Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Bachelor's diploma thesis

in Computer Science

Precipitation Data Processing and Visualisation Web System

Marin Karamihalev

286781

Siyana Ivanova

286780

thesis supervisor
dr inż. Rafał Jóźwiak

WARSAW 2020

…………………………	…………………………	………………………...
Supervisor’s signature	Author’s signature	Author’s signature
Podpis promotora	Podpis autora	Podpis autora

Abstract

This document presents the results of our thesis project: an automatic web system for downloading, processing, and visualizing precipitation data obtained via NASA's Global Precipitation Measurement Mission. GPM consists of a network of satellites observing global rain and snowfall and has the stated goal of enhancing scientific understanding of the planet's water and energy cycles as well as advancing the accuracy of weather forecasting.[1] The Precipitation Data Processing and Visualisation Web System will serve as a tool to aid the agricultural community in more effectively gauging the timing and amount of precipitation that has occurred, thus improving farmers' ability to determine crop maintenance needs and predict crop yields. The system's functionalities take into account the specific requirements of the agricultural community and enable more efficient use of global precipitation data on farmland through accessing and processing specific geospatial data (using the GeoJSON standard) over a given time period. The system provides extensive visualization options and simplifies the integration of processed data with forecasting models. The system consists of three main components: a downloader for the necessary GPM files, a web server with a REST API which comes with a set of data operations and transformation tools, and a web app which visualises the data and allows the user to interact with it.

Keywords: Precision agriculture, geotiff, data processing, visualisation, web application, web service

Abstrakt

Niniejszy dokument przedstawia wyniki projektu dyplomowego - automatycznego systemu internetowego do pobierania, przetwarzania i wizualizacji danych o opadach uzyskanych w ramach globalnej misji pomiarowej NASA (GPM NASA). GPM składa się z sieci satelitów obserwujących globalne opady deszczu i śniegu i ma na celu zwiększenie naukowego zrozumienia cykli wodnych i energetycznych planety, a także zwiększenie dokładności prognozowania pogody. Funkcjonalności system uwzględniają specyficzne wymagania społeczności rolniczej oraz umożliwią bardziej efektywne wykorzystanie danych o globalnych opadach na lądzie, w szczególności poprzez dostęp i przetwarzanie względem określonych danych geoprzestrzennych (wykorzystując standard GeoJSON) oraz przedziałów czasowych. System zapewnia szerokie możliwości wizualizacyjne oraz uprasza integrację przetwarzanych danych z modelami prognostycznymi. System składa się z trzech głównych komponentów: usługi pobierania niezbędnych plików z GPM, web-serwera z REST API zapewniającego zestaw narzędzi do operacji i transformacji danych oraz aplikacji internetowej, która wizualizuje dane i pozwala użytkownikowi na interakcję z nimi.

Słowa kluczowe: Rolnictwo precyzyjne, geotiff, przetwarzanie danych, wizualizacje, aplikacja internetowa, serwer internetowy

Warsaw, 27 January 2020

DECLARATION

I hereby certify my authorship of the portions of the Engineering Thesis under the title of “Precipitation Data Processing and Visualisation Web System” specified in the Project Schedule and Division of Labour section of this work, completed under the guidance of our supervisor dr inż. Rafał Jóźwiak.

OŚWIADCZENIE

Oświadczam, że moją część pracy inżynierskiej (zgodnie z podziałem zadań opisanym w pkt. Project Schedule and Division of Labour pracy dyplomowej) pod tytułem “Precipitation Data Processing and Visualisation Web System”, której promotorem jest dr inż. Rafał Jóźwiak wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

…………………………	………………………..
Author’s signature	Author’s signature
Podpis autora	Podpis autora

Table of contents

Introduction

Precision agriculture is an innovative field which uses satellite imagery to infer the special conditions under which certain crops can be grown most efficiently depending on their location and its specific meteorological conditions. This has been made possible by advances in both engineering and computer science, allowing for the use of GPS and other technologies, coupled with software developed for agricultural purposes, in sophisticated crop management. Our project, the Precipitation Data Processing and Visualisation Web System, is an example of such software, created in order to help monitor meteorological phenomena and making decisions based on the satellite data obtained from NASA's Global Precipitation Measurement mission.

The goal of the Precipitation Data Processing and Visualisation Web System is to provide accurate and practical information about weather events which affect crop development. The meteorological data has multiple uses in agricultural farming, notably for making various informed choices related to the amount of product (fertiliser or other agronomic treatments) which needs to be used at a particular section of land; historical as well as near real-time data can be utilised to this effect.

Precision agriculture methods can be used to reduce environmental impact as well as allow farmers to save on costs. With our system we hope to improve the accuracy of such services, which are provided by firms such as our consultant SatAgro,[2] and so facilitate even more efficient use of agricultural lands. The Precipitation Data Processing and Visualisation Web System collects the requested data from NASA's GPM mission, processes it, and makes it available for SatAgro to include in their own system's tools for farmers (e.g. prescription maps). The visualisation app component, on the other hand, allows direct interaction with the data so that the user can observe weather patterns and relevant precipitation values contained in NASA's files.

With the Precipitation Data Processing and Visualisation Web System, we hope to introduce a more user-friendly tool for use specifically in the context of precision agriculture. Existing options, like SatAgro's current app (shown below in *Fig 1*), use data from different, less accurate sources. We hope to improve the precision of such services by using the satellite data provided by NASA. While it was previously possible to view the data from the GPM mission using NASA's own Global Precipitation Viewer (*Fig 2*), the GPV lacks tools which allow the user to focus on a particular region defined by a geometrical shape. Such functionality is particularly desirable when the satellite data is to be used for purposes related to precision agriculture, since individual farming fields can easily be defined in this manner. Our system allows the user to define regions in a geojson format and request not only overall precipitation values, but also snow, ice, or liquid precipitation percent values specifically.

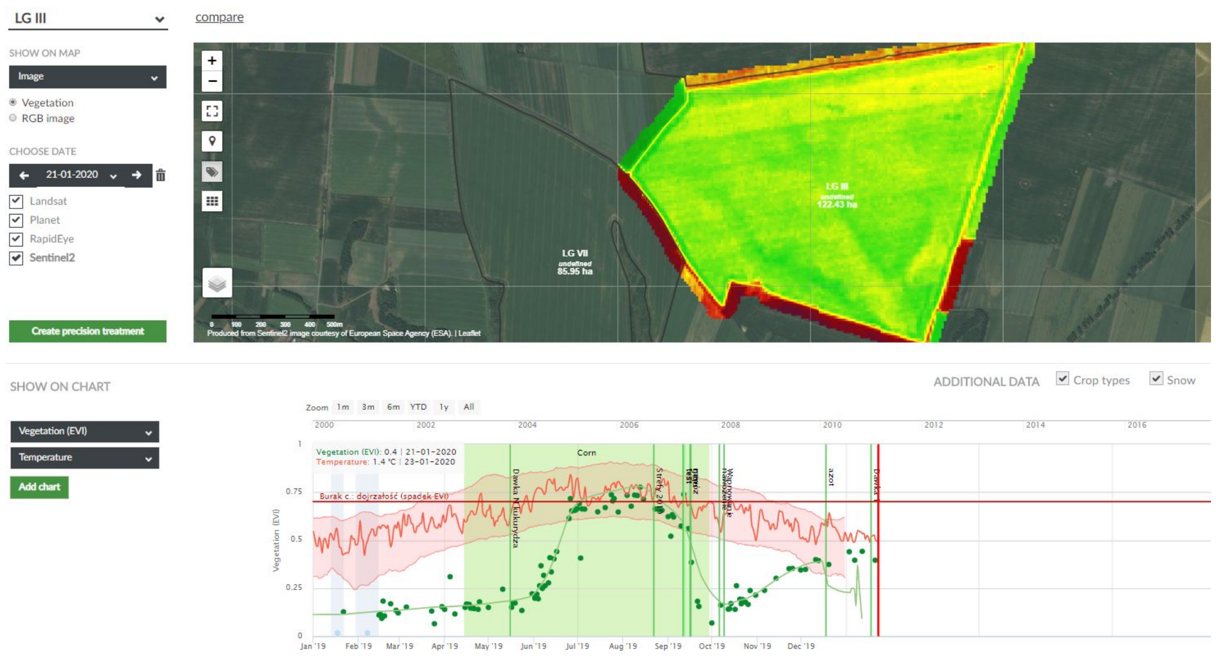


Fig 1 SatAgro's application

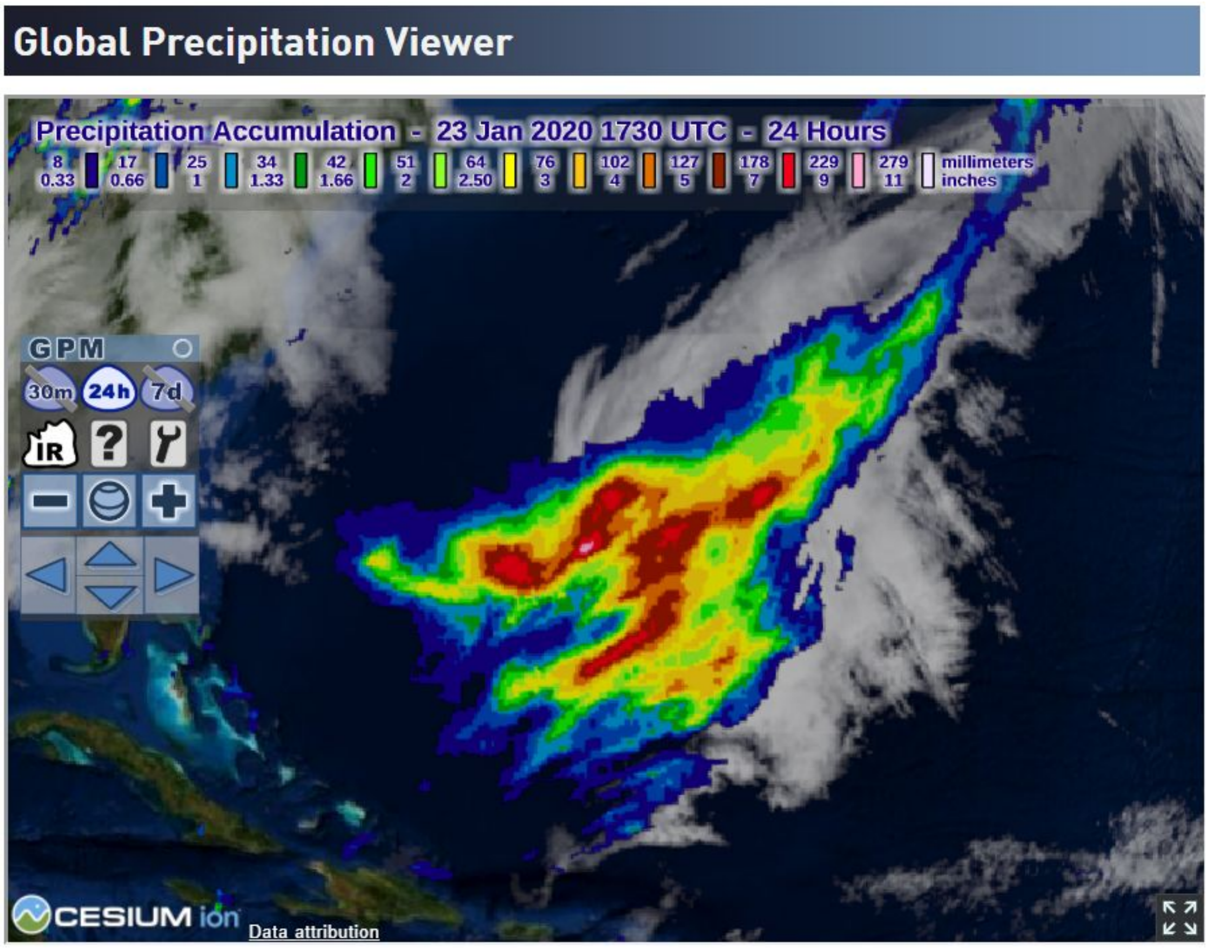


Fig 2 NASA's Global Precipitation Viewer

Chapter 1: Requirements Specification

1.1 Functional Requirements

The requirements regarding functionality which we determined must be present in the project are described in the following table, which lists the desired system responses for the various use cases that may arise.

