))(i)?;
9           Ok((out, (first, second)))
10      }
11
12      let (_, (west, north)) = skip_and_find_values(input, ”UpperLeftPointMtrs=”).ok()?;
13      let (_, (east, south)) = skip_and_find_values(input, ”LowerRightMtrs=”).ok()?;
14
15
16  // Coordinates are in millions
17  // Do not be fooled, this is not meters but actually degrees!
18  // UpperLeftPointMtrs=(150000000.000000,-20000000.000000)
19  // LowerRightMtrs=(160000000.000000,-30000000.000000)
20      Some(TileBounds {
21          north: north as i32 / 1_000_000,
22          south: south as i32 / 1_000_000,
23          west: west as i32 / 1_000_000,
24          east: east as i32 / 1_000_000,
25      })
26  }
```

3.4 Garstang model as software

When implementing light pollution model as software the goal was to stay as close to the mathematical version discussed in chapter 2.3 as possible. When writing math as Rust code the readability of the code becomes really poor, and in order to mitigate this as much as possible variable names and symbols were chosen to reflect the original model variable names and symbols.

Although using the symbol θ instead of the name ”theta” as variable name makes the code a bit cryptic, if given proper documentation such as this thesis the software is easier to understand and check for correctness if the code is as close to the math as possible. If the model is ever improved upon it is done so mathematically first, and afterwards the code will be updated to reflect the new mathematical model, not the other way around. This however does not mean errors cannot be made while implementing, for this reason a debugging system was implemented.

Source code 3.11: The first few lines of the Garstang 1989 function, most UTF-8 characters can be declared as variables in Rust. This which is levraged to produce code that is as close to the mathematical model as possible.

```rust
pub fn garstang_1989_calc(LP: f64, mut distance: f64, H: f64, A: f64,obs_direction: Vector3D) -> f64 {
    if distance < 0.325 {
        distance = 0.325;
    }

    let N_m: f64 = 2.55 * ten_to_pow(19); // Particle density at sea level
    let σ_r: f64 = 4.6 * ten_to_pow(-27); //aerosol scattering coefficient
    const c: f64 = 0.104; // Molecular scale height in km^-1
    const π: f64 = std::f64::consts::PI;
    const R: f64 = 0.325; // center from sides of a pixel or square city in km
    const K: f64 = 0.5; // Atmospheric clarity
    const E: f64 = 6371.; // Radius of the earth sea level
    const F: f64 = 0.15; // Fraction of light being emitted upwards
    const G: f64 = 0.15; // Amount of light being reflected from the ground
    const gamma: f64 = 1.0 / 3.0; // Arbitrary value, don't change...
    const ε: f64 = 16. / (9. * π);

    let a = 0.657 + 0.059 * K;
    ...
```

In order to debug the integration, Rust's conditional compilation was used, as illustrated on line 3 in 3.12. Print statements were used in place of a conventional debugger. The choice to use print statements stems from the fact that a debugger often provides information that is too detailed. During debugging, it becomes crucial to examine several integrations, compare them, and identify what may be contributing to an error or producing an incorrect value.

It's important to note that this code is excluded during production compilation, ensuring it has no impact on the final production code. This is done by the attribute on line 3 in 3.12 which tells the compiler to not include this code when compiling for production.

Source code 3.12: Garstang model debug implementation, which allows integral accuracy verification and optimization through comparison between each step along the integral.

```
pub fn garstang_1989_calc(LP: f64, mut distance: f64, H: f64, A: f64,obs_direction: ...
     Vector3D) -> f64 {
    ...
    #[cfg(build = ”debug”)]
    if DBG_LVL ≥ 1 {
        println!(””);
        println!(”---------------DEBUG-LEVEL-1-------------”);
        println!(”norm_dir: {norm_dir:?}, dir_rotated: {dir_vec_OQ_norm:?}, S: {S:?} , ...
             OS_angle: {OS_angle}, SR: {EH}, , dx: {dx}, dy: {dy} ”)
    }
    ...
    ...
    #[cfg(build = ”debug”)]
    if DBG_LVL ≥ 2 {
        let theta = rad_to_degθ();
        let phi_start = rad_to_degΦ(_start);

        println!(”---------------DEBUG-LEVEL-2-------------”);
        println!(
            ”X: {X:?}, XO: {XO:?} , θ: {theta} , Φ_start: {phi_start}, u_prev: {u_prev} ”
        )
    }
    ...
```

Rust standard types have extensive mathematical functions built right in, these were leveraged to a large extent allowing the model to be written completely within Rust's standard library. In addition Rust's type aliasing system was also leveraged allowing for more readable code, here at 3.13 one can see the definition of two types that are technically the same however their use cases are different.

Source code 3.13: Type aliasing allowing for much more readable code, even though these types are fundamentaly the same, they convey different meanings.

