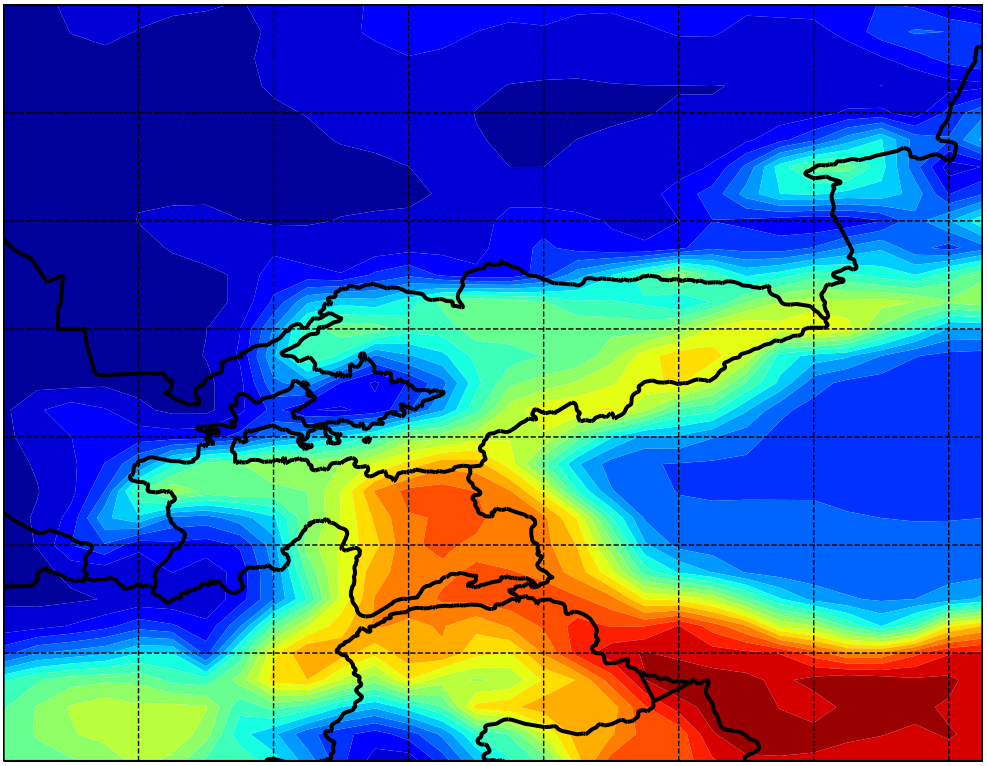
American University of Central Asia

Internship Report RS RAS



Place of Internship

Research Station

of the Russian Academy of Sciences, Bishkek

Program

Applied Mathematics and Informatics

Student

Moritz M. Konarski

Group

2017

November 14, 2020 – Bishkek

Contents

1 Introduction

This report covers the tasks and results of my internship at the Federal State Budgetary Institution of Science Research Station of the Russian Academy of Sciences in Bishkek (RS RAS). RS RAS is a subsidiary of the Ministry of Science and Higher Education of the Russia Federation and employs 137 people. Since 1978 it has been researching seismic processes and developing geodynamic models. The internship took place from the 7th of September 2020 to the 7th of November 2020 and was conducted remotely due to the continuing COVID-19 pandemic. My AUCA supervisor for this internship was Olga Zabinyakova, Scientific Secretary of RS RAS and my RS RAS supervisor was Sanzhar Imashev, Acting Head of the Laboratory for Integrated Research of Geodynamic Processes in Geophysical Fields.

The internship was split into an educational section and an industrial section. According to my internship dairy form, the aim of the educational internship was to acquire the knowledge necessary to understand the research carried out by RS RAS. During the industrial part of the internship I should participate in a certain part of their work or in work that is similar to theirs. To fulfill these requirements my RA RAS supervisor gave me the following tasks for my educational internship:

1. familiarize yourself with web resources providing access to NASA Earth Remote Sensing data;
2. familiarize yourself with the scientific data format netCDF (Network Common Data Form);
3. study libraries used to work with the netCDF format in various computing environments.

For the industrial internship I was tasked to:

1. register on the NASA Earthdata platform to access satellite data;
2. develop a library for working with netCDF files in the Python programming language (using satellite data as an example);
3. develop a computer application for data visualization and reanalysis of NASA MERRA2 satellite data.

These tasks are outlined in my internship diary form. In the next section of this report I will cover the educational part of the internship and talk about NASA Earth Remote Sensing data, the netCDF data format, and libraries used to work with netCDF files.

Then, I will detail the industrial part of the internship and talk about registering on the NASA Earthdata platform, downloading NASA MERRA2 data, and the development of the Python application. The Python application development will be split into multiple sections. As part of the industrial internship section the development process will be described. In a separate section the finished program with all of its components will be discussed.

As a practical example of using the program, I will follow the suggestion of my supervisor and analyze the vertical structure of the air temperature (variable `T`) to show its vertical gradient, the stratopause, and the behavior around the ozone layer.

2 Educational Internship

2.1 NASA Remote Sensing Data

NASA Remote Sensing data is available via the Earth Science Data Systems (ESDS) Program (see here). The program covers the data acquisition, processing, and distribution with the goal to enable the widespread use of NASA mission data. The data is available for free and their software is publicly available as Open Source Software.

Part of ESDS is the NASA Goddard Earth Sciences (GES) Data and Information Services Center (DISC) which provides data on atmospheric composition, water and energy cycles, and climate variability. The GES DISC provides over 3.3 Petabytes of data, including the MERRA2 dataset. MERRA2 (the Modern-Era Retrospective analysis for Research and Applications version 2) focuses on historical climate reanalysis using satellite data. The dataset I worked with (M2I3NPASM) contains data from January 1st, 1980 until October 1st, 2020 (at the time of writing). It covers the whole globe with measurements taken every 3 hours. M2I3NPASM includes 14 measured variables in addition to latitude, longitude, time, and a pressure level. The measured variables include the surface pressure, specific humidity, eastward and northward wind, and temperature. The measurements are done on a cube sphere grid (add explanation) and later processed to fit the standard latitude and longitude grid. This processing generally involved a bilinear interpolation of data values.

The data is provided on the GES DISC website (see here) and can be accessed from there. The website gives the option to download only a subset of the data by selecting a certain time range, latitude and longitude range, and group of variables. This is advantageous because a full file for 1 day is about 1.1 GB in size. For my internship I worked with a subset of the data that includes all variables but is restricted to a latitude of 34°N to 48°N and a longitude of 65°E to 83°E. This restricts the area to Kyrgyzstan and sections of all surrounding countries. Additionally, the file size is reduced to a manageable 6 MB per file so that a whole year of data only takes up 2.2 GB (in 365 files), the same space two files of the complete data would take up.

2.2 The netCDF Data Format

The M2I3NPASM data is provided in a file format called the Network Common Data Form (netCDF). NetCDF is made up of a data format and libraries that can read and

write its data. NetCDF is developed and maintained by Unidata, a community of research institutions, with the goal of sharing geoscience data and the tools to use and visualize it. Unidata is funded by the National Science Foundation, a US government agency that promotes and supports science and research. Unidata also maintains libraries (programming interfaces) for C, Java, and Fortran. Based on these, multiple other interfaces are available, including one for Python.

The netCDF data format is specifically designed to hold scientific data. According to the Unidata website, the netCDF data format has the following features:

- **Self-Describing**. A netCDF file includes information about the data it contains.
- **Portable**. A netCDF file can be accessed by computers with different ways of storing integers, characters, and floating-point numbers.
- **Scalable**. Small subsets of large datasets in various formats may be accessed efficiently through netCDF interfaces, even from remote servers.
- **Appendable**. Data may be appended to a properly structured netCDF file without copying the dataset or redefining its structure.
- **Sharable**. One writer and multiple readers may simultaneously access the same netCDF file.
- **Archivable**. Access to all earlier forms of netCDF data will be supported by current and future versions of the software.

The NASA M2I3NPASM data is available in "classic" netCDF–4 format, meaning that it is backwards compatible. These files have 4 dimensions:

1. longitude in degrees east (meaning west is represented as negative),
2. latitude in degrees north (making south negative),
3. pressure in hPa,
4. time in minutes since the first time point in a file.

To be self-describing, the NASA M2I3NPASM netCDF files contain information about, among others, the intitution that created the file, date and time of the beginning and end of the dataset, and the minimum and maximum latitude and longitude values. The files furthermore contain metadata for each of the measured variables. The most important of these are the fill values that identify missing data, the long name, a full version of the short variable name abbreviation, and the units of the variable.

2.3 NetCDF Libraries

Because I am working with the Python programming language for my internship I require a netCDF library that works in that programming language. Unidata provides an interface between the netCDF library for the C programming language and Python. This interface is called netCDF4. It has most of the features of the C library and enables the creation and reading of netCDF files using Python. This interface is used

in my industrial internship work to work with the netCDF files downloaded from GES DISC.

3 Industrial Internship

3.1 NASA Earthdata

3.1.1 Registration

To access data on the NASA Earthdata platform (which includes GES DISC), a registered account is required. The purpose of the registration is for NASA to improve their service and to offer notifications and saved preferences. The Earthdata account then needs to be linked to a GES DISC account to access the M2I3NPASM dataset I am working with. A full guide can be found under this url.

3.1.2 Downloading

To actually download GES DISC Data there are many options, but `wget` might be preferable. `wget` is a utility for downloading files from the internet. It is generally used as a command line tool and it is available for most platforms. The steps necessary to set up wget to download data from GES DISC are outlined here. The process simply involves setting up your login information in a local file. Then, one needs to acquire the urls that point to the data that is meant to be downloaded. As described above in the section on NASA remote sensing data, a subset of the dataset (e.g. M2I3NPASM) should be specified to significantly reduce the download time. Then the GES DISC website will provide one or more download links, generally in a `txt` file. This file can then be given to `wget` (following the platform-specific instructions) to download these files. Once the files have been downloaded, they are ready to be analyzed.

3.2 Python Application Development

This subsection is based on the notes in my internship diary and will explain the process of developing the Python application. It will not go into great technical detail because I want to reserve that for the description of the finished program. Additionally, every program written during my internship contributed to the final program and thus when I describe the final result I will be indirectly describing the important results obtained along the way. The decision to work with Python was made in our first meeting and we chose it because everyone had had at least some experience with it and Python has a rich ecosystem of libraries that would enable us to complete all the parts of our assignment.

3.2.1 Simple Beginnings – 10.09. to 20.09.

The first step was to download the netCDF data from the GES DISC website to start working with it. I created the required account (as outlined above) and downloaded 3 days worth of data of the M2I3NPASM dataset using `wget`. Then, I develped a simple command line python program that enabled me to more comfortably download large quantities of files. This program used the Python requests library to download the files, and not `wget`. This program has since been retired because `wget` is completely sufficient and there is no need for this program.

The next simple command line program I wrote listed all the available variables in a netCDF file. This was the first time I worked with these files and thus I had to find out how it works. I chose the `netCDF4` Python library for this task because it is developed by the same group that created the file format itself. I also had to find out how to access the variable names that are stored in a netCDF file.

Once I had familiarized myself with the file format, I wrote a simple program capable of creating a heat map graph of a data type that only has latitude and longitude dimensions. This program was not flexible and most values were hard-coded, but I forced me to explore how to plot two–dimensional heat maps using Python. I used the `matplotlib` library to create the plot itself and to save it as a picture. To create the map features I used the `cartopy` library which specializes in creating all kinds of maps.

