

Finding water level changes via satellite imagery

Creating a python package collaboratively

Please read this assignment carefully.

This coursework is concerned with the creation of a library to analyse data from different instruments on-board a satellite. The instruments provide daily updates from a piece of land that can be used to track water levels at different locations.

This assignment asks you to work collaboratively within your team to create a package. For that you will need to write some code for querying, loading and analysing a dataset about different portions of land in a map. We will describe how the code must behave, but it is up to you to fill in the implementation. The package needs to follow all the good practices learnt in the course, that is, the package should: be **version controlled**; include **tests**; provide **documentation and doctests**; set up **command line interfaces**; and be **installable**. Besides this, you will also need to modify an existing implementation of a provided script to make it **more readable, more efficient, and measure its performance**.

The collaboration aspect should be organised and managed using GitHub.

The exercise will be semi-automatically marked, so it is very important that your solution adheres to the correct interface, file and folder name convention and structure, as defined in the rubric below. An otherwise valid solution that doesn't work with our marking tool will not be given credit.

For this assignment, you can use the Python standard library and other libraries you may wish to use (but make sure they are clearly set as dependencies when installing your package). Your code should work with Python 3.9.

First, we set out the problem we are solving. Next, we specify the target for your solution in detail. Finally, to assist you in creating a good solution, we state the marking scheme we will use.

1 Background information

The Irish Space Agency has launched Aigean, an Earth observation satellite to monitor an area around Lough Ree. Recently, rainfall has decreased in the area, and during the latest years droughts have become more frequent and more severe. With the instruments on board Aigean the scientific community will be able to obtain better data about the water levels and the erosion of the land, and therefore will be able to generate more accurate predictions.

However, the Irish Space Agency sadly hasn't provided any software tools to do this analysis!

Thankfully, Aoife O'Callaghan, a geology PhD student at the Athlone City Institute, has set the objective to solve this problem by creating an open-source package to analyse Aigean data. Aoife has some ideas of what she would like the package to do, but she doesn't have a research software development background beyond how to install and use Python libraries. That's why Aoife has contacted you!

You and your group members agree this is a great tool to offer to the community and have decided to put all your brains together to come up with an easy-to-use Python library to analyse and visualise Aigean satellite data.

What do we know? What do we have? What do we want?

1. Aigean has multiple instruments, as an starting point we only need to focus on the imagers and the radar.
2. There are three imagers on board of the spacecraft. Their only differences are in their resolution (how much area they cover per pixel) and their field-of-view (how much they can see in a single image).
3. The three imagers are called: Lir, Manannan and Fand.
 - **Lir** has the largest field-of-view, but the smaller resolution with a pixel size of 20 m per pixel;
 - **Manannan** provides a smaller field-of-view with a better resolution of 10 m per pixel; and
 - **Fand** has the smallest field-of-view but a very high resolution of 1 m per pixel.
4. The radar is called **Ecne** and it provides three measurements for the deepest areas in the region.
5. Each instrument provides data in a different format, but the imagers share a common set of metadata.
6. A number of images are taken every day, however not all the land is fully covered in a single day, it depends on the satellite orbits. Ecne, however, takes always measurements of the same points.
7. All the data is available at the Irish Space Agency webservice archive.
8. The Python library - **aigeanpy** - should be able to query, download, open, process and visualise the satellite images.
9. We want to create three command line tools to provide access to some functionality from outside Python.
10. We have a script from a post-doc of Aoife's group that implements the so-called **k-means algorithm** for clustering data points. We want to include it in our library too! It will help people to analyse different land areas based in their parameters.
11. We are also interested on how to make our code, specifically the k-means algorithm, more efficient. This will be used to analyse Ecne's data.
12. We want this tool to be used by any researcher, so it needs to be easy to install and use. This includes having good documentation about how to use it and how to acknowledge it in the publications that benefit from it.
13. And we also want to make it easier to others to contribute so we need to provide information about how we would like others to contribute.

Let's look at what we've got access to already:

1.1 The data archive webservice


The Irish Space Agency data archive is located at: <https://dokku-app.dokku.arc.ucl.ac.uk/isa-archive/> and their main page provides some information about how to query this service.

The website offers two services. One is used to **query the catalogue**, and the other to **download a file** from the archive.

The results from the query service are provided as JSON files with the properties of the observations found in the specified time range (and instruments). These files include information about the date and time of the observations, the instrument used, the field of view observed and the filename where that observation is stored. We can download that files using the filename as an argument to the download service. The format from the observation files vary depending on the instrument (specified in the following section).

Read the information on the [archive website](#) to understand how to query the service, what parameters are accepted and what are the defaults.

We need to create a set of tools within the Python package to query and download the files. They need to be available from `aigeanpy.net.query_isa` and `aigeanpy.net.download_isa`. They must accept all the parameters listed on the website. Additionally, the `download_isa` need to allow the user to specify where to download the file (`save_dir`).

 **Path** objects can also be used to **write_bytes** into a file. Check [Path](#) and [request's Response](#) documentation to see how you could write the content of a `requests.Response` into a file.

1.2 Different instruments, different file types

Data from each instrument is provided in a different type of file.

Lir uses the [Advanced Scientific Data Format](#) (ASDF). The `asdf` Python library can read them and extract the data and metadata from these files.

Manannan uses [Hierarchical Data Format 5](#) (HDF5). As with the `asdf`, this type of file contains the data and the metadata together. The [h5py Python library](#) can load them.

The Fand instrument stores the data in [numpy format](#) and the metadata in [JSON files](#). `numpy` files can be read from [NumPy's load](#) and the Python Standard Library provides support to [load JSON files](#). The archive provides that pair of files in a single zip file (for which Python Standard Library also provides a module to load: [zipfile](#)).

