POLITECNICO MILANO 1863

"Python-based Polygon and Raster Generation and Visualization of Different Telecommunication Global Maps in QGIS and Geoserver"

Student:
Fatemeh Soleimaniansomarin
10898039

Project supervisor:
Maria Antonia Brovelli

Table of contents

Contents

1. Introduction

This project was conducted as a 2-month summer internship in International Telecommunication Union (ITU). The International Telecommunication Union (ITU) stands as the specialized United Nations agency dedicated to information and communication technologies. With its primary objective being the global interconnection of people, ITU is steadfast in its commitment to facilitating international connectivity within communication networks. A ubiquitous presence worldwide, this United Nations entity holds a preeminent position across numerous domains associated with ICT. The standards, protocols, and international agreements set forth by ITU play an indispensable role in ensuring the active functionality of the global telecommunication system.

In the field of telecommunication services, spatial analysis plays a crucial role in enhancing service quality and network efficiency. This project focuses on generating both vector and raster data from flat telecommunication datasets, leveraging the powerful capabilities of Python and QGIS. By visualizing and analyzing this data, we aim to create detailed maps that highlight key telecommunication metrics such as rain zones and noise zones. These visual tools are crucial for understanding the spatial distribution of various factors affecting telecommunication performance and for making informed decisions to optimize service delivery.

Geographic Information Systems (GIS) have proven indispensable across numerous fields, including telecommunication, due to their ability to manage, analyze, and visualize spatial data effectively. In telecommunication, GIS is instrumental in planning and optimizing network infrastructure, monitoring service quality, and addressing coverage issues. This project's core objective is to transform raw telecommunication data into meaningful spatial representations. By generating vector and raster layers, we provide a comprehensive view of areas with varying levels of rain intensity and noise interference, which are critical for network planning and management.

The insights derived from these maps facilitate better telecommunication services by identifying problem areas, optimizing resource allocation, and improving overall network reliability and user experience. This report delves into the methodologies employed, the data processing techniques, and the analytical outcomes, illustrating the significance of spatial analysis in telecommunication infrastructure and underscoring the broader impact of GIS in advancing various sectors.

This dual-sided project required a comprehensive understanding of geospatial concepts, data processing techniques, and proficient programming skills. The resulting output of vector data encompassed three widely used geospatial formats: JSON, GEOJSON, and Geo-package, providing the ability to visualize the generated polygons seamlessly in various Geographic Information System (GIS) platforms. Simultaneously, the data-to-raster conversion enabled a more nuanced representation of the data, enhancing analytical capabilities and visualization in GeoTIFF format.

The polygons and raster data, presented in these formats, hold significant potential for insightful analysis in the telecommunication area. In this report, I will elaborate on the methodologies employed, the challenges encountered, the innovative solutions devised, and the outcomes achieved during this project.

The successful completion of this dual-purpose project underscores the critical role of geospatial data processing and visualization in modern data-driven applications. It highlights the potential to drive informed decision-making and advance various domains, emphasizing the importance of both polygon generation and raster data conversion using Python scripts.

2. Project Development:

2.1. Vector generation:

2.1.1. Overview:

The goal of this part is to create JSON, GEOJSON and Geo-Package from the flat files of the data that are supposed to be polygons showing different values.

JSON, which stands for JavaScript Object Notation, is a lightweight data-interchange format that is easy for humans to read and write, and easy for machines to parse and generate. It is a text-based format that is often used to transmit data between a server and a web application as an alternative to XML.

JSON is built on two main structures:

- **Objects:** An object is an unordered collection of key-value pairs. Each key-value pair in an object is called a property. Properties are separated by commas, and the key and value are separated by a colon. An object is enclosed in curly braces {}.
- **Arrays:** An array is an ordered list of values. Values can be strings, numbers, objects, arrays, or other data types. Arrays are enclosed in square brackets [].

JSON provides a standardized way to structure data that can be easily understood by different programming languages, making it a popular choice for data interchange on the web. It's commonly used for configuration files, APIs, and data storage.

GeoJSON is a format for encoding geographic data structures using JavaScript Object Notation (JSON). It provides a standardized way to represent geographical features, such as points, lines, and polygons, along with their associated attributes. GeoJSON is commonly used in web-based mapping applications and Geographic Information System (GIS) software to store and exchange spatial data in a human-readable and machine-parsable format. It allows for the easy sharing and integration of geographical information across various platforms and applications.

GeoPackage is an open standard for storing and sharing geospatial data, aiming to provide a platform-independent, file-based format that combines both vector and raster data in a single SQLite database. It enables efficient storage and access to complex geospatial information, including geometry, attributes, and tile matrices, all within a single portable file. GeoPackage supports a wide range of geographic data types and allows for seamless interoperability across diverse geospatial applications, making it a versatile choice for managing and distributing geographic data.

The maps that I created are RZ(Rin Zone), NZ(Noise Zone), RDARA(Regional and Domestic Air Route Area), VOLMET(weather information), and MWARA(Major World Air Route Area) maps.

Each of these maps plays a critical role in the overall framework of telecommunication services, especially in sectors that rely heavily on precise and reliable data for operational efficiency and safety.

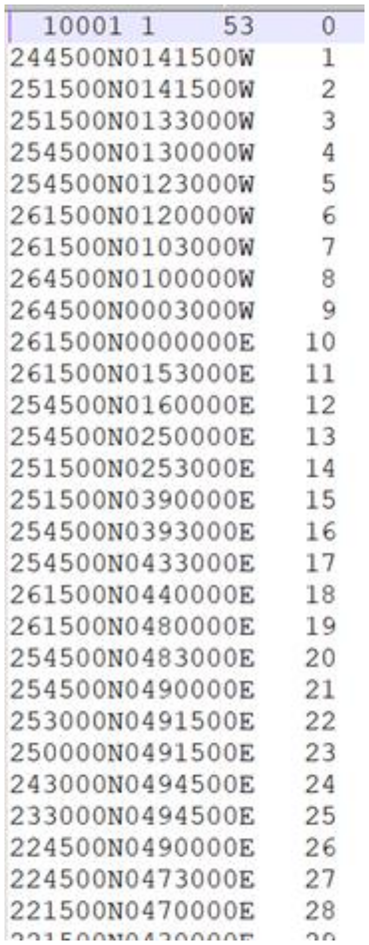
2.1.2. Data structure:

We had some data from different maps that needed plotting. The structure of data is that one file contains the coordinates of points and the other file contains the values. Data is grouped in different groups based on values, so each group of points with the same value indicates one contour and one polygon on the final map.

The structure of files ending with LINES-IDX is two columns. The first one is the contour index, and the second one is the value of that contour. For the Python script to read the text file and to be able to spate the columns, we want to separate each column with a tab, so I edited the original file and saved it in a new file FILES-IDSX-edited.

The file ending with LINES are structured as:

For each contour first the contour index, and the first and last point numbers in one line then in the following lines for each point we have Latitude, Longitude, and point number. This repeats for each contour.



```
 10001 1     53     0
244500N0141500W     1
251500N0141500W     2
251500N0133000W     3
254500N0130000W     4
254500N0123000W     5
261500N0120000W     6
261500N0103000W     7
264500N0100000W     8
264500N0003000W     9
261500N0000000E    10
261500N0153000E    11
254500N0160000E    12
254500N0250000E    13
251500N0253000E    14
251500N0390000E    15
254500N0393000E    16
254500N0433000E    17
261500N0440000E    18
261500N0480000E    19
254500N0483000E    20
254500N0490000E    21
253000N0491500E    22
250000N0491500E    23
243000N0494500E    24
233000N0494500E    25
224500N0490000E    26
224500N0473000E    27
221500N0470000E    28
```

Figure 1. Dataset LINES.txt for one of the maps.

2.1.3. Plotting and generating polygons:

2.1.3.1. Python code to plot the polygons:

In the python script" Creating-jsons-and-splitting-and-gpkg" (annex 1) steps are explained completely. But the general idea is that, in the first section of the script, we get the lines file, separate each contour with contour ID, and the points and coordinates related to that contour ID. We also convert the coordinates from degrees, minutes, and seconds to degrees.

Then, we read the values files and we assign the values to each contour ID. We also define the attributes of the polygons that we want to create. The type of our geometry is "polygon", we will also have columns value, contour ID, and features at the end. Now we can create a JSON file for each contour. In the next section of the script, we merge JSON files to create one JSON file for all contours. And, in the last section, we create a Geo Package that can be visualized in QGIS.

2.1.3.2. Visualizing overlapping polygons:

Upon exporting the file into QGIS, a comprehensive assessment of the maps is essential. This evaluation is crucial to identify and rectify potential errors, notably instances of overlapping contours. Often, numerous contours might be concealed by a larger, overarching contour, rendering their visibility problematic unless the file adopts a transparent style. Furthermore, post the merging of files into a singular map, it's not uncommon to encounter duplicated contours that necessitate modification or removal.

In maps like Ground Conductivity Maps, we have a vector consisting of polygons or contours of different values. When we categorize the polygon based on the value attribute, we can see that some features overlap with each other.

In this case, the polygons of the same layer have an overlap or have a so-called self-overlap, the overlapping polygons can be recognized by a tool in QGIS. The tool is a “polygon self-intersection”. It is a tool from the SAGA plugin. It shows you the polygons that intersect with others.

To remove the overlap by following these steps:

- Select the overlapping polygons
- Use the “difference” tool and make the difference of the layer with the selected features of the same layer.
- Copy the selected features and paste them into a new layer.
- Merge the differenced layer with the copy of selected features.
- In case different vector layers overlap each other, we simply use the “difference” tool in QGIS to remove the overlap, between every two polygons.

There were also cases where the overlapping polygons shouldn’t be fixed because they show completely different values. In the case of VOLMET, MWARA, and RDARA maps, the polygons have an overlap. We just need to make them transparent, so all of them are visible, by following these steps.

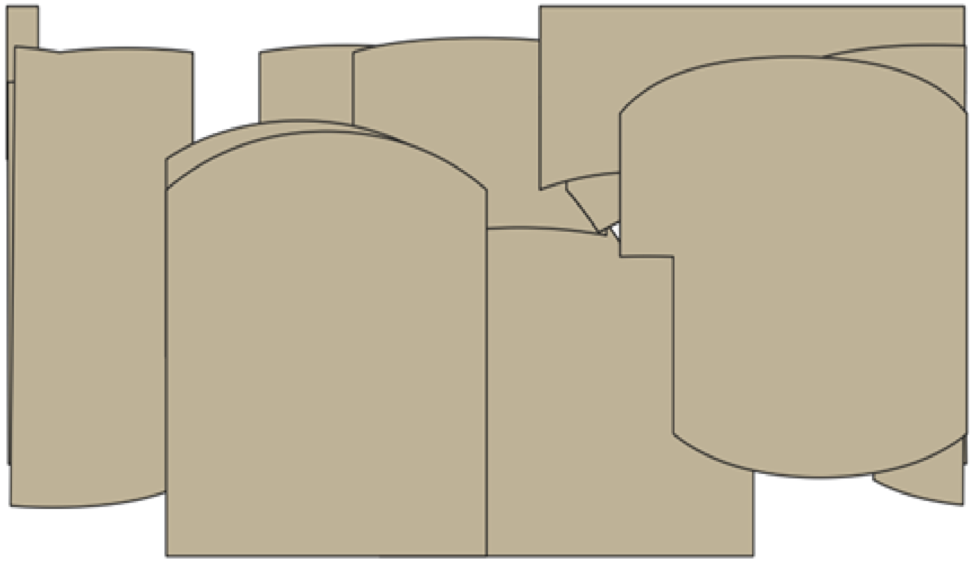
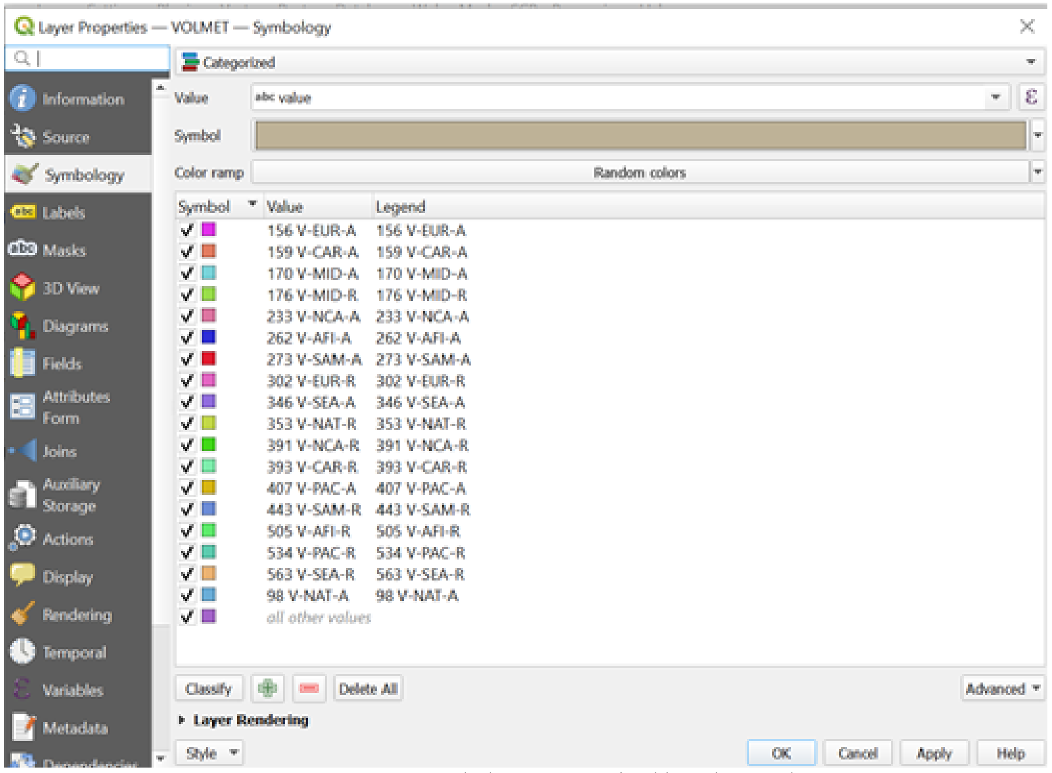