```
pub type Vector3D = (f64, f64, f64);
pub type Point3D = (f64, f64, f64);
```

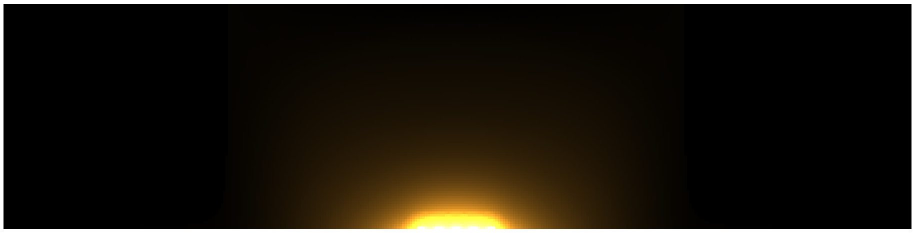

Figure 3.2: Image showing a test of the Garstang model, the image shows the night sky affected by city in the horizon.

Testing the light pollution model required the development of a straightforward program. This software was designed to iterate over every pixel of the sky, generating an image from the observer's perspective. Fast computations became possible through the benefit of the mathematical optimization; the calculation of the integral through the angle ϕ. This fast computation speed allowed for the generation of a sequence of images which illustrate how the night sky is affected by a city as distance increases. These images were then used to create a GIF animation, the animation can be viewed on the homepage of the repository on GitHub (link).

3.5 Generating light pollution maps

3.5.1 Design Considerations

The map generation software processes GeoTIFF files containing satellite data, it then applies the light pollution model and maps the result to a gradient and produce a image which shows the propagation of light pollution in the atmosphere. Due to the time restriction, this project has yet to create a world spanning map, though a map has been generated on a tile which covers most of the United Kingdom. The United Kingdom tile was selected because it is easy to recognize geographically and it has above average light pollution when comparing it to the rest of the tiles. A key aspect of the project is to optimize the software such that it is efficient and the map generation is as fast as possible.

Each tile is 2400x2400 which becomes approximatly 1780x1780 km near the equator, there are 5760000 pixels in each tile. Reducing the amount of processing done for each pixel is key, several methods were considered however only one proved to be fast enough to consider.

In their 2015 paper P. Cinzano, F. Falchi, and C.D. Elvidge [15] wrote about the time it took for them to generate their world atlas however since the author could not gain access to their software it is unknown what algorithm they employed for computing their map.

From the authors perspective there are two main ways of producing the map. These can be rather simply conveyed through two questions, the questions are framed as if it was the center pixel posing them.

1. How much am I being affected by the pixels around me?
2. How much am I affecting the pixels around me?

The algorithm for the first question is here named ”the crawl method” since the most efficient implementation of this question crawls along the image. The nature of this question requires it to be asked once for all pixels when generating the map.

The second question has been named here as ”the stencil method”, since it can be thought of as a stencil being placed in an area with its center being the source.

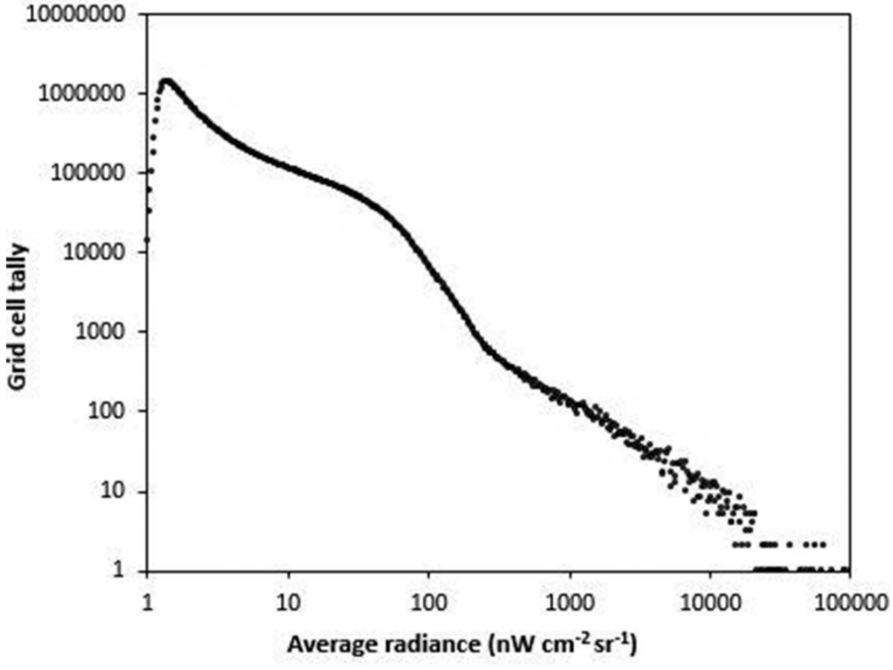

Figure 3.3: Radience distribution show how most values are below 10000 (Source: [18])

Over 99% of the nonzero radiance values contained in the dataset are between 1 and 20000 (see fig 3.3). Rather than recalculating these values every time, a cache can be built by calculating the light pollution value at 100m intervals for each radiance value. By setting the max distance to 300km the cache becomes a vector with the dimensions of 20000x3000, producing 60 000 000 possible combinations. Although this is memory intensive, it speeds up computations by a huge amount, this cache is read only after creation and therefore can safely be shared between threads, lookup is O(1) on heap allocated memory.

3.5.2 The crawl method

The crawl method can be implemented using a rather naive approach, by going through every pixel and calculating each neighbouring pixels contribution within a radius. This naive approach is demonstrated in the figure 3.4 as the first row of images.

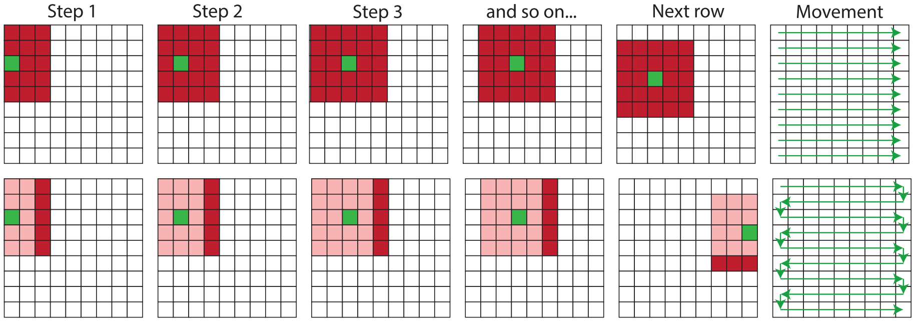

Figure 3.4: Figure shows how the crawl method works, and its movement pattern.

The more optimized version of the crawl method is shown in the second row of images in figure 3.4. This version does not check all surrounding pixels every time, instead it keeps a list of all the pixels which have a value that can contribute. By keeping a list of meaningful values it can skip checking every surrounding pixel every time it moves, and instead only check the leading edge where its moving towards. By checking the leading edge it can then update the list and keep working with a smaller dataset. Values are removed from the list if they are outside the radius.

The problem with this optimized crawling method is that it becomes rather complicated because the optimal movement pattern as shown in rightmost square in fig 3.4 makes the code a bit more bloated.

Even though the optimized version of the pixel iteration method is faster it still has to consider every single pixel multiple times if n is the number of pixels and the radius is 200 pixels it still has to go through every single pixel at near 200 times, this is because it must check pixels it has checked at some earlier point when going backwards.

3.5.3 The stencil Method

The stencil method proved to be the fastest method, however this method does require some preprocessing where every pixel in the tile is checked such that zeroes and invalid values can filtered out. The only thing left after filtering are pixels which can affect neighbouring pixels.

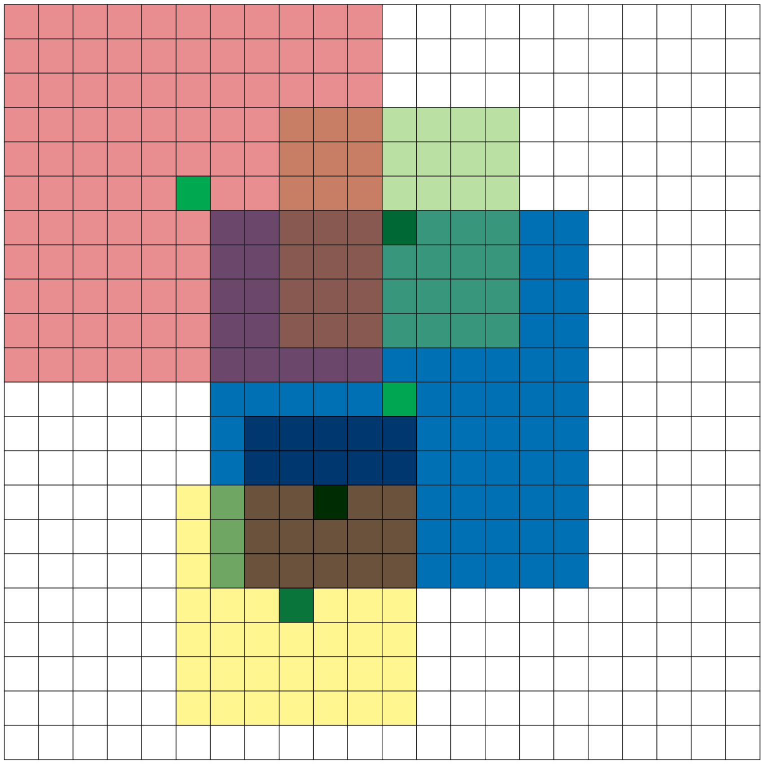

Figure 3.5: Figure shows how the stencil method works, each green square is the luminous source which emits light and affects an area around it.

The algorithm then becomes rather simple; for each pixel that has a relevant value calculate the distance to every pixel it affects within a radius, and apply the value which can be found in the cache. The resulting image is identical to the crawl method but much faster since it does not need to consider every pixel, it does however have to consider overlapping pixels several times.

3.5.4 Implementing the Garstang cache

The code for generating the cached light pollution values is rather straightforward, instead of caching an entire 2d stencil for every light value, the cache is linear in with a length of 3001. The intended use case is that for any radiance pixel value we can directly index the cache by that value and again by the distance.

The matrix in 3.1, shows the cache structure. The row indexes denoted by r are the radiance values and the column indexes d are the distance values.

Due to the fact that small radiance values affect small areas the tiles should be dynamic in size. For this reason the last value in the distance indexed vector is reserved to inform; at what distance the light pollution value went to 0. This value at index 3000 is therefore not a light pollution value but rather a distance value which can be used to determine the radius of the stencil.

$$\text{garstang\_cache} = \begin{pmatrix} (r_0, d_0) & (r_0, d_1) & \cdots & (r_0, d_{2999}) & (r_0, d_{3000}) \\ (r_1, d_0) & (r_1, d_1) & \cdots & (r_1, d_{2999}) & (r_1, d_{3000}) \\ \vdots & \vdots & \ddots & \vdots & \\ (r_{20000}, d_0) & (r_{20000}, d_1) & \cdots & (r_{20000}, d_{2999}) & (r_{20000}, d_{3000}) \end{pmatrix} \tag{3.1}$$

An important factor in the map generation is where in the night sky the measurement is taken from. In this case the map is generated by sampling the direction towards the city at 45° angle, this direction was arbitrarily chosen, this choice should however be justified in the future and part of the calibration step.

Source code 3.14: Garstang cache implementation, this loads the cache from file or generates it if the file is missing. The generation takes around 5-10 minutes depending on the scaling factor which determines the radius of a tile.