Finally, the Ecne instrument doesn't take images, but infers some measurements of the 300 deepest areas in the region. The measurements are turbulence, salinity and algal density for these points. They are stored in CSV.

💡 Ideally a user shouldn't need to unzip the file before loading it with the library. The `io.BytesIO` class can help you to load the file in memory. Take a look at how it's used on the [exemplar at the beginning of our course notes](#).

1.2.1 Getting the coordinates right

Arrays are stored in Python as (rows, columns). However, we normally refer to places in a map as (x, y) coordinates (with x running from left to right, and y running from bottom to top). Also when displaying an image in [matplotlib with imshow](#), by default, you'd get the axis as its origin is in the top-left corner and positive y-values going downwards. For this library, we will need to manage two type of coordinate systems: pixels and earth.

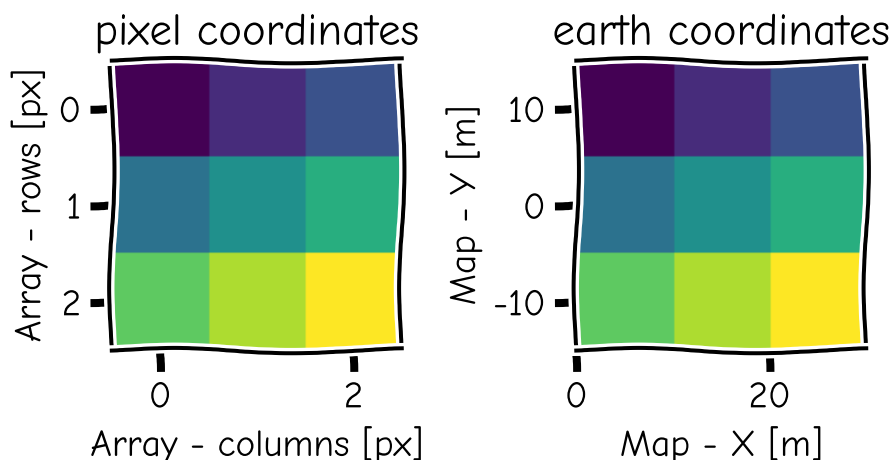


Figure 1: Difference between the two coordinate systems. The plot in the left shows the default when visualising an array, the (0, 0) is on the top left corner. On the right, the same array is shown as a map, which a set of (x, y) coordinates are represented within a pixel. In this case each pixel corresponds to 10 meters and the origin is within the second row, and the first column.

To ease the conversion between the coordinate systems you'll need to create two helper functions, which are called `earth_to_pixel` and `pixel_to_earth`.

⚠ Depending on the resolution of the image, a pixel may correspond to multiple earth coordinates. In such cases, `pixel_to_earth` should provide the coordinate of the centre of the pixel.

■ When asked to convert earth coordinates to pixels, and the coordinate falls in the centre of four pixels, then a possible approach is to give the top-right corner ■. But any other approach is correct too! Be consistent with your choice and test for edge cases.

Each image will come with an array of a particular size (and shape) and the metadata will provide the resolution (in meters per pixel), the earth x- and y-coordinates (in meters) as the (lower, upper) boundaries for each axis. The field of view (i.e., the difference between the boundaries) divided by the resolution should give you the shape of the array (in the (columns, rows) order).

```
>>> my_array.shape
(5, 6)
>>> my_metadata['resolution']
10
>>> my_metadata['xcoords']
(0, 60)
>>> my_metadata['ycoords']
(-10, 40)
>>> fov = (my_metadata['xcoords'][1] - my_metadata['xcoords'][0],
...        my_metadata['ycoords'][1] - my_metadata['ycoords'][0])
>>> fov
(60, 50)
>>> (fov[0] / my_metadata['resolution'], fov[1] / my_metadata['resolution'])
(6, 5)
```

1.2.2 An object to manipulate the data

Since the three imagers are observing the same geological properties, all of them could be grouped as of the same family. Aoife would like to get a function such that given a filename will generate an object where we could then proceed for the analysis. The function should be called `get_satmap` and used as follows:

```
>>> from aigeanpy.satmap import get_satmap
>>> lir_map = get_satmap('aigean_lir_20220222_170523.asdf')
>>> lir_map.data
array([[0 1 ...],
       [...   ],
       [... n]])
>>> lir_map.meta
{'resolution': 1,
 'xcoords': (20, 30),
 'ycoords': ...,
 ...
}
```

i.e., `data` will give the array, and `meta` will give a dictionary with the metadata of the file.

You are free to implement the `SatMap` class as you like, using any design patterns if needed. However, try to make it in a way that would make it simpler to add new imagers in the future.

1.2.3 Requirements

The class should provide the following attributes:

- `.fov` should give the field of view of the image loaded;
- `.centre` will give the earth coordinates of the centre of the image;
- `.shape` must provide the shape of the array as when called on a numpy array.

The class should allow to operate with + and - with other objects of the same type. But with the following restrictions:

- Only allow to + two images from the same instrument (and with the same resolution) taken on the same day.
- Whereas - could only happen when they are taken on different days, but still from the same instrument.

What does it mean to + or - images?

In this case, + means to collate the two images, as if trying to build a puzzle. For example if we got an image covering the (0,0)-(10,10) range and another from (12, 5)-(22,15), then we would end up with a “canvas” that goes from (0,0)-(22,15).