Figure 2. VOLMET map overlapping



Categorized

Value: value

Symbol

Color ramp: Random colors

Symbol	Value	Legend
✓ 🟪	156 V-EUR-A	156 V-EUR-A
✓ 🟧	159 V-CAR-A	159 V-CAR-A
✓ 🟦	170 V-MID-A	170 V-MID-A
✓ 🟩	176 V-MID-R	176 V-MID-R
✓ 🟥	233 V-NCA-A	233 V-NCA-A
✓ 🟦	262 V-AFI-A	262 V-AFI-A
✓ 🟥	273 V-SAM-A	273 V-SAM-A
✓ 🟪	302 V-EUR-R	302 V-EUR-R
✓ 🟪	346 V-SEA-A	346 V-SEA-A
✓ 🟩	353 V-NAT-R	353 V-NAT-R
✓ 🟩	391 V-NCA-R	391 V-NCA-R
✓ 🟩	393 V-CAR-R	393 V-CAR-R
✓ 🟨	407 V-PAC-A	407 V-PAC-A
✓ 🟦	443 V-SAM-R	443 V-SAM-R
✓ 🟩	505 V-AFI-R	505 V-AFI-R
✓ 🟩	534 V-PAC-R	534 V-PAC-R
✓ 🟧	563 V-SEA-R	563 V-SEA-R
✓ 🟦	98 V-NAT-A	98 V-NAT-A
✓ 🟪	*all other values*	

Figure 3. Symbology: Categorized based on “value”

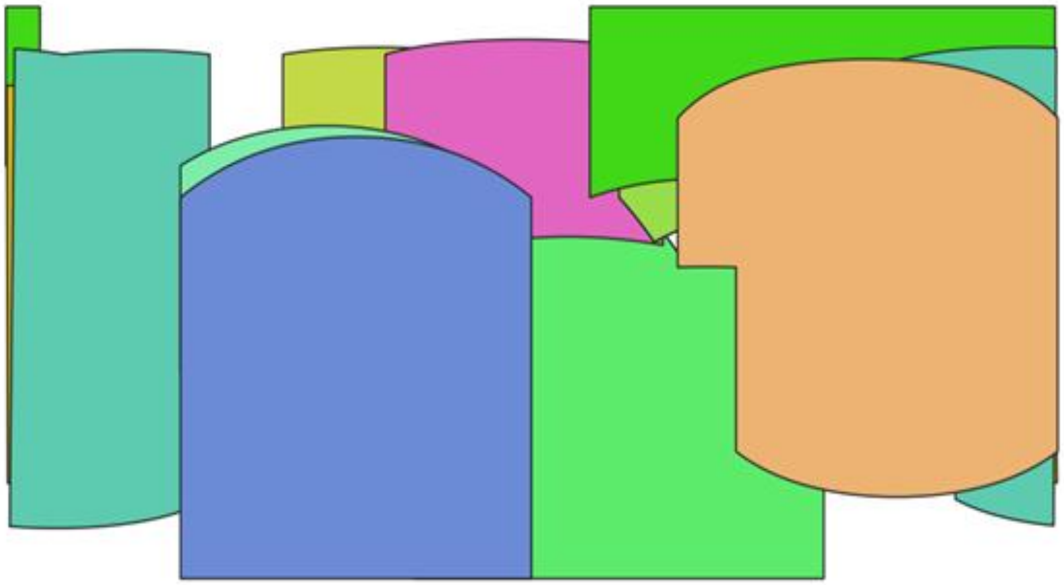
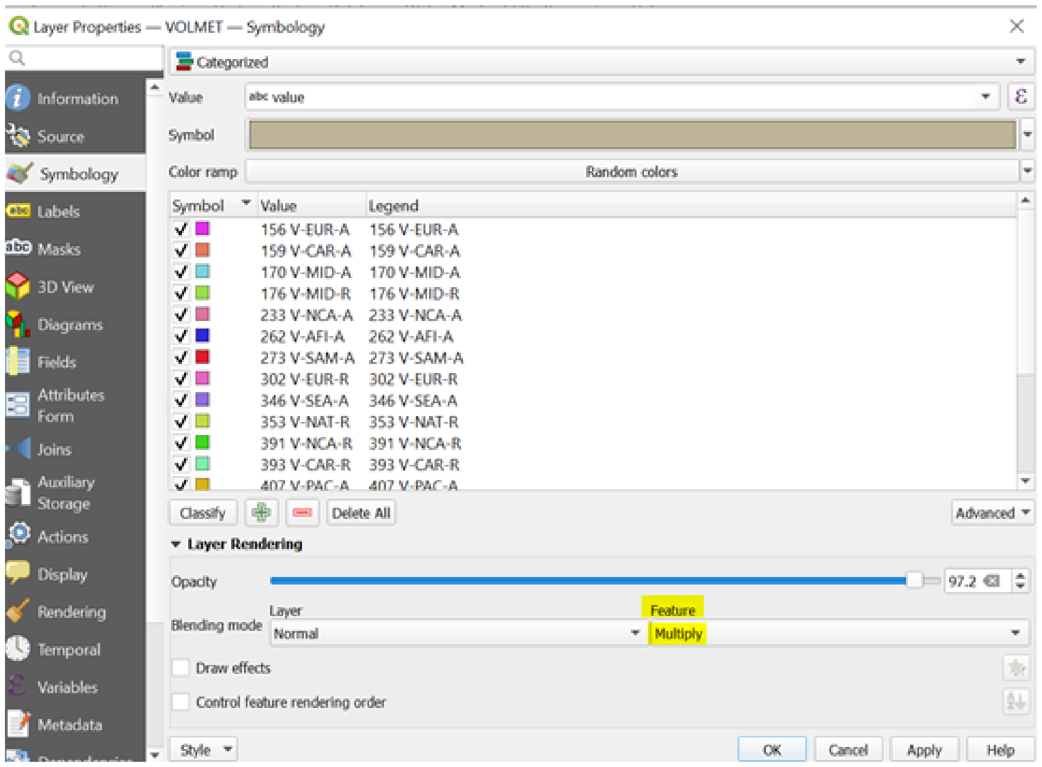


Figure 4. Categorizing result of VOLMET map



Layer Properties — VOLMET — Symbology

Categorized

Value: value

Color ramp: Random colors

Symbol	Value	Legend
✓ 🟪	156 V-EUR-A	156 V-EUR-A
✓ 🟧	159 V-CAR-A	159 V-CAR-A
✓ 🟦	170 V-MID-A	170 V-MID-A
✓ 🟩	176 V-MID-R	176 V-MID-R
✓ 🟥	233 V-NCA-A	233 V-NCA-A
✓ 🟦	262 V-AFI-A	262 V-AFI-A
✓ 🟥	273 V-SAM-A	273 V-SAM-A
✓ 🟪	302 V-EUR-R	302 V-EUR-R
✓ 🟪	346 V-SEA-A	346 V-SEA-A
✓ 🟩	353 V-NAT-R	353 V-NAT-R
✓ 🟩	391 V-NCA-R	391 V-NCA-R
✓ 🟩	393 V-CAR-R	393 V-CAR-R
✓ 🟨	407 V-PAC-A	407 V-PAC-A

Layer Rendering

Opacity 97.2

Blending mode — Layer: Normal — Feature: Multiply

Figure 5. Symbology of the layer VOLMET, we put features mode on Multiply.

So all layers are visible:

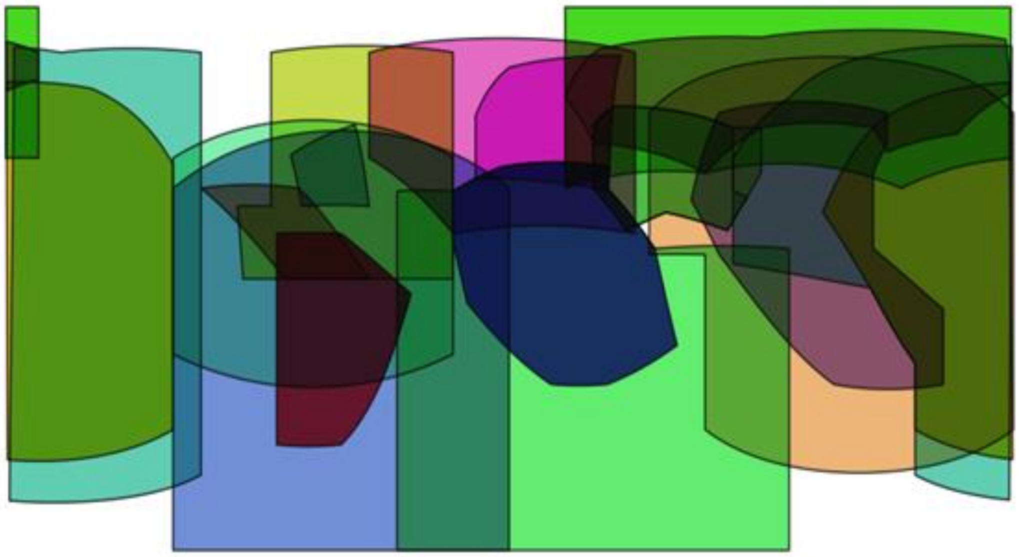
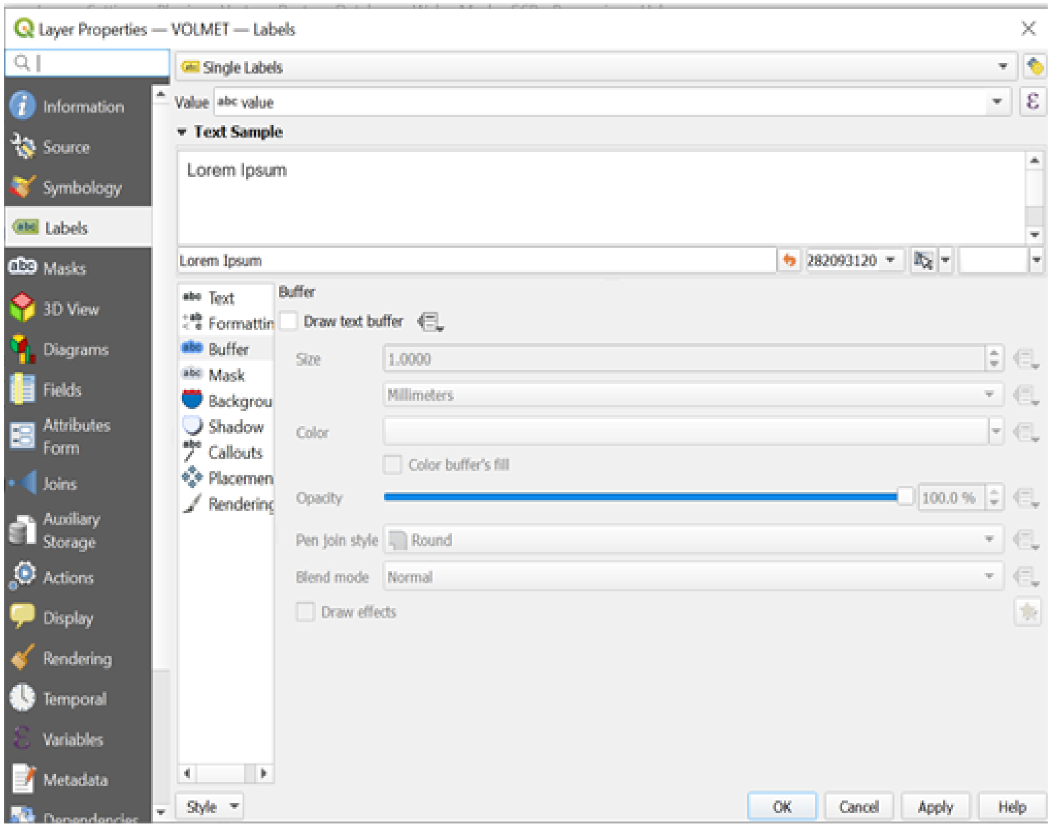


Figure 6. VOLMET final view.