3.2.2 Figuring out Data Storage – 22.09. to 01.10.

Because M2I3NPASM data comes in 1 file per day it can be cumbersome to work with data that covers more than a single day (many files would have to managed at once). Furthermore, each file includes multiple variables (because one generally does not know which specific variable will be needed) which means that there is unnecessary data once one decides to analyze a single variable. To solve both of these issues, my supervisor suggested that I extract one particular variable from multiple netCDF files and save all the data into a single file. Now a file format for these files needed to be found.

The data in netCDF files is stored in multi-dimensional arrays because this structures resembles the structure of the data most closely. Thus our data format should also support multi-dimensional arrays. After considering multiple alternative options (Parquet, HDF5, netCDF), with my supervisor's advice I decided on NPZ files, a format for the `numpy` Python library. This format is convenient because `numpy` is one of the most popular scientific python computing libraries and used as a backend for many other libraries meaning that it is widely usable. Also, NPZ files natively support multi-dimensional arrays which makes them a good fit for our data. The compression of NPZ files is another advantage because it saves space when the file becomes large. NPZ files were chosen over NPY files, which are also files used by the `numpy` library,

because NPY files are not compressed and thus take up more space. NPY files also only hold one single multi-dimensional array while NPZ files can hold multiple arrays. This enables me to store all the necessary data in a single file (the actual data plus data for the dimensions time, latitude, longitude, level).

Unfortunately NPZ files are not self-describing and cannot hold information about the data that they contain. To not lose the information about our data that the netCDF files hold, another file type to store this information was required. For this purpose I chose JSON files, which can easily be read from and written to using Python and most other programming languages. Conveniently, the Python dictionary data type – a list if key-value pairs, e.g. `"age": 21` where `"age"` is the key and `21` is the value – can be easily converted to JSON and stored for later use. This is the approach I decided to implement.

After choosing these files types, I started to develop a program that would take every netCDF file in a directory and take the data of one variable from each file and put it into one large multi-dimensional array. It also extracts the values for the latitude and longitude as well as the start time of the first file and the end time of the last file. The data, latitude, and longitude are then saved to a single NPZ file. Then the important metadata – the minimum and maximum values of data and dimensions, start and end time, measurement intervals, units, variable names – are extracted, put into a Python dictionary, and then saved as a JSON file.

3.2.3 Improving the Program – 07.10. to 22.10.2020

After both basic versions of a heat map plotting program and a data extraction program were developed, I worked on making them work together and iteratively improving them. I also started to develop a graphical user interface (GUI).

Data Processing. When the processed data was first being used in plotting some bugs became apparent. The most severe of these was a mistake I made in handling the masked arrays contained in the netCDF files. These arrays contain data which might not be valid for certain values of the dimensions. In the netCDF files all of these values are filled with a special value called the `_FillValue`. This value is specified in the netCDF files and can be used to process the files correctly. If it is not handled correctly, the fill value will be interpreted as a proper value and make any plot unusable. What I did to fix this issue was to replace every occurrence of the fill value with the `numpy` data type `numpy.NaN` (a specfic value meaning Not a Number). This makes it simple to ignore these values in calculations because there can be no confusion about whether or not the numbers are valid. The fill value is also being saved to the metadata file so that it can be used at a late date.

Plotting. The main changes to the plotting program in this part of the development was the creation of a program that can plot time series, unifying the heat map and time series plotting program, and enabling the plotting programs to work with the above mentioned `numpy.NaN` values.

To plot a time series one has to select data for a specific point in space and then plot all the values of that point from a start date and time to an end date and time. The most challenging element here is the conversion of a user-entered date and time in the format `YYYY-MM-DD H` to a computer-readable date and time object and then to an array index that can be used to access the data. This was achieved by reading in the user-specified date and time as text, using a Python standard library function to convert it into a `datetime` data type. Then one can find the difference between the user-specified date and time and the start date and time of the data which can then be used to find the index of the array that corresponds to the given time.

Unifying the heat map and time series plotting involved copying the code of both programs to a single file and then setting up the command line program to accept the types of input required to plot each of the types of plots.

To get the plotting programs to work with `numpy.NaN` values, the only change I had to make to the programs was to change the functions that find the minimum and maximum values to functions that ignore `numpy.NaN` values. If this is not done, any operation involving a `NaN` value will itself result in `NaN`. Now the `NaN` values are left out of computations and will not invalidate the results.

GUI. To create a GUI I needed to become familiar with the basics of `PyQt5`, the graphics library used in this project. PyQt5 is a Python library that offers bindings to the C++-based Qt library. Qt covers, among others, wireless connectivity, web browsing, and traditional user interface (UI) development. Only the UI development part of the library will be used in this project. A simple GUI in PyQt5 can be created in a few lines of code. The only requirement is to create a `QApplication`, which is the main application that is run when the code is executed. If one also wants to display something to the user, a GUI element needs to be created. One of the simplest such object is a button, which simply get a text and an action that is performed when it is clicked. The action is performs is a normal Python function and can thus do anything a Python function could do – display text, open another window, or create a graph.

3.2.4 Developing a GUI – 22.10. to 09.11.

Now that I had developed a working data processor program, a working plotting program, and a very basic GUI, I could start putting the three of them together to create a functional GUI. In general this process involved re–writing my existing programs in an object-oriented programming (OOP) style. This was necessary because as individual programs they were procedural, meaning they would execute a set

number of commands in a set order and then exit. Now though, a new command could be called at any time or values might need to be modified while the program is running, so OOP was a good idea. Additionally, PyQt5 is written using an OOP approach, so it is just natural to use one, too. OOP also allows code to be organized into smaller, purpose-built classes that make code management simpler. During this process I discovered many bugs in the original programs that had not come up before and I was forced to rethink and improve my existing programs. The following paragraphs give an overview of this process.

A better way to plot dates. In the procedural version of the time series plotting program I needed to create an array of time and date pairs to use as x axis data for the time series plot. From the included metadata I could read the number of measurements per day and through the user–specified start and end date I had a time range. Thus I could find the number of measurements for the time range. For each of these values for which I only knew the index in they array, I now needed to create a piece of text that has the date and time of the measurements. I painstakingly wrote long and hard-to-read code that performed this task but still had some problems. Most notably, to set the values on the axis, I manually specified them which meant that all of them were displayed – having one label for each day when graphing data for a whole year is very messy. I found that a simpler way to do this was to use a feature of the `pandas` library. This mathematical and statistical library has a series feature that can also create a series of dates. I simply needed to specify the beginning and end dates and times and then the interval of time between each measurement and a series would be generated in a single line of code. This made my code much more readable and correct.

Optimizing the data processor. The procedural version of the data processor reads every netCDF source file in one-by-one and then copies the desired data from it to a `numpy` array. Each of these arrays then needs to be added to a larger array that saves all the data until it is finally saved as an NPZ file. The simplest way to do this is to create an original array, load the first piece of data into it and then use the `ndarray.append(array)` function (`ndarray` is the `numpy` name for a multi-dimensional array) which takes the `ndarray` and appends `array` to the end of it. This works perfectly well, but it is very slow. While developing a GUI for the data processor it was so slow that my operating system was giving me warnings that my program must have crashed because it was not responding. The reason for this dismal performance was that the `append` function does not append in-place. That means that each time the function is called, all the data from both the `ndarray` and the `array` are copied to a new location. If this is done many times and especially as the main array gets larger this operation is very slow. Thus my goal was to append the array in-place – meaning that I would not have to copy all the data each time – and I tried to find a function

that would enable this. Because I could not find one I decided to use the following approach. I know at the very beginning of the processing how many files will be processed. I also know how many measurements from each file I will need to copy. Thus I can create one large, empty, appropriately sized array before the copying starts and then simply put the data from each file in the place that it belongs. This keeps the data from being copied each iteration and thus dramatically improved the speed of the program.

Threading. Even after optimizing the data processor and speeding it up considerably, I was still getting warnings about my program freezing while it was running. Additionally, when I tried to update the progress bar that I planned to use to show the user how far the extraction had progressed, it did not work but jumped to 100% once the process had completed. After researching online and reading the documentation I realized that while my data processing was going on, the event loop that controls the program was being blocked. The even loop is the loop that continuously checks if a button has been pressed etc. If this loop is blocked my program becomes unresponsive, which is what lead to warnings about it freezing and the progress bar not updating. The solution to this problem is to use separate processes to do the time intensive tasks and to have the main program simply run the event loop and keep the program responsive. PyQt5 has a feature called `QThread` which is a process that can run on its own. It is started from the main program but then runs in parallel with it until it is completed or otherwise stops. While it is running, a `QThread` can send messages to the main program through `pyqtSignal`s, which can contain any data type Python supports, from `None` (for a simple indication that something happened) to text (maybe a status indicator). If the data processor is started as a thread it no longer freezes the main program and when a `pyqtSignal` is used to send progress information to the main program the progress bar can be properly updated. This way of implementing more time intensive processes worked so well that I used if for every single such case afterwards.

Deciding on the GUI structure. After I had created a simple GUI that fit the needs of the data processor – directory selection, variable selection, button to start processing, and a progress bar – I needed to decide how to move forward with the GUI. I could develop small but separate GUIs for each step in the process, create a main menu where buttons would move the user to the desired window or popped up a new one, but I decided to implement it using tabs (like tabs in a web browser). Tabs would have the advantage that I would only need a single main window and wouldn't have to switch between multiple independent windows. This would also enable the user to look at more than one tab, e.g. plan which data to select while some data was being processed. To create these tabs I used the PyQt5 `QTabWidget` which makes the

creation and management of tabs easy. It allows, for example, to get an index of which tab is currently activated and it can also activate a specific tab to move the user to a new tab.

HelperFunctions for buttons and labels. My GUI is made up of a relatively small number of UI elements. The most common ones are the label (which simply displays text) and the button (as mentioned above). There are at least two of both on every single tab in my program. While creating them is not hard, it does take a couple lines of code. The basic lines needed are the initialization, setting the desired text, and setting its size and position. This would take about 3 lines of code (although some of these commands can be combined in the initialization). Because I found myself writing these 3 lines of code over and over again I created a class called `HelperFunctions.py` that contained two functions that created a button or a label in one line. The function itself uses the exact same commands that I was using in my code, but now I didn't have to type them anymore. This step made my code more readable and smaller in size.

A table to select data parameters. After the data has been processed from netCDF files to NPZ files, the program should allow the user to export or plot a part or all of the data. This requires the user to be able to specify which range of dates, latitudes, longitudes, and levels they want to export or plot. The GUI for the selection should be precise but also simple and intuitive. I decided to do this in the form of a table. This table allows me to display the dimensions name, its units, and the minimum and maximum possible values. Then there are empty fields where the user must enter the minimum and maximum values he wants their subset of data to have. Choosing a table for this purpose means that all the relevant data can be displayed together in a compact fashion.

3.2.5 Fixing Bugs – 10.11. to 13.11.

NetCDF file recognition. When I submitted the finished program to my supervisor, he pointed out that my program did not handle all netCDF files correctly. The files that I had downloaded and worked with had the file extension `.nc4` (for netCDF4) and my program only recognized these. The files he wanted to use had the extension `.nc`, but my program rejected them even though they were valid netCDF files. To fix this bug I had to not only check for files with the `.nc4` extension, but also for those with the `.nc` extension. Thankfully, this bug was an easy fix.

Help file in the browser. My original help section was simply another tab that had subtabs with a paragraph of text that briefly explained the purpose and usage of each of the components. My supervisor pointed out that this was not very helpful nor nice

to look at and he provided my with a website template that I used to create a better–looking help section. This help section is available in the top menu bar of the window. The website opens in a tab in a browser and now contains pictures of the tabs of the program for illustration.