```
fn get_or_create_garstang_cache()->Vec<Vec<u32>>{
    let mut garstang_cache = vec![vec![0; MAX_DIST + 1]; 20_000];
    let dir_xy = 1. / (2. as f64).sqrt();
    let obs_dir: (f64, f64, f64) = (-dir_xy, 0., dir_xy);

    //check if cache file exists
    if let Some(cache_file) = get_cache_from_file() {
        for l in 0..garstang_cache.len() {
            for v in 0..garstang_cache[0].len() {
                garstang_cache[l][v] = cache_file[l][v];
            }
        }
        return garstang_cache;
    }

    for light in 0..20_000 {
        for dist in 0..MAX_DIST {

            // DNB VIIRS data is in nano watts per square cm
            // Garstang model uses total luminosity in lumen
            // Conversion from nW/cm^2 to l/m^2 is 0.000000001*683*100*100 = 0.00683
            // Total luminosity is then 0.00683*750*750 = 3841.875
            let lp = garstang::garstang_1989_calc(
                (light as f64) * 3841.875 / 0.15, // LP
                (dist as f64) / 10., // Distance in km from hecto meters
                0., //H
                0., //A
                obs_dir, //dir
            ) * 100000.; //Scale the output since we need the more significant figures and ...
                it will be saved as a u64.