Table 1.1 Functional requirements

Number	Use case	System response
1 - Web service		
1.1	Must download all sets of files needed	The downloader component connects to the NASA FTP server, locates all needed files which have not yet been obtained, and downloads them
1.2	Must provide the total precipitation amount in a region for a period	Request for a given region and a period is parsed, then the transformation tools locate the corresponding files and perform geospatial operations to return the set of sums (one per day in the range) of precipitation for the region
1.3	Must provide a new image containing the precipitation over a region for each given date in a set	Request for a given region and a period is parsed, then the transformation tools locate the corresponding files and perform geospatial operations to return the new tif file containing one band with the precipitation over the region per day in the range
1.4	Must provide access to the early and the late products whenever available	Since the late product is not always available at the NASA server, the application performs a check for which product is ready to download at the time; the late product is always prioritised as it is more accurate: if it is available, it will be the one provided by the application and if not, the early product will be downloaded instead
1.5	User must be able to choose the type of file to be downloaded and processed	On the NASA server there are four types of file: precipitation (all types), liquid, ice, and liquid percent. The application allows the user to choose which type of file to work with.
2 - Visualisation app		
2.1	User must be able to visually explore the precipitation data	The visualisation app allows the user to select dates, a region in a chosen shape, and a type of precipitation to view over a map of the world

2.2	Must be able to load a local tif file and visually explore its contents	The app contains the option to load a tif file and explore its contents over the world map in a similar manner as with the precipitation data from the NASA server
2.3	User must be able to save a new image to their local computer with the precipitation for a region in a given period	After selecting the desired options (dates, region, type of precipitation), the user can save a tiff file locally with this information, which can be viewed again later
2.4	User must be able to select a range of dates and allocate a specific region on the map	Request is composed with all information provided by the user in a format accepted by the server API
2.5	User must be able to explore different bands of a loaded image	Interactive controls which allow inspecting the bands one by one are provided: one band corresponds to one day and they are viewable separately

1.2 Non-Functional Requirements

The table below presents the goals that were set for the system's quality. This includes requirements for its compatibility with SatAgro's existing codebase, its adaptability to changes in requests or configurations, its ease of use (facilitated by a high standard for clear documentation), its security, and its ability to provide a smooth and stable user experience even when encountering problems on the side of the NASA server.

Table 1.2 Non-functional requirements

Number	Requirement	Solution
1 - Compatibility and integration		
1.1	Able to be integrated into SatAgro's existing codebase	Using Python for the entirety of the web service so that it can easily be integrated into SatAgro's existing codebase; using the Django Rest Framework to comply with client request
2 - Adaptability and modifications		
2.1	Flexible and reasonably modifiable should changes in the NASA database occur	Code for downloading files is modular and can be adapted to reflect changes in the database e.g. filename alterations, which files should be downloaded, which operations to execute post-download

2.2	Adaptable to changes in what information is needed from the NASA server	Modular code allows easy changes to choice of files and performed operations
2.3	Able to provide a configurable and convenient way to download the data	External configuration file allows changes to the basic steps of the download procedure; easy to add scripts to the downloader component which perform new specific tasks
3 - Usability and documentation		
3.1	Fast and efficient at processing the data	Region calculation executed only once per request; load only those segments of each file necessary for the given region
3.2	Easy to run and use	Installation, execution and configuration instructions for all components included in the readme file
3.3	Has a user manual readable to users unacquainted with the system	User manual specifies simple, easy instructions for setting up and testing the system, creating scheduled downloader tasks, starting the web service and using the visualisation app
4 - Security		
4.1	Up to the company's security standards	Using the Django Rest Framework (per the client's request) for its built-in security features
5 - Stability		
5.1	Able to deal with problems occurring on the side of NASA's server	If the server is down, there will be more attempts to connect to it until a result is obtained by the downloader
5.2	Able to deal with data being unavailable	If a file (e.g. the latest near real-time file) has not come out yet on the NASA server, the scheduling of the downloader allows checking for it again in an hour

1.3 Risk Analysis

For a thorough analysis of the potential risks associated with the process of implementing our solution as well as a strategy for dealing with those risks, we chose the SWOT method. The two tables below discuss the positives as well as the negatives which we faced when creating this system and detail the ways we chose to manage them. These management strategies proved effective in creating a product which is as well-considered and robust as possible.

Table 1.3 Strengths and opportunities

Strength	Opportunity
Having access to an advisor as well as an industry professional	Can get code review from experts and testing in an industry setting in order to improve the code
Project will be used in a real-life situation	Testing is purposeful and there are clear goals for what the system should be able to do
The code is modular and extensible	New functionality is relatively easy to add and changes to already existing code do not require full revamps
Ease of deployment	Reduces effort and the chance of mistakes on the client's side when using the system
Early start	Reduces the chance of missing a deadline and allows more time for potential changes or dealing with unexpected issues
Functional-first approach to tools development	Ease of extension, composability, and testing

Table 1.4 Weaknesses, threats, management strategies

Weakness	Threat	Management strategy
Inexperience in the field of geospatial science	Making mistakes related to handling the geospatial information which must be processed by the system	Ask experts from SatAgro who work in this field for explanations and advice - apply what is learned
Inexperience in making a complete functional system to be deployed in a real-life setting	Not meeting the company's standards; misunderstanding the clients' wishes; being overwhelmed by the many functions the system must perform	Break the task down into smaller components (e.g. downloader, tools); comprehensive, frequent testing; asking for the company's feedback whenever large changes are introduced
Large project	Missing a deadline	Starting early and asking the supervisor and client for feedback frequently
System can currently only work with tiff files	Inflexibility of the system	Making sure the structure of the downloader and transformation tools allows for the easy addition of more tasks and/or endpoints
API lacks flexibility	Cannot work with different parameters, e.g. a more specific period of time within one day	Making the system extensible and flexible so that more endpoints can be added with ease

Chapter 2: Design

2.1 Technology Selection

Languages

For the web service, we elected to work in Python as per the advice we received from SatAgro experts, as this would ease integration on their end and make any potential extensions to the system easier to implement for their team. Furthermore, the language provides a wide variety of libraries which we were able to make use of, and it allows for readable, easy-to-maintain code.

For the visualisation app, our chosen language is JavaScript due to its versatility and its suitability (over, it would be fair to say, almost any other currently popular language) for web-based applications. In addition, it provides apt opportunities for rapid prototyping, as well as various libraries to choose from.

Frameworks and Libraries

The following table describes the various libraries and frameworks which we have chosen to use in our solution and their utility to the system.

Table 2.1 Frameworks and libraries

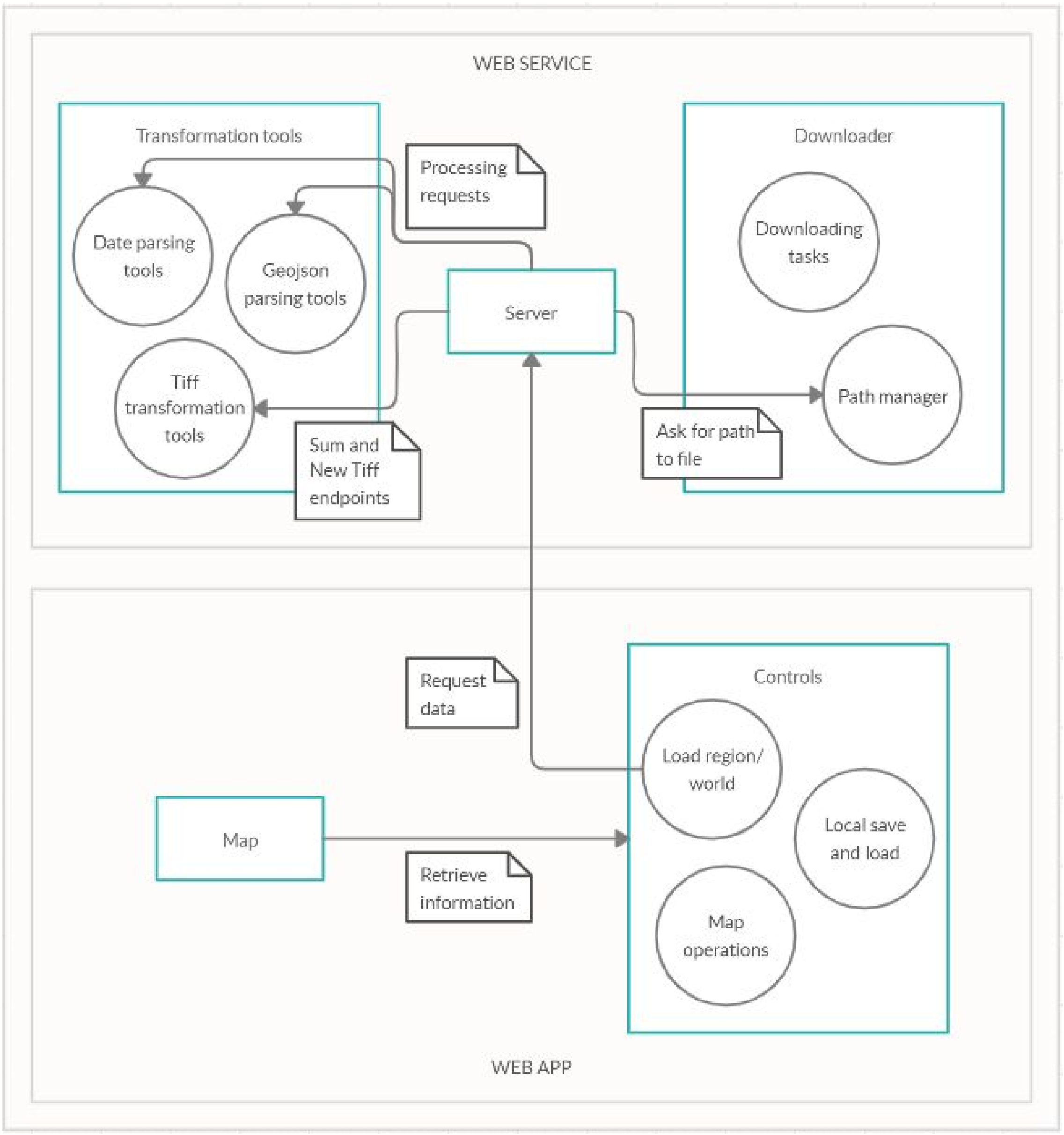
Name	Description	Used in
Visualisation App		
Svelte	JavaScript framework; a reactive way of building user interfaces which ensures that the application remains fast-running during the entire span of a user's interaction with it[3] - this is an important feature for our system as any large slow-downs in displaying the map and the precipitation values would greatly decrease the usability of the app	Building app structure and UI
Leaflet	One of the most popular mobile-friendly JS libraries for displaying maps,[4] it provides simple and functional utilities and is efficient; we chose this library as it does not slow the application down and has built-in functions which make it easy to create a lot of the desired functionality for the web app like displaying the world map and being able to select a region	Map-related functionality

Georaster	A library that allows parsing an array buffer into a georaster object which is then passed on to Georaster-layer-for-Leaflet; one of the few tools available to perform this function, also chosen for being lightweight and minimalist	Parsing server response
Georaster-layer-for-Leaflet	Library which extends the layer type from Leaflet to be applicable to georaster files - it is the bridge between Leaflet and Georaster	Map compatibility
Colormap	Provides colour coding for the precipitation values: the colour scheme overlays the map, showing different values as different shades and thus visually representing the amount of precipitation over a given location; chosen for being easy to use and providing a wide variety of palettes to choose from	Displaying precipitation values visually
Web Service		
Django	A web framework with many built-in security and scalability features as well as handling for various common web programming tasks;[5] the client requested that we use this framework when developing the server component of the web service for security and compatibility reasons	Server
Pytest	Testing framework chosen for its extensibility which allows specific types of tests like benchmarks; used for all unit, integration, and end-to-end testing in the web service	Testing
Shapely	A popular and widely well-regarded library containing different geometric shapes and operations on them, e.g. finding the bounding box of a polygon; we use this library for transforming an incoming geojson into a shape and calculating the coverage (applying a clipping mask to the map region)	Parsing geojson
Rasterio	Performs operations like loading, creating, or extracting metadata from a geotiff - all very crucial to the system; chosen for this purpose as it is very well documented and robust	Raster operations for geotiff
Numpy	A principal library to have whenever performing scientific calculations in Python as it includes a wide range of tools and functions[6] - for our purposes, we made use of its matrix operations in order to calculate the coverage and apply the resulting clipping mask to the values in the tiff	Matrix operations for geotiff

Operating systems

The system is OS-independent due to the choice of languages and libraries. However, most of the testing has been performed on Linux (due to the client's setup) and on Windows 10 (due to our setup). No testing has been performed on Mac OS.