Save all plot pdfs to a single file. When my program creates multiple files during the plotting phase, each plot is saved in its own file. For most file types this is the only way this works, but for the PDF format there are other options. My program still saved each plot to its individual PDF. My supervisor pointed out that the better way to do this is to save the plots to one single PDF file that contains one plot per page. I implemented this approach and it increased reduces the number of created PDF files and makes them easier to manage.

Fix the bug on Windows. I developed my GUI on Linux and thus I wanted to make sure it would also work on Windows. I installed all the necessary libraries and started my program. The error I encountered came from the fact that Windows does not allow a colon (":") to be part of a file name while Linux does. The problem arose when files that had time values in them were saved (a file of time "12:00" for example). On Linux this works fine, but on Windows it generates an error. To fix this error, I replaced the colons by dashes ("-") when the platform is Windows.

4 Python Satellite Data Program

This section details how to setup and use the program and how it works. First, the setup is discussed and then each of the successive stages are detailed. The information about different functions and ways to use Python and the libraries are taken from the documentation. Because many different libraries and elements work together, I will not point out every single one but name the required one here.

4.1 Setup

The programs and data required to use the program are the following:

- netCDF data files (preferably from M2I3NPASM because it has been tested),
- a working installation of the Anaconda Python distribution (available here).

Then one has to get the code to the program and set up the required anaconda environment. The code can either be downloaded from my Github repository which can be found here, or I can provide it upon request.

Setting up the anaconda environment is simple. Navigate to the `programs` folder in my internship repository. Then, open that folder in a terminal (it must be `cmd.exe` on windows, Powershell will not work). In the terminal, type `conda env create --file`

`internship_gui.yml` (on Windows, use `conda env create --file internship_gui_win.yml`). This command will create a Python environment with all the required dependencies for my program. The installation might take a while. The resulting environment will have the same name as the file it was created from. To use the environment is has to be activated with `conda activate internship_gui` (or `conda activate internship_gui_win`. Once the environment is activated, navigate to the `gui_program` folder. To execute the program, run the command `python3 main.py` (`python main.py` on Windows). You should see a window that looks like Figure 4.1.

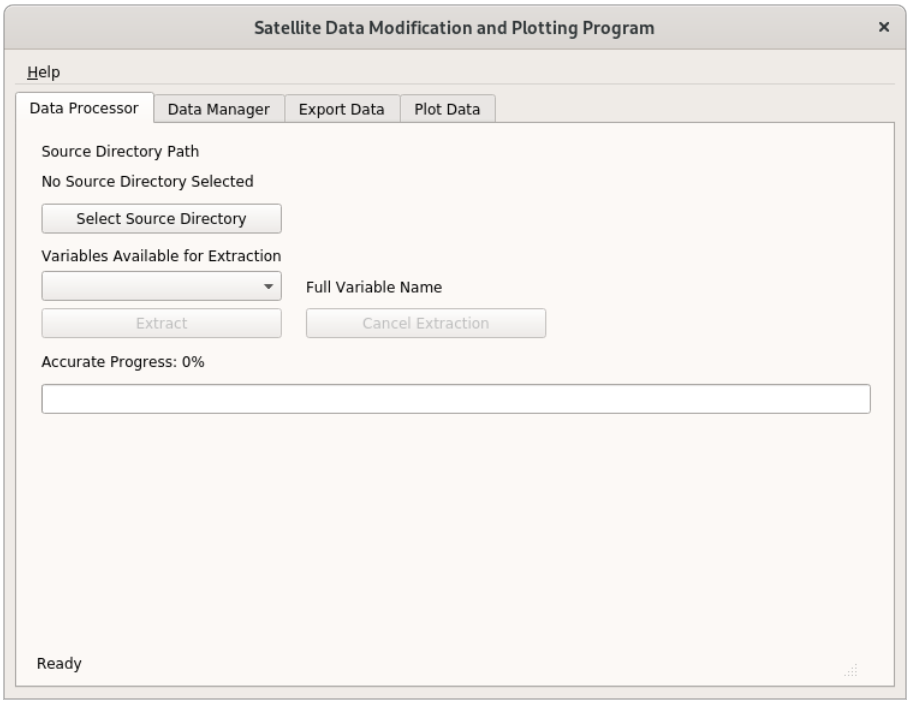


Figure 4.1: The Data Processor Tab

4.2 Data Processing

The first screen of the program is the Data Processor as seen in Figure 4.1. The purpose of the Data Processor is to take netCDF files as input, extract a single variable from them and save it to a NPZ file. It is made up of two classes, the `DataProcessor.py` class, a `QThread` that performs the processing, and the `DataProcessorTab.py` class which handles the UI. The steps to using it are the following:

1. Click the **Select Source Directory** button. An input dialog (Figure 4.2) will appear where you will have to select the folder where the netCDF files are located. The validity of the specified directory will be checked.

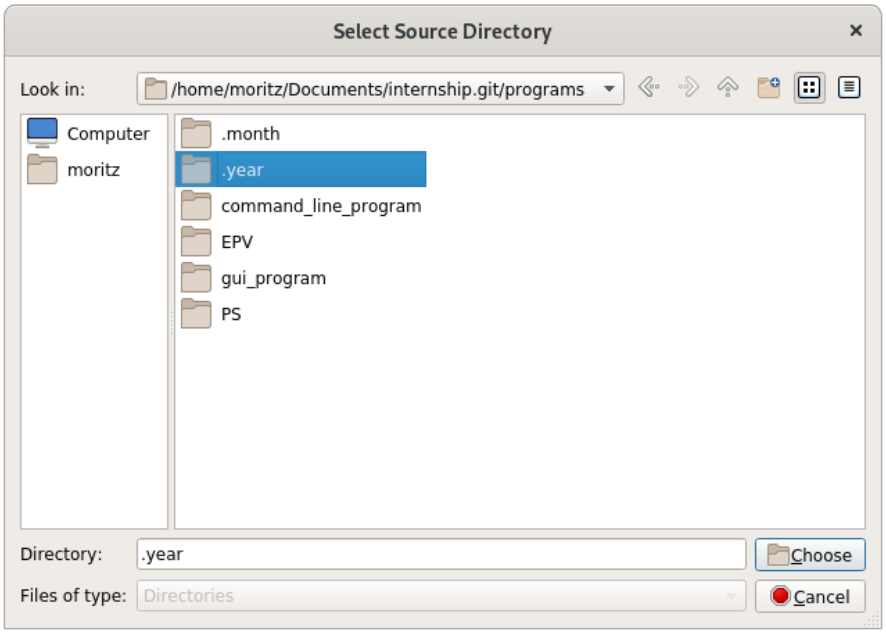


Figure 4.2: The Source Directory Selection Popup

2. After **Choose** is clicked, the directory is checked to make sure it actually contains netCDF files. Then, the available variables are extracted from the first file. They are displayed in a drop-down list in their short form and then the selected one's full name is displayed (see Figure **??**). Here the user must choose one of the variables to extract.

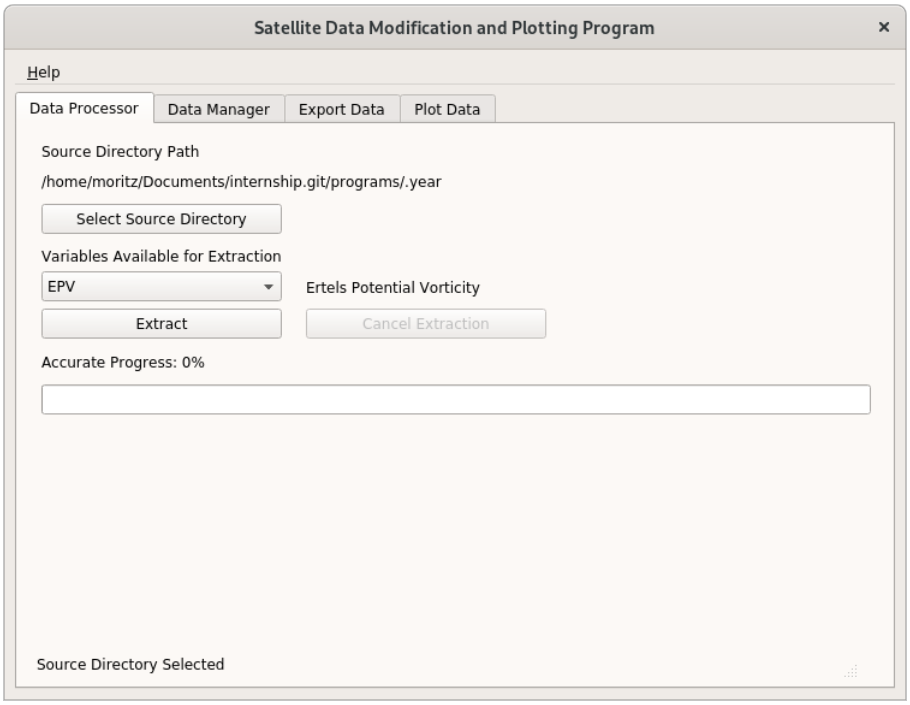


Figure 4.3: Data Processor with loaded source Directory

3. When the choice is made, click **Extract**. A popup asking for a destination

directory will appear where the user should indicate where the files should be saved. In case the extraction takes too long or the user changes their mind, the **Cancle Extraction** button will stop the extraction process. Figure 4.4 shows the extraction process.

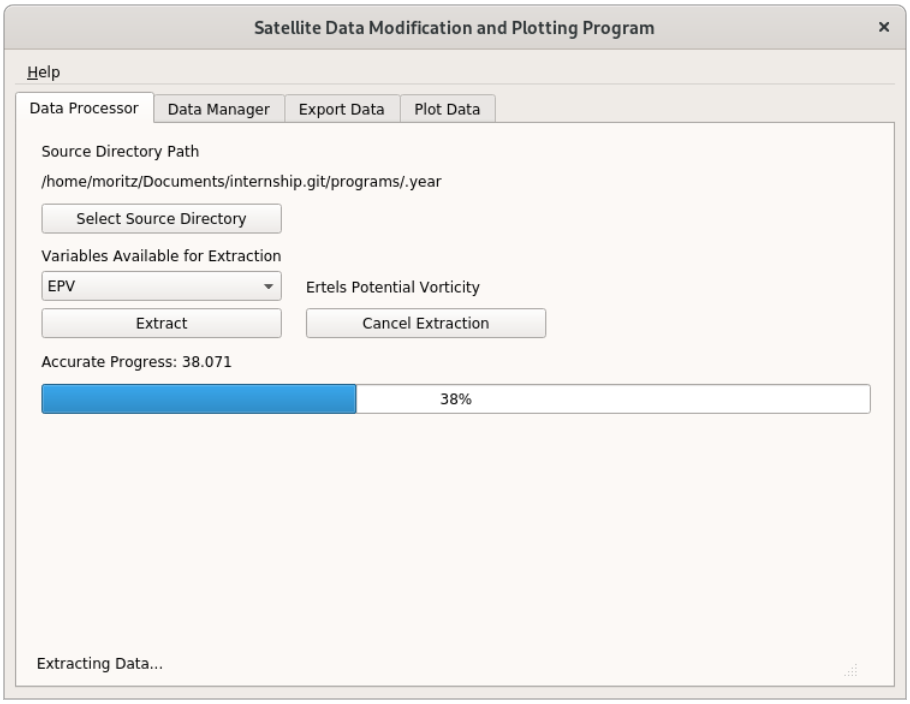


Figure 4.4: Extraction in Progress

This process will create a new directory in the directory the user selected. This directory will be named after the short name of the variable. It will contain the NPZ file named `<short variable name>.npz`, which will contain all the relevant data, and a file called `metadata.json`, which contains all the important metadata information that was contained in the netCDF file.