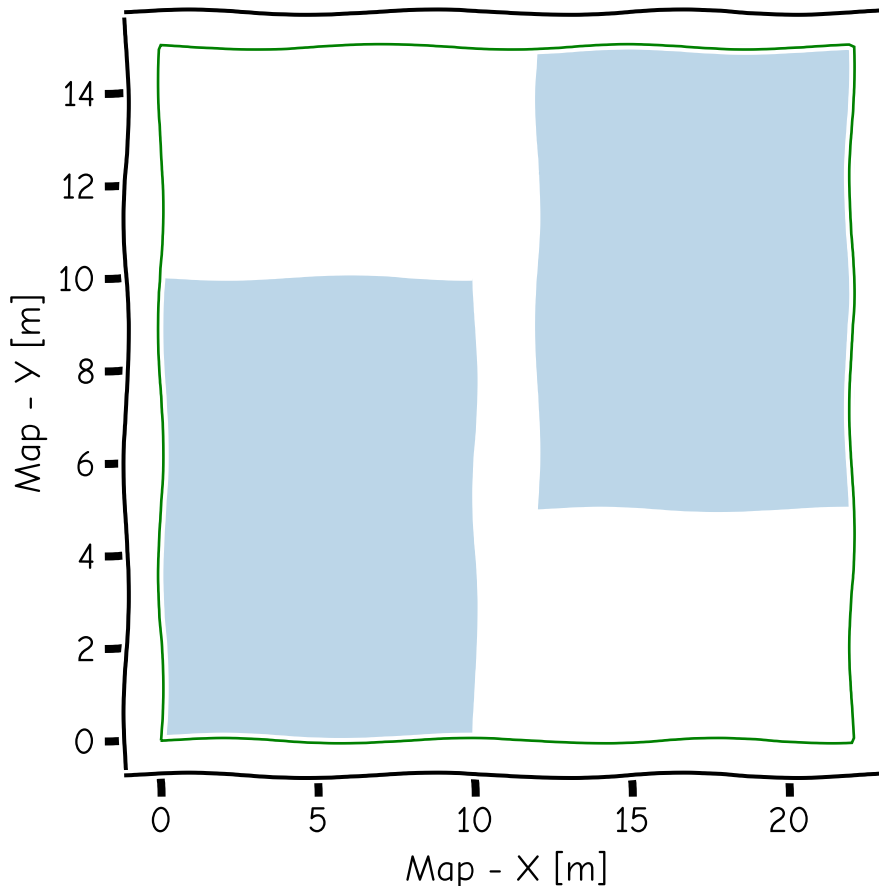


Figure 2: Example of adding two images together to create a larger image, shown by a green line. A bit of space has been left around the edges to aide visibility of the final image size.

When the two images are overlapping, the values of the overlapping areas should not be added. Otherwise it would give the wrong impression. Values should be the same when observed the same day.

On the other hand, when using - we want to obtain a difference image to measure change between the days. Therefore, in this case, it will only work when the data is overlapping. Therefore, if we have taken an image yesterday covering (0,0)-(10,10), and today another in the range of (5, 5)-(15, 15), the resultant image should be the difference between the both for the range (5, 5)-(10, 10).

In the cases that a user tries to add or subtract two satmaps when they are not from the same instrument, the program should raise an exception. Similarly if trying to add images from different days or subtract images

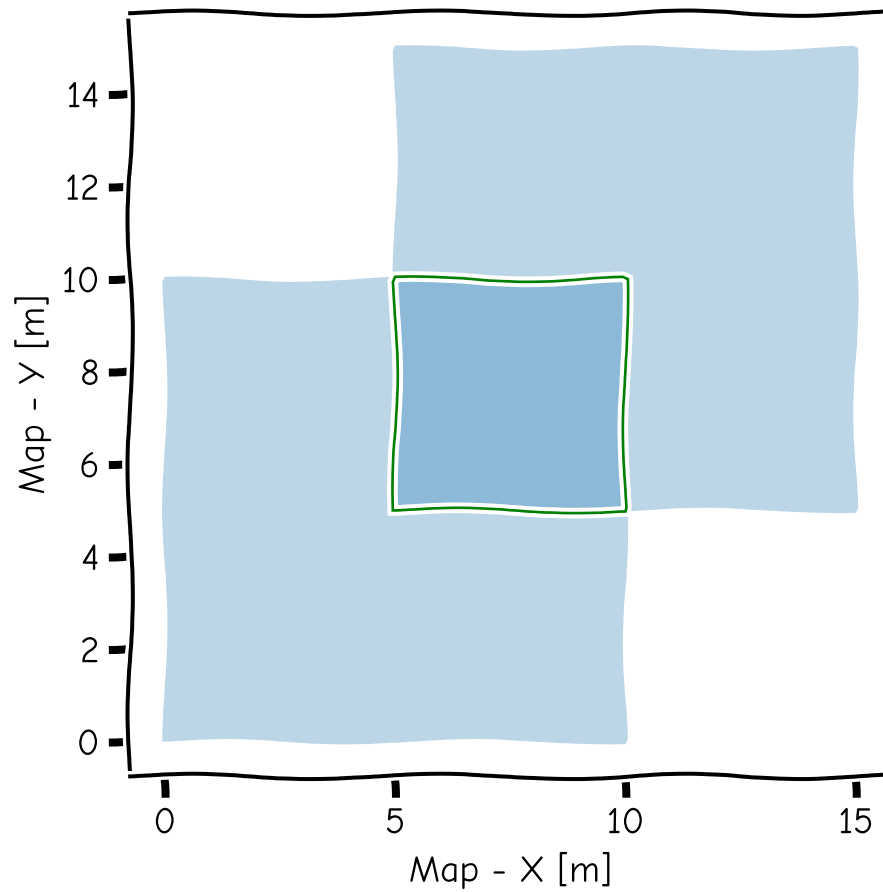


Figure 3: Example of subtracting two images together to create a difference image, shown by a green line. A bit of space has been left around the edges to aide visibility of the final image size.

from the same day. An exception should also be raised if the user attempts to subtract two non-overlapping satmaps.