We can also label the polygons. At first by selecting "**layer properties**", then select the tab "**label**" and then choose "**single label**" as shown in Figure 7.



Layer Properties — VOLMET — Labels

Single Labels

Value: abc value

Text Sample

Lorem Ipsum

Buffer

Draw text buffer

Size: 1.0000

Millimeters

Color

Color buffer's fill

Opacity: 100.0 %

Pen join style: Round

Blend mode: Normal

Draw effects

Figure 7. How to label the polygons

2.1.3.3. Splitting issue:

The maps we created are 180-degree maps that typically refer to maps that cover the entire longitudinal range of the Earth, from -180 degrees to 180 degrees. This can cause some challenges while mapping.

When we generate the polygons using the Python script, some polygons may have some problems that we need to fix by code, manually, or by correcting their JSON files.

The first problem is about 180-degree maps and to solve the problem we need to split the polygon into two polygons, plot each one, and then join them together.

For this concern, I used the splitting Python script (annex 2), in which we need to insert the troubled contours IDs and the splitting point longitude.

The splitting point should be a Longitude within which two polygons shouldn't connect. It means that we create one polygon with the coordinates whose longitudes are less than the splitting point and another polygon for those with longitudes more than that. Then we merge them together.

For example, this one is a VOLMET map and polygon number 10412. As can be seen, before splitting the map was exactly the opposite of what it was supposed to be.

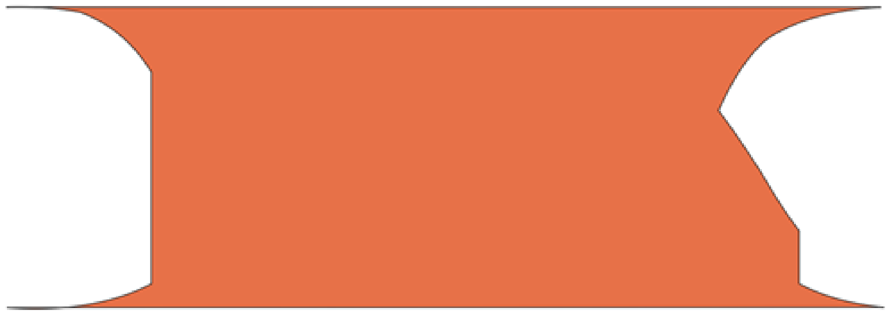


Figure 8. VOLMET map: The result of plotting with a Python script, presenting a splitting issue.

Figure 9. VOLMET map: After solving the splitting issue.

Also, note that the splitting point doesn't have to be a fixed point. It can be anywhere between two polygons.

The way that I defined the splitting Longitude is that I created a new layer in QGIS. Then I created a point between two polygons and then with the identifier tool, I found out its longitude. I create the point by following the steps below and also shown in figure 10, 11, and 12:

- At first we select the "**Layer**" tab
- Then click on "**Create a new layer**"
- Then, "**New shapefile layer**"
- Then, put fields as shown in Figure 11, and click on "**OK**"
- Then, choose "**identify feature**" and click on the point. You can see the coordinates as shown in Figure 12.

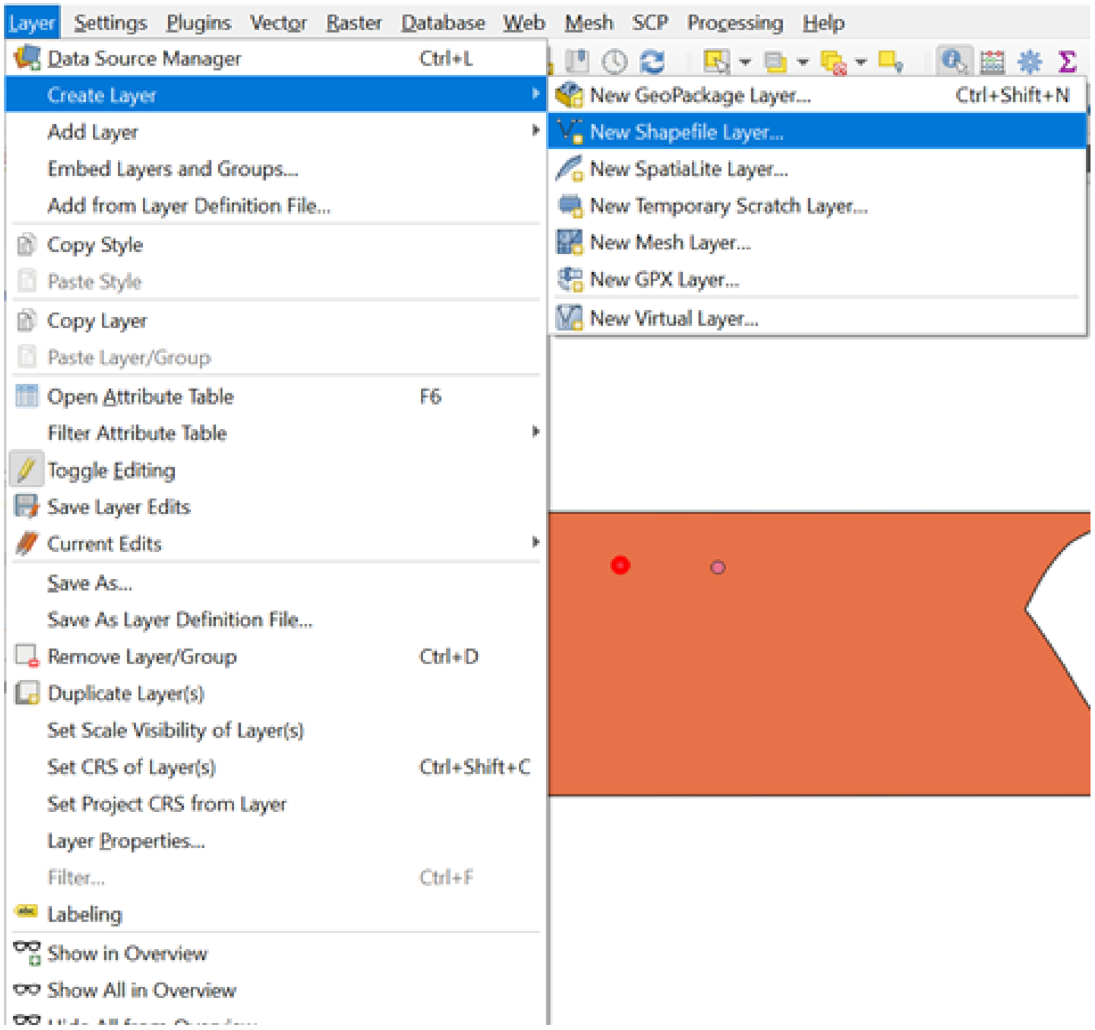
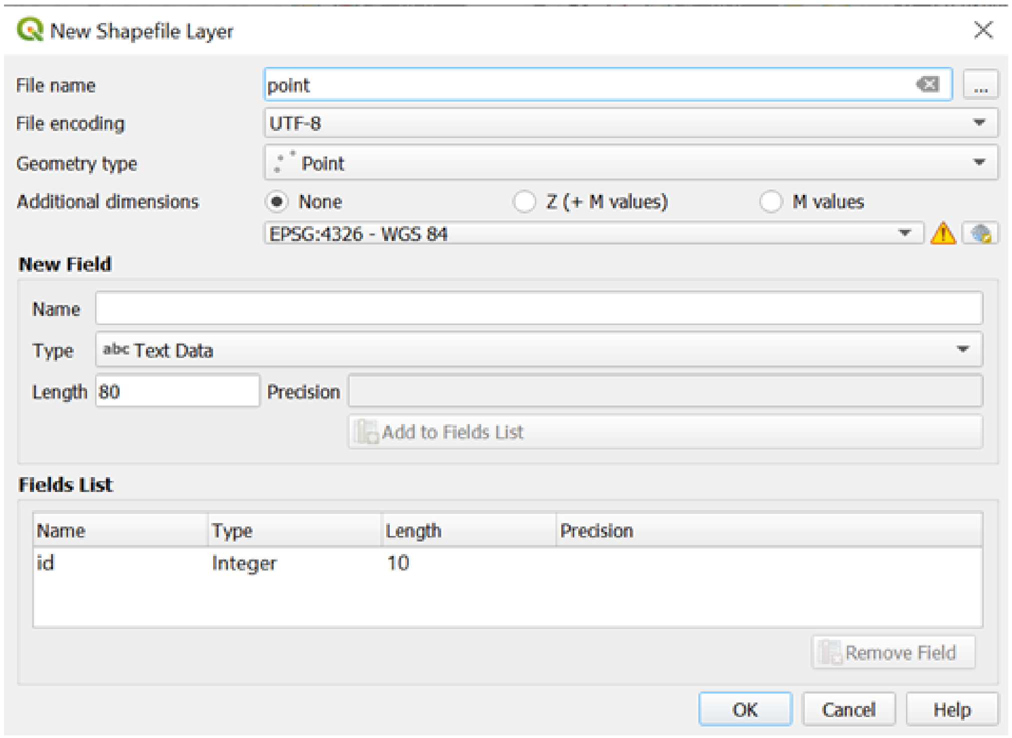


Figure 10. Creating a new layer as a point.



New Shapefile Layer

File name	point
File encoding	UTF-8
Geometry type	Point
Additional dimensions	(● None) ○ Z (+ M values) ○ M values
	EPSG:4326 - WGS 84

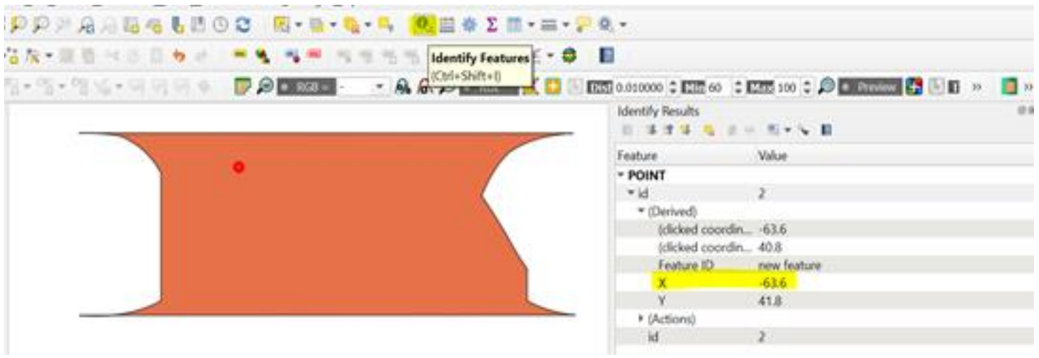
New Field

Name	
Type	abc Text Data
Length	80
Precision	

Fields List

Name	Type	Length	Precision
id	Integer	10	

Figure 11. Creating a point layer.



Feature	Value
POINT	
id	2
(Derived)	
(clicked coordin...	-63.6
(clicked coordin...	40.8
Feature ID	new feature
X	-63.6
Y	41.8
(Actions)	
id	2

Figure 12. Finding the coordinates of a point.