            // If the value is below threshold set last index to the max relevant distance
            // This index will then be checked later in order to provide the size of the ...
                stencil.
            if lp < 1. {
                if light % 100 == 0 {
                    println!(”source light: {light}, SETTING MAX INDEX: {dist} light_max: ...
                        {} ,light_min: {},”, garstang_cache[light][0], ...
                        garstang_cache[light][dist]);
                }

                //The size of the stencil in pixels
                garstang_cache[light][MAX_DIST] = dist as u32;
                break;
            }
            garstang_cache[light][dist] = lp as u32;
        }
        if garstang_cache[light][MAX_DIST] == 0 && light % 10 == 0 {
            garstang_cache[light][MAX_DIST] = MAX_DIST as u32 - 1;
            println!(”source light: {light}, SETTING MAX INDEX: {MAX_DIST} light_max: {} ...
                ,light_min: {}”, garstang_cache[light][0], garstang_cache[light][2999]);
        }

    }

    //Save the cache to file such that this calculation can be skipped in future
    write_cache_to_file(garstang_cache.clone());
    garstang_cache
}
```

The Garstang model returns values which are below zero, these must therefore be scaled up with some reasonable factor such that the values become greater than 1 and the stencils have a reasonable size. The size of the stencil is determined by when it falls down below one, this makes the scaling value important for knowing how big the tiles should be. The size of the stencil directly affects how many pixels need to be written too, and also how big the overlap is in areas and therefore the speed of the map generation software.

These scaling values are arbitrary at the moment decided upon only by what seems to fill up the distance array to a reasonable range. The values are arbitrary only because of the time limit for this project, there is a clear and well documented intention of calibrating the software such that it produces accurate results, further discussed in chapter 4.

Accelerating the calculation involves enabling multiple threads to operate on the same result matrix simultaneously. Yet, this can lead to race conditions. A race condition can occur when a thread aims to update a value in the result_matrix. This task involves reading the current value, adding to that value, and then writing a new value back. If another thread updates this same value between this threads reading and writing, the contribution of that other thread is lost, leading to an incorrect sum.

These race conditions are however solved using rusts atomic types. The result_matrix is a 2d array which holds 2400x2400 unsigned 64 bit atomic integers. These atomic integers work like any other integer except for the fact that they are thread safe, atomic data types are guaranteed to order reads with writes if they operate on the same atomic reference, this ensures that the sum is added up sequentially such that it remains correct. The atomic data type is the fastest thread safe data type there is in Rust, due to the fact that they do not require locks or other sync mechanisms which would add overhead.

The result matrix is of the ParMatrix type which holds the 2d atomic vector. Normally in order to operate on data with multiple threads the compiler requires the implementation of Sync and Send. However in our case the underlying atomic data type is thread safe, but the compiler has no way of knowing that we will always be operating on the underlying data type and would therefore not allow us to compile. For this reason we need to tell the compiler that this type is thread safe, and to do that without actually implementing anything requires the unsafe keyword on line 5 and 6.

Source code 3.15: Atomic paralell matrix type, this type is thread safe. Though unsafe is used on line 5 and 6 the way the datastructure is used is safe.

```
 1  pub struct ParMatrix {
 2      pub inner: Vec<Vec<AtomicU64>>,
 3  }
 4
 5  unsafe impl Send for ParMatrix {}
 6  unsafe impl Sync for ParMatrix {}
 7
 8  impl ParMatrix {
 9
10      pub fn new(size_y:usize , size_x:usize)->ParMatrix{
11          let mut matrix: Vec<Vec<AtomicU64>> = vec![];
12          for _ in 0..size_y {
13              let mut line = vec![];
14              for _ in 0..size_x {
15                  line.push(AtomicU64::new(0))
16              }
17              matrix.push(line);
18          }
19
20          ParMatrix { inner: matrix }
21      }
22
23      pub fn write(&self, x: usize, y: usize, v: u64) {
24          self.inner[y][x].fetch_add(v, Ordering::Relaxed);
25      }
26
27      pub fn read(&self, x: usize, y: usize) -> u64 {
28          self.inner[y][x].fetch_max(0, Ordering::SeqCst)
29      }
30  }
```

The map generator spawns a number of threads where each works on one stencil at a time. Due to the fact that each pixel is constant in degrees latitude and longitude 2.5.2 in order to calculate the distance the projection needs to be compensated for. This distance calculation is done through the use an external library which uses the WGS84 ellipsoid model. The Rust geo library which is used has multiple distance calculation algorithms that can be used. In this case the geodesic distance formula given by Charles F.F Karney [23] with a high accuracy is used. The repeated computation of this distance is however very costly which is why a compromise was made.

The distortion is calculated for the width and height of one pixel (line 10 and 21 3.16) and it is then multiplied up by the relative euclidean distance from the center pixel. Though this is not as accurate as calculating the distance from the source to every affected pixel every time. This part of the software should be a part of the calibration step in order to check for correctness when calculating distances. When compensating for the distortion the stencil is not necessarily a perfect square anymore which is why the stencil size for both x and y directions are calculated at line 27 and 28.

Source code 3.16: Stencil generation source code, shows how the stencils are created and distortion due to the projection is compensated for.