The `SatMap` class will require two more methods: `.mosaic` and `.visualise`. `.mosaic` will allow to combine images as when using `+` but allowing mixing instruments (with different resolution!). The signature for `.mosaic` needs to be as follows:

```
def mosaic(self, otherMap: SatMap, resolution: int, padding: bool) -> SatMap:
    ...
```

If the `resolution` is not provided it should use the one of the two satmaps with larger detail, and expand the other to that level. `padding` should be `True` by default, i.e., to return an image with blanks (`NaNs`) on it. However, if `padding` is set as `False` the resultant image should only cover maximum portion without blanks on the image.

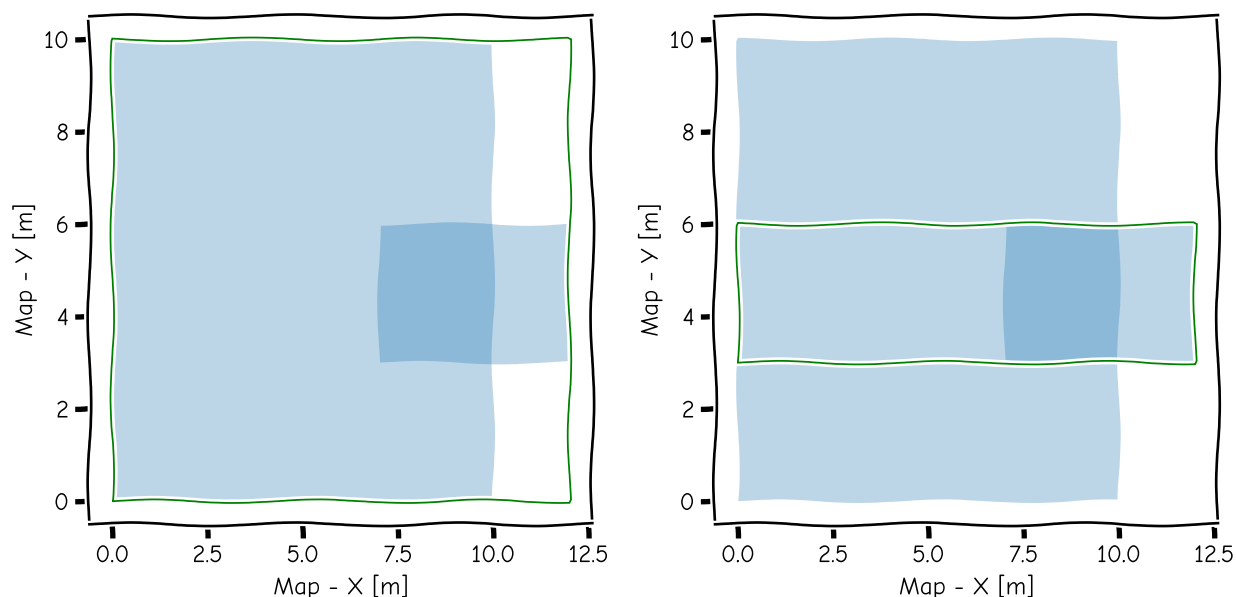


Figure 4: Example of a mosaic with (left) and without (right) padding. The result is shown by a green line.

i To rescale the arrays you can use [scikit-image's `rescale`](#), when you need to **downscale** it, they recommend you use `downscale_local_mean`. If using them, pay attention to the **mode** and **order** arguments as their default most probably won't give you what you want. Also pay attention to the **scale** or **factor** and how that relates to the changes in resolution we need. For example, if you want to change an array with resolution 1 to downsample it so it gets a resolution value of 2, then the factor would be $1/2$. If you want the opposite, to rescale an image of resolution 2 into 1, then the scale is 2, as you will be increasing by two every pixel in the original image.

Note that changes in resolution may necessitate changes in field-of-view. In this case, your code should raise an appropriate exception.

Finally, the last method required on `SatMap` is `.visualise`. It should have the following signature:

```
def visualise(self, save: bool, savepath: Union(Path, str), **kwargs):
```

`save` should be `False` by default, and `savepath` be the current directory. If `save` is set to `True`, then the image should not be displayed on the screen and saved in the required path with the following pattern: `{observatory}_{instrument}_{date}_{time}_{extra}.png`, where the date and time are formatted

as YYYYmmdd (e.g., 20221231) and HHMMSS (e.g., 120034) respectively.

The visualisation should show the axis as in earth coordinates and with the proper orientation of the image. Any perceptually uniform sequential colormap may be used, with the exception of when visualising a difference image - in this case, a diverging colormaps must be used. Check [matplotlib's colormap reference](#).

⚠ Remember to update the metadata when using `+`, `-` and `mosaic` so that the `.visualise` method knows how to display and save these properly.

1.3 Command line interfaces

The package should expose three **command-line interfaces**. One will query the archive webservice and download the latest image, another will print the metadata from one or more files on screen, and the last one will generate a mosaic from a list of files provided. Let's see them in more detail in the sections below.

All the command line interfaces should: - check the validity of the inputs before making any query or operation, and produce helpful messages if they are invalid. - produce a meaningful error message if, for example, there's no internet connection, the webapp is inaccessible, the images can't be put together, or the metadata can't be reached.

💡 When a command line interface fails it's common that the exit value is different than 0. Check the [sys.exit function](#) to know how to use it. Additionally, whatever error message (such as which files failed to be processed), are streamed to the `stderr`. You can do so by writing in the [sys.stderr file object](#).

1.3.1 Getting the latest image of the archive

The program should be named `aigean_today`. It shall accept two arguments, one to specify the instrument - if the instrument is not specified then it will download the most recent observation - and the other to whether we want to generate a png after download.