Once the extraction process is complete, the user can either continue using this program and switch to the Data Manager tab, or take the created folder and use the NPZ file with any `numpy`–compatible program.

4.3 Data Management

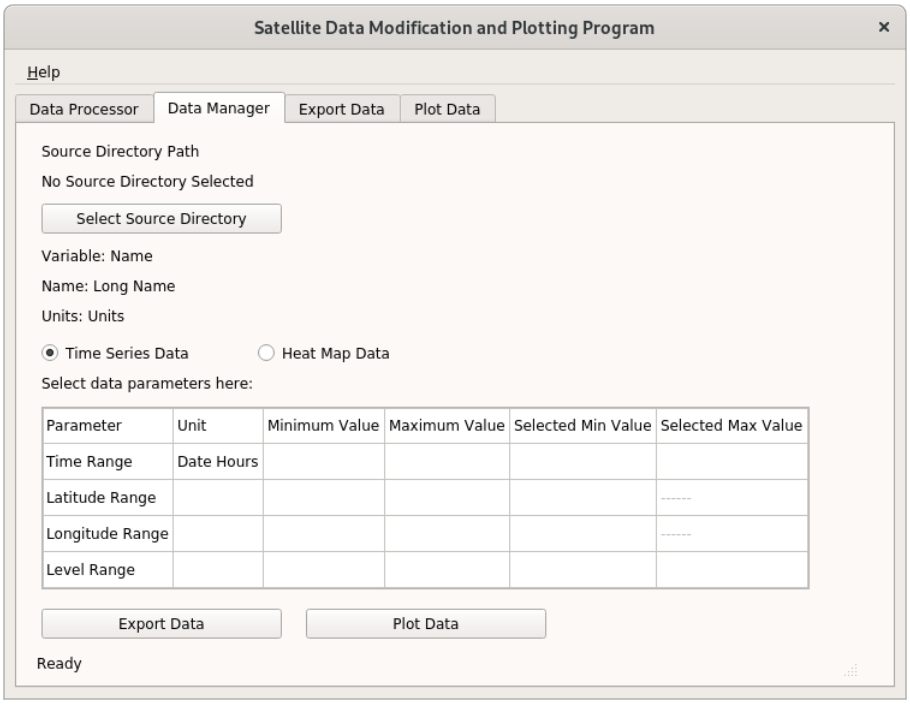


Figure 4.5: The Data Manager Tab

The Data Manager seen in Figure 4.5 is responsible for selecting the data subset the user intends to use and the type of data the user intends to use. The data subset is defined by setting minimum and maximum values for the dimensions of the data. The type of data is either a time series, where a location is fixed and all data for that point is available, or a heat map where a range of latitude and longitude is selected and then all data for that is selection is used. The Data Manager, much like the Data Processor, has two parts, the `DataManager.py`, responsible for providing the data, and the `DataManagerTab.py`, which is responsible for the GUI. The Data Manager is used like this:

1. Click the **Select Source Directory** button. An input dialog will appear where you will have to select the folder that was created by the Data Processor (the validity of the folder will be checked).
2. After the directory has been successfully selected, the variable name, long variable name, and the unit of the variable will be displayed. Furthermore, the table at the bottom will be filled in with the minimum and maximum values that are available according to the file that was provided. Figure 4.6 illustrates this step.

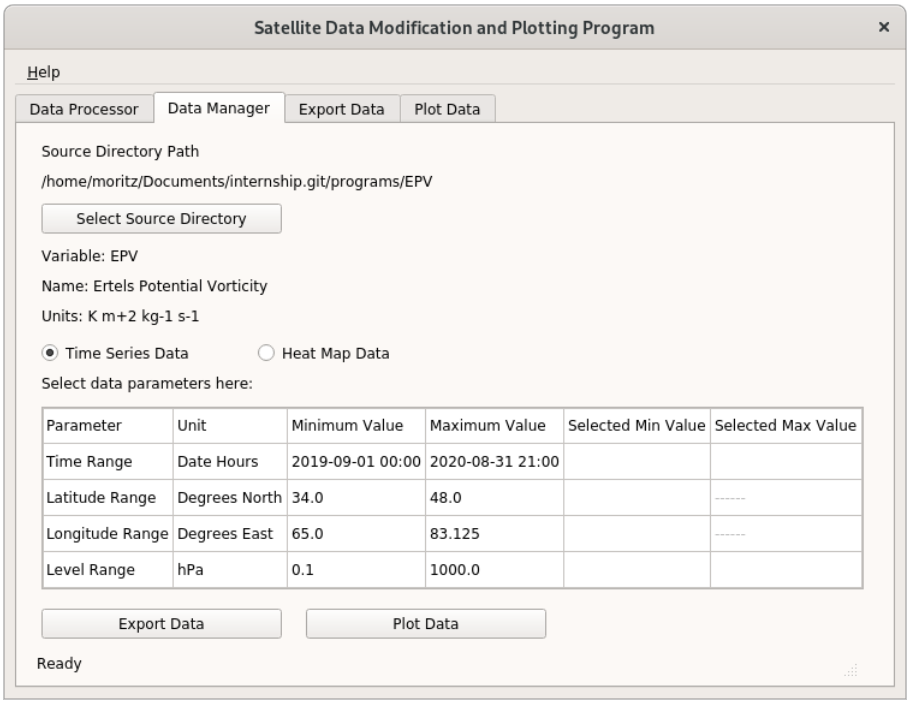


Figure 4.6: Data Manager with loaded Directory

3. With the directory selected, the user now needs to choose the limits of the data dimensions by entering them into the table. All user input will be validated and in case a value is outside of the possible values, an error message is shown. Furthermore, all entered values will automatically be corrected to the closest available value. The radio buttons determine if time series data (see Figure 4.7) or heat map data (see Figure 4.8) is selected and the table is adjusted accordingly. If the selected data does not have a level associated with it, the last row of the table is disabled (see Figure 4.9).

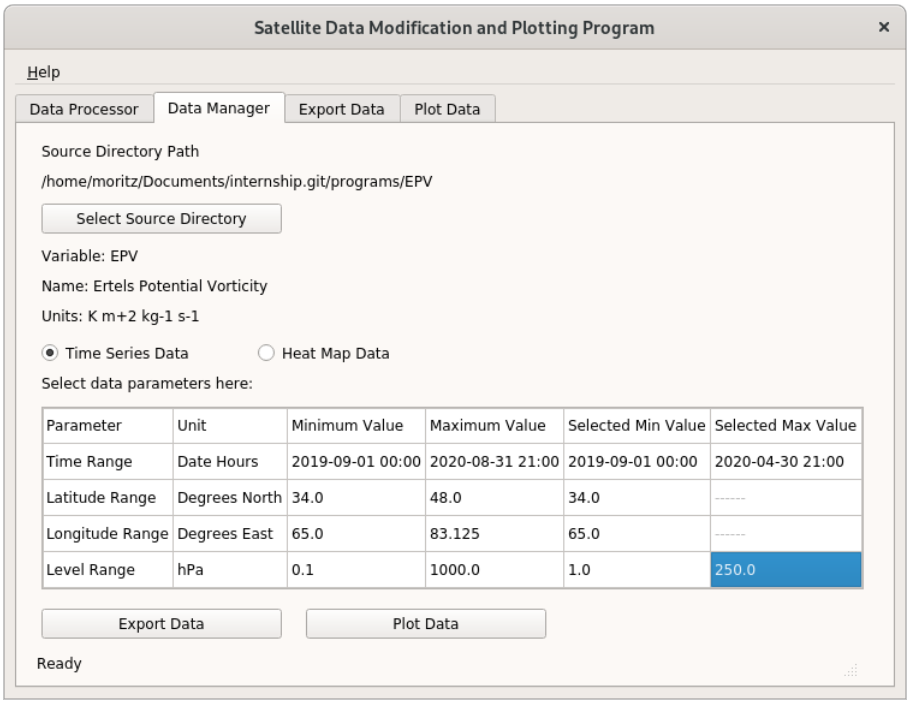


Figure 4.7: Data Manager in Time Series Mode

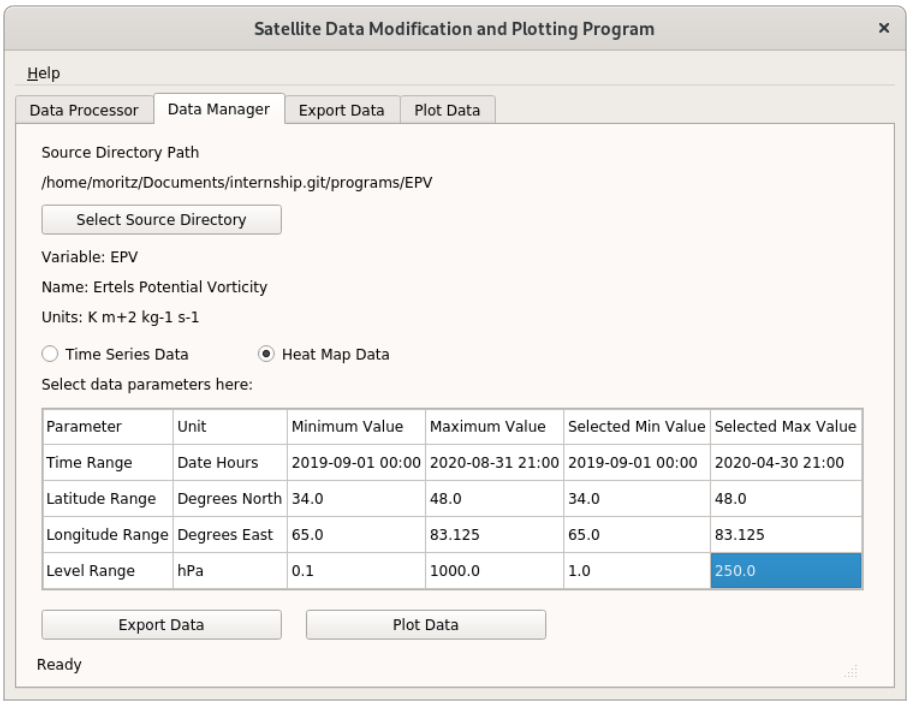


Figure 4.8: Data Manager in Heat Map Mode

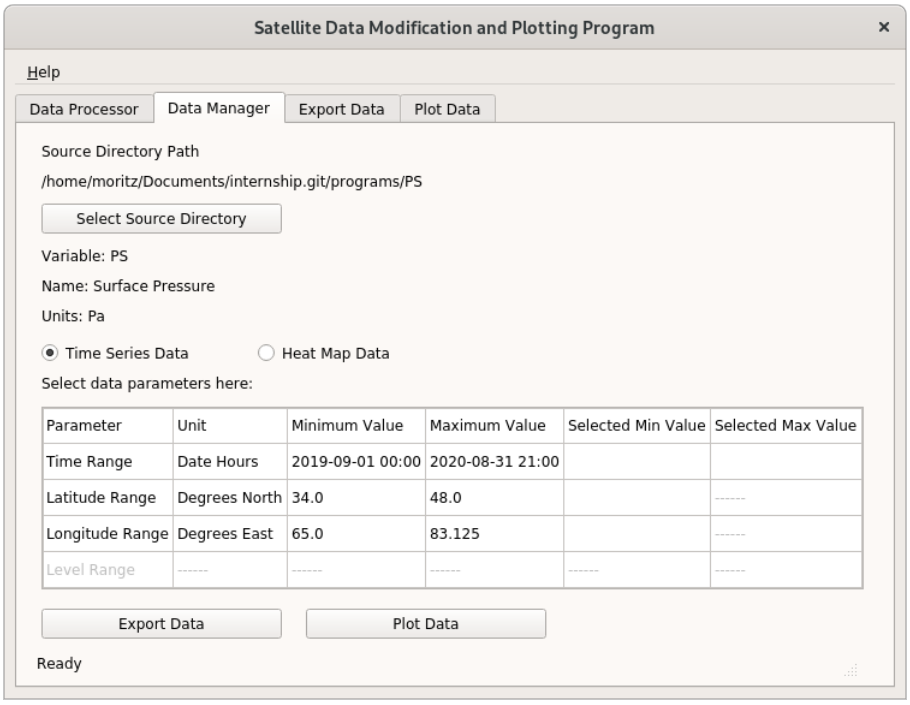


Figure 4.9: Data Manager with data without level

4. When all data fields have been filled, the user may either press the **Export Data** button or the **Plot Data** button. Each of the buttons will prepare the associated data and then switch to the corresponding tab.