2.2. Raster generation:

2.2.1. Overview

In the field of telecommunications, particularly for satellite and radio communication systems, understanding the atmospheric conditions that affect signal propagation is crucial. Integrated Cloud Liquid Water Content (ICLWC) rasters, derived from flat files containing Latitude, Longitude, pixel size and digital numbers of each pixel, provide detailed spatial representations of the total amount of liquid water in clouds over specific periods. These flat files were converted to rasters using Python.

These Integrated Cloud Liquid Water Content (ICLWC) rasters are essential for identifying and analyzing areas with high cloud water content, which can significantly impact signal attenuation and overall communication quality.

The **annual ICLWC rasters** offer a comprehensive view of the average cloud water content over the course of a year. This long-term data is invaluable for strategic planning and infrastructure development, helping to identify regions where persistent cloud cover may necessitate additional measures to ensure reliable communication services.

The **monthly ICLWC rasters** provide more granular insights into the seasonal variations and short-term fluctuations in cloud liquid water content. These rasters are crucial for operational planning and real-time decision-making, allowing telecommunication providers to anticipate and mitigate potential disruptions caused by monthly changes in atmospheric conditions.

By integrating these ICLWC rasters into telecommunication analysis, service providers can enhance the accuracy of their signal propagation models, optimize network performance, and ensure robust communication links in varying weather conditions. This integration ultimately leads to improved service quality and reliability for end-users.

2.2.2. Data structure:

We have the flat files for Annual and Monthly cloud water content, and for each annual and monthly we have the data with different probabilities of water content and mean and std of them.

The flat files, as mentioned before, contain Latitude and Longitude, pixel size, and digital number (DN) representing the value of the pixels, which need to be converted to rasters for better visualization.:

In the metadata of each data, we also can find a table (Table 2), representing the pixel size, number of columns and rows, and other spatial features that are necessary in raster generation.

Table 1. Map file characteristics

Parameter	Value
Format	ASCII
Upper left corner latitude	−90° N
Latitude increment	+0.25°
Upper left corner longitude	−180° E
Longitude increment	+0.25°
Number of rows	721
Number of columns	1 441
Column separator	Space
Row separator	Windows (CR LF)

The data was available in both monthly and annual periods. For each map, there is a latitude file and a longitude file, representing the latitude and longitude of each pixel, along with a value file containing the digital number for the data. Using all of this information, we created raster files for better visualization and analysis of this data.

2.2.3. Generating GeoTIFF file by Python script:

A Python script named “georeferencingWaterCloud” (annex 3) is used to get the data of cloud liquid water content and to create the Geotiff and georeference them.

GeoTIFF, short for Geographic Tagged Image File Format, is a widely used raster image format that includes geographic metadata and spatial reference information. It extends the TIFF file format to include georeferencing data, enabling geospatial data to be embedded within the image file. This georeferencing information typically includes geographic coordinates, coordinate system details, projection, and other parameters necessary to accurately position and align the image with real-world geographical features. GeoTIFFs are extensively used in GIS (Geographic Information System) applications, remote sensing, cartography, and various other fields where accurate spatial representation of imagery and associated geographic context is crucial.

The Python script undertakes a comprehensive geospatial data processing procedure. Initially, it retrieves latitude and longitude data from designated text files, namely 'Lat.txt' and 'Long.txt'. Following this, the script processes multiple text files containing field strength data associated with distinct geographical coordinates. The acquired data is meticulously organized into structured lists and subjected to calculations to derive new values contingent upon defined longitude ranges. Subsequently, the script effectuates a conversion of the processed data into NumPy arrays, facilitating subsequent analytical operations.

In a subsequent phase, the script systematically determines the geographic extents and resolutions of the dataset. This information serves as a basis for generating GeoTIFF files, a format that encapsulates geographic metadata and facilitates precise alignment with real-world geographic features. Notably, the script orchestrates the categorization of resultant GeoTIFF files into an 'intermediate' directory. Furthermore, it applies compression and optimization strategies, culminating in the creation of a 'final' directory housing the processed GeoTIFF files, accompanied by their respective overviews.

In summation, the script adeptly assimilates, processes, and transmutes geospatial and field strength data into GeoTIFF files, augmenting their applicability in geographic information systems and subsequent analytical endeavors.

The output of each annual and monthly data is a set of statistical maps with different percentages of water could and mean and standard deviation maps.

2.3. Geoserver

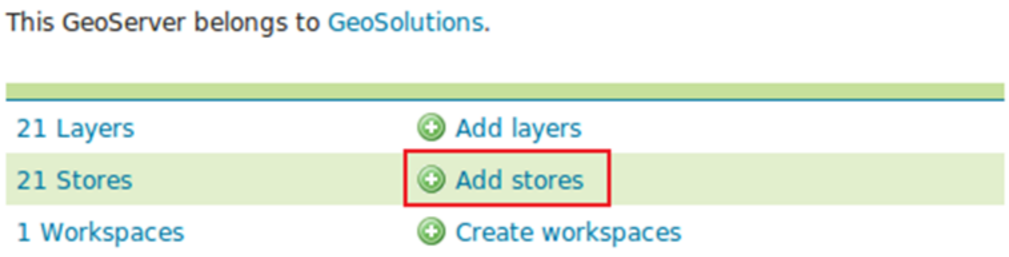
GeoServer is an open-source server software designed to facilitate the sharing, processing, and editing of geospatial data over the Internet. It acts as a platform for serving geospatial information in various formats, such as maps, images, and other location-based data, adhering to standardized protocols like Web Map Service (WMS), Web Feature Service (WFS), and Web Coverage Service (WCS).

With GeoServer, users can publish and manage geospatial data from various sources, including databases, shapefiles, and more. It supports a wide range of data formats and enables users to customize styling, projections, and other aspects of how the data is presented. This makes it a valuable tool for creating interactive web-based maps and applications that require geospatial information.

In summary, GeoServer helps organizations and individuals effectively share and utilize geospatial data on the web by providing a reliable and flexible platform for serving geospatial information in a standardized and accessible manner.

For uploading GeoTIFFs on GeoServer we must follow the following steps:

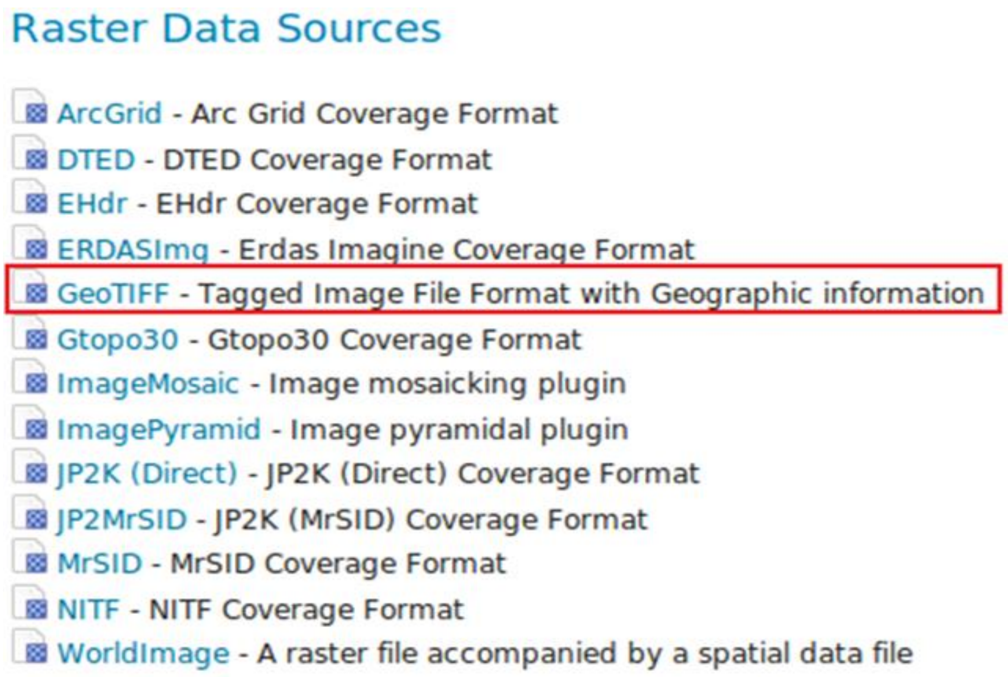
- Open the web browser and navigate to the GeoServer Welcome Page.
- Select Add Stores from the interface.


This GeoServer belongs to GeoSolutions.

21 Layers	Add layers
21 Stores	Add stores
1 Workspaces	Create workspaces

Figure 13. Add stores.

Select GeoTIFF - Tagged Image File Format with Geographic information from the set of available Raster Data Sources.

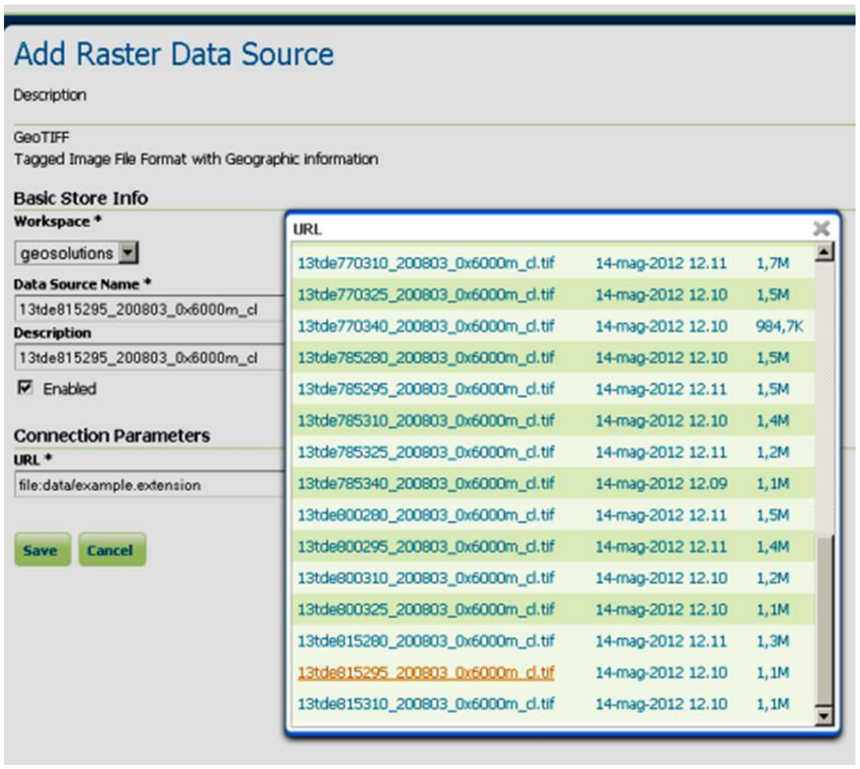

Raster Data Sources

- ArcGrid - Arc Grid Coverage Format
- DTED - DTED Coverage Format
- EHdr - EHdr Coverage Format
- ERDASImg - Erdas Imagine Coverage Format
- GeoTIFF - Tagged Image File Format with Geographic information
- Gtopo30 - Gtopo30 Coverage Format
- ImageMosaic - Image mosaicking plugin
- ImagePyramid - Image pyramidal plugin
- JP2K (Direct) - JP2K (Direct) Coverage Format
- JP2MrSID - JP2K (MrSID) Coverage Format
- MrSID - MrSID Coverage Format
- NITF - NITF Coverage Format
- WorldImage - A raster file accompanied by a spatial data file

Figure 14. Choose Raster Data Sources

Specify a proper name (as an instance, 13tde815295_200803_0x6000m_cl) in the Data Source Name field of the interface.

Click on the browse link to set the GeoTIFF location in the URL field.