2.2 System Architecture and Module Design



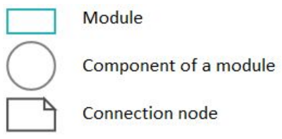


Fig 2.1 Overview of system architecture

Structure and Connections

The system is divided into two parts: the web service, which collects, transforms, and redistributes NASA GPM data; and the web app or visualisation app, which allows interaction and visual representation of the data. The web service consists of three main modules: the downloader, the server, and the transformation tools, while the web app contains a map and a set of controls.

As shown in *Fig 2.1*, the server is responsible for all connections between other modules in the system, including the link between its two “sides” - this separation of concerns contributes to the malleability of the code, allowing for less time- and work-intensive changes and preventing any issues arising in components which are not being altered. Within the web service, the server connects to the downloader’s path manager component in order to obtain the path to the downloaded files, and to the transformation tools to process requests (geojson and data parsing tools) and provide endpoints (tiff tools). On the other hand, the visualisation app requests from the server the data required for its load region/ load world functionality. Within the app itself, the map module connects to the controls to retrieve the information that is to be visualised, e.g. tiff image bands.

Modules

As *Fig 2.1* illustrates, each module in the system consists of one or more components. The server (a module in the web service) and the map (in the visualisation app) are without any subdivisions as each performs a singular task - facilitating data access and displaying the data, respectively.

All other modules can be broken down into the components which comprise them. For the downloader, these are the path manager and the download tasks; for the transformation tools, the date and geojson parsers and the tiff tools. As for the controls (located in the web app), they are subdivided into three types: loading the world or a region, loading or saving a local geotiff/geojson, and map operations for specific date and region requests.

The specific responsibilities of each module and component are listed below in table form.

Responsibilities: Web Service

Table 2.2 Web service responsibilities

Module/ Component	Responsibility
Server	Providing consistent access to the data
Downloader	Keeping track of and downloading all necessary data

Downloading tasks	Download procedures
Path manager	Keeping track of correct pathing and its format
Transformation tools	Transforming the data
Date parsing tools	Correct date format
Geojson parsing tools	Transforming geojson into geometric shapes
Tiff transformation tools	Loading tiff files and applying required mathematical operations

Responsibilities: Web App

Table 2.3 Web application responsibilities

Component	Responsibility
Controls	User interaction
Load region/world	Acquiring the information requested by the user
Local load and save	Allowing use of local files
Map operations	Allowing specific user requests (date, region)
Map	Displaying data

Main Components

As previously discussed, the web service consists of three modules - the server, the downloader, and the transformation tools - while the visualisation app has two, the controls and the map. These modules, in turn, are composed of several components, as seen in *Fig 2.1*. In the present section we go into detail regarding how these components operate. We have elected to use an imperative paradigm in our web service code as opposed to an object-oriented one since we do not need to keep track of many complex variables; the complexity of the problem our system is aimed to solve lies instead in processing the data obtained from the files. Due to this, using object-oriented design would require more code and introduce needlessly complicated structures while not being more efficient or effective than imperative programming. Therefore, we have chosen to represent the main components of this part of the system with activity diagrams in order for the diagrammatic depiction to correctly reflect and be consistent with our programming paradigm.

Web Service

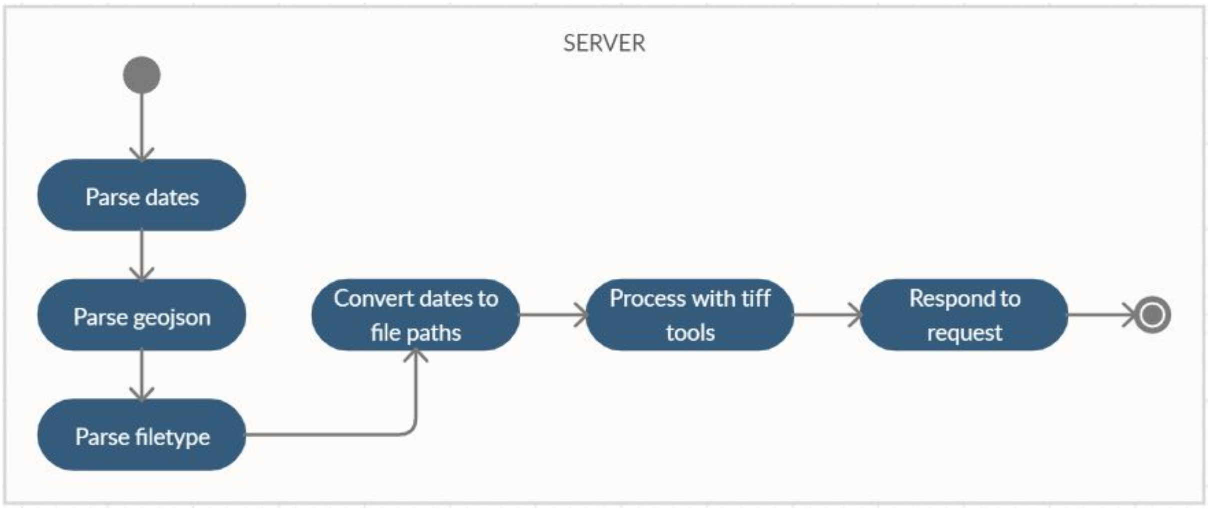


Fig 2.2 Server activity diagram

As shown in *Fig 2.2*, whenever a new request is made to the server, after parsing all needed elements, it converts the parsed dates into file paths and connects to the path manager in order to locate the needed files. Then, it passes the gathered information to the transformation tools where it can be processed, before finally responding to the request.

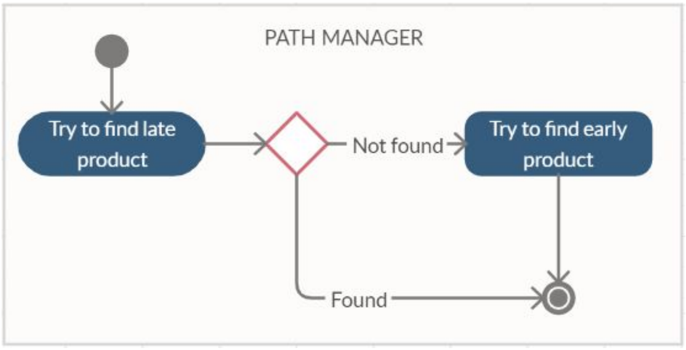


Fig 2.3 Path manager activity diagram

The path manager, described in *Fig 2.3*, always aims to find the late product, as it contains more accurate precipitation information for a given date. However, it may not be available at the time of some requests, as it does not appear on the NASA server until the following day - in this case, the path manager finds the early product, which provides less precise information but is available on the same day it refers to.

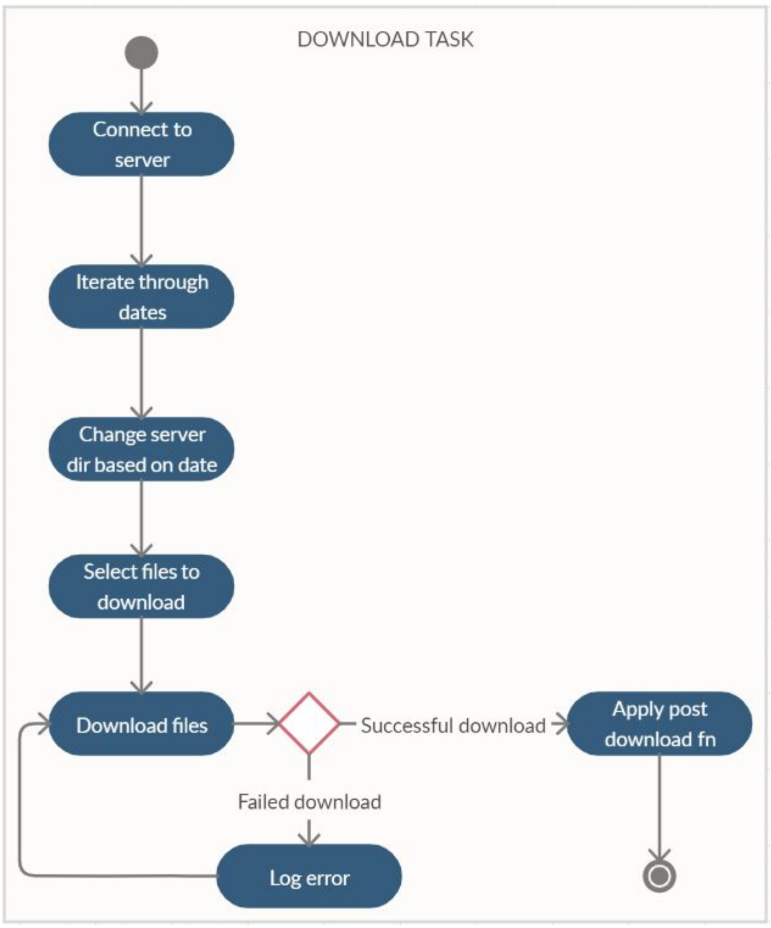


Fig 2.4 Download task activity diagram

The main component of the downloader (*Fig 2.4*) is responsible for the different downloading tasks (early or late product). After connecting to the server, it iterates through the requested dates and selects the files that must be downloaded. If some of the files are already present in the download directory, they will be skipped. After all the files are obtained, post-download operations like unzipping and renaming according to the convention in the configuration file are applied to them.

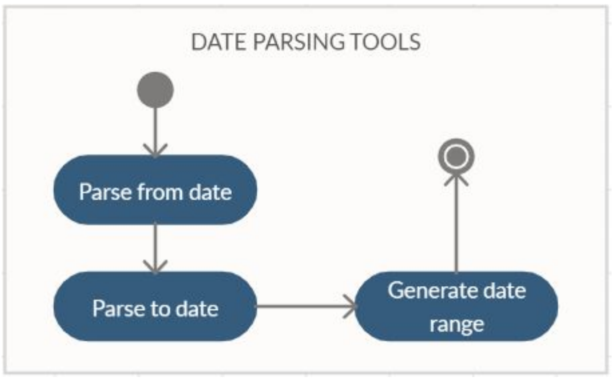


Fig 2.5 Date parsing tools activity diagram

When the date parsing tools (*Fig 2.5*) receive a new pair of beginning and end dates, both are parsed, and then a date range in the correct format is generated. The range goes on to be passed on by the server and used by the path manager to locate the needed files.