The values entered in the Data Manager are persistent, meaning that if the user exports some data first and then wants to plot the same data, they only need to go back to the Data Manager tab and press **Plot Data**.

4.4 Data Exporting

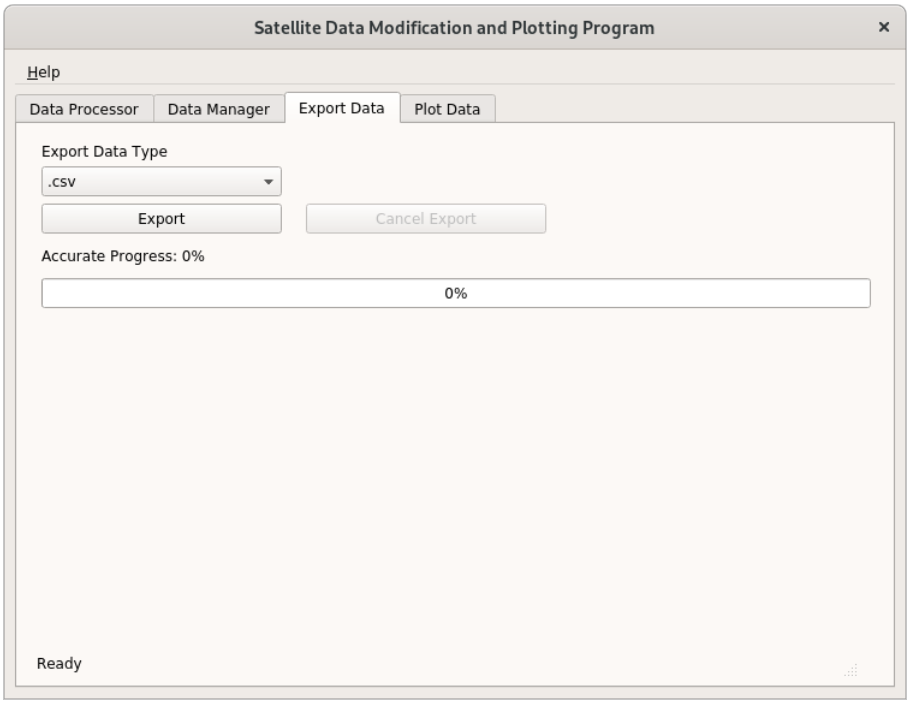


Figure 4.10: The Data Export Tab

If the user pressed the **Export Data** button, they will be taken to the Export Data tab. Again, this tabs functionality is coded in the `DataExporter.py` class and the GUI is in the `DataExporterTab.py` class. Figure 4.10 shows what the user sees. To use the data manager, the following actions are required:

1. The user is required to choose one of the 4 options of export data types from the drop-down list seen in Figure 4.10. The available options are `.cvs` (comma separated values), `.zip` (compressed file), `.xlsx` (an MS Excel file), or `.html` (a website file to be opened in the browser).
2. After making their choice, the user needs to press the **Export** button to begin the export process. Because this process might produce 1000s of files depending on the data settings specified, the program will give the user a warning about the number of files that will be created (see Figure 4.11).

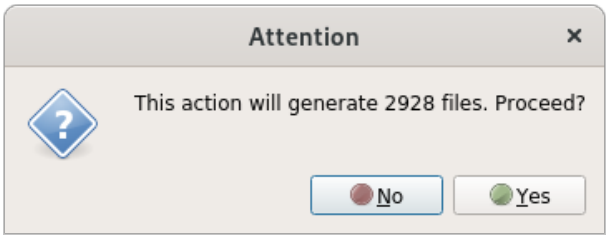


Figure 4.11: The File Number Warning Message

If the user clicks **Yes**, the export begins. Then, the program will ask for a destination directory where the files will be saved in a new directory called `<variable name>-exported`. If the process takes too long or the user simply wants to interrupt it, they need to press the **Cancel Export** button. The progress bar will show the extraction progress as illustrated by Figure 4.12

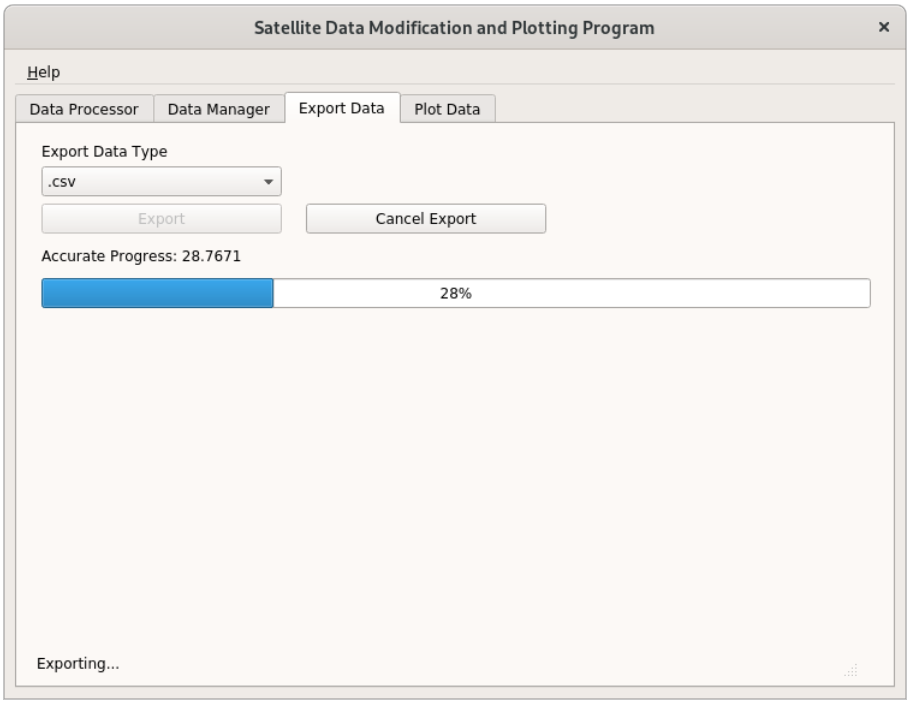


Figure 4.12: Data Export in progress

When heat map data is being exported, a file will be created for each time step and for each level in each time step. This may lead to the large number of files that the warning message (Figure 4.11) is meant for.

When time series data is being exported, a file will be generated for each level but because time series have a time span, all selected time values will be in one file. This leads to a manageable number of files.

The exported files will be saved in files with the following file names:

- **Heat Map**: `Heat Map <long variable name> <date and time> <minimum latitude and longitude values>-<maximum latitude and longitude values> <pressure level>.<file type>` (the pressure will be omitted if the data does not include it). An example of this type of file name is `Heat Map Ertels Potential Vorticity 2019-09-01 00:00 (34.0N, 65.0E)-(48.0N, 83.125E) 1.0 hPa`.
- **Time series**: `Time Series <long variable name> <start date and time>-<end date and time> <latitude and longitude> <pressure level>.<file type>` (the pressure will be omitted if the data does not include it). An example of this type of file name is `Time Series Ertels Potential Vorticity 2019-`

2019-09-01 00:00:00	237.30182
2019-09-01 03:00:00	236.86859
2019-09-01 06:00:00	236.79724
2019-09-01 09:00:00	236.82132
2019-09-01 12:00:00	237.73062
2019-09-01 15:00:00	238.40472
2019-09-01 18:00:00	238.4401
2019-09-01 21:00:00	238.20715
2019-09-02 00:00:00	237.49396

Figure 4.13: Temperature `T` exported as `.csv`

`09-01 00:00-2019-09-10 00:00 (34.0N, 65.0E) 1.0 hPa`.

To demonstrate the result see Figure 4.13 which contains the exported data for air temperature `T` in Kelvin at a pressure of 250 hPa at 34°N, 65°E from 01.09.2019 00:00 to 30.09.2019 21:00. The data has been cut off to save space.

4.5 Data Plotting

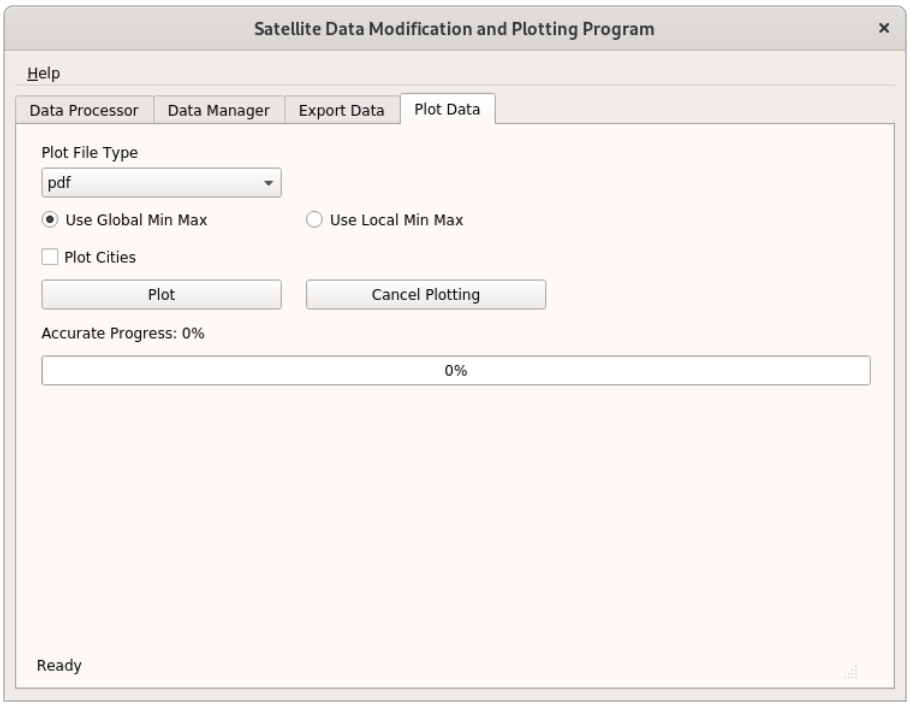


Figure 4.14: The Data Plotter Tab for Heat Map Data

If the user chooses to push the **Plot Data** button in the Data Manager tab, they will be shown the Data Plot tab. As with all other tabs, the UI is handled by a class called `DataPlotterTab.py` and the plotting itself is handled by the `DataPlotter.py`. The

GUI the user will see if they chose to plot heat map data is illustrated in Figure 4.14. If the user chose time series data, Figure 4.15 will be shown (the **Plot Cities** check box will be disabled).