The syntax must follow this:

```
aigean_today [--instrument <instrument>] [--saveplot]
```

The acceptable inputs for `--instrument` (or `-i`) is one of the four instrument names (`lir`, `manannan`, `fand` or `ecne` - case insensitive!). If the `--saveplot` (or `-s`) flag is passed, then that indicates that the program should save the generated PNG figure. `--saveplot` will only take effect if the file downloaded is from one of the three imagers, otherwise it should message to the user that the file downloaded can't be visualised.

All the data needs to be downloaded or created in the directory where it's run.

1.3.2 Extracting the metadata information

This command should be invoked as `aigean_metadata`. The only arguments that can be passed to it is a file or list of them. The output will be the set of **key: value** available in the metadata, one per line. If multiple files are passed, then each line needs to contain the file name at the start as shown in the example below.

```
aigean_metadata <filename_i> [<filename_j> ...]
```

The output for one file should be like:

```
archive: ISA
observatory: Aigean
```



```
instrument: Lir
obs_date: 2022-12-01 16:29:14.832
...
```

and if more than one file is being parsed, then the output should be like:

```
aigean_lir_20221201_162914.asdf:archive: ISA
aigean_lir_20221201_162914.asdf:observatory: Aigean
aigean_lir_20221201_162914.asdf:instrument: Lir
aigean_lir_20221201_162914.asdf:obs_date: 2022-12-01 16:29:14.832
...
```

The order of the files should be shown as parsed to the command. This is useful as then you could run **grep** on it's output to filter some results, e.g.,

```
$ aigean_metadata * | grep "xcoords"
```

```
aigean_lir_20221130_061711.asdf:xcoords: (125, 300)
aigean_fan_20221101_103152.zip:xcoords: (-190, 190)
aigean_man_20221201_123942.hdf5:xcoords: (735, 1125)
aigean_man_20221201_171848.hdf5:xcoords: (420, 780)
...
```

If the program fails because some of the passed files are not aigean files or they are corrupted, the program should show the correct ones in screen and at the end print a list of the files that were incorrectly read. For example:

```
aigean_man_20221201_123942.hdf5:xcoords: (735, 1125)
aigean_man_20221201_171848.hdf5:xcoords: (420, 780)
```

```
These files failed while being processed
- aigean_fand_sample.zip
- aigean_lir_20221209_162914.asdf
- example.py
```

1.3.3 Creating a mosaic from the command line

The last programme, **aigean_mosaic**, will work similarly to the **aigean_metadata** command, in term of inputs. It will accept two or more filenames and process them all. Additionally it must also accept a resolution argument. The syntax for this command would look like:

```
aigean_mosaic --resolution 1 <filename_i> <filename_j> [<filename_k> ...]
```

The command should provide after completion the filename to the visualisation of the resulting satmap.

```
$ aigean_mosaic --resolution 10 aigean_man_20221201_*
aigean_man_20221201_123942_mosaic.png
```

If at least one of the input files fails then the command should error without providing any intermediate results.

1.4 The k-means algorithm

To analyse the Ecne data we will be using the k-means algorithm from the post-doc on Aoife's group. We will need to finish with an analysis function such as:

```
analysis.kmeans(filename: Union(Path, str), clusters: int, iterations: int) -> list
```

where should only accept CSV files like the ones from Ecne, a number of how many clusters the user wants and how many iterations to execute. The default for these values must be 3 and 10 respectively.

Here we have a brief description of how the algorithm works, but note that you do not need to understand it in detail: the focus of the exercise is on how the code and data are structured. Importantly, **you are not expected to (in fact, should not) make changes to the algorithm itself!**

The goal of the algorithm is to separate data points into groups. In the original version we got (`clustering.py`), each point is represented as a tuple of its coordinates. The algorithm then proceeds to form three clusters of nearby points by following these steps:

1. Pick three points randomly to be the initial centres of the clusters (line 10).
2. Assign each data point to a cluster. This is done by computing its distance from all cluster centres, and assign it to the nearest centre (lines 18-21).
3. Update the centre of each cluster by setting it to the average of all points assigned to the cluster (lines 23-25).
4. Repeat steps 2-3 for the desired number of times.

In the end, each cluster will (ideally) contain points that are close to each other, and far from the other clusters. The code then prints some basic statistics about the resulting clusters.

Note: You may find it useful or interesting to visualise the output of the algorithm, that is, how the points are clustered. To do that, you can use this or a similar piece of code at the end of the code provided:

```
from matplotlib import pyplot as plt
fig = plt.figure()
ax = fig.add_subplot(projection='3d')

for i in range(3):
    alloc_ps = [p for j, p in enumerate(ps) if alloc[j]==i]
    ax.scatter([a[0] for a in alloc_ps], [a[1] for a in alloc_ps], [a[2] for a in alloc_ps])

plt.show()
```

This will plot the three clusters in different colours. Naturally, you will need to adjust that code as you make changes to the clustering code if you want it to keep working!. This is purely for your own benefit – **the version you submit should not perform any plotting.**

💡 An extra note for those interested: this version of the code is non-deterministic; that is, every time you run it, you may get different results. If you want to remove this randomness – for example, to test that it still behaves as expected as you make changes –, you can think about setting the seed of the random generator in the beginning of the file. Note that this is in no way required for this exercise, and in fact, would change the behaviour of the code, so **don't include changes like this** in your final submission!

2 Your tasks

2.1 Team work

This is a collaborative effort. It's up to you how to distribute the work between the team. Read more at [How to work](#) and [Marking](#) sections below to understand how this part would be evaluated.

2.2 Interfaces and packaging

Your final product should be in the form of a Python package that other users can install. This should include documentation about the package and its contents.

2.2.1 Packaging

The final code should be in the form of a **package** called **aigeanpy**. Someone should be able to install it (and its dependencies) by navigating to the directory containing the package code and running

```
pip install .
```

Note: Do **NOT** upload your package to PyPI or any other public package repository!