Add Raster Data Source

Description

GeoTIFF
Tagged Image File Format with Geographic information

Basic Store Info

Workspace *
geosolutions

Data Source Name *
13tde815295_200803_0x6000m_cl

Description
13tde815295_200803_0x6000m_cl

☑ Enabled

Connection Parameters

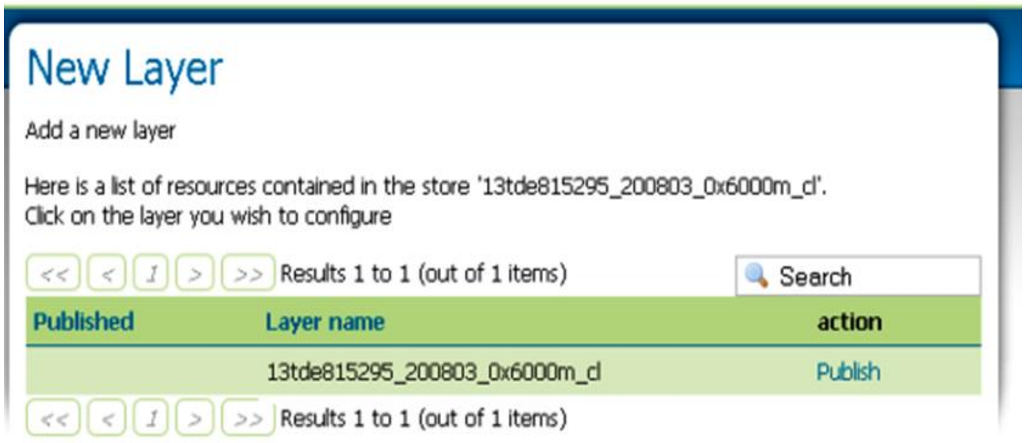
URL *
file:data/example.extension

URL		
13tde770310_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,7M
13tde770325_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,5M
13tde770340_200803_0x6000m_cl.tif	14-mag-2012 12.10	984,7K
13tde785280_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,5M
13tde785295_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,5M
13tde785310_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,4M
13tde785325_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,2M
13tde785340_200803_0x6000m_cl.tif	14-mag-2012 12.09	1,1M
13tde800280_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,5M
13tde800295_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,4M
13tde800310_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,2M
13tde800325_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,1M
13tde815280_200803_0x6000m_cl.tif	14-mag-2012 12.11	1,3M
13tde815295_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,1M
13tde815310_200803_0x6000m_cl.tif	14-mag-2012 12.10	1,1M

Figure 15. Add raster data source

Click Save.

Publish the layer by clicking on the publish link.



New Layer

Add a new layer

Here is a list of resources contained in the store '13tde815295_200803_0x6000m_cl'.
Click on the layer you wish to configure

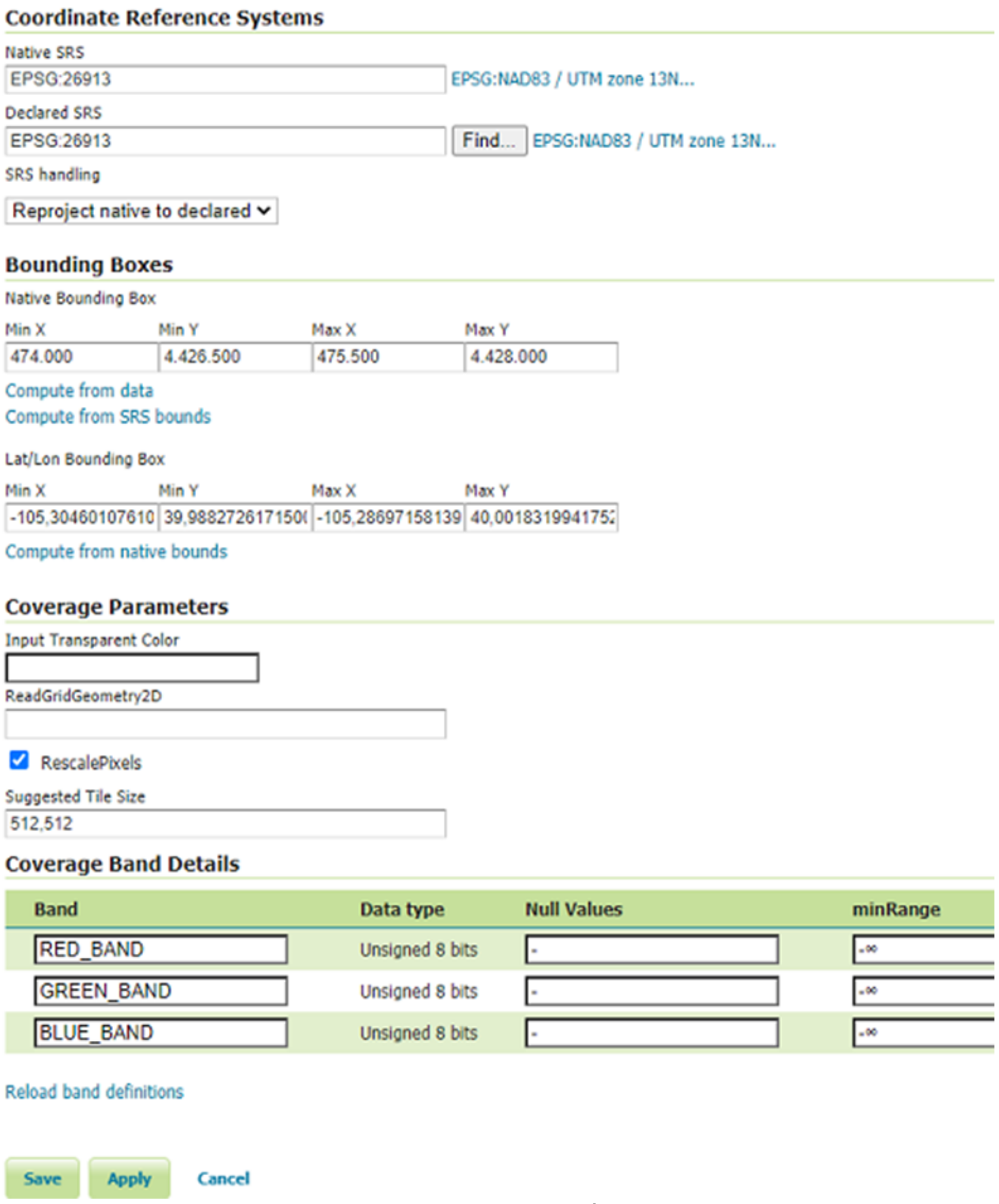
Results 1 to 1 (out of 1 items)

Published	Layer name	action
	13tde815295_200803_0x6000m_cl	Publish

Results 1 to 1 (out of 1 items)

Figure 16. Publish the layer

Check the Coordinate Reference Systems and the Bounding Boxes fields are properly set and click on Save.



Coordinate Reference Systems

Native SRS

EPSG:26913 EPSG:NAD83 / UTM zone 13N...

Declared SRS

EPSG:26913 Find... EPSG:NAD83 / UTM zone 13N...

SRS handling

Reproject native to declared

Bounding Boxes

Native Bounding Box

Min X	Min Y	Max X	Max Y
474.000	4.426.500	475.500	4.428.000

Compute from data

Compute from SRS bounds

Lat/Lon Bounding Box

Min X	Min Y	Max X	Max Y
-105,30460107610	39,988272617150(	-105,28697158139	40,001831994175;

Compute from native bounds

Coverage Parameters

Input Transparent Color

ReadGridGeometry2D

☑ RescalePixels

Suggested Tile Size

512,512

Coverage Band Details

Band	Data type	Null Values	minRange
RED_BAND	Unsigned 8 bits	-	-∞
GREEN_BAND	Unsigned 8 bits	-	-∞
BLUE_BAND	Unsigned 8 bits	-	-∞

Reload band definitions

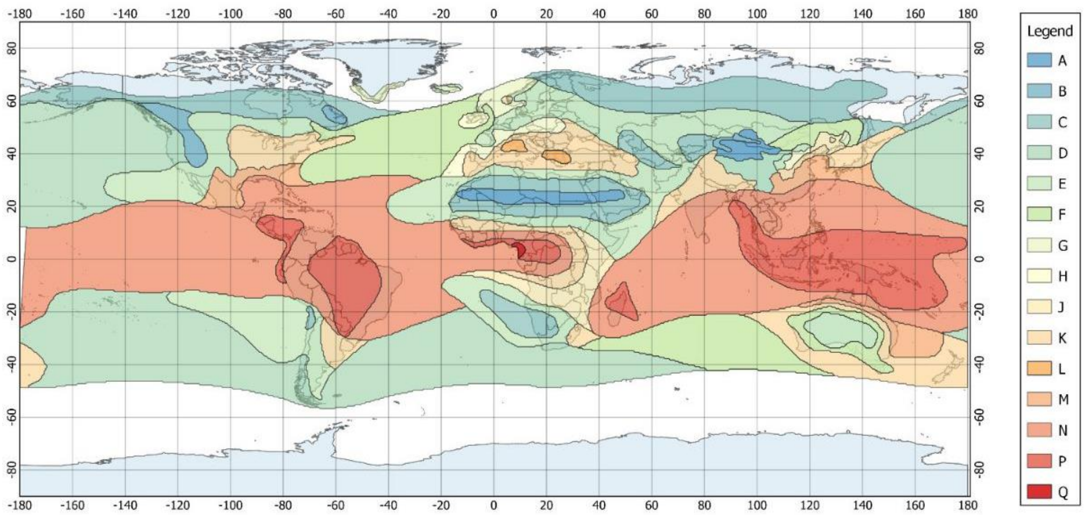
Figure 17. Coordinate Reference Systems

At this point the GeoTIFF is being published with GeoServer. You can use the layer preview to inspect the data.

3. Results:

3.1. Vector results:

Here are the results of generated polygons:



The designations employed and the presentation of material on this map do not imply the expression of any opinion whatsoever on the part of ITU and of the Secretariat of the ITU concerning the legal status of the country, territory, city or area or its authorities, or concerning the delimitation of its frontiers or boundaries.