```
 1  //Calculate projection distortion in x
 2  let p1 = point!(
 3      x: offset_lat + (s_x as f64 * PIXEL_DIM),
 4      y: offset_lon + (s_y as f64 * PIXEL_DIM)
 5  );
 6  let p2 = point!(
 7      x: offset_lat + ((s_x + 1) as f64 * PIXEL_DIM),
 8      y: offset_lon + ((s_y) as f64 * PIXEL_DIM)
 9  );
10  let Δ_x = (p1.vincenty_distance(&p2).unwrap() / 100.) as f32;
11
12  //Calculate projection distortion in y
13  let p1 = point!(
14      x: offset_lat + (s_x as f64 * PIXEL_DIM),
15      y: offset_lon + (s_y as f64 * PIXEL_DIM)
16  );
17  let p2 = point!(
18      x: offset_lat + ((s_x) as f64 * PIXEL_DIM),
19      y: offset_lon + ((s_y + 1) as f64 * PIXEL_DIM)
20  );
21  let Δ_y = (p1.vincenty_distance(&p2).unwrap() / 100.) as f32;
22
23  // This calculates how wide and tall the stencil is
24  let stencil_size_x = (garstang_cache_ref[c as usize][3000] as f32/Δ_x) as i32;
25  let stencil_size_y = (garstang_cache_ref[c as usize][3000] as f32/Δ_y) as i32;
26
27  let s_x = s_x as i32;
28  let s_y = s_y as i32;
29
30  // Generates a stencil by writing directly to atomic array
31  for y_stencil_index in -stencil_size_y..stencil_size_y {
32      for x_stencil_index in -stencil_size_x..stencil_size_x {
33          let dist = length(Δ_x * x_stencil_index as f32, Δ_y * y_stencil_index as f32);
34
35          //If the distance is greater than 299.9km
36          if dist > 2999. {
37              continue;
38          }
39          let mut dist = dist as usize;
40
41          //If the distance is less than 300m
42          //This only applies where we are calculating the center pixel
43          if dist < 3 {
44              dist = 3;
45          }
46          let x = s_x + x_stencil_index;
47          let y = s_y + y_stencil_index;
48
49          if x < 0 || x ≥ dim || y < 0 || y ≥ dim {
50              continue;
51          }
52
53          result_image_ref.write(
            x as usize,
            y as usize,
            garstang_cache_ref[c as usize][dist] as u64,
        );
    }
}
```

Once all the values in the source dataset have been processed an image is generated by reading each result value and mapping it to a color. This was achieved through two external libraries named image and colorgrad. In order resent the result in a meaningful way the values must be scaled down, and order to be able to see more detail in the lower range of values as well as in the higher range the square root is taken before scaling the values by some factor (line 10 3.17).

The scaling makes the difference between the largest and the smallest value smaller, which helps when the gradient picks colors based on values between the range of 0.0 to 1.0. In this case the image is for demonstration purpose and in order to get a more accurate result software which maps raw calibrated values to a gradient would have to be written, and is discussed more in the calibration chapter.

Source code 3.17: The source code which handles the gradient application and conversion to an image. The scaling values are arbitrary, chosen for their ability to provide contrast

```rust
pub fn generate_image() {
    let mut generated_image: RgbaImage = ImageBuffer::new(2400, 2400);

    let gradient = generate_gradient();

    let result = stencil();

    for y in 0..2400 as usize {
        for x in 0..2400 as usize {
            let scaled = (result.read(x, y) as f64).sqrt() / 355.;
            if scaled > 0. && y % 500 == 0 {
                println!(”scaled {scaled}”);
            }
            let color = gradient.at(scaled).to_rgba8();
            let mut alpha: u32 = (scaled * 5000.) as u32;
            if alpha > 255 {
                alpha = 254;
            }
            generated_image.get_pixel_mut(x as u32, y as u32).0 =
                [color[0], color[1], color[2], alpha as u8];
        }
    }