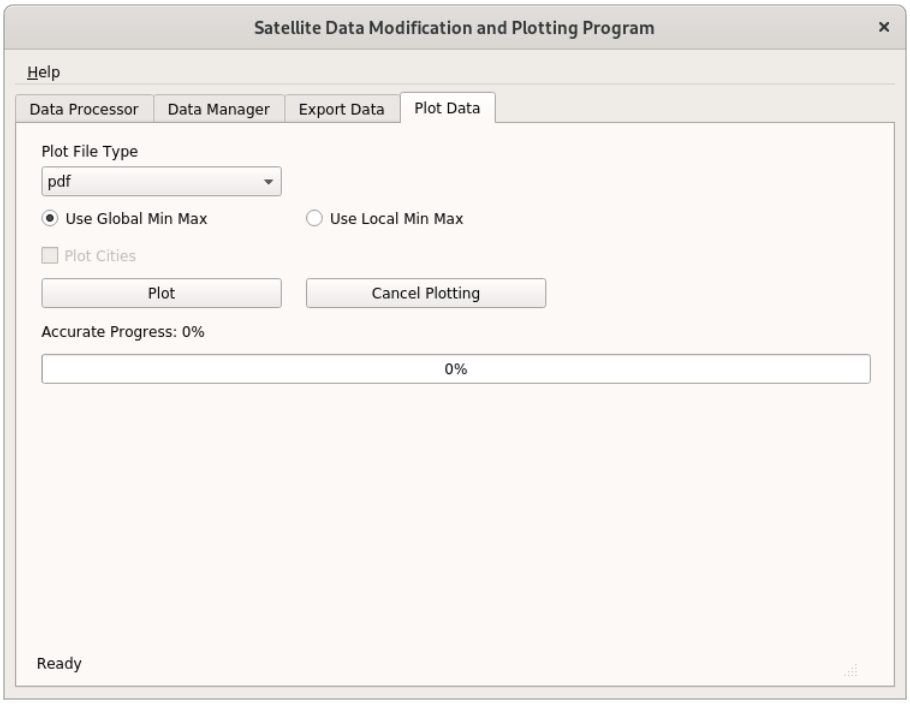


Figure 4.15: The Data Plotter Tab for Time Series Data

The steps to using this tab are very similar to the Data Exporter tab; they are:

1. The user chooses one of the 5 options of plot data types from the drop-down list seen in Figure 4.14. The available options are `.pdf` (portable document format), `.png` (portable network graphics), `.eps` (encapsulated PostScript), `.svg` (scalable vector graphics), or `.jpeg`.
2. The next option is whether or not to use global minima and maxima or to use local minima and maxima. Using global minima and maxima means that if the user selected a range of 2 days, the minimum and maximum values of all measurements in these 2 days are used as the axis limits. If the user chooses local minima and maxima, the minimum and maximum value for each individual plot will be used. The former makes is easier to compare multiple plots with each other, while the latter creates more fitting individual plots.
3. If the user is plotting heat map data they can choose whether or not to plot cities on their heat map. The cities that will be plotted are Bishkek, Almaty, Kabul, Tashkent, and Dushanbe.
4. After making these choices, the user needs to press the **Plot** button to begin the plotting process. This process might produce 1000s of files depending on the data settings specified, so the program will give the user a warning about the number

of files that will be created (see Figure 4.11 above). If the user clicks **Yes**, the export begins. Then, the program will ask for a destination directory where the files will be saved in a new directory called `<variable name>-plotted`. If the process takes too long or the user simply wants to interrupt it, they need to press the **Cancel Plotting** button. The progress bar will show the extraction progress as illustrated by Figure 4.12

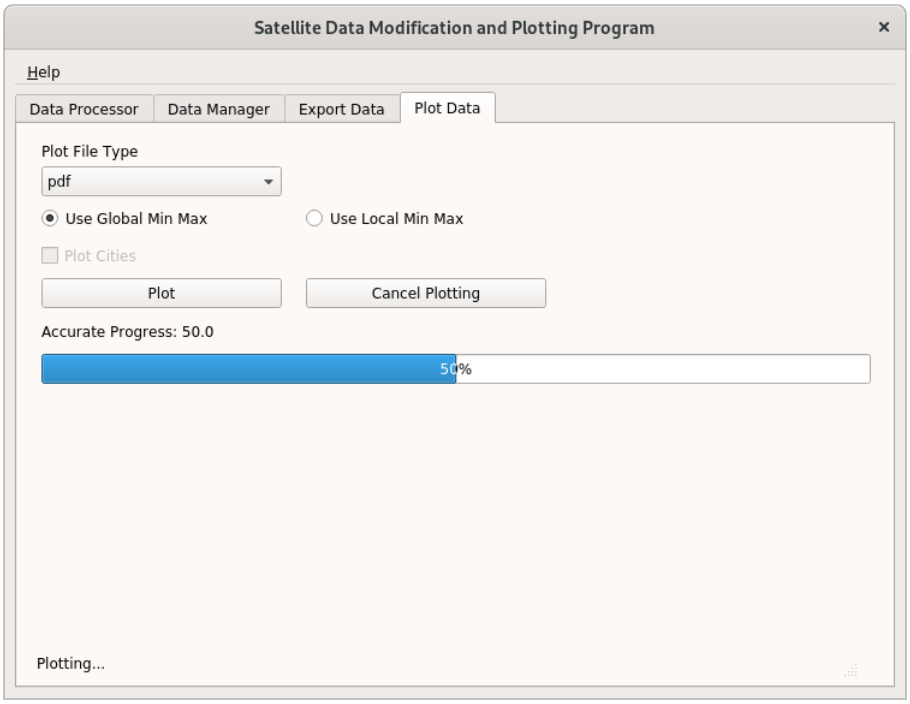


Figure 4.16: Data Plotter in Progress

When heat map data is being exported, a file will be created for each time step and for each level in each time step unless the plot data type is PDF. If PDF is chosen all the plots will be put into one large PDF file with a page for each of the plots. If PDF is not the file type, a large number of files may be generated.

When time series data is being plotted, a file will be generated for each level but because time series have a time span, all selected time values will be in one file. This leads to a manageable number of files.

The exported files will be saved in files with the save files names as specified in the Data Export section above.

Sticking with the temperature example from above, the time series plot of the same data that is shown in Figure 4.13 is plotted in Figure 4.17. In Figure 4.18 a heat map of the air temperature on 01.09.2019 at 00:00 at a pressure of 250 hPa is graphed for a region from (34°N, 65°E) to (48°N, 83.125°E).

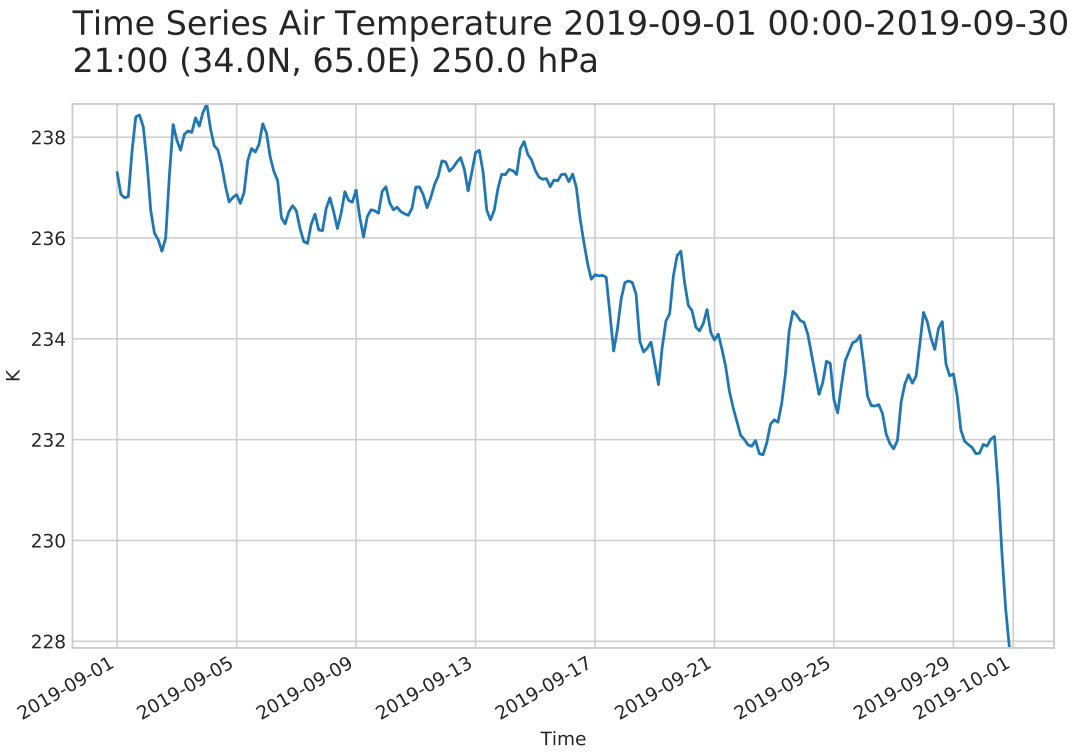


Figure 4.17: Time Series Plot for Air temperature

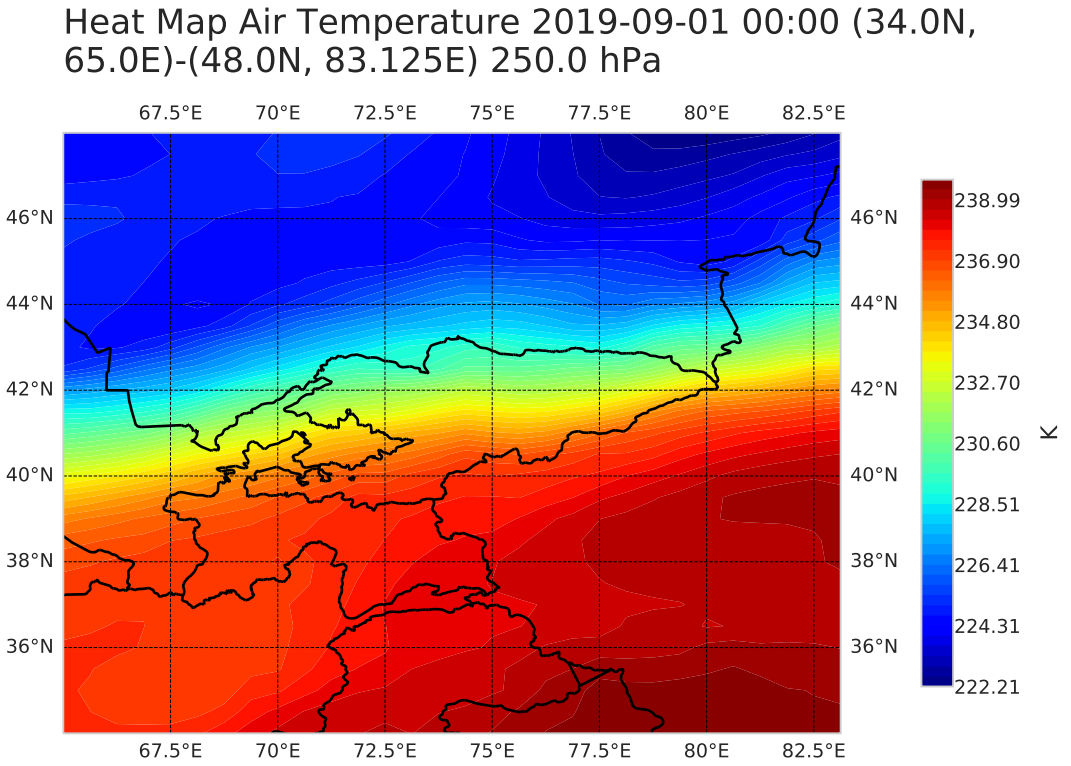


Figure 4.18: Heat Map Plot for Air temperature

Figure 4.19 shows the same plot as Figure 4.18, but with city plotting enabled.