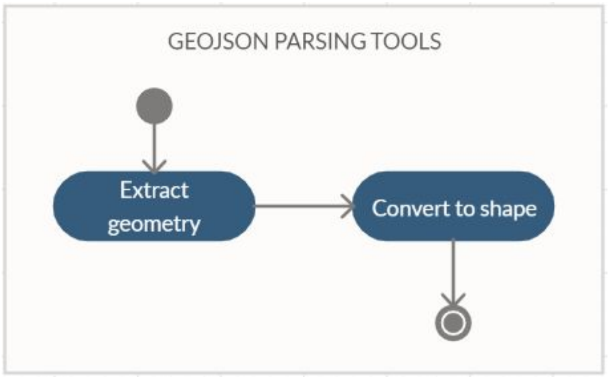


Fig 2.6 Geojson parsing tools activity diagram

The geojson parsing tools' sole purpose is to extract the geometric data from a geojson and convert it to a shape that can be used by the tiff tools, as shown above in *Fig 2.6*.

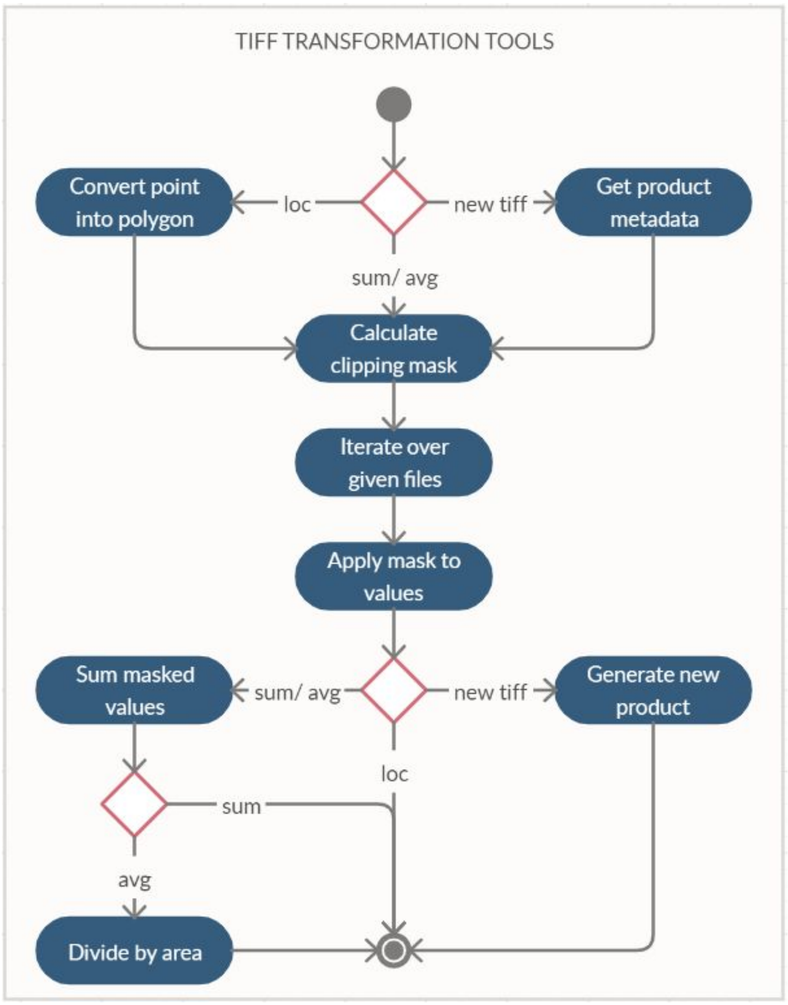


Fig 2.7 Tiff transformation tools activity diagram

The tiff transformation tools can perform several functions depending on the request, as seen in *Fig 2.7*. The sum endpoint produces the total amount of rain and snowfall over a region; the avg endpoint returns the average precipitation over it; the loc endpoint provides the estimated amount of precipitation over a specific 0.1 by 0.1-degree square; and the new tiff endpoint generates a geotiff file containing information about the precipitation over the region.

Visualisation app

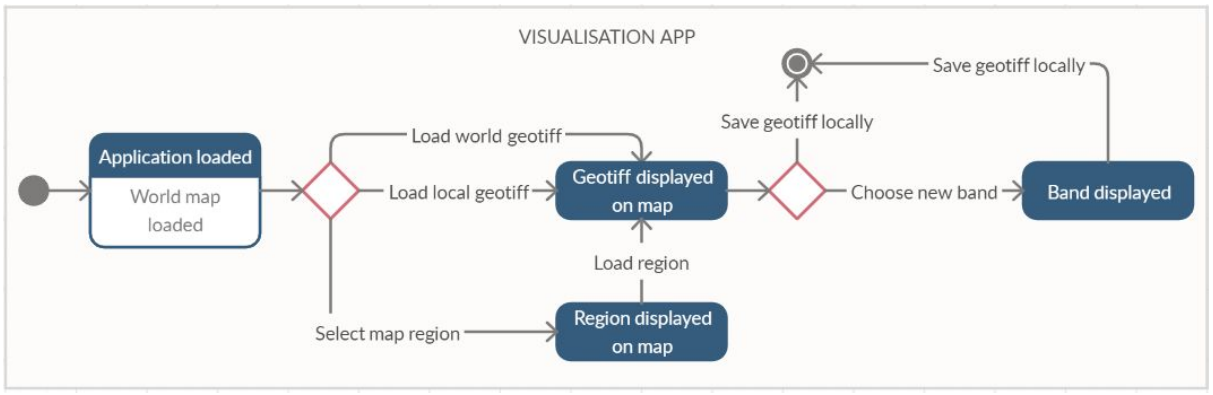


Fig 2.8 Visualisation app overview state diagram

In the visualisation app, whose workflow is shown in *Fig 2.8*, the map and controls components work synchronously to create the desired functionality, hence their representation in a single cohesive state diagram which overviews the whole app. When the user starts the web app, the world map is loaded and one can choose to proceed in one of three ways using the controls: load a geotiff file for the whole world, load a geotiff file locally, or select a region on the map by either using the geometric tools, loading a json file containing a shape, or through the geojson viewer. The region selection will be displayed before the precipitation values are displayed over it. After viewing the relevant geotiff file, the user may save it locally or use the map operation controls to choose a new band to display. As the visualisation app also serves as the system's only GUI, its components and uses are further discussed in the User interface section.

2.3 Communication

Web Service

The web service communicates with the NASA server using FTP in order to get the files for a request. The communication is facilitated by the Python library ftplib7, which deals with connecting to, navigating, and downloading from an FTP server. Per the client's request, we also use the Django framework for all server-related purposes due to its built-in security features.

Visualisation App

The communication protocol the web app uses is HTTP, which is handled by JavaScript. Additionally, we use the JavaScript Fetch8 API in order to facilitate communication between the visualisation app and the web service. Fetch is an API which deals with network requests and responses, used for fetching resources needed by the app such as the tiff files shown in the GUI.

2.4 External Interfaces

In the following table we list the standards and file formats that our system uses to acquire external information it needs, and describe the utility each of them provides.

Table 2.4 File formats and standards

File format/ standard	Description	Used in
Geotiff	A file format based on tiff, specifically geared towards georeferencing images[9]	Entire web service; chosen out of the available formats on the NASA server because of client's wishes
Py	Python's script file format	All configuration files: parameter specific and logging config files for the downloader and the server
EPSG4326	A cartography standard used by NASA and GPS satellite navigation[10]	Geotiff files all use this format; used in the map component of the visualisation app

2.5 User Interface

The visualisation app also serves as the system's UI. The web service does not require building a graphical user interface since all its tasks can be easily accomplished using command lines (see the User manual). In *Fig 2.9*, a simple use case diagram shows the ways in which the user interacts with the UI in order to examine various geotiff/ geojson files.

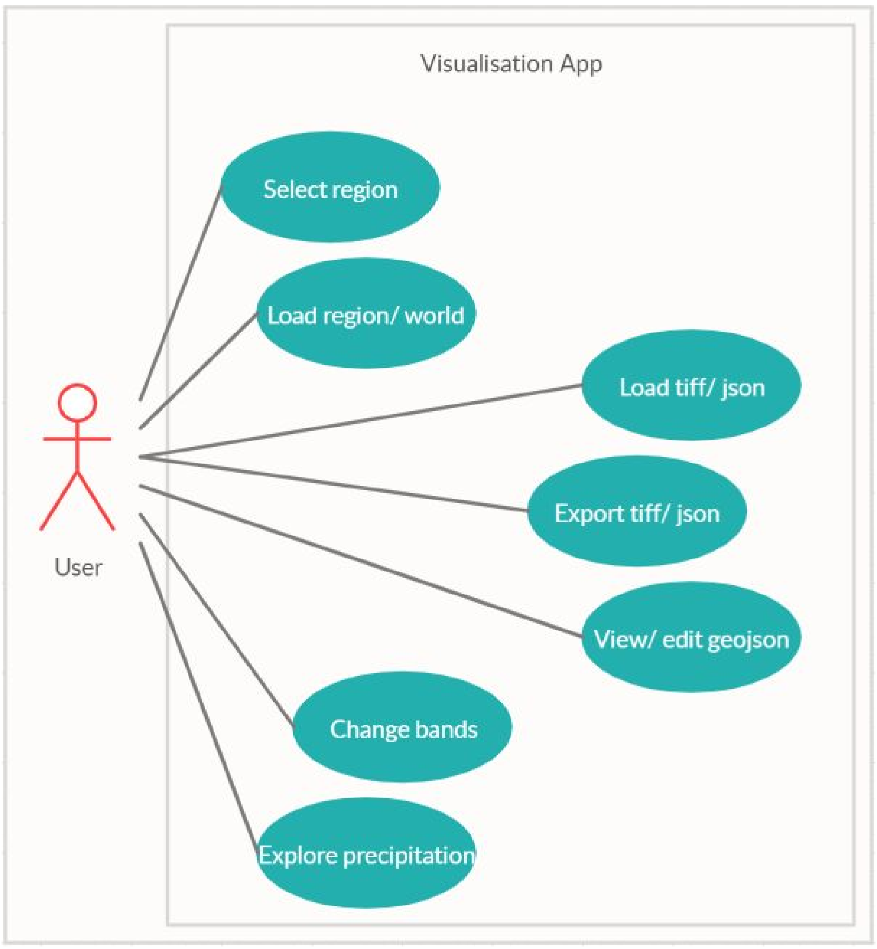


Fig 2.9 Use case diagram for the web app

As previously mentioned, the data can be explored as an overlay on a world map using the various options, e.g. region selection tools, date range widgets, band switching, etc. A full discussion of the app's functionalities can be found in Chapter 4: Using the Visualisation App.

Chapter 3: Testing

The most crucial parts of the system are functions which must take arguments and return very specific results, for example the transformation functions central to the web service. Therefore, our testing approach included a variety of testing files containing example data which were used to confirm that these functions are working correctly and yielding the expected results.

For singular functions performing a separate task within a component, like the “rounding” functions found in the tiff tools (which round up or down to the closest first decimal, regardless of the number’s sign), we created unit tests; on the other hand, integration tests were needed for those parts of the program which use several modules, e.g. the function which checks if data is currently available. Additionally, we included benchmark tests to ensure that the calculations for the sum of the overall precipitation run at an appropriate speed for different periods of time. We tested for the sum over 1 day and over 30 days.