2.2.2 Command line interface

The package should expose three **command-line interface** through the **aigeanpy** command. The user should be able to pass some arguments and pass some flags.

2.2.3 Library-style interface

Once the package is installed, it should expose the following functions:

1. a `net.query_isa` function which accepts the same parameters as the webservice and returns a data structure with the details of the response.
2. a `net.download_isa` function that accepts a filename obtained from querying the archive, and a `save_dir` parameter that accepts either a string or `Path` identifier with the location of where to save the downloaded file. For example,

```
>>> from aigeanpy.net import download_isa
>>> lir_map = download_isa('aigean_lir_20220222_170523.asdf', save_dir='..')
```

3. a `satmap.get_satmap` function which returns a `SatMap` object.

```
>>> from aigeanpy.satmap import get_satmap
>>> lir_map = get_satmap('aigean_lir_20220222_170523.asdf')
```

A `SatMap` object should provide the following attributes and methods:

- `.meta` a dictionary containing all the metadata of the observation.
- `.data` should return the array containing the data of that observation.
- `.shape` the shape of the array kept under `.data`.
- `.fov` the width and height of the image in earth coordinates (also named field of view).
- `.centre` the coordinates of the centre of the image in earth coordinates.
- `.mosaic(another_satmap)` combines both `SatMaps` into a single one.
- `.visualise()` generates a plot that can be saved as a PNG if requested.
- `lir_map + another_lirmap` will produce a `SatMap` with the combination of both.
- `lir_map - different_day_lirmap` will produce a difference `SatMap` where the overlap areas have values of the resulting difference.
- `print(lir_map)` should produce an output such as

```
>>> print(lir_map)
<AIGEAN/LIR: (x0,y0) - (x1,y1) r m/px>
```

where AIGEAN/LIR refers to **observatory/instrument** names, `(x0,y0)` and `(x1,y1)` are the bottom-left and top-right earth coordinates of the image, and `r` the resolution of that object in meters per pixels.

4. `analysis.kmeans()` should accept an Ecne file and return as many lists as clusters required, each containing the indices of the measurements for each group. It may accept the `clusters` and `iterations` arguments, using 3 and 10 respectively by default if they are not provided.

2.2.4 Documentation

The code should contain enough information that will explain to users what it does, how to run it, and any other important details. This information will come in a variety of formats.

Firstly, the code should have **docstrings** that explain, for example, what functions do and what arguments they take. The docstrings should be in the [numpy format](#). You should also use **comments** to clarify any particular points in the code that you feel require more explanation. The package you create should contain any **metadata files** that you find appropriate (as also discussed in the course notes).

The submission should include the sources to generate documentation pages using the **Sphinx** framework. Besides the automatically generated information of the methods available in the library it should also provide:

1. An description of what the package does,
2. a short user guide on how to install it and do a simple operation,
3. a guide for developers with information about how to contribute to the repository, how to test it, what style is followed, etc., and
4. a tutorial section with at least one example of how to use the library for a particular analysis. You can invent the tutorial as you like. Here are some examples:
 - Query and download images for a particular date
 - Generate an animation of 10 days of observation using difference images.
 - Find all the images in a directory that covers a particular area.
 - Measuring the area of water on a mosaic (using scikit-image).

The documentation should be in a directory named `docs`. We will run the following commands to build and check your documentation:

```
cd repository/docs
make html
python -m http.server -d build/html 8080
```

(after the above, your documentation should be viewable in a browser at <http://localhost:8080>)

2.3 Validation and robustness

The new code should include checks on the validity of the inputs, and **raise appropriate errors** if the users give inputs that don't make sense. At a minimum, the following situations should cause an error:

- a user tries to query using wrong date formats
- a user tries to make a query when they don't have a working internet connection
- a user tries to load a non Aigean file
- a user tries to combine (+ or -) **SatMaps** that are incompatible for these operations.
- a user tries to generate a difference image or a mosaic without padding and there's no any overlapping area.

As you create the code, we want you to add checks that ensure that it behaves correctly. This will be primarily achieved by **unit tests** in the **pytest** framework. The final code should include:

- tests for converting coordinates between earth and pixel coordinate systems;
- tests for any of the functions and public methods listed above;
- negative tests, where that makes sense;
- tests mocking services that require internet connection;
- usage examples in documentation strings that can be run using **doctest**.

You should also set up these tests to run automatically when you push to GitHub or open a pull request, using a **Continuous Integration** platform. You can use GitHub Actions for this. However, keep in mind that there's a finite number of credits, approximately 100 min per group and per month, so use them wisely - [see how many more credits testing on Mac costs compared with Linux](#). If you abuse the system you will be affecting other groups and you may be penalised for that (We will be monitoring the usage so we can warn you before it's too late).

Also think about what other measures you can take that help you check that the code does what is expected and handles user input sensibly.

2.4 Incorporating kmeans

2.4.1 Refactoring kmeans for readability

The kmeans that we have provided you with is not very clearly written. There are various changes that would make it more readable, as discussed in class and in the notes. We want you to pick **five** of those changes (or more if you feel like it!) and apply them to the code. Each change should be accompanied by a commit with a relevant message. It is fine if a change is split over multiple commits, but do not mix different types of changes in the same commit.

To keep easier track of these changes create an issue on your given repository with a checklist with a meaningful explanation of the changes you intent to do. Each time you apply one of these changes to the whole file, commit and add the commit number(s) to the item of the list that refers to.

For example, you can create a checklist in markdown as:

```
- [x] Modify something to make it readable; a9d018253655b3eeb5107b6cd0576ea44c5e5b8b
- [ ] Change this unknown thing to something known;
```

Note that you can edit an issue as many times you need to by clicking the three dots button on the top right corner of the issue comment.