    generated_image
        .save(format!(”./map_generation/tests/uk_test.png”))
        .unwrap();
}
```

Chapter 4

Calibration

Due to the time limit on the project calibration of the software was not achievable, instead a list of curcial points which should be calibrated and why is provided. Calibration will be the next natural step in the process of making this project a viable product which can be deployed by Time and Date AS.

The light pollution model is crucial software which requires calibration. The angle at which the observer is observing the night sky should be decided upon and reasoned about.

Calculation of total luminosity of the pixel which compensates for the fact that the satellite is sampling through the atmosphere, is another crucial point which should be calculated to a higher degree of accuracy.

Compensation for the VIIRS DNB sensors spectral response compared to the actual wavelengths light sources have should also be corrected for.

For now the output of the Garstang model is just the sky brightness value, it should however be calculated into a value which can directly map to the Bortle scale.

In addition the gradient used to create the images should not necessarily be a linear mapping from 0 to 1 as it is now, instead each color should be mapped to a value and the values can be arbitrarily placed to produce a gradient within any numeric range.

Chapter 5

Results and discussion

Each of the images below are 2400 by 2400 pixels, and each of them took about 3 seconds to produce using an AMD Ryzen 5950X at 3.4 Ghz using 32 threads regardless of whether it was filtered or not, which means generating the tiles is not a bottleneck when processing a single tile.

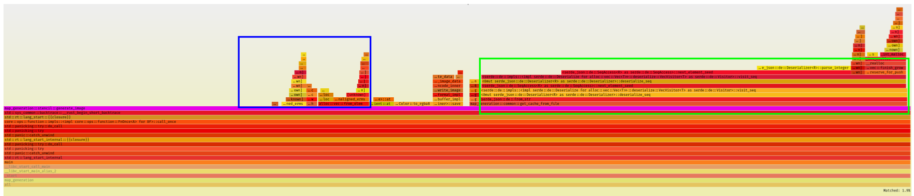

Figure 5.1: Image of process performance testing using rust flamegraph, this graph shows where the time is spent in the software. The blue square denotes the section of time which can be attributed to calculating the result. This shows that deserializing the large cache marked by the green square is actually slower than calculating the result.

There is unfortunately no simple way to compare with other implementations of light pollution maps, if one assumes that this scales linearly and there are 460 non overlapping land tiles [22] generating for the entire world would take somewhere around $460 * 3/60 = 23$ minutes.

The tiles generated and shown in figures 5.2, 5.3, 5.4 and 5.5, though not precise in terms of being able to tie a gradient colour to a meaningful

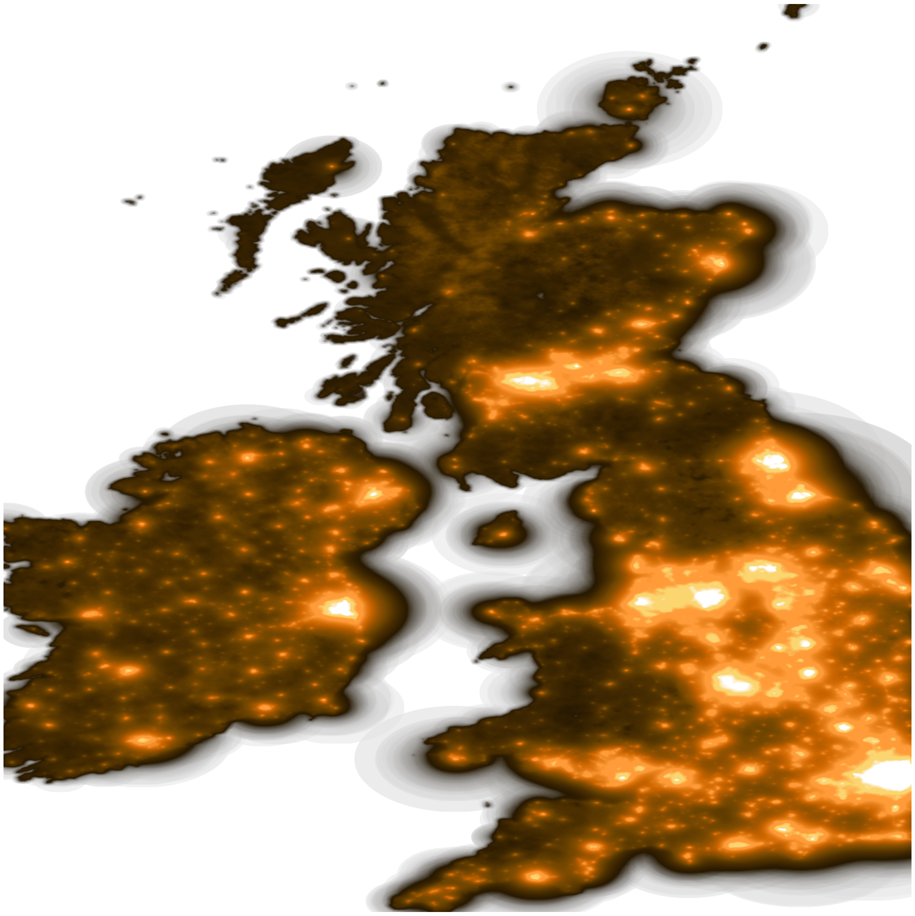

Figure 5.2: Image of United Kingdom light pollution levels using a gradient which emulates light emission including all values from the source data (not calibrated)