The following table details all tests.

Table 3.1 Testing

Test name	Location	Test description
test_find	test_path_manager.py	Generate correct name based on a pattern, filetype, and date
test_exists	test_path_manager.py	Assert that a file exists given a pattern, filetype, and date
test_locate_late	test_path_manager.py	Correctly locate late run data for a date
test_locate_early	test_path_manager.py	Correctly locate early run data for a date
test_date2path	test_server_tools.py	Given a date in milliseconds, return a path as string
test_process_data	test_server_tools.py	Given mock correct and incorrect requests, does not generate or generates an error
test_find_available	test_server_tools.py	Given mock correct and incorrect requests, does not generate or generates an error
test_milli2date	test_tools_date_parser.py	Convert milliseconds to date
test_parse	test_tools_date_parser.py	Parse dates into milliseconds, given date format

test_eval_date	test_tools_date_parser.py	Parse date into milliseconds, given date format
test_date_range	test_tools_date_parser.py	Generate correct number of days given date range
test_eval_milli	test_tools_date_parser.py	Convert milliseconds into correct date, given date format
test_parse	test_tools_geojson_parser.py	Convert geojson into correct representation
test_point2poly	test_tools_geojson_parser.py	Generate correct shape given point
test_read_window	test_tools_tif.py	Read specific window given geometry without error
test_apply_mask	test_tools_tif.py	Shape of applied mask is the same as the percentage cover for given geometry
test_sum	test_tools_tif.py	Produce correct values for test data
test_avg	test_tools_tif.py	Produce correct values for test data
test_avg_ice	test_tools_tif.py	Produce correct values for test data
test_avg_liquid	test_tools_tif.py	Produce correct values for test data
test_avg_liquidPercent	test_tools_tif.py	Produce correct values for test data
test_sum_mini_region	test_tools_tif.py	Produce correct values for test data
test_sum_ice	test_tools_tif.py	Produce correct values for test data
test_sum_liquid	test_tools_tif.py	Produce correct values for test data
test_sum_liquidPercent	test_tools_tif.py	Produce correct values for test data
test_new_tiff	test_tools_tif.py	Test that sum of new file values equals result of sum endpoint
test_new_tiff_mini_region	test_tools_tif.py	Generate new file without error
test_new_tiff_liquidPercent	test_tools_tif.py	Generate new file without error
test_new_tiff_liquid	test_tools_tif.py	Generate new file without error
test_new_tiff_ice	test_tools_tif.py	Generate new file without error
test_get_meta	test_tools_tif.py	Extract metadata without error
test_mini_region_cover	test_tools_tif.py	Correctly calculate percentage cover for a small region

test_round_up	test_tools_tif.py	Correctly “round up”
test_round_down	test_tools_tif.py	Correctly “round down”
test_benchmark_sum_1day	test_tools_tif.py	Test amount of time for creating a sum for 1 day
test_benchmark_sum_30day	test_tools_tif.py	Test amount of time for creating a sum for 30 days

The system was also extensively tested by our consultant SatAgro throughout the process of iterative development. The firm has performed a myriad of acceptance tests and intends to integrate the Web Service into their own system which provides precipitation information to their farmer clients.

Chapter 4: Using the Visualisation App

Although the web app is hosted online at https://data-exploration-app.netlify.com/, it can also be run locally, provided that the server is running, as detailed in the installation instructions. To use the app, some downloaded files are necessary: the only way to acquire them is through using the server, as the visualisation/ exploration component is separate for security reasons and cannot download files. *Fig 4.1* details how one can use the app.

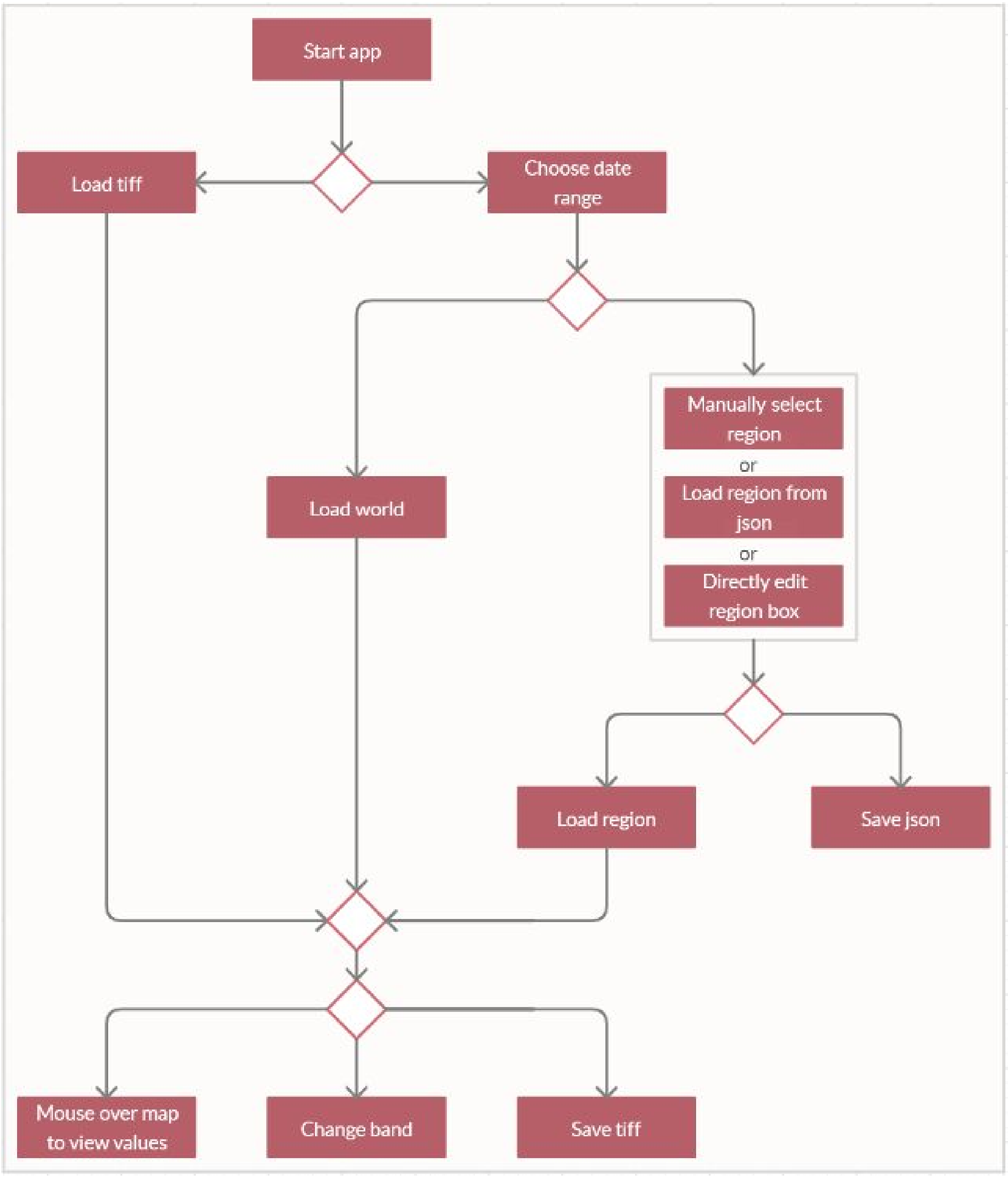
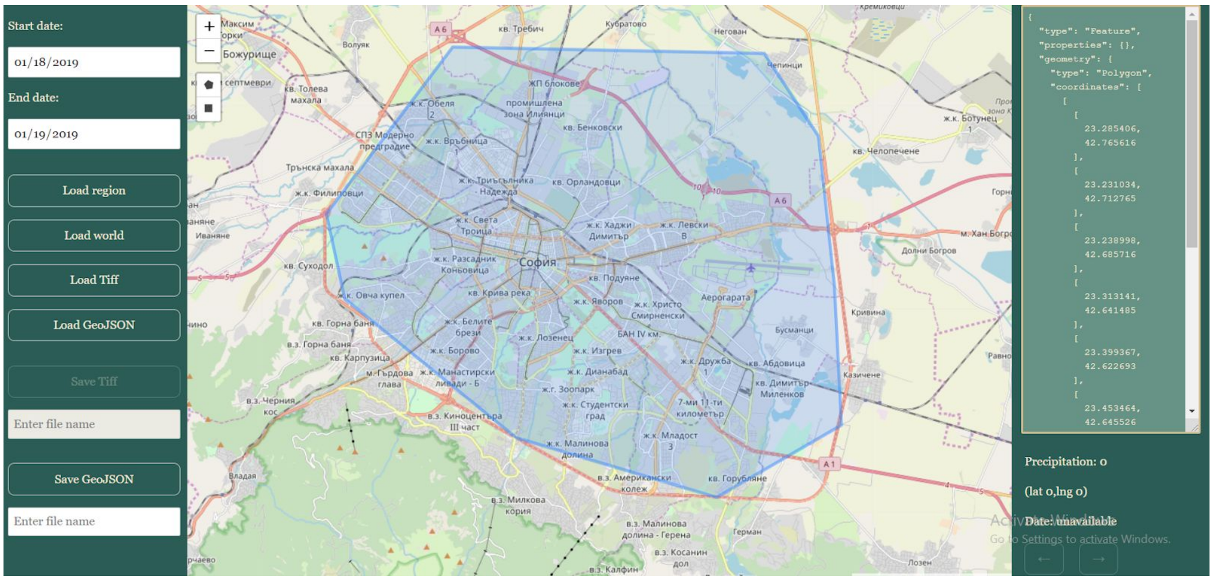


Fig 4.1 Using the visualisation app

When using the app, there are several options for visualising and exploring the data obtained through the server (*Fig 4.1*). The user may simply load a tiff file and view its contents by mousing over the map to see the precipitation values at the corresponding coordinates (i.e. the 0.1 by 0.1-degree square under the cursor), or they may change the band between different dates, given that the loaded file contains information for more than one day.

Alternatively, the user may pick a date range and load the visual data for either the whole world or a selected region, and then once more use the mouse or the band tool, or choose to save a tiff file to their machine; to explore precipitation this way, the files for the chosen dates must be available and downloaded.