For the submission, take a screenshot of that issue and name it `refactoring_steps.png`.

Do not introduce external libraries like **numpy** at this stage; focus on readability over performance. Remember not to change the algorithm itself, although you are free to improve, for instance, how the data is read, as long as the input and output remain the same.

As part of these changes, we want to make sure that the code is easy to run. Specifically, we want to be able to call the code in the file in two ways. Firstly, the file should include a function called **cluster**, which should take a list of points (as tuples) and, optionally, a number of iterations, and perform the analysis. Secondly, it should have a command-line interface (created using **argparse**) so that a user can call it by specifying a file and, optionally, a number of iterations, like this:

```
python clustering.py samples.csv [--iters 20]
```

In either case, if the number of iterations is not specified, it should be set to 10. Creating these interfaces does not count as one of the five changes you should make.

💡 **Tip:** Read through the code first before making any changes. Pick one possible improvement, apply it, then commit your changes before starting on another type of improvement. This will help ensure that your code still works as you change things, and keep your history clearer.

⚠️ **Note:** Do not set any precision when returning or printing the centres obtained.

2.5 Using NumPy

The initial code is written in plain Python, without using external libraries. One way of improving its performance is using a dedicated library for numerical computations, and this is what this part is concerned

with.

Create a new file called `clustering_numpy.py`. Starting from the plain Python version (either the version we gave you or your cleaned up version), introduce `numpy` structures and functions to make the code more concise and (hopefully!) faster. Your final version of the file should use `numpy` as much as possible, avoiding, for example, Python lists or `for` loops (where possible).

The new file should also expose the same interface as before: a command-line interface, and a `cluster` function. The user should be able to run it like this:

```
python clustering_numpy.py my_samples.csv [--iters 20]
```

2.6 Integration with aigeanpy

One of both versions (either with and without `numpy`) should be available to be called from the package via the `analysis.kmeans()` function. Note that the output of that function should not return the centres, but a lists with the measurement indices that belong to each group.

2.7 Measuring performance

We would like to see how the two versions of the code (with and without `numpy`) compare in terms of performance, especially as the input grows in size. For this part, create a script `performance.py` that runs the two versions on different input files which contain an increasing number of points, ranging from 100 to 10,000. Plot the time required against the size of the input (number of points), using a single plot for both versions. Include the script and the plot in your final submission, in a file called `performance.png`. Remember to label your axes and lines/points clearly, and make sure you commit the final plot to the repository!

(The execution time also depends on how many times we run the main loop of the algorithm. For this analysis, keep the number of iterations of the assign/recentre loop fixed to 10, as in the original version.)

Tip: You may find it useful to write a script that calls your code for the different input files, so that you can run the whole analysis with a single command. Some versions of the code may take a long time to run, so you may want to include some progress monitoring, such as printing a message when a file has been processed. In any case, make sure to not include this or other input/output when measuring time!

3 How to work

3.1 Collaboration

You will work in groups of 4-5 people to accomplish the tasks above. How you split the work within the group is entirely up to you. You may want to assign one aspect of the work (*e.g.*, tests) to each person, and have them be responsible for it throughout the project. Alternatively, you can decide to split the total work into smaller units (“sprints”), and within each of those allocate some of the smaller tasks to each person. Or you can come up with a different scheme!

Similarly, how you communicate is up to you. You can use some of the tools and practices we have mentioned in class (such as issue tracking), over whatever platform is convenient.

We will ask each team to meet with one instructor approximately halfway through the exercise, to see how the collaboration is going.

3.2 Suggestions for successful team work

1. Introduce yourself, either from the start or as opportunities arise share your strengths, weakness, and your values (what’s important for you and why)

2. Define a set of policy/rules about how to interact and what's expected and what's unacceptable from the group. You can adopt a code of conduct ([contributor covenant](#) [Carpentries Python Software Foundation's](#), ...)
3. Define roles for activities. These roles could be for the duration of the project, for a fragment of it, or changed daily. Some roles could be easier to transfer from day to day, for example [in each of your meetings you could have a facilitator, gatekeeper, timekeeper and a note-taker](#) and that won't disrupt the evolution of the project, whereas other roles, like a lead-developer, may need some global knowledge or skills that may not be easily and quickly transferable to change them with a high frequency.
4. Decide the set of tools to use to keep the whole team in the same page. Remember, the power is not in the tool you choose, but how effectively you use them (is it the right tool for the task?).
 1. Be aware of what is needed to run that tool (do you need to create a new account? Is it accessible for everyone? Are there some concerns such as privacy, political or moral of using that tool?)
 2. Spend time explaining how to effectively use that tool to everyone in the team in case it is new for them or they are unaware of certain features. Provide some resources for future references.
 3. Keep discussions (and most importantly decisions!) accessible to everyone. If possible use a common place to record decisions and track tasks. For example, having to scroll up and down through an endless chat or forum to find who is doing what is not very efficient.
5. Establish a methodology for reviewing and collaborating on the code that your team will produce (branch naming convention, branching strategy, who merges and when).
6. Communicate, communicate and communicate... and be careful with assuming that you or others have understood what has been said! Express what you've understood to get confirmation that your understanding is correct.

3.3 Version control

You are expected to use `git` throughout the project, and work on the GitHub repository that we will provide you with.

You should use the GitHub issue tracker to record planned work and issues that come up. Changes to the code should be made through pull requests rather than committing directly to the main branch. Make sure that pull requests are only merged after being reviewed and approved first. In your submission, include files that evidence your use of issues and pull requests. Specifically:

- `issues.png`: a screenshot of open and closed issues in your repository;
- `pr.png`: a screenshot of open and closed pull requests in your repository;
- `pr_link.txt`: a text file containing the URL of a pull request that you consider representative of your work.