Figure 18. Rain Zones final map

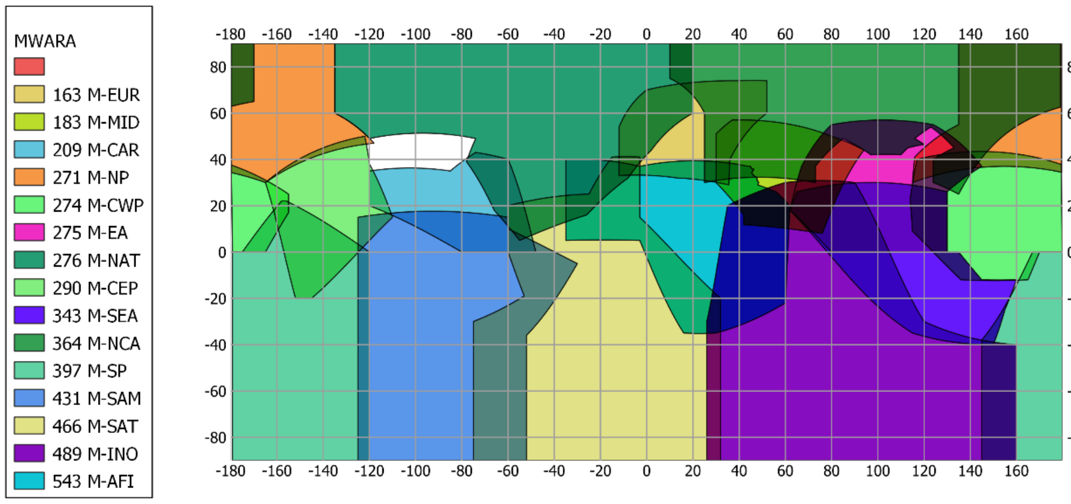


Figure 19. MWARA final map

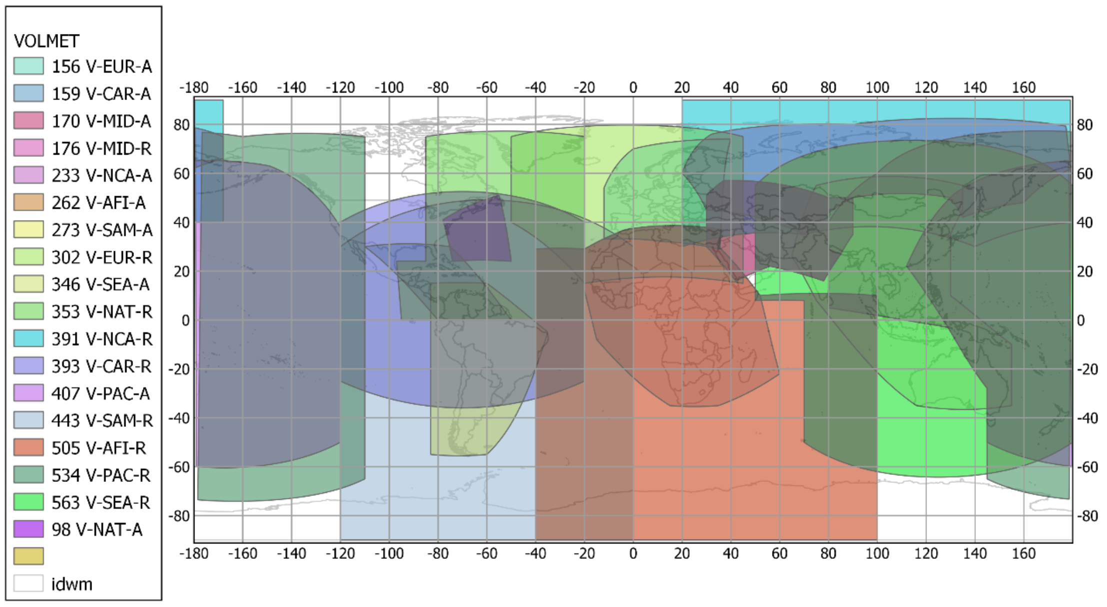
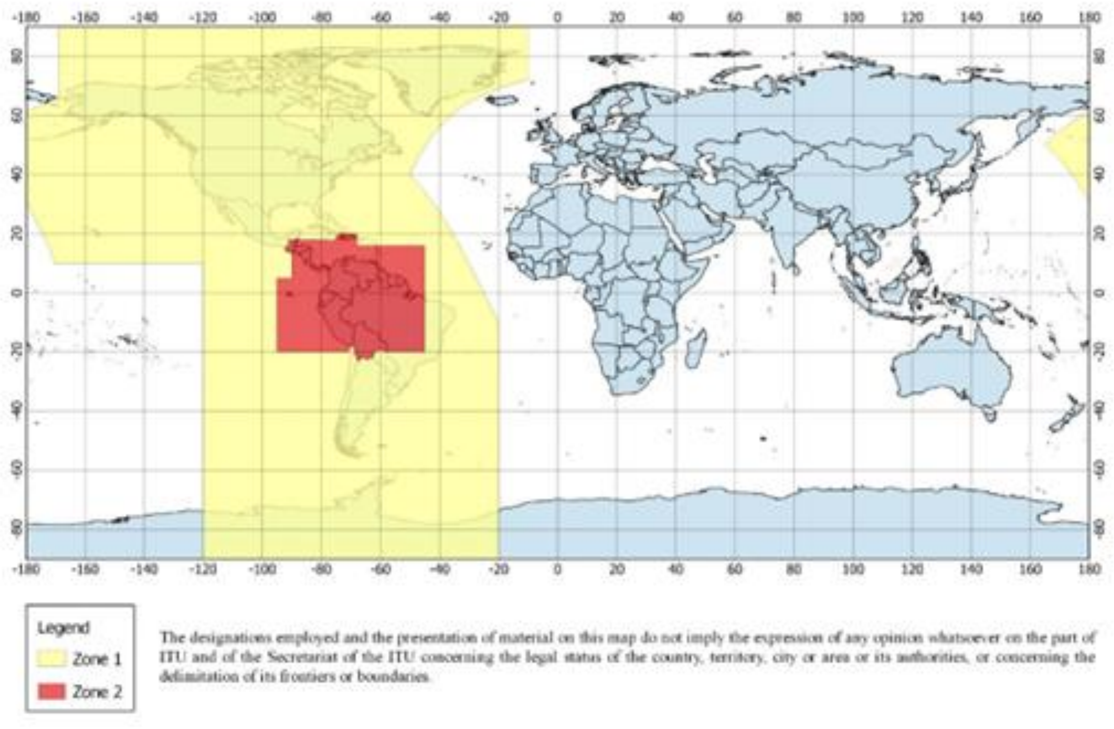


Figure 20. VOLMET final map



The designations employed and the presentation of material on this map do not imply the expression of any opinion whatsoever on the part of ITU and of the Secretariat of the ITU concerning the legal status of the country, territory, city or area or its authorities, or concerning the delimitation of its frontiers or boundaries.

Figure 21. RJ88 NZ Zones final map

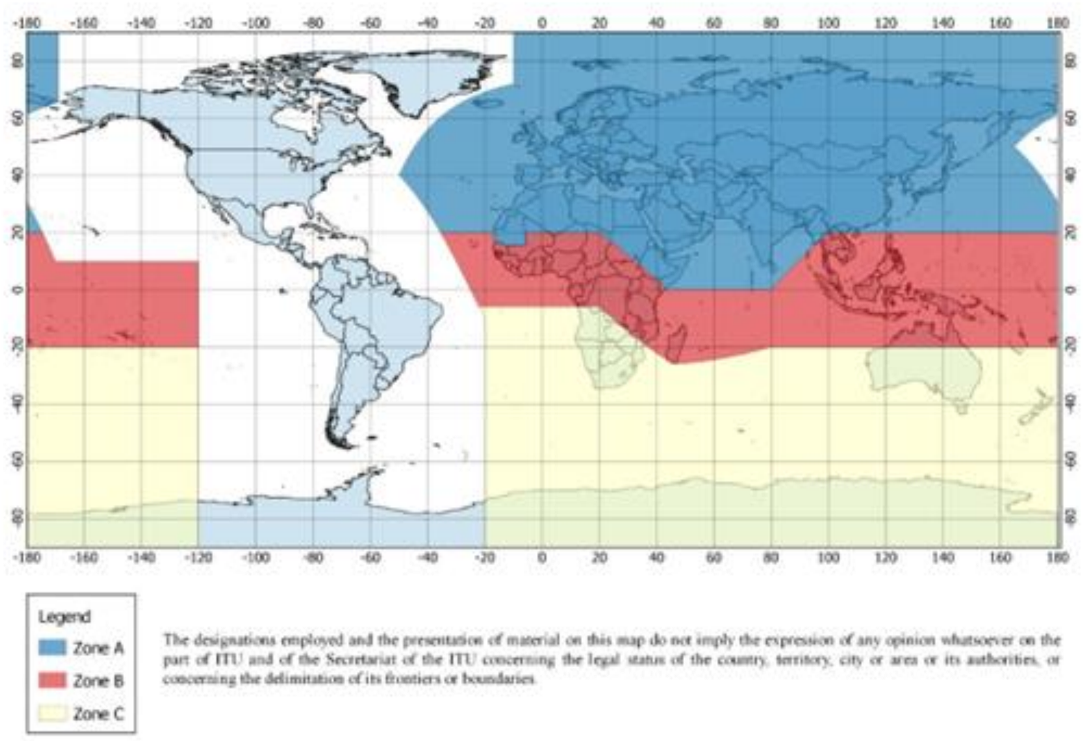

Legend
Zone A
Zone B
Zone C
The designations employed and the presentation of material on this map do not imply the expression of any opinion whatsoever on the part of ITU and of the Secretariat of the ITU concerning the legal status of the country, territory, city or area or its authorities, or concerning the delimitation of its frontiers or boundaries.

Figure 22. GE75 Noise Zones final map

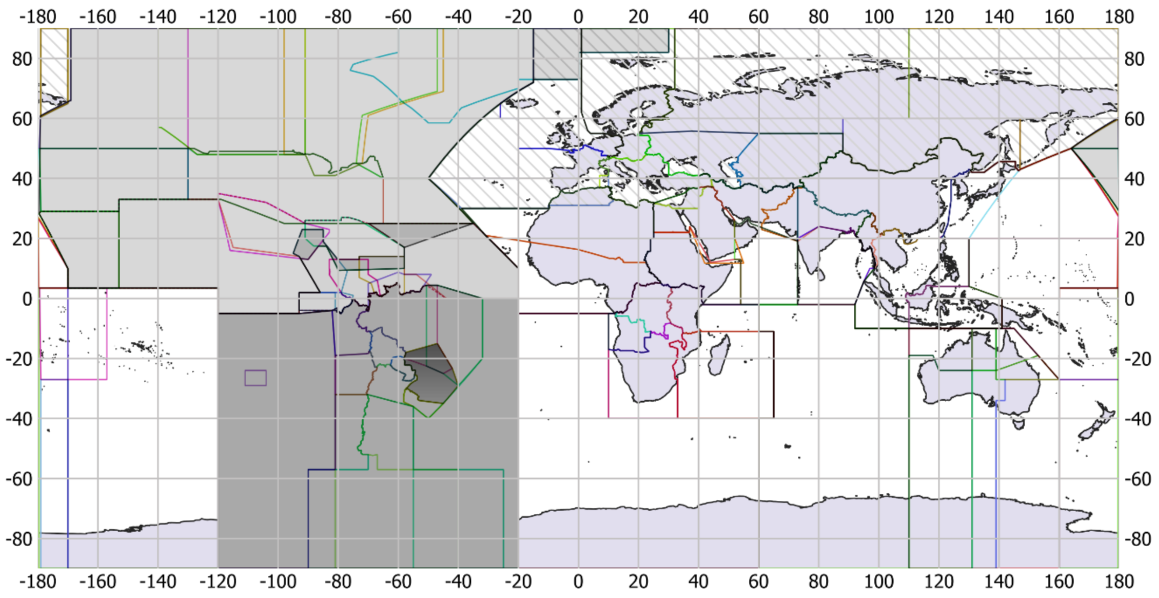


Figure 23. RDARA final map

3.2. Raster results:

As mentioned before, I generated the raster for annual and monthly water content data with different percentages of water content for more than 40 flat files . Here are some samples of the GeoTIFF outputs generated by the Python script.

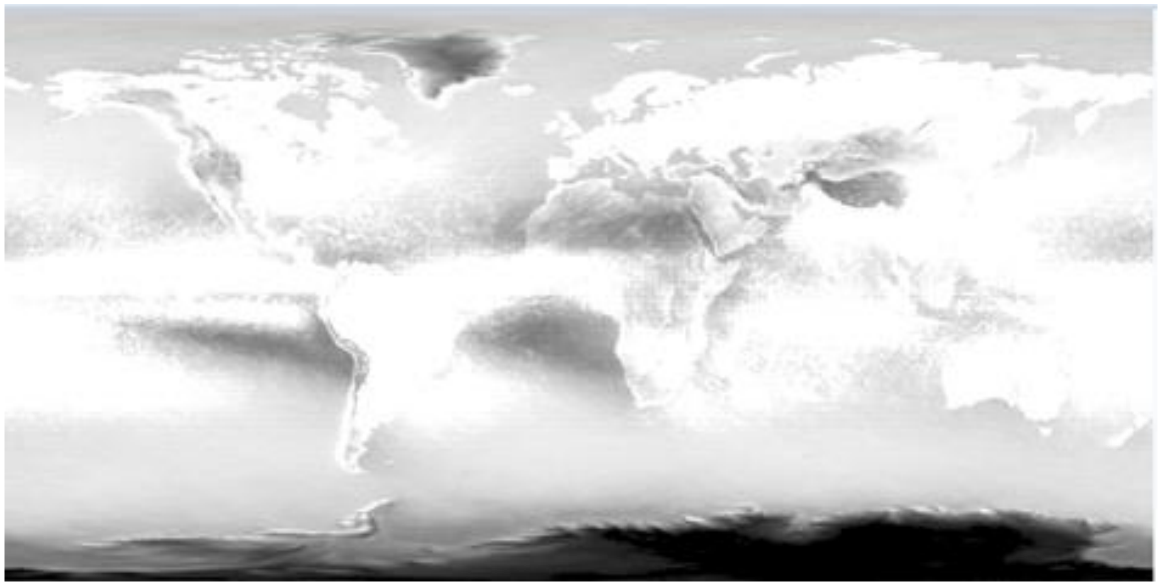


Figure 24. Annual integrated cloud liquid water content with the probability of 0.01 %

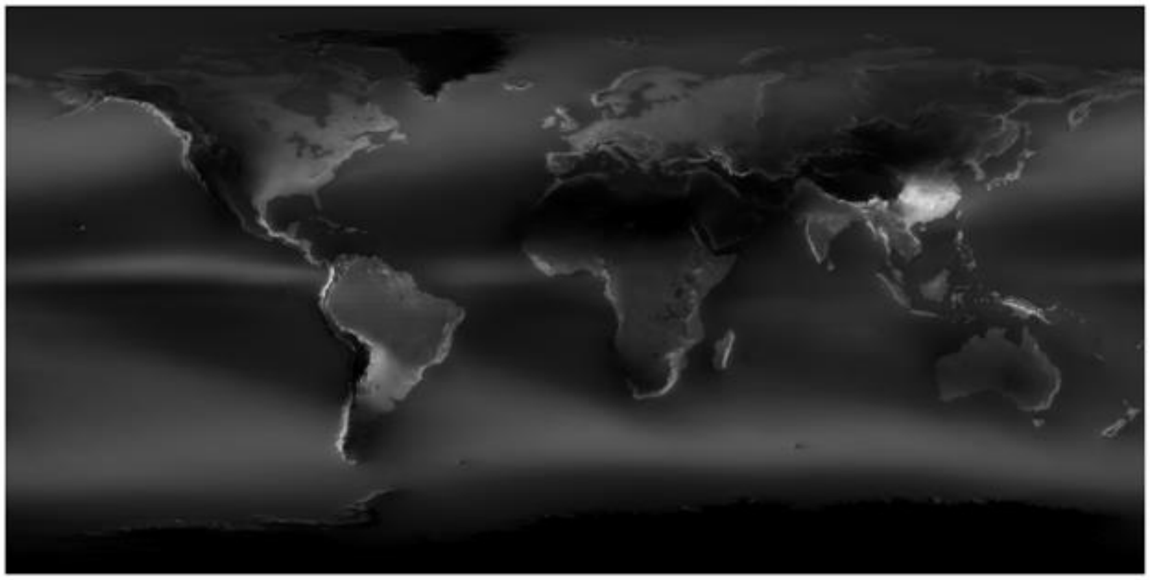


Figure 25. Annual integrated cloud liquid water content with the probability of 5 %