When it comes to selecting the region, there are several options at the user's disposal: it can be selected manually using the map tools for a rectangle or polygon shape, loaded from a json file, or directly entered in a json format into the region box. After selection, the region may be loaded to show precipitation (using the Load Region button) if data is available for it, or it may be saved as a json file so it can be reused later by loading that file.



```
{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          23.285406,
          42.765616
        ],
        [
          23.231034,
          42.712765
        ],
        [
          23.238998,
          42.685716
        ],
        [
          23.313141,
          42.641485
        ],
        [
          23.399367,
          42.622693
        ],
        [
          23.453464,
          42.645526
```

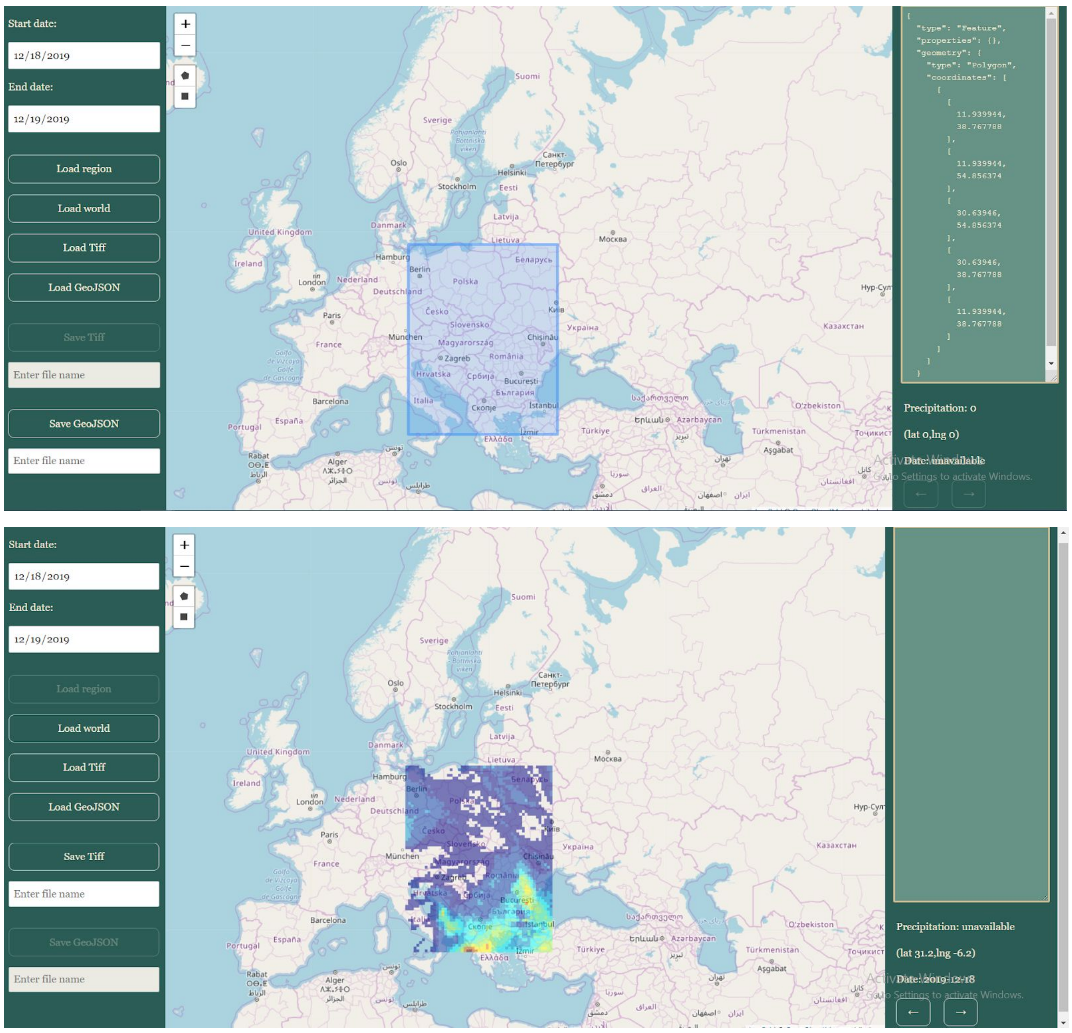
Fig 4.2 Controls and region selection

The screenshot in *Fig 4.2* showcases the functionality of the visualisation app GUI. At the top are the various controls at the user's disposal:

- the Start and End Date fields - upon a mouse click, each shows a calendar widget
- the Load Region and Load World buttons which load the data for a region and the whole world respectively
- the Precipitation Widget, which shows the amount of precipitation on top of a given 0.1 x 0.1 degree square (the accuracy provided by GPM) when it is moused over
- the Band Widget, where the user can use the arrow buttons to switch between bands (each one showing one day of precipitation within the date range)

- the Save Tiff button and a field in which to provide the desired file name
- the Load Tiff button, which allows the user to load a local tiff file and display it on the map
- the Save GeoJSON button which can be used as a tool to “bookmark” a region for later use by saving it in a geojson format
- the Load GeoJSON button
- the JSON Viewer which shows the coordinates of the selected region and can also be used to manually enter a region in a json format

On the map itself, an example of a region selection is shown in blue. Next, the user may manipulate the selection with the controls, for example load the colour representation of the precipitation, choose a range of dates, and then view the data for each of them by clicking through the different bands. Below in *Fig 4.3* and *FIg 4.4* we show examples of a region with loaded data and of a tiff file for the whole world (or, technically, the world region).



Start date: 12/18/2019
End date: 12/19/2019
Enter file name

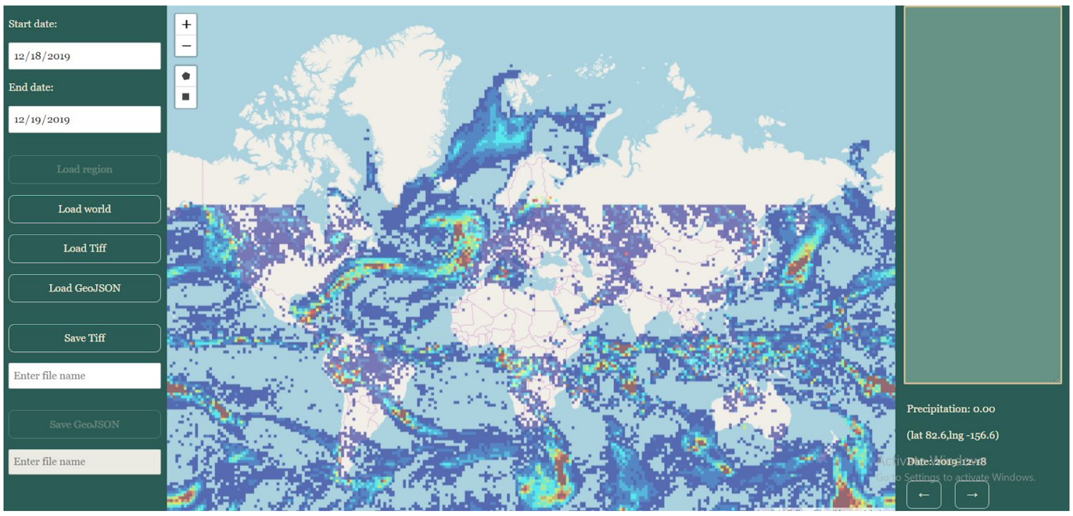
```
{
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          11.939944,
          38.767788
        ],
        [
          11.939944,
          54.856374
        ],
        [
          30.63946,
          54.856374
        ],
        [
          30.63946,
          38.767788
        ],
        [
          11.939944,
          38.767788
        ]
      ]
    ]
  }
```

Precipitation: 0
(lat 0,lng 0)
Date: unavailable

Start date: 12/18/2019
End date: 12/19/2019
Enter file name

Precipitation: unavailable
(lat 31.2,lng -6.2)
Date: 2019-12-18

Fig 4.3 Selected and then loaded region



Precipitation: 0.00

(lat 82.6,lng -156.6)

Date: 2019-12-18

Fig 4.4 Loaded world

In *Fig 4.4*, there is a cutoff line at the top of the map due to the lack of weather data being collected for that area by NASA's GPM mission.

Chapter 5: Using the Web Service

In the present chapter we aim to show an example of the system's usage outside of what has mostly been discussed so far: putting in a server request, downloading the data, and exploring it in the visualisation app. It is possible to use the web service on its own, separate from the app. Here we consider how a user may get and utilise web service data with a simple Python script. Of course, this is just a single example which includes the sum and avg endpoints, and it is possible for the user to create their own scripts utilising the other endpoints as well.

It is assumed that the user has the server running locally and has downloaded some data for the date range which interests them. For instructions on how this can be done please refer to Appendix 2 and Appendix 3.

The requests Python library is used to make requests to the server in this example. The data downloaded is for the first two weeks of January 2020, and refers to a rectangular area over London - a region created using the visualisation app.

```
import requests

# Local server url
base_url = "http://127.0.0.1:8000/"

# Set date range of interest
from_date = '2019-11-15'
to_date   = '2020-01-15'

# Specific urls for sum and avg
sum_url = base_url + f'sum?from={from_date}&to={to_date}'
avg_url = base_url + f'avg?from={from_date}&to={to_date}'

# Area of interest
area = {
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          -0.385266,
          51.356175
        ],
        [
          -0.385266,
          51.606589
        ],
        [
          0.166141,
          51.606589
        ],
        [
          0.166141,
          51.356175
        ],
        [
          -0.385266,
          51.356175
        ]
      ]
    ]
  }
}
```

Once all this has been set up, we may proceed to make the requests and parse the server's responses.

```
# Request sum and avg
sum_resp = requests.get(url= sum_url, json= area)
avg_resp = requests.get(url= avg_url, json= area)

# Parse json response
sum_data = sum_resp.json()
avg_data = avg_resp.json()
```

Now that we have the data we need, we may use it in a variety of ways. In this example we have chosen to create some plots showing the total and average precipitation over our chosen region on the given dates. This is done using the numpy and pyplot libraries. First, the data must be formatted so that it is readable when displayed on a plot.

```
# Used to convert timestamp into human-friendly date format
import datetime
def fts(ts):
      return
datetime.datetime.fromtimestamp(ts).strftime("%d/%m/%Y")

# Used for compatibility with plotting library
import numpy as np
```

# Extract values from response and convert to numpy array
```
sums = np.array([v for [v, _] in sum_data])
avgs = np.array([v for [v, _] in avg_data])
```

# Convert dates for readability
```
dates = np.array([fts(d) for [_, d] in sum_data])
```

Now the plots can be created, starting with the one showing the precipitation sum.

# Used to plot data
```
import matplotlib.pyplot as plt
```

# Plot sums as line chart
```
fig, ax = plt.subplots()
ax.plot(dates[0:31], sums[0:31])
fig.autofmt_xdate()
plt.xlabel('Sums over 15/11/2019 - 15/12/2019')

plt.show()
```

This is the resulting plot.

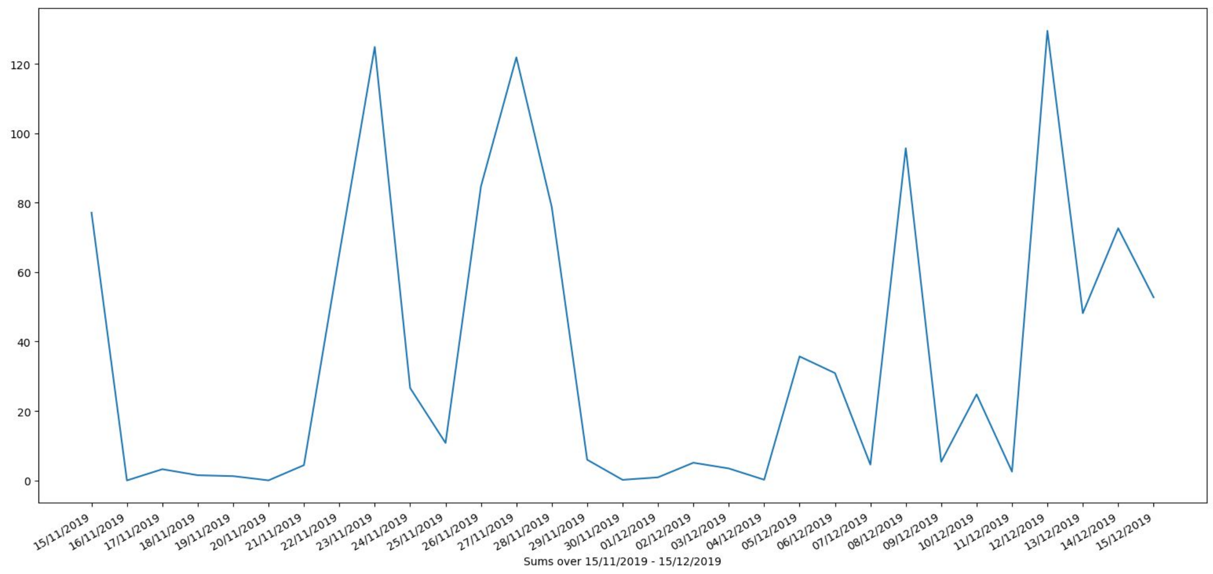


Fig 5.1 Precipitation sum

Next, the average precipitation plot is created.