To get a list of open and closed issues on the same page, go to the issues page of your repository and filter with only: `is:issue` (similarly use `is:pr` on the pull requests page).

3.3.1 Where is the repository for my group?

You should have received an invitation to join a GitHub repository named `aigeanpy-Working-Group-XX` where XX is your group number, under the [UCL-COMP0233-22-23](#) organisation. The repository is initially empty. If you don't see the invitation (check your emails or GitHub notifications), or it has expired, e-mail the teaching team (arc-teaching@ucl.ac.uk) with your GitHub username. You can also find your invite by going to: <https://github.com/UCL-COMP0233-22-23/aigeanpy-Working-Group-XX> changing XX for your group number.

3.3.2 Can we change the settings of our repository?

By default, you do not have enough permissions to alter the settings of your repository. This is to avoid the possibility that, for example, someone deletes it. A member of a group can be provided with increased ("admin") permissions to have full access to the repository settings. Agree between your groups who should

have these elevated permissions and email the teaching team, including in CC all the other members of your group. Tell us who you want to be given admin access and their github username.

Alternatively, if you want a setting changed, email us and we can make the change for you.

3.4 Deliverables

You must submit your exercise solution to Moodle as a single uploaded gzip format archive. (You must use only the tar.gz, not any other archiver, such as .zip or .rar. If we cannot extract the files from the submitted file with gzip, you will receive zero marks.)

To create a tar.gz file you need to run the following command on a bash terminal:

```
tar zcf filename.tar.gz directoryName
```

For example, if you group is working group 31, then you'll create your file as:

```
tar zcf working_group_31.tar.gz working_group_31
```

The folder structure inside your tar.gz archive must have a single top-level folder, whose folder name is your team name (*e.g.*, `working_group_31`), so that on running

```
tar xzf working_group_31.tar.gz
```

this folder appears. This top level folder must contain all the parts of your solution. Specifically, it should include a directory named **repository** with the following:

- the final code for the package, its tests and documentation sources
- the **performance.py** script and two performance plots, as described above, within a **benchmark** directory.

Note: We will only mark the **main** branch of the repository you submit!

In summary, your directory structure as extracted from the `working_group_xx.tar.gz` file should look like this:

```
working_group_xx/
├── repository/
│   ├── .git/
│   ├── <files and directories for the package, tests and documentation>
│   ├── benchmark/
│   │   ├── performance.py
│   │   └── performance.png
│   └── <other files you think are required>
├── refactoring_steps.png
├── issues.png
├── pr.png
└── pr_link.txt
```

Because this is a group assignment, only one member of the team needs to submit the the exercise solution to Moodle. Make sure you agree on who submits!

4 Marking

4.1 IPAC

Your submitted assignment will be marked as a single project for the whole group. As part of your final submission, you will also be required to assess the rest of your team. These two factors (group mark and peer evaluation) will determine your personal grade, using the IPAC methodology (Individual Peer Assessment of Contribution to group work).

We will ask you to evaluate your group members (and yourself) on the following criteria:

- communicating and sharing knowledge
- good team-working skills (such as respect, listening, leadership)
- quality of research and application of skills
- time and effort contributed
- overall value to the team's success

Note that the purpose of the scheme is not to set students against each other. Due to how IPAC works, falsely claiming that you have done most of the work and giving poor evaluations to your fellow group member is unlikely to artificially raise your own grade.

Once the group work is submitted, and soon after the submission deadline, the IPAC will be available on Moodle. This will need to be submitted individually by each member of the team. You'll have a week to fill it up.

4.2 Marking scheme

Total: 100 marks

- **Packaging and interfaces (20%)**
 - Installable package (2 marks)
 - Appropriate metadata (including version number and other properties) (1 mark)
 - Which packages code (but not tests), correctly. (1 mark)
 - Which specifies library dependencies (1 mark)
 - Which points to the entry point functions (1 mark)
 - Which allows to import `query_isa`, `download_isa`, `get_satmap` and `kmeans` from the library (2 mark)
 - Contains three other metadata files. Hint: How to use the package, how to reference it, who can copy it (3 marks)
 - Command-line interfaces correctly installed from the package (2 mark)
 - Command-line interfaces correctly passes the arguments (2 mark)
 - Documentation - Sphinx builds and provides appropriate documentation for all the functions and classes (2 marks)
 - Documentation - Includes the user and developers guide, and a use case tutorial (3 marks)
- **Code Structure, Style and Functionality (25%)**
 - Code readability (6 marks)
 - Clear code and file structure for all the functions and classes definitions (4 marks)
 - Avoids repetition through out all the package (3 marks)
 - Accepts passing the arguments required for each of the requirements asked (6 marks)
 - Objects are represented with the correct information (1 marks)
 - Objects methods and operations work as required (3 marks)
 - File and images produced are properly named and annotated (e.g., file names, axis labels, structure) (2 marks)
- **Validation and robustness (20%)**
 - Appropriate input checks meaningful error messages (3 marks)

- Meaningful error messages for wrong inputs or external problems (e.g., no-internet connection) (3 marks)
- Unit tests (positive and/or negative) for at least 12 functions or methods (6 marks)
- Mock tests (4 marks)
- Doctests (2 marks)
- Continuous integration (2 marks)
- **Refactoring and Performance (25%)**
 - At least 5 changes to the `clustering.py` provided code keeping its functionality (5 marks)
 - Exposing cluster function as described and provide a command line interface (4 marks)
 - Conversion to NumPy (works correctly, uses arrays, avoids loops, ...) (8 marks)
 - Performance plots and script (8 marks)
- **Ways of working (10%)**
 - Git commits of reasonable size with meaningful messages (4 marks)
 - Consistent use of issues and pull requests (6 marks)