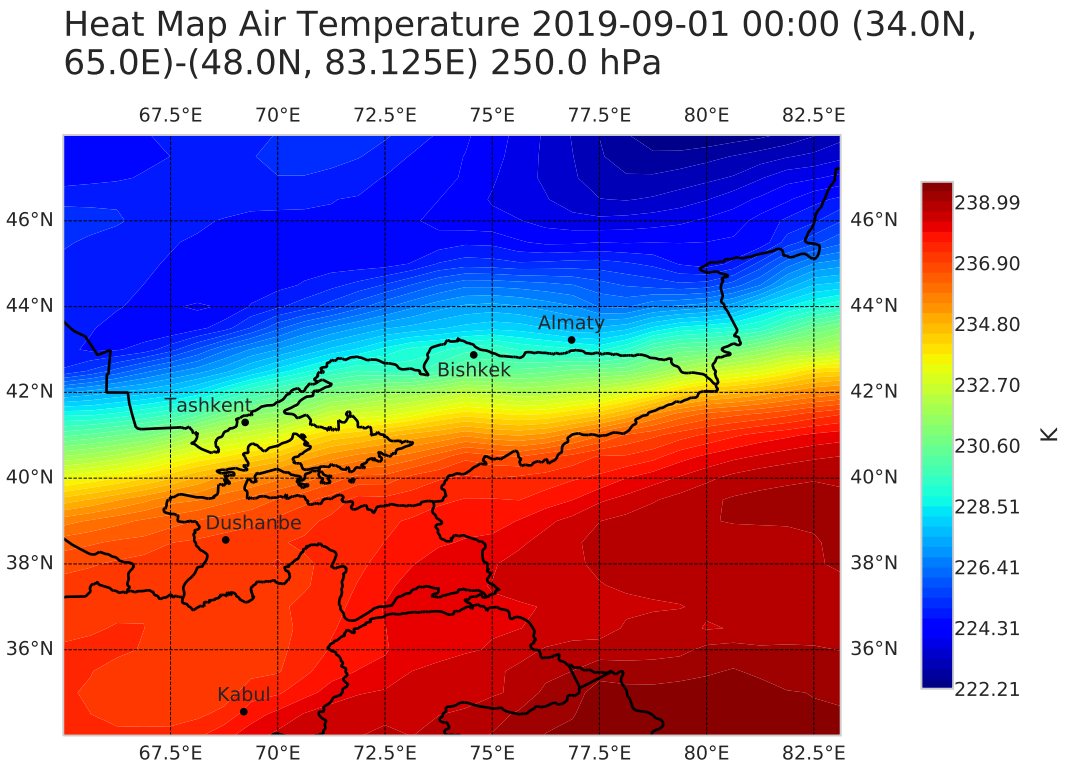


Figure 4.19: Heat Map Plot for Air Temperature with cities

4.6 Help

The last and maybe most important section of the program is the help section. It can be accessed by clicking on the text **Help** in the menu bar at the top of the program window as seen in every figure of the program window, e.g. Figure 4.16. This will open a local html file in a browser tab. This website is based on a template given to me by my supervisor to create the help section. It contains a table of contents, a short introduction to what the program does, and then a short description of what each of the 4 tabs do, similar to the one provided in this section. In Figure 4.20 the menu and introduction are shown as they appear in a browser.

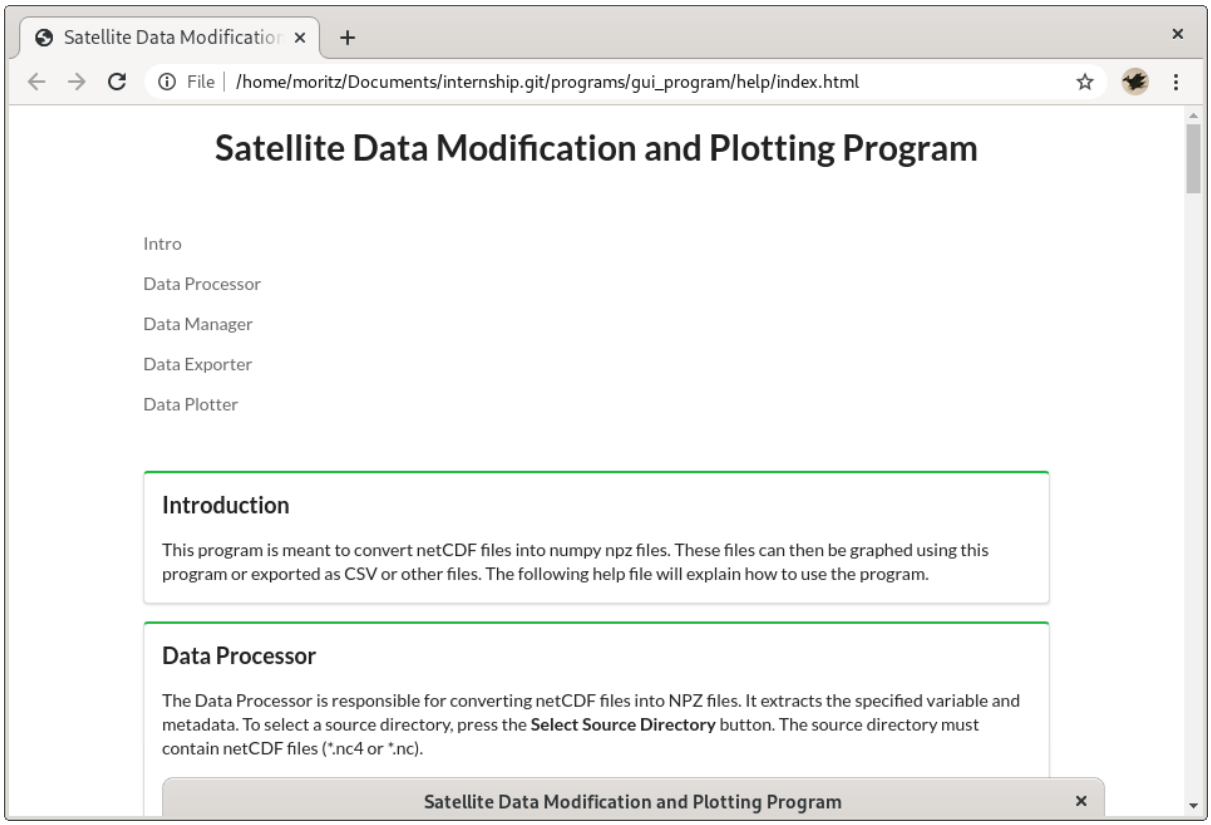


Figure 4.20: Screen Shot the Help section

5 Data Investigation

In this section I will use my program and the skills I learned during this internship to explore the vertical structure of air pressure, as per my supervisor's suggestion. I will show the tropopause zone, the vertical gradient of the air temperature and explain why it is like it is, and the inverse gradient caused by the ozone layer.

5.1 The Troposphere and Tropopause

The Troposphere is the lowest layer of the atmosphere – starting on the surface it reaches to an average height of 12 km. The air pressure in the troposphere decreases non-linearly as the height above the surface increases. The temperature in the troposphere also generally decreases when the pressure decreases. At the top of the troposphere the temperature is -63°C or about 210K. The pressure range of the troposphere is about 1000 hPa to 200 hPa according to »».

The boundary between the troposphere and the next layer of the atmosphere, the stratosphere, is called the tropopause. The tropopause can be demarcated by multiple factors. One factor is the inversion of the temperature gradient of the troposphere,

meaning that the temperature starts to increase with a decrease in pressure, not to decrease like before. Another metric by which to define the tropopause is by a sharp increase in potential vorticity, a measure of the rotation of the air masses. The third factor indicating the tropopause is the abrupt increase of the ozone mixing ratio. The ozone mixing ratio is a measure of how much ozone is part of the air by either mass or volume.

To investigate these behaviors, I modified my plotting program to plot all pressure levels of a certain variable for a specific date and time and position. The following subsections use these plots to explain the behavior above. To be comparable, all plots are using the exact same data: M2I3NPASM data for the 01.09.2019 at 12:00 for the position (40°N, 70°E).

5.1.1 Air Temperature

As stated above, the air temperature is expected to drop continuously as the pressure decreases from 1000 hPa to about 200 hPa. After that, in the tropopause, the temperature is expected to sharply increase. Figure 5.1 illustrates this behavior.

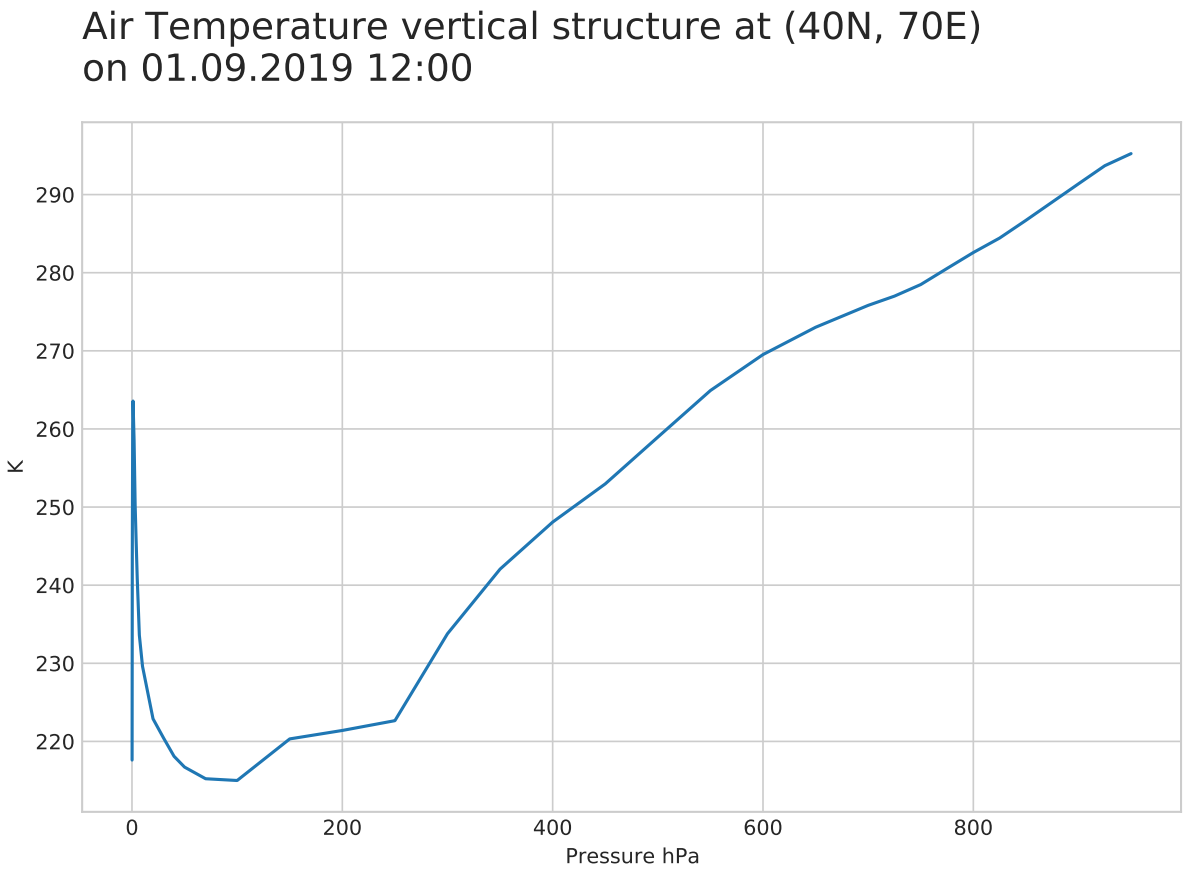


Figure 5.1: Air Temperature by Air Pressure

We see that after about 200 hPa the temperature decreases to about 210K and then

sharply increases, as expected.