# Plot averages for first 2 weeks of january
```
avg_week1, avg_week2 = avgs[47:54], avgs[54:61]

plt.plot( avg_week1, color='skyblue', linewidth=2)
plt.plot( avg_week2, color='green', linewidth=2)
plt.legend(['01/01/2020 - 07/01/2020', '08/01/2020 - 14/01/2020'],
loc='upper left')

plt.show()
```

The plot looks as follows.

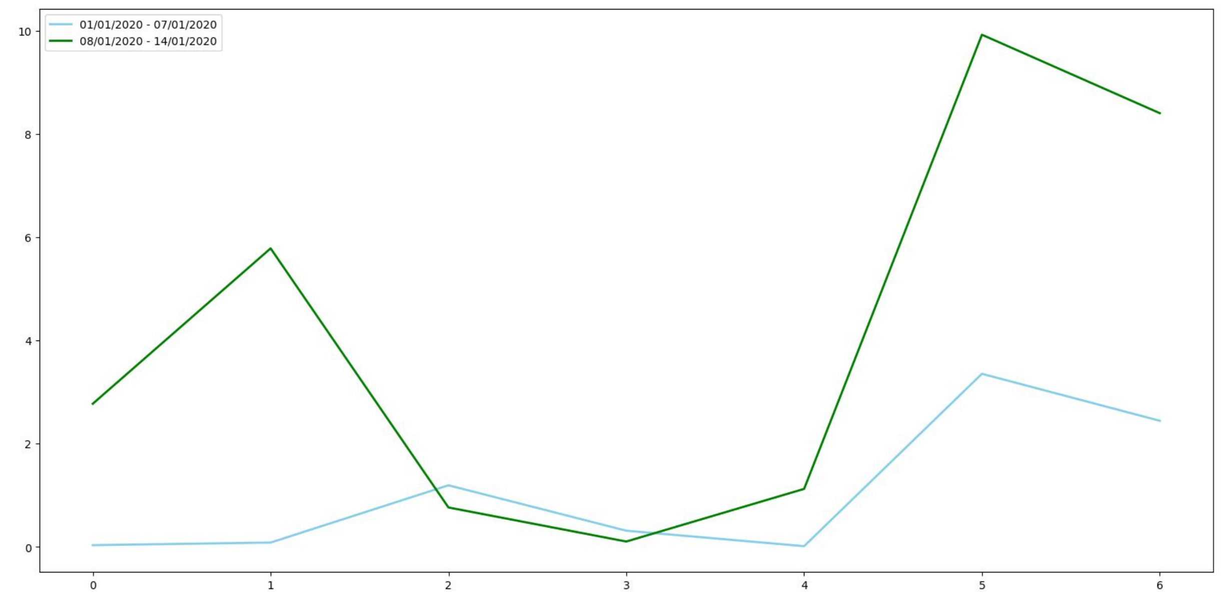


Fig 5.2 Average precipitation

The code-only version of this example is available in Appendix 4.

Chapter 6: Conclusions and Outcomes

The Precipitation Data Processing and Visualisation Web System fulfilled all requirements which were set up at the beginning of the project. It went through an iterative process of development and was improved upon over the course of several months, following the feedback of our supervisor Dr. Jóźwiak and our consultant SatAgro. The firm plans to apply our project as part of their own precision agriculture service which aims to assist farmers in cutting back on product amounts, resulting in reductions in cost and a more environmentally friendly way of farming. We are content with the results we achieved with this project and thankful for the opportunity to interact with the intriguing and productive field of precision agriculture.

Naturally, any project can be improved upon in some way or another. Our system was tested on both small and large regions, and we have determined that using the satellite weather data from the GPM mission is more practical for larger pieces of land, as the format the data comes in uses relatively big unit pieces of land (0.1 by 0.1 degrees). Of course, it is still possible to utilise this data when working with smaller regions, e.g. separate fields, but we believe the system stands to benefit from extensions such as other sources of data, or even from including other types of files available on the NASA GPM mission website (for the scope of this project, we only used tiff). Furthermore, it could be interesting to add even more endpoints to the server, for example ones which use vector based math to produce sums over a whole month or a longer period, instead of only a day. Lastly, more complexity could be added to the visualisation app component: more options like the ability to view various statistics would be a curious extension of its concept.

Overall, we believe the Precipitation Data Processing and Visualisation Web System achieved the goals we set for it and for ourselves as programmers and computer scientists, and we look forward to possible improvements on it in the future. We sincerely thank Dr. Jóźwiak and the team of SatAgro for their indispensable help.

Sources Cited

In order of appearance in-text

1. Jenner, Lynn. “GPM - Global Precipitation Measurement.” *NASA*, NASA, 18 Feb. 2015, https://www.nasa.gov/mission_pages/GPM/main/index.html.
2. “Tap into Satellite Data and Unleash Profits from Your Farm.” *SatAgro*, SatAgro, https://www.satagro.pl/#satagro.
3. “Svelte.” *Svelte*, https://svelte.dev/.
4. “An Open-Source JavaScript Library for Interactive Maps.” *Leaflet*, https://leafletjs.com/.
5. “Django Overview.” *Django*, https://www.djangoproject.com/start/overview/.
6. “NumPy.” *NumPy*, https://numpy.org/.
7. Mike. “Python 101: An Intro to Ftplib.” *The Mouse Vs. The Python*, 23 June 2016, http://www.blog.pythonlibrary.org/2016/06/23/python-101-an-intro-to-ftplib/.
8. “Fetch API.” *MDN Web Docs*, https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API.
9. “GeoTIFF.” *NASA*, NASA, https://earthdata.nasa.gov/esdis/eso/standards-and-references/geotiff.
10. “4326.” *EPSG*, https://epsg.io/4326.

Vocabulary

Downloader	The system component which retrieves the data
GPM mission	NASA's Global Precipitation Measurement mission
NASA server	The NASA FTP server where all the precipitation data is stored
Precision agriculture	A crossover between computer science and agriculture: a method for increasing the efficiency and effectiveness of farming by using satellite data
Server	The system component which parses incoming requests and passes them on to the transformation tools
Tiff file	A tagged image file format; we use specifically the geotiff extension of this format
Transformation tools	The system component which performs geospatial operations on the data
Visualisation app	The system component used for visualising and interacting with the data
Web service	The system component responsible for data retrieval and processing; consists of the downloader, server, and transformation tools

List of Abbreviations

API	Application Programming Interface
FTP	File Transfer Protocol
GeoJSON	Geographical JavaScript Object Notation
GPM	Global Precipitation Measurement
GPV	Global Precipitation Viewer
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
REST API	Representational State Transfer API
TIFF	Tagged Image File Format

List of Figures

List of Tables

List of Appendices

Deployment Documentation	Contains technical documentation regarding the system's requirements and the configuration needed to run it
Installation Instructions	Contains a step-by-step guide to installing and running the system
User Manual for the Web Service	Contains instructions for using the web service part of the system, including examples for all endpoints and required input formats
Web Service Example	Contains the code-only version of the example usage given in Chapter 5
Code Snippets	Contains example pieces of code central to the system
Project Schedule and Division of Labour	Contains a rough guide to the timeline on which the project was completed and an approximate division of labour between the authors

Appendices

Appendix 1: Deployment Documentation

Requirements

- Python 3.7 +
- NodeJs
- NPM
- Svelte

Table a1.1 Required libraries

Library Name	Version	Use
Visualisation App		
colormap	2.3.1	Generates the colors that represent precipitation
sirv-cli	0.4.4	Used in development as a mock server
svelte-extras	2.0.2	Provides additional Svelte components
Web Service		
affine	2.3.0	Dependency
aiofiles	0.4.0	Dependency
APScheduler	3.6.1	Dependency
arrow	0.15.2	Parser for different date formats
attrs	19.1.0	Dependency
Django	2.2.7	Server framework
djangorestframework	3.10.3	Toolkit for Rest Api building
Fiona	1.8.8	Dependency
httpcore	0.3.0	Dependency
httptools	0.0.13	Dependency
multidict	4.5.2	Dependency

numpy	1.17.2	Used for matrix calculations
py	1.8.0	Dependency
pyparsing	2.4.2	Dependency
pytest	5.2.0	Test framework
python-dateutil	2.8.0	Dependency
pytz	2019.3	Dependency
rasterio	1.0.28	Used mainly for reading and writing tiff files
requests	2.22.0	Dependency
requests-async	0.5.0	Dependency
Shapely	1.6.4.post2	Used for parsing geojson
six	1.12.0	Dependency
snuggs	1.4.7	Dependency
tzlocal	2.0.0	Dependency
urllib3	1.25.6	Dependency

Configuration

Windows 10 is the best option as most testing has been done on it, but Ubuntu 16.04 is also possible.

Configuration files

Each component has a *logging_config.py* file which handles where to save logs and how to format them. Apart from that, there are two more configuration files:

- *server_config.py*: basic configurations for the server such as download path, server date format, etc.
- *config.py* (in downloader): handles the filename patterns for the downloaded files and the date formats.

Appendix 2: Installation Instructions

How to Install

Python 3.7 or newer required. Tested on Windows 10 and Ubuntu 16.04. On Windows, it is a safer option to install Anaconda and install the project inside a new environment.

Step 1: Copy the contents of the CD into a directory on your computer.

Step 2: Using a console inside the main project folder run the command below (on Windows, use the Anaconda prompt for this step):

```
pip install -r requirements.txt
```

How to Run the Web Service

Step 1:

```
cd <webservice directory in the project folder>
```

Step 2: Set the DJANGO_SECRET_KEY environment variable (platform dependent) to your preference.

Step 3:

```
python manage.py runserver <PORT> --settings=api.settings.prod
```

Port defaults to 8000.

How to Run the Downloader

The downloader is a separate component of the project and is meant to be run on its own.

While in the webservice directory, type the following to download the latest files:

```
python download_latest.py
```

To download all files starting from the date defined in downloader/config.py type:

```
python download_all.py
```

The downloader can be configured by changing the variables inside the config.py file:

- Username and password: login information for the nasa gpm ftp server; an account can be made at https://registration.pps.eosdis.nasa.gov/registration/

- Output_filename: can pick variables - year, month, day, original name (the name that was on the server)
- Output_dir: base directory in which the files will be downloaded
- Start_date: for the download_all.py script; will download all late files from this date up to now (that are available)
- Date_format: The format of the starting date
- Filetype: The type of data: precipitation (“all”), ice, liquid, or liquidPercent

It is best to have the downloader be run periodically, for instance using crontab. Below is an example set to run every 60 min. Early files are released once every 3 hours. 1/3 of the time is chosen in case of an error on the NASA server. If the files have already been downloaded, they will not be downloaded again.

```
0 0 0/1 1/1 * ? * python3 download_latest.py
```

Keep in mind that if you are using a Python virtual environment for the project, then you should add to the script above the activation for it. For instance:

```
0 0 0/1 1/1 * ? * source envs/projectenv; python3
download_latest.py
```

How to Run the Visualisation App Locally

Requires Node.js and NPM to be installed.

In a console inside the app folder type

```
npm install
npm run dev
```

It is on port 5000.

Appendix 3: User Manual for the Web Service

Rest API

Basic request requirements:

- `from`: Request should contain from and to fields (with date in the preferred format).
- `to`: Request should contain geojson.
- (Optional) `date_format`: Request can contain a date_format field to indicate which format the dates are given in. The default is “YYYY-MM-DD.”
- (Optional) `type`: Request can contain a type field that indicates which type of file to use. The available options are: [all, liquid, liquidPercent, ice]. Set to 'all' by default.
    - Additional information about file types can be found in "IMERG GIS product description.pdf" - provided by NASA.
- (Optional) `round_to`: number of digits after the decimal point. Defaults to 2.