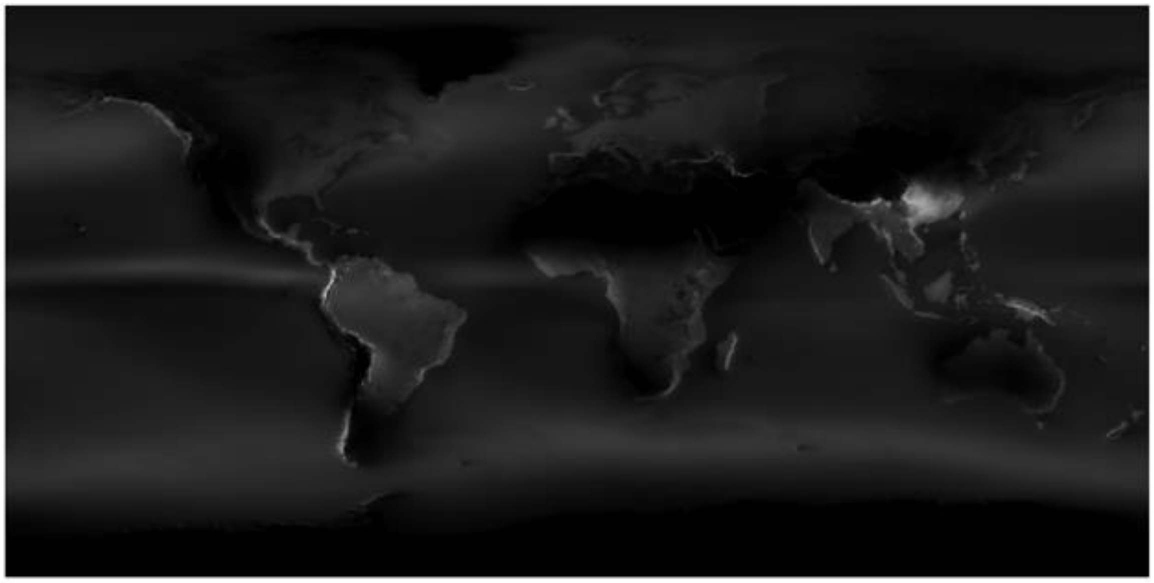


Figure 26. Annual integrated cloud liquid water content with the probability of 20 %

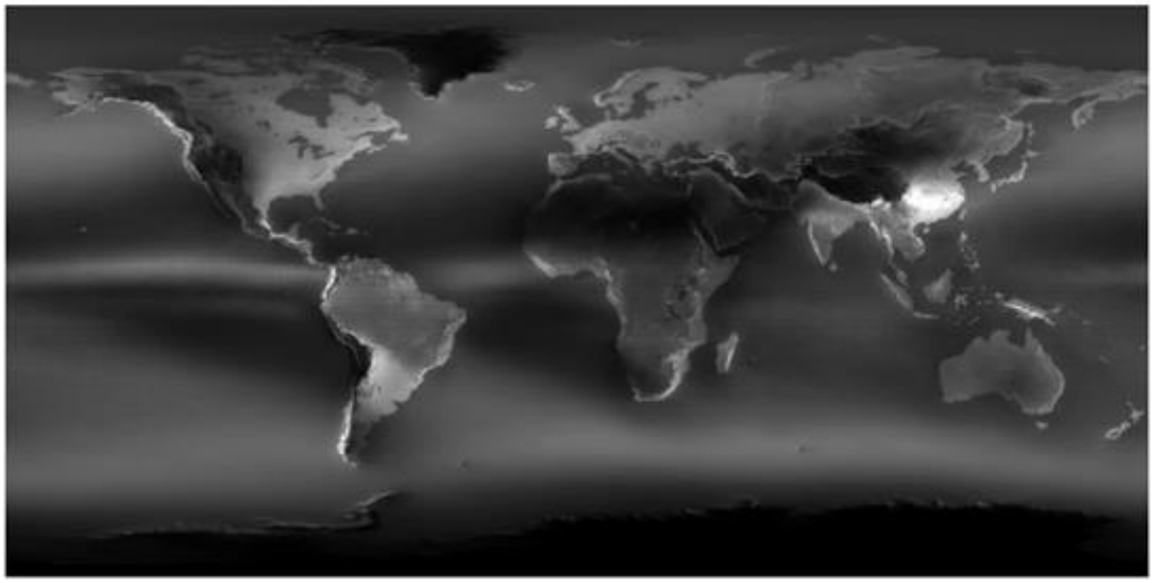


Figure 27. STD of Annual integrated cloud liquid water content

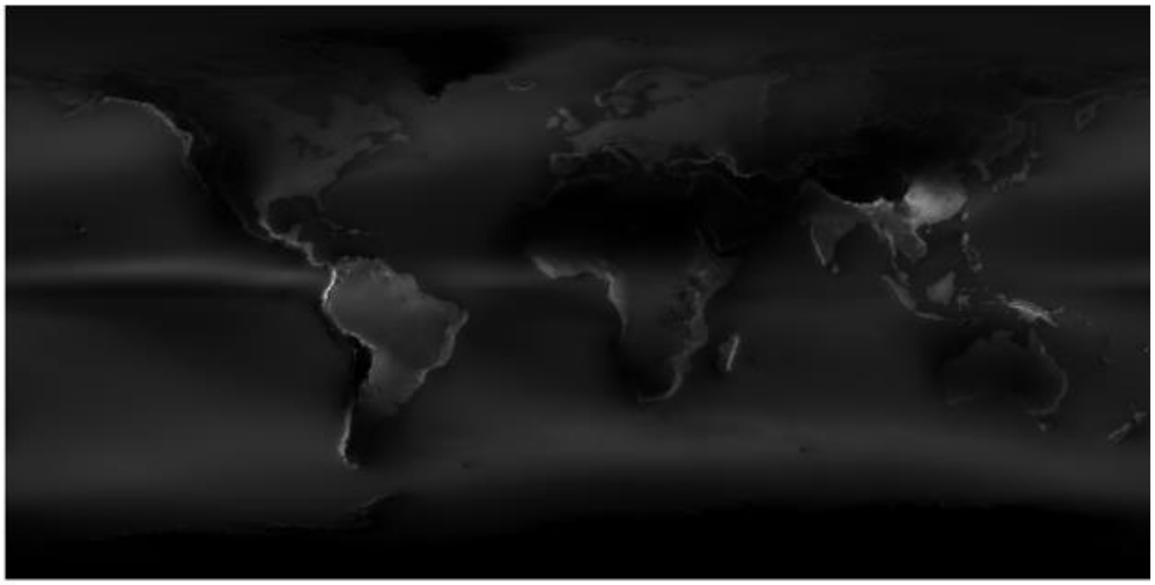


Figure 28. Mean of Annual integrated cloud liquid water content

4. Reference

https://docs.geoserver.geo-solutions.it/edu/en/adding_data/add_geotiff.html

https://www.itu.int/en/ITU-R/information/Pages/default.aspx

https://www.itu.int/en/about/Pages/vision.aspx

https://www.itu.int/pub/R-REC

5. Annex

5.1. Annex 1:

__Section1#__ Preparing text files and converting them to JSON-polygon types, and also assigning the values to each polygon based on their index!

```
import json
from collections import OrderedDict
import os

class GeoPoint:
    def __init__(self, longitude, latitude):
        self.longitude = longitude
        self.latitude = latitude


class Contour:
    def __init__(self):
        self.noPoints = 0
        self.points = []


IDWM_Contours = {}

def parse_coordinates(coordinates):
    latDeg = int(coordinates[0:2])
    latMin = int(coordinates[2:4])
    latSec = int(coordinates[4:6])
    NS = coordinates[6]
    lonDeg = int(coordinates[7:10])
    lonMin = int(coordinates[10:12])
    lonSec = int(coordinates[12:14])
    EW = coordinates[14]

    longitude = lonDeg + lonMin / 60.0 + lonSec / 3600.0

    if EW == "W":
        longitude = -longitude

    latitude = latDeg + latMin / 60.0 + latSec / 3600.0

    if NS == "S":
        latitude = -latitude

    return longitude, latitude

def parse_lines(lines):
    for line in lines:
        trimmed = line.strip()
        parts = trimmed.split()

        if len(parts) == 4:  # index reference 10001 1 5745 0
            contourID = int(parts[0])
            first = int(parts[1])
            last = int(parts[2])
            dummy = int(parts[3])

            if contourID not in IDWM_Contours:
                IDWM_Contours[contourID] = Contour()

            IDWM_Contours[contourID].noPoints = last
        elif len(parts) == 2:  # points
            coordinates, point_no = parts[0], int(parts[1])

            if contourID not in IDWM_Contours:
                IDWM_Contours[contourID] = Contour()

            longitude, latitude = parse_coordinates(coordinates)
            IDWM_Contours[contourID].points.append(GeoPoint(longitude, latitude))


def geo_point_to_dict(geo_point):
    return OrderedDict([("type", "Point"), ("coordinates", [geo_point.longitude, geo_point.latitude])])


def contour_to_json(contour, value):
    geometry = OrderedDict([
        ("type", "Polygon"),
        ("coordinates", [[(point.longitude, point.latitude) for point in contour.points]])
    ])
    feature = OrderedDict([
        ("type", "Feature"),
        ("geometry", geometry),
        ("contour_id", contour_id),
        ("value", value)
    ])

    return json.dumps(feature, indent=2)


def read_values_file(file_path):
    values = {}
    with open(file_path, "r") as file:
        for line in file:
            parts = line.strip().split("\t")
            contour_id = parts [0]
            value = " ".join(parts[1:3])
            values[int(contour_id)] = value
            #print(f"Contour ID: {contour_id}, Value: {value}")
    return values


file_path = ".../LINES.txt"

with open(file_path, "r") as file:
    lines = file.readlines()

parse_lines(lines)

values_file_path = ".../LINESIDX.DAT"
contour_values = read_values_file(values_file_path)

output_directory=".../OUTPUT/JSONs"
for contour_id, contour in IDWM_Contours.items():
    json_data = contour_to_json(contour, contour_values.get(contour_id, 0))  # Assign a default value of 0
if not found in the file
    output_file_path = os.path.join(output_directory,f"contour_{contour_id}.json")

    with open(output_file_path, "w") as output_file:
        output_file.write(json_data)

print("done")
```

__Section2__*:#Convert the json files to one gpkg file*

```
import os
import json
import pandas as pd
import geopandas as gpd
from shapely.geometry import shape

# Create an empty DataFrame to hold the data
data = []

# Set the path to the directory containing the JSON files
json_files_directory = ".../OUTPUT/JSONs"

# Loop through the JSON files and read them into the DataFrame
for file_name in os.listdir(json_files_directory):
    if file_name.endswith(".json"):
        file_path = os.path.join(json_files_directory, file_name)
        with open(file_path, "r") as file:
            json_data = json.load(file)
            data.append(json_data)
```

```
# Convert the list of JSON data to a DataFrame
df = pd.DataFrame(data)

# Convert the DataFrame to a GeoDataFrame by creating a 'geometry' column
geometry = [shape(geom) for geom in df['geometry']]
gdf = gpd.GeoDataFrame(df, geometry=geometry)

# Convert the GeoDataFrame to a GeoPackage file
output_directory = ".../OUTPUT/GPKG"
output_gpkg_file = os.path.join(output_directory, "output_file.gpkg")
gdf.to_file(output_gpkg_file, driver="GPKG")

print("done")
```