5.1.2 Potential Vorticity

The second indicator of the tropopause is a sharp increase in the potential vorticity that should occur around 200 hPa. The satellite data includes Ertel's Potential Vorticity (EPV), which shall he used here. Figure 5.2 illustrates the shart increase in potential vorticity. The x axis is $\log_{10}$-based to make the transition more clear.

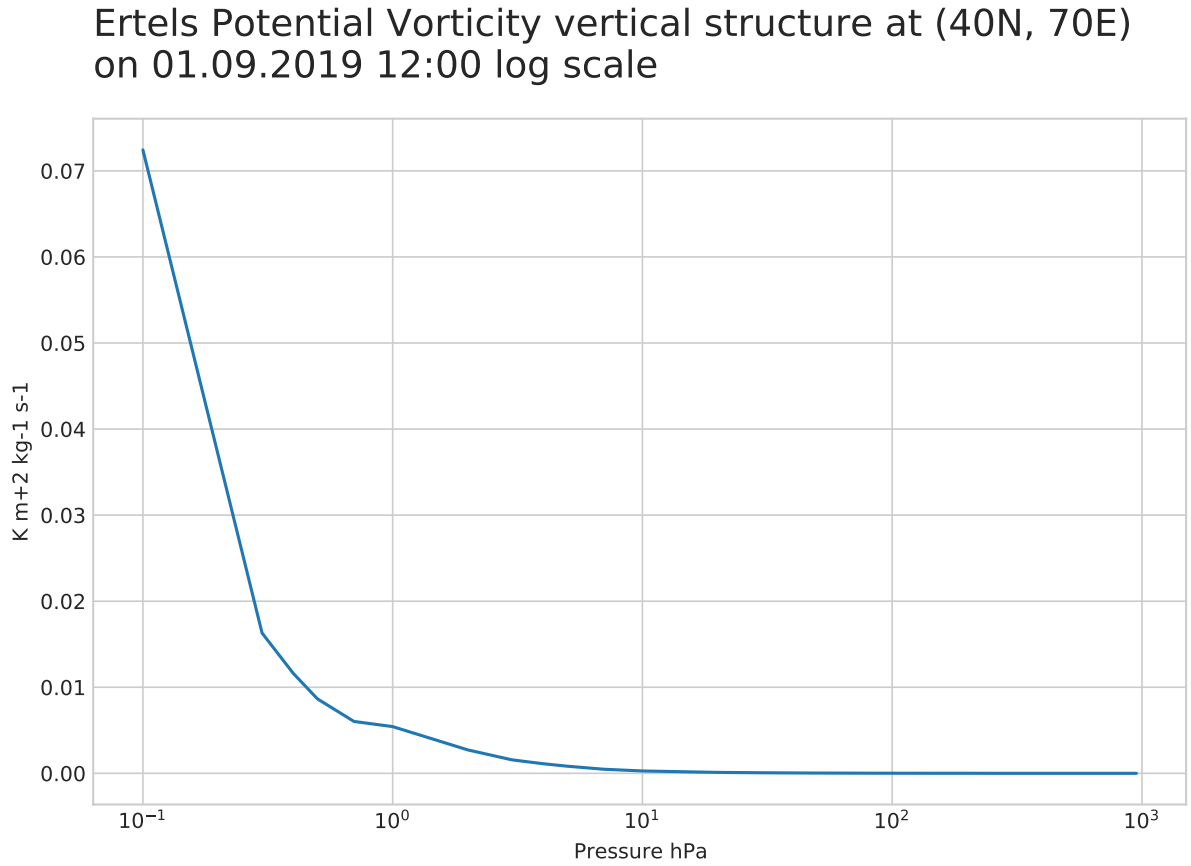


Figure 5.2: Ertel's Potential Vorticity by Air Pressure on a $\log_{10}$ scale

5.1.3 Ozone Mixing Ratio

The third marker of the tropopause is the increases ozone mass mixing ratio, represented by the `O3` variable of the M2I3NPASM dataset. In Figure 5.3 we see that as the pressure approaches approximately 10 hPa, the ozone mass mixing ratio increases sharply.

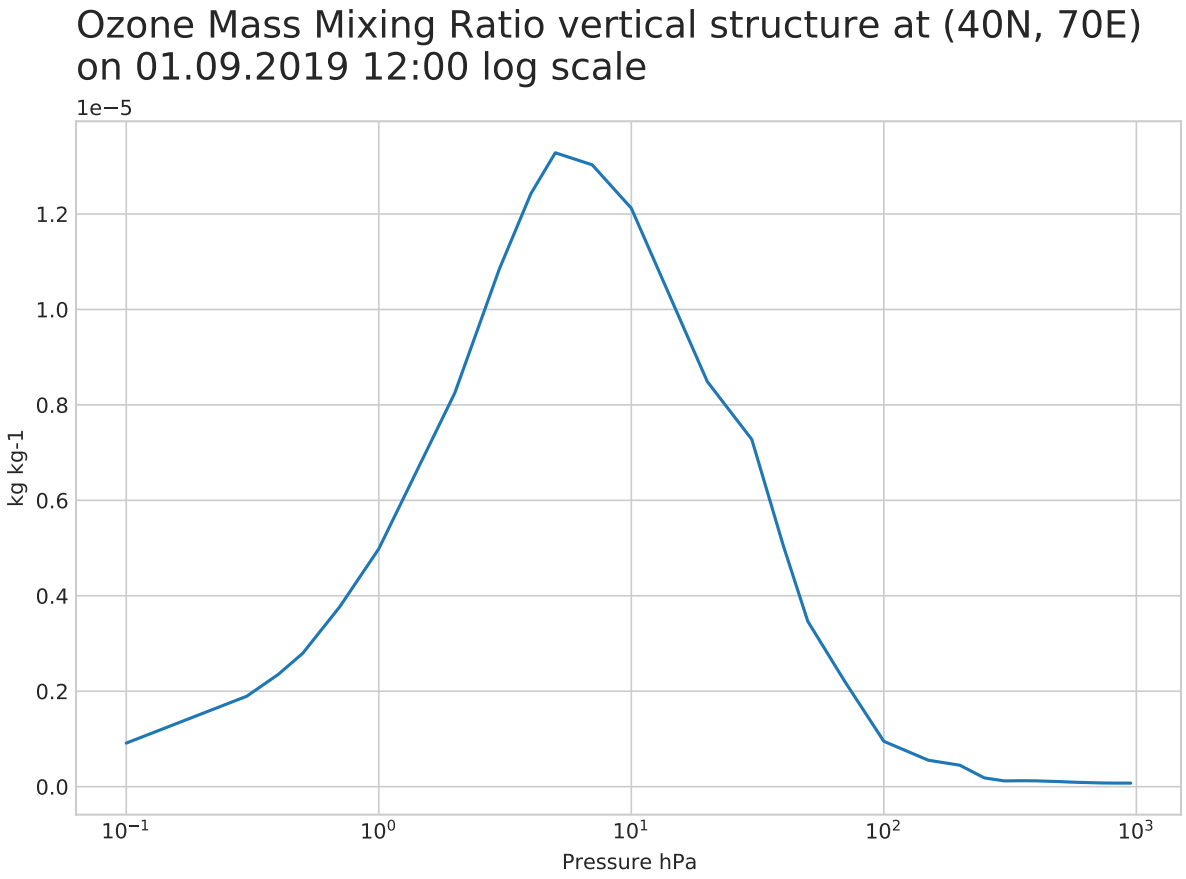


Figure 5.3: Ozone Mass Mixing Ratio by Air Pressure on a $\log_{10}$ scale

Figure 5.3 supports our assumptions about the ozone mass mixing behavior in the tropopause.

5.2 Reasons for Temperature Inversion

According to »», the temperature inversion in the tropopause is caused by heating as a result of sunlight absorption by molecules and aerosols in the tropopause.

As Figure 5.3 showed, the ozone mass mixing ratio increases sharply as the pressure decreases to about 10 hPa. While most ozone is contained in the stratosphere, where it absorbs ultraviolet radiation, some is also contained in the troposphere, as shown. Ozone absorbs ultraviolet radioation and heat is resealed in the process. The ozone in the troposphere might absorb ultraviolet light that got through the troposphere and release heat in the process which in turn warms up the tropopause.

6 Conclusion

During this 2 month internship at RS RAS Bishkek I studied the ways in which one can access NASA satellite data, which formats they come in, and how to work with

said data. I also learned how to develop a program that can read, process, export, and plot the satellite data using Python and the relevant libraries. All of these skills will be very useful for my future.

Before this internship I never realized how simple it was to access satellite NASA data, which I always thought was very interesting. Furthermore, these skills could be useful for either future research I might do or for a job I might have. Gaining experience with scientific data formats like netCDF and the libraries used to work with these formats falls into the same category.

Developing a Python program using a popular GUI library is also a valuable skill to have, especially because I study Applied Mathematics and *Informatics* and might work in that field as well.

To conclude I think that I learned many useful skills during my internship that will help me be a better student and maybe even a better researcher one day.

References

[1] "NumPy reference," 06.2020. Release 1.19.0. Retrieved from: https://numpy.org/doc/1.19/numpy-ref.pdf.

[2] W. KcKinney and Pandas Development Team, "Pandas: powerful python data analysis toolkit," 08.2020. Release 1.1.3. Retrieved from: https://pandas.pydata.org/docs/pandas.pdf.

[3] "The netcdf user's guide," Accessed 23.10.2020. URL: https://www.unidata.ucar.edu/software/netcdf/docs/user_guide.html.

[4] "Netcdf4 module documentation," Release 1.5.4. Accessed 23.10.2020. URL: https://unidata.github.io/netcdf4-python/netCDF4/index.html.

[5] J. Hunter, D. Dale, E. Firing, M. Droettboom, and the matplotlib development team, "Matplotlib," 09.2020. Release 3.3.2. Retrieved from: https://matplotlib.org/3.3.2/Matplotlib.pdf.

[6] "Cartopy documentation," Release 0.18.0. Accessed 23.10.2020. URL: https://scitools.org.uk/cartopy/docs/latest/.

[7] "Python documentation: json – json encoder and decoder," Release 3.9.0. Accessed 23.10.2020. URL: https://docs.python.org/3/library/json.html.

[8] "Python documentation: Built-in types – dict," Release 3.9.0. Accessed 23.10.2020. URL: https://docs.python.org/3/library/stdtypes.html#typesmapping.

[9] "Python documentation: datetime – basic date and time types," Release 3.9.0. Accessed 23.10.2020. URL: https://docs.python.org/3/library/datetime.html.

[10] "Ipython documentation," Release 7.18.1. Accessed 23.10.2020. URL: https://ipython.readthedocs.io/en/stable/.