Example geojson:

```
{ "geometry": {
        "type": "Polygon",
        "coordinates": [
          [
            [
              -88.24218749999999,
              41.50857729743935
            ],
            [
              -59.765625,
              -40.97989806962013
            ],
            [
              39.0234375,
              12.897489183755892
            ],
            [
              12.3046875,
              54.57206165565852
            ],
            [
              -88.24218749999999,
              41.50857729743935
            ]
          ]
        ]
    }
}
```

Precipitation cumulative series

From this section onwards, examples use curl. Please remember to remove new lines when copying them to the command line.

Request total precipitation over an area:

- Url: “/sum?from=YYYY-MM-DD&to=YYYY-MM-DD”
- GET only
- Example:

```
curl -X GET -H "Content-Type: application/json" -d
"{\"geometry\":{\"type\": \"Polygon\", \"coordinates\":
      [[[20.021939277648926, 52.552976197007524],
      [20.02585530281067, 52.554913612568825],
      [20.026885271072388, 52.55413735195836],
      [20.022990703582764, 52.552206425665396],
      [20.021939277648926, 52.552976197007524]]]}}"
      "http://localhost:8000/sum?from=2019-11-01&to=2019-11-15"
```

Precipitation average series

Request average precipitation per unit of area in given geometry:

- Url: “/avg?from=YYYY-MM-DD&to=YYYY-MM-DD&over=meters”
- GET only
- Additional Parameters:
    - over: "meters" or "degrees" (squared)
- Example:

```
curl -X GET -H "Content-Type: application/json" -d
"{\"geometry\":{\"type\": \"Polygon\", \"coordinates\":
      [[[20.021939277648926, 52.552976197007524],
      [20.02585530281067, 52.554913612568825],
      [20.026885271072388, 52.55413735195836],
      [20.022990703582764, 52.552206425665396],
      [20.021939277648926, 52.552976197007524]]]}}"
      "http://localhost:8000/avg?from=2019-11-01&to=2019-11-15&over
=meters"
```

Precipitation over a location

Request precipitation around a point (creates square of 0.1 degree size centered around the point):

- Url: “/loc?from=YYYY-MM-DD&to=YYYY-MM-DD”
- GET only
- Example:

```
curl -X GET -H "Content-Type: application/json" -d
"{\"geometry\":{\"type\": \"Point\", \"coordinates\":
      [20.021939277648926, 52.552976197007524]}}"
      "http://localhost:8000/loc?from=2019-11-01&to=2019-11-15&roun
d_to=3"
```

Precipitation cumulative imagery

Request a tiff of precipitation over an area:

The first band contains percentage coverage info. Subsequent bands contain precipitation amounts (band names are dates).

```
GET /tiff?from=...&to=...
```

Imagery logging

Request which files are available on the server. Returns an array of pairs, such as `[['2019-10-10', true], ... ]`. The date format is kept the same as the one provided in the request.

```
GET /available?from=...&to=...
```

Running the tests

To run the tests type the command below into a console while inside the main folder of the project:

```
pytest -v --disable-warnings
```

Appendix 4: Web Service Example

```
# Example of getting and using web service data
# Server running locally and data for the dates of interest is downloaded

# Using requests library to make the actual request to the server
import requests

# Local server url
base_url = "http://127.0.0.1:8000/"

# Set date range of interest
from_date = '2019-11-15'
to_date   = '2020-01-15'

# Specific urls for sum and avg
sum_url = base_url + f'sum?from={from_date}&to={to_date}'
avg_url = base_url + f'avg?from={from_date}&to={to_date}'

# Area of interest - rectangle over London, made using visualization app
area = {
  "type": "Feature",
  "properties": {},
  "geometry": {
    "type": "Polygon",
    "coordinates": [
      [
        [
          -0.385266,
          51.356175
        ],
        [
          -0.385266,
          51.606589
        ],
        [
          0.166141,
          51.606589
        ],
        [
          0.166141,
          51.356175
        ],
        [
          -0.385266,
          51.356175
        ]
      ]
```

```
        ]
      }
    }
```

# Request sum and avg

```
sum_resp = requests.get(url= sum_url, json= area)
avg_resp = requests.get(url= avg_url, json= area)
```

# Parse json response

```
sum_data = sum_resp.json()
avg_data = avg_resp.json()
```

# Used to convert timestamp into human-friendly date format

```
import datetime
def fts(ts):
return datetime.datetime.fromtimestamp(ts).strftime("%d/%m/%Y")
```

# Used for compatibility with plotting library

```
import numpy as np
```

# Extract values from response and convert to numpy array

```
sums = np.array([v for [v, _] in sum_data])
avgs = np.array([v for [v, _] in avg_data])
```

# Convert dates for readability

```
dates = np.array([fts(d) for [_, d] in sum_data])
```

# Used to plot data

```
import matplotlib.pyplot as plt
```

# Plot sums as line chart

```
fig, ax = plt.subplots()
ax.plot(dates[0:31], sums[0:31])
fig.autofmt_xdate()
plt.xlabel('Sums over 15/11/2019 - 15/12/2019')

plt.show()
```

# Plot averages for first 2 weeks of january

```
      avg_week1, avg_week2 = avgs[47:54], avgs[54:61]
      plt.plot( avg_week1, color='skyblue', linewidth=2)
      plt.plot( avg_week2, color='green', linewidth=2)
      plt.legend(['01/01/2020 - 07/01/2020', '08/01/2020 - 14/01/2020'],
loc='upper left')

      plt.show()
```

Appendix 5: Code Snippets

Web Service

# From the tiff tools: tiff window reading
```
def read_window(path, geometry):
  try:
    with rio.open(path) as data:
      minx, miny, maxx, maxy = geometry.bounds

      minx, miny = round_down(minx), round_down(miny)
      maxx, maxy = round_up(maxx), round_up(maxy)

      window = rio.windows.from_bounds(minx, miny, maxx, maxy, TIF_AFFINE)

      return data.read(TIF_BAND, window=window), None
  except Exception as e:
      return None, e
```

# From downloader: download_task task execution
```
def run(self):
        # Connect to server
        with FTP(self.server) as ftp:
            ftp.login(passwd=self.username, user=self.username)

            cwd = "./"
            ftp.cwd(cwd)

            # Iterate through given dates
            for date in self.dates:
                # Check if file for given date needs to be downloaded
                if not self.needs_download(date):
                    logger.info(f'Does not need download for
{date.year}-{date.month}-{date.day}')
                    continue

                # Find and change to dir of file to download
                remote_dir = self.remote_dir_func(date)
                if cwd != remote_dir:
                    ftp.cwd(remote_dir)
                    cwd = remote_dir

                # Get available files in dir and select which to download
                available_files = ftp.nlst()
                to_download = self.select_files_func(date, available_files)

                if len(to_download) == 0:
                    logger.info(f'Nothing to download for
{date.year}-{date.month}-{date.day}')
                    continue
                # Download each selected file
                for download_item in to_download:
                    downloaded = True
                    save_path = os.path.join(self.save_dir, download_item)
                    dirname = os.path.dirname(save_path)
                    if not os.path.exists(dirname):
                        os.makedirs(dirname)
                    try:
                        ftp.retrbinary(f'RETR {download_item}',
open(save_path, 'ab+').write)
                    except Exception as e:
                        logger.error(f'Could not download: {download_item}')
                        logger.error(f'\tError: {e}')
                        downloaded = False

                    # Run post download operation
                    if downloaded:
                        logger.info(f'Downloaded: {download_item}, saved as:
{save_path}')
                        self.post_download_func(download_item, save_path,
date)
```

Visualisation app

// Parse array buffer into georaster

```
const parse_raster = (arrayBuffer) => {
    buffer = arrayBuffer.slice(0)
    parseGeoraster(arrayBuffer)
        .then(georaster => {
            // Store new georaster
            tif_raster = georaster
            // Round bounding values
            tif_raster.xmin = round_down(tif_raster.xmin)
            tif_raster.ymin = round_down(tif_raster.ymin)
            tif_raster.xmax = round_up(tif_raster.xmax)
            tif_raster.ymax = round_up(tif_raster.ymax)
            // Reload visual
            reload_tif_layer()
            // Clear drawn layer
            if (draw_layer)
                drawnItems.removeLayer(draw_layer)
            // Clear json region
            region = ""
        }).catch(err => {
            alert("Could not parse")
            console.error(err)
        })
}
```

// Get tiff file from server

```
fetch(addr + `/tiff?from=${start_date}&to=${end_date}&type=all`, {
      method: "post",
      mode: "cors",
      body: region,
      headers: {
            "Content-Type": "application/json"
      }
      })
      .then(response => response.arrayBuffer())
      .then(arrayBuffer => {
            // parse tif file into georaster
            parse_raster(arrayBuffer)
      })
      .catch(err => {
            alert("Could not load the region")
            console.error(err)
      })
```

Appendix 6: Project Schedule and Division of Labour

Authors' note: Here we present a rough distribution of the labour which went into creating this work, as per faculty guidelines. We kindly ask the reader to keep in mind that this project was a collaboration, and a strict, definitive division is unfeasible due to the active involvement of both authors in all aspects of the finished product.

Table a6.1 Completion timeline and responsibilities

Task	Time frame	Responsible author
Project stubs	June 2019	Siyana Ivanova
Basic project structure	June 2019	Siyana Ivanova
Initial versions of modules	July 2019	Marin Karamihalev
Basic downloader	July 2019	Marin Karamihalev
Downloader updates (as requested by SatAgro)	July 2019	Siyana Ivanova
Early-stage testing	July 2019	Marin Karamihalev
Improvements to tiff tools	August 2019	Siyana Ivanova
Initial version of visualisation app	August 2019	Siyana Ivanova
Unit testing	August 2019	Marin Karamihalev
New versions of downloader and tiff tools	September 2019	Marin Karamihalev
Improvements to processing time	September 2019	Marin Karamihalev
Improvements to testing	September 2019	Marin Karamihalev
New version of visualisation app	October 2019	Siyana Ivanova
Ability to view tiff file in visualisation app	October 2019	Marin Karamihalev
Ability to choose file types in downloader	October 2019	Siyana Ivanova
Logging	November 2019	Marin Karamihalev
Extensions to testing	November 2019	Marin Karamihalev
Django REST framework	November 2019	Marin Karamihalev

Initial documentation	November 2019	Siyana Ivanova
Design document	November 2019	Siyana Ivanova
Chapter 1	November 2019	Siyana Ivanova
Technology selection, Communication, External Interfaces sections: Chapter 2	November 2019	Marin Karamihalev
System Architecture and Module Design, User Interface sections: Chapter 2	November 2019	Siyana Ivanova
Region selection in web app	December 2019	
Saving and loading capabilities of web app	December 2019	Siyana Ivanova
Additional endpoints: avg, loc	December 2019	Marin Karamihalev
Technical documentation	December 2019	Siyana Ivanova
Abstract and introduction	December 2019	Siyana Ivanova
Deployment documentation	December 2019	Siyana Ivanova
Web app visual design	January 2020	Siyana Ivanova
Python documentation	January 2020	Marin Karamihalev
Chapter 3 - text	January 2020	Siyana Ivanova
Chapter 3 - table	January 2020	Marin Karamihalev
Chapter 4	January 2020	Siyana Ivanova
Chapter 5	January 2020	Marin Karamihalev
Chapter 6	January 2020	Siyana Ivanova
Utility sections	January 2020	Siyana Ivanova
Appendix 1	January 2020	Siyana Ivanova
Appendix 2	January 2020	Marin Karamihalev
Appendix 3	January 2020	Marin Karamihalev