Section3:*#Convert the json files to one json file*

```
import os
import json

# Create an empty list to hold the JSON data
json_data_list = []

# Set the path to the directory containing the JSON files
json_files_directory = "OUTPUT/JSONs"

# Loop through the JSON files and read them into the list
for file_name in os.listdir(json_files_directory):
    if file_name.endswith(".json"):
        file_path = os.path.join(json_files_directory, file_name)
        with open(file_path, "r") as file:
            json_data = json.load(file)
            json_data_list.append(json_data)

# Save the list of JSON data as a single JSON file with an array of objects
output_json_file = "output_file.json"
with open(output_json_file, "w") as file:
    json.dump(json_data_list, file, indent=2)
```

#To convert the json files to one Geojson, in order to be readable in QGIS

```
import os
import json
import pandas as pd
import geopandas as gpd
from shapely.geometry import shape

# Create an empty DataFrame to hold the data
data = []
```

```
# Set the path to the directory containing the JSON files
json_files_directory = "…/OUTPUT/JSONs"

# Loop through the JSON files and read them into the DataFrame
for file_name in os.listdir(json_files_directory):
    if file_name.endswith(".json"):
        file_path = os.path.join(json_files_directory, file_name)
        with open(file_path, "r") as file:
            json_data = json.load(file)
            data.append(json_data)

# Convert the list of JSON data to a DataFrame
df = pd.DataFrame(data)

# Convert the DataFrame to a GeoDataFrame by creating a 'geometry' column
geometry = [shape(geom) for geom in df['geometry']]
gdf = gpd.GeoDataFrame(df, geometry=geometry)

# Convert the GeoDataFrame to a GeoJSON file
output_geojson_file = "output_file.geojson"
gdf.to_file(output_geojson_file, driver="GeoJSON")
```

5.2. Annex 2:

```
###split the contours by this code!
import os
import json
from collections import OrderedDict

class GeoPoint:
    def __init__(self, longitude, latitude):
        self.longitude = longitude
        self.latitude = latitude


class Contour:
    def __init__(self):
        self.noPoints = 0
        self.points = []

def separate_polygon(points):
    l1 = []
    l2 = []

    for gp in points:
        if gp.longitude < -47: #put the splitting Longitude here
            l1.append(gp)
        else:
            l2.append(gp)

    #print("Avg Longitude:", avg_longitude)
    print("Points in L1:", len(l1))
    print("Points in L2:", len(l2))

    return l1, l2

def close_contour_if_needed(points):

    if len (points) >= 2 and points[0] != points[-1]:
        points.append(points[0])

IDWM_Contours = {}

def parse_coordinates(coordinates, contourID):
    latDeg = int(coordinates[0:2])
    latMin = int(coordinates[2:4])
    latSec = int(coordinates[4:6])
    NS = coordinates[6]
    lonDeg = int(coordinates[7:10])
    lonMin = int(coordinates[10:12])
    lonSec = int(coordinates[12:14])
    EW = coordinates[14]

    longitude = lonDeg + lonMin / 60.0 + lonSec / 3600.0

    if EW == "W":
        longitude= -longitude

    latitude = latDeg + latMin / 60.0 + latSec / 3600.0

    if NS == "S":
        latitude = -latitude

    return longitude, latitude

def parse_lines(lines):
    contourID = None  # Initialize contourID outside the loop
    for line in lines:
        trimmed = line.strip()
        parts = trimmed.split()

        if len(parts) == 4:
            contourID = int(parts[0])
            if contourID not in IDWM_Contours:
```

```
            IDWM_Contours[contourID] = Contour()
        IDWM_Contours[contourID].noPoints = int(parts[2])
    elif len(parts) == 2 and contourID is not None:
        coordinates, point_no = parts[0], int(parts[1])
        longitude, latitude = parse_coordinates(coordinates, contourID)
        IDWM_Contours[contourID].points.append(GeoPoint(longitude, latitude))

def geo_point_to_dict(geo_point):
    return OrderedDict([("type", "Point"), ("coordinates", [geo_point.longitude, geo_point.latitude])])


def contour_to_json(contour, value):
    geometry = OrderedDict([
        ("type", "Polygon"),
        ("coordinates", [[(point.longitude, point.latitude) for point in contour.points]])
    ])
    feature = OrderedDict([
        ("type", "Feature"),
        ("geometry", geometry),
        (("value", value)),
        ("contour",int(contour_id))
    ])
    return json.dumps(feature, indent=2)

#when we have two value columns
def read_values_file(file_path):
    values = {}
    with open(file_path, "r") as file:
        for line in file:
            parts = line.strip().split("\t")
            contour_id = parts [0]
            value = " ".join(parts[1:3])
            values[int(contour_id)] = value
            #print(f"Contour ID: {contour_id}, Value: {value}")
    return values

file_path = ".../LINES.txt"

with open(file_path, "r") as file:
    lines = file.readlines()

parse_lines(lines)

values_file_path = ".../LINESIDX.txt"
contour_values = read_values_file(values_file_path)

for contour_id in [10413,10412,10411,10410]: #ID of troubled contours
```

```
    if contour_id in IDWM_Contours:
        l1, l2 = separate_polygon(IDWM_Contours[contour_id].points)
        IDWM_Contours[contour_id].points = l1
        if len(l2) > 0:
            new_contour_id = contour_id + 1000
            new_contour = Contour()  # Create a new contour instance
            new_contour.points = l2
            IDWM_Contours[new_contour_id] = new_contour  # Add the new contour to the dictionary
            #print("New Contour ID:", new_contour_id)
            #print("Number of Points in New Contour:", len(new_contour.points))

        close_contour_if_needed(IDWM_Contours[contour_id].points)
        #print("Contour", contour_id, "Points after closure:", [gp.longitude for gp in
IDWM_Contours[contour_id].points])

output_directory="…/OUTPUT"
for contour_id, contour in IDWM_Contours.items():
    json_data = contour_to_json(contour, contour_values.get(contour_id, 0))  # Assign a default value of 0
if not found in the file
    output_file_path = os.path.join(output_directory,f"contour_{contour_id}.json")

    with open(output_file_path, "w") as output_file:
        output_file.write(json_data)
```

5.3. Annex 3:

```
import sys
import numpy as np
import os
import glob
from osgeo import gdal, gdal_array, osr

drc = os.getcwd()

l = open(os.path.join(drc, 'Lat.txt'), "r")
f = open(os.path.join(drc, 'Long.txt'), "r")

lon = f.read().split('\n')
lat = l.read().split('\n')

while lon[-1] == '':
    lon.remove(lon[-1])
while lat[-1] == '':
    lat.remove(lat[-1]) # open read, divide by line and remove possible blank line for lat and lon
for i in range(0,len(lon)):
    lon[i]=lon[i].split()
    lat[i]=lat[i].split()
    lon[i].pop(-1) # remove the last value of lat and lon (the 360, equivalent to the 0)
    lat[i].pop(-1)


for subdir in glob.glob(os.path.join(drc, '*')):
    for subdir2 in glob.glob(os.path.join(subdir, '*')):
        if os.path.isdir(subdir2):
            for p in glob.glob(os.path.join(subdir2, '*.txt')):

                d = open (p,"r")
                data = d.read().split('\n') # split the data with all the "break line"
                while data[-1] == '': # remove the last lines of txt file if they are empty
                    data.remove(data[-1])
                for i in range(0,len(lon)): # separate the lines individually
                    data[i]=data[i].split()
                    data[i].pop(-1)



                a = int(float(len(lon[0]))/2)
                A=[0]*len(lon) # creates an empty list "A" with len(lon) rows, frist half of values (0 - 180)
                Alon=[0]*len(lon)
                B=[0]*len(lon) # creates an empty list "B" with len(lon) rows, for the second half of values (1
80-360)
                Blon=[0]*len(lon)
                BA=[0]*len(data) # Creates a new list for the new list, row = number of rows in data
                BAlon=[0]*len(data)
                BAlat = lat

                for i in range(0, len(lon)): # for each one of this row
                    A[i] = Alon[i] = [0]*len(lon[0][:(a+1)]) # creates columns up to the value "180" (a + 1 be
cause it neglicted the last value of a otherwise)
                    A[i] = data[i][:(a+1)] #fill the new list (half of the lon) with the fieldstrenghs values
                    Alon[i] = lon[i][:(a+1)]
                    B[i] = Blon[i] = [0]*len(lon[0][a:])# creates columns from "180" up to the end (360)
                    B[i] = data[i][a:] # fill the new list with (other half of the lon) with the fieldstrenghs values
                    Blon[i] = lon[i][a:]
                    for j in range (0, len(Blon[i])):
                        Blon[i][j] = float(Blon[i][j]) - 360 # transform B values in negative
                    BA[i] = BAlon[i] = [0]*len(lon[i])
                    BA[i] = B[i] + A[i] # fill the new list with the values of B list followed by A list
                    BAlon[i] = Blon[i] + Alon[i]

                arr = np.array(BA) #change de type of datas from list to np.array
                latmid = np.array(BAlat)
                lonmid = np.array(BAlon)
                array = arr.astype(np.float) # change the type of datas from str to float
                latfin = latmid.astype(np.float)
                lonfin = lonmid.astype(np.float)
                xmin,ymin,xmax,ymax = [lonfin.min(),latfin.min(),lonfin.max(),latfin.max()]
##     # find the min and max for lon and lat
```

```
                nrows,ncols = np.shape(array)
                xres = ((xmax - xmin)/float(ncols-1)) # measure the resolution of x axis. the -1 is for the -180
added
                yres = ((ymax - ymin)/ float(nrows-1))

                geotransform =((xmin - (xres/2)),xres,0,(ymax + (yres/2)) ,0, -yres) # neither is taken in acou
nt (values usually -180, 90)
                base = os.path.splitext(p)[0]
                base = base + '.tif' # create the name of the futur tif file
                GeoTIFF_interm = os.path.join(drc, 'GeoTIFF_interm')
##                 print(base, xres)
                if not os.path.exists(GeoTIFF_interm):
                    os.makedirs(GeoTIFF_interm)
                output_raster = gdal.GetDriverByName('GTiff').Create(os.path.join(GeoTIFF_interm, base),
ncols, nrows, 1, gdal.GDT_Float32)
                srs = osr.SpatialReference()
                srs.ImportFromProj4('+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs')
                #print("Spatial Reference: ", srs.ExportToWkt())
                output_raster.SetProjection(srs.ExportToWkt())
                #print("Projection Set: ", output_raster.GetProjection())  # Print the projection
                output_raster.SetGeoTransform(geotransform)
                output_raster.SetMetadataItem("AREA_OR_POINT",'Point')
                output_raster.GetRasterBand(1).WriteArray(array)   # Writes my array to the raster
                output_raster.GetRasterBand(1).SetNoDataValue(-9999)  # Replace -9999 with your desired
NoData value
                output_raster.FlushCache()
                output_raster = None

for subdir in glob.glob(drc + '\*'):
    for subdir2 in glob.glob(subdir + '\\*'):
        if os.path.isdir(subdir2 + '\\GeoTIFF_interm') == True:
            os.chdir(subdir2)
            GeoTIFF_final = 'GeoTIFF_final'
            if not os.path.exists(GeoTIFF_final):
                os.makedirs(GeoTIFF_final)
            os.chdir(subdir2 +'\\GeoTIFF_interm')
            for p in glob.glob ('*.tif'):
                os.chdir(subdir2 +'\\GeoTIFF_interm')
                base = os.path.splitext(p)[0]
                gtif = gdal.Open(p)
                inf = gdal.Info(gtif)
                os.chdir(subdir2 + '\\GeoTIFF_final')
                translateoptions = gdal.TranslateOptions(gdal.ParseCommandLine("-of Gtiff -co TILED=YE
S -co BLOCKXSIZE=512 -co BLOCKYSIZE=512 -co COMPRESS=DEFLATE"))
                gdal.Translate(base + "_f.tif", gtif, options = translateoptions)

            for p in glob.glob('*.tif'):
                gtif = gdal.Open(p,1)
                inf = gdal.Info(p)
                gdal.SetConfigOption('COMPRESS_OVERVIEW', 'DEFLATE')
                gdal.SetConfigOption('GDAL_TIFF_OVR_BLOCKSIZE', '256')
```

```
gtif.BuildOverviews("NEAREST", [2,4,8,16,32])
inf = gdal.Info(p)
print (inf)
```