light pollution value, still demonstrate that it is very possible to do so in a performant way.

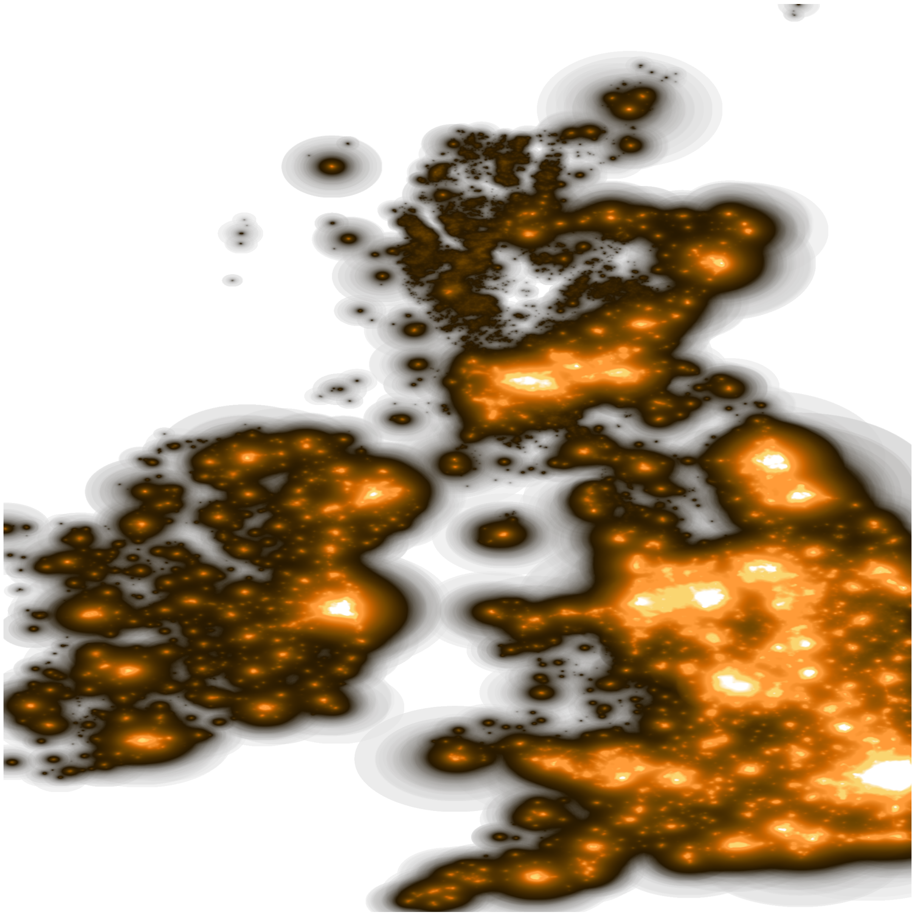

Figure 5.3: Image of United Kingdom light pollution levels using a gradient which emulates light emission, excluding values below 10 (not calibrated)

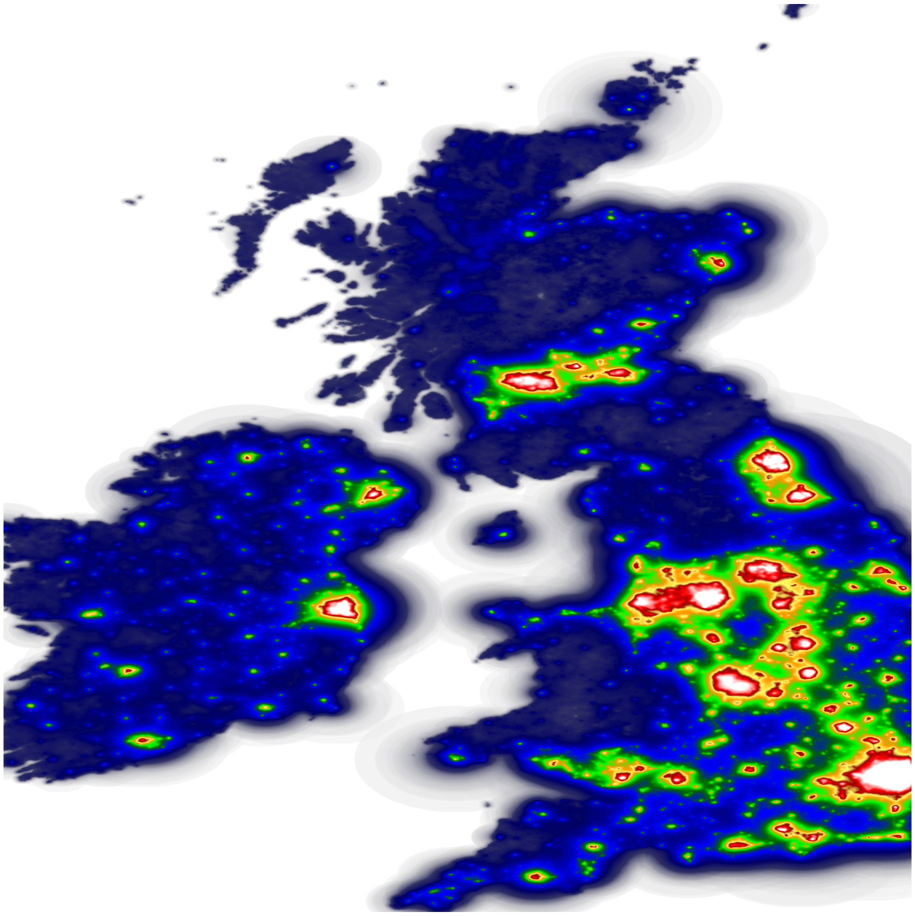

Figure 5.4: Image of United Kingdom light pollution levels using Bortle scale including all values from the source data (not calibrated)

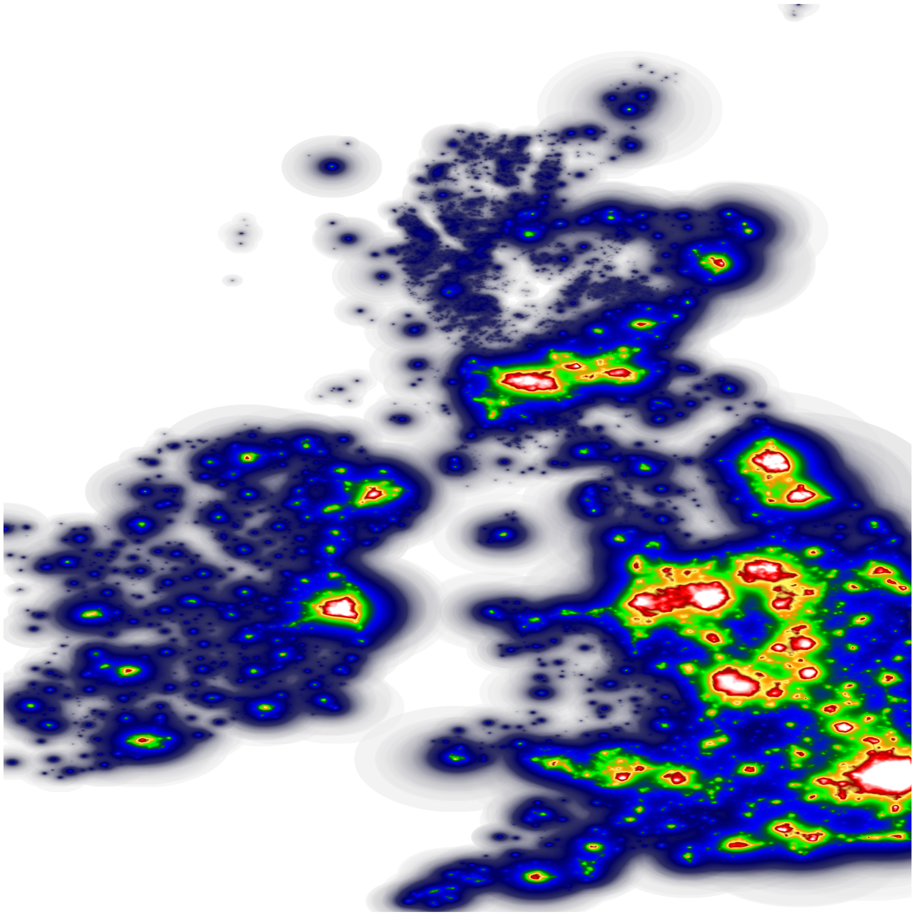

Figure 5.5: Image of United Kingdom light pollution levels using Bortle scale, excluding values below 10 (not calibrated)

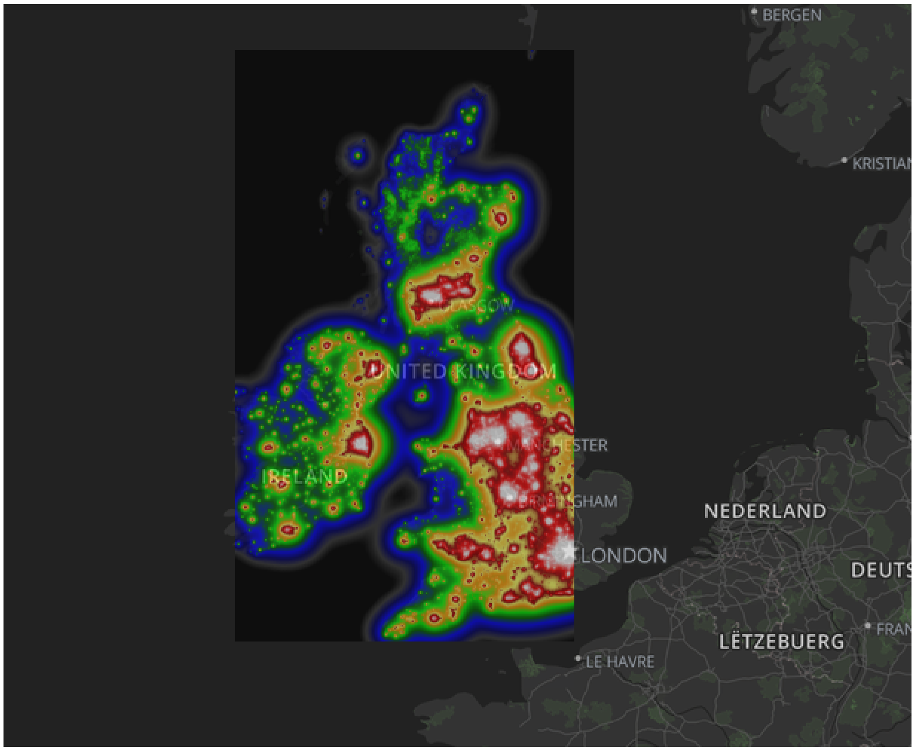

Figure 5.6: Demonstration of an interactive map application, the tile is applied as an overlay to leaflet which is a javascript map This demo is accessible in the repository.

Chapter 6

Conclusion

The fact that the software is open source might speed along the calibration process and encourage others to contribute and help maintain an open and accurate light pollution model which anyone can use.

The lack of time mentioned several times in this thesis stems from the enormous amount of research required, over 30 papers were thoroughly read, and information from many sources was processed. The satellite data proved to be more of a challenge than first expected, due to mislabelling of data and erroneous assumptions on the part of the author.

A special thanks goes to everyone involved at Time And Date, and especially Dr. Frank Tveter, a great colleague and brilliant astrophysicist who helped me understand the finer details of astrophysics and R. H. Garstang's model.

Bibliography

[1] NASA&apos;s Black Marble — blackmarble.gsfc.nasa.gov. `https://blackmarble.gsfc.nasa.gov/`. [Accessed 15-May-2023].

[2] R. H. Garstang. Model for artificial night-sky illumination. *Publications of the Astronomical Society of the Pacific*, 98(601):364, mar 1986.

[3] R. H. Garstang. Dust and light pollution. *Publications of the Astronomical Society of the Pacific*, 103(668):1109, oct 1991.

[4] Kurt W. Riegel. Light pollution. *Science*, 179(4080):1285–1291, 1973.

[5] Kohei Narisada and Duco A. Schreuder. *Light Pollution Handbook*. 2004.

[6] Mark W. Miller. Apparent Effects of Light Pollution on Singing Behavior of American Robins. *The Condor*, 108(1):130–139, 02 2006.

[7] Avalon C.S. Owens, Précillia Cochard, Joanna Durrant, Bridgette Farnworth, Elizabeth K. Perkin, and Brett Seymoure. Light pollution is a driver of insect declines. *Biological Conservation*, 241:108259, 2020.

[8] CALLUM J. MACGREGOR, MICHAEL J. O. POCOCK, RICHARD FOX, and DARREN M. EVANS. Pollination by nocturnal lepidoptera, and the effects of light pollution: a review. *Ecological Entomology*, 40(3):187–198, 2015.

[9] Christopher C. M. Kyba, Thomas Ruhtz, Jürgen Fischer, and Franz Hölker. Cloud coverage acts as an amplifier for ecological light pollution in urban ecosystems. *Plos One*, 6(3):1–9, 03 2011.

[10] Ron Chepesiuk. Missing the dark: Health effects of light pollution. *Environmental Health Perspectives*, 117(1):A20–a27, 2009.

[11] informationeso.org. How light pollution affects the dark night skies — eso.org. `https://www.eso.org/public/belgium-fr/images/dark-skies/?lang`. [Accessed 14-May-2023].

[12] P. J. Treanor. A simple propagation law for artificial night-sky illumination. *The Observatory*, 93:117–120, June 1973.

[13] R. H. Garstang. Night sky brightness at observatories and sites. *Publications of the Astronomical Society of the Pacific*, 101(637):306, mar 1989.

[14] P. Cinzano, F. Falchi, and C.D. Elvidge. The first World Atlas of the artificial night sky brightness. *Monthly Notices of the Royal Astronomical Society*, 328(3):689–707, 12 2001.

[15] Fabio Falchi, Pierantonio Cinzano, Dan Duriscoe, Christopher C. M. Kyba, Christopher D. Elvidge, Kimberly Baugh, Boris A. Portnov, Nataliya A. Rybnikova, and Riccardo Furgoni. The new world atlas of artificial night sky brightness. *Science Advances*, 2(6):e1600377, 2016.

[16] P. Cinzano and F. Falchi. The propagation of light pollution in the atmosphere. *Monthly Notices of the Royal Astronomical Society*, 427(4):3337–3357, 12 2012.

[17] Nasa. Npp mission brochure - nasa. `https://www.nasa.gov/pdf/596329main_NPP_Brochure_ForWeb.pdf`. [Accessed 14-May-2023].

[18] Christopher D Elvidge, Kimberly Baugh, Mikhail Zhizhin, Feng Chi Hsu, and Tilottama Ghosh. Viirs night-time lights. *International Journal of Remote Sensing*, 38(21):5860–5879, 2017.

[19] Npp. Suomi National Polar-orbiting Partnership Media Toolkit— nasa.gov. `https://www.nasa.gov/pdf/596877main_NPP_PressKit_Color.pdf`. [Accessed 14-May-2023].

[20] Catalog of Earth Satellite Orbits — earthobservatory.nasa.gov. `https://earthobservatory.nasa.gov/features/OrbitsCatalog/page2.php`. [Accessed 14-May-2023].

[21] Push-broom scanning method.png - Wikimedia Commons — commons.wikimedia.org. `https://commons.wikimedia.org/wiki/File:Push-broom_scanning_method.png`. [Accessed 14-May-2023].

[22] Black marble user guide version 1.2 — blackmarble.gsfc.nasa.gov. `https://viirsland.gsfc.nasa.gov/PDF/BlackMarbleUserGuide_v1.2_20210421.pdf`, 2021. [Accessed 14-May-2023].

[23] Charles F. F. Karney. Algorithms for geodesics. *Journal of Geodesy*, 87(1):43–55, jun 2012.