pyveg

Release 1.0.0

Nick Barlow, Camila Rangel Smith, Samuel Van Stroud, Jesse F. A

Nov 19, 2020

THE PYVEG PACKAGE

The *pyveg* package implements functionality to download and process data from Google Earth Engine (GEE), and analyse images from periodic vegetation patterns (PVP) in arid and semi-arid ecosystems. PVP images are quantified using network centrality metrics. The results of the analysis can be used to search for typical early warning signals of an ecological collapse. Google Earth Engine Editor scripts are also provided to help researchers discover locations of ecosystems which may be in decline.

pyveg is being developed as part of a research project looking for evidence of early warning signals of ecosystem collapse using remote sensing data. pyveg allows such research to be carried out at scale, and hence can be an important tool in understanding changing arid and semi-arid ecosystem dynamics.

CHAPTER

ONE

THE PYVEG PACKAGE

1.1 Introduction

The pyveg package is developed to study the evolution of vegetation patterns in semi-arid environments using data downloaded from Google Earth Engine.

The code in this repository is intended to perform two main tasks:

1. Download and process GEE data:

- Download satellite data from Google Earth Engine (images and weather data).
  - Downloaded images are divided into 50x50 pixel sub-images, network centrality metrics are used to describe the pattern vegetation are then calculated on the sub-image level. Both colour (RGB) and Normalised Difference Vegetation Index (NDVI) images are downloaded and stored on the sub-image level.
  - For weather collections the precipitation and temperature "images" are averaged into a single value at each point in the time series.
- The download job is fully specified by a configuration file that can be generated by specifying the details of the data to be downloaded via prompts (satellite to use, coordinates, time period, number of time points, etc.).

2. Time series analysis on downloaded data:

- Time series analysis of the following metrics: raw NDVI mean pixel intensity across the image, vegetation network centrality metric, and precipitation.
  - The time series are processed (outliers removed and resampled to avoid gaps). All time series of each sub-image are aggregated into one summary time series that is used for analysis.
  - The summary time series is de-seasonalised and smoothed.
  - Residuals between the raw and de-seasonalised and smoothed time series are calculated and used for an early warning resilience analysis.
- Time series plots are produced, along with auto- and cross-correlation plots. Early warning signals are also computed using the ewstools package, including Lag-1 autocorrelation and standard deviation moving window plots. A sensitivity and significance analysis is also performed in order to determine whether any trends are statistically significant.
- Time series summary statistics and resilience metrics are saved into files.
- A PDF report is created showcasing the main figures resulting from the analyses.

Other functionalities:

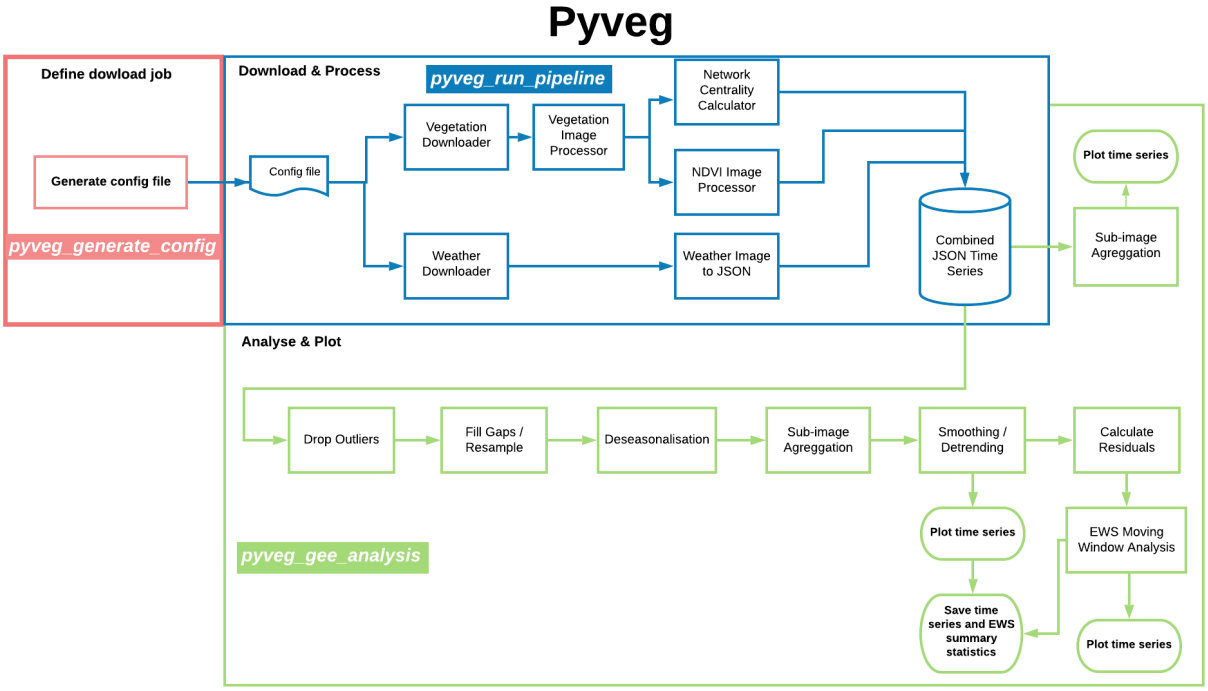
pyveg also has other minor functionalities:

- Analysis of a collection of summary data that has been created with the pyveg pipeline (downloading + time series analysis).

- Simulate the generation and evolution of patterned vegetation
- A stand-alone network centrality estimation for a 50x50 pixel image.
- A functionality to upload results to the Zenodo open source repository

1.1.1 pyveg flow

The diagram below represents the high level flow of the main functionalities of the `pyveg` package. For each main component there is a CLI console scripts defined, that is shown in the diagram.



The `pyveg` program flow.

The full ReadTheDocs documentation for this `pyveg` can be found in this link.

This page contains an installation guide, and some usage examples for this package.

1.2 Installation

`pyveg` requires Python 3.6 or greater. To install, start by creating a fresh `conda` environment.

```
conda create -n veg python=3.7
conda activate veg
```

Get the source.

```
git clone https://github.com/alan-turing-institute/monitoring-ecosystem-resilience.git
```

Enter the repository and check out a relevant branch if necessary (the `develop` branch contains the most up to date stable version of the code).

```
cd monitoring-ecosystem-resilience
git checkout develop
```

Install the package using `pip`.

```
pip install .
```

If you are using Windows and encounter issues during this stage, a solution may be found here: https://github.com/NREL/OpenOA/issues/37. If you plan on making changes to the source code, you can instead run `pip install -e .`.

Before using the Google Earth Engine API, you need to sign up with a Google account here, and authenticate. To authenticate, run

```
earthengine authenticate
```

A new browser window will open. Copy the token from this window to the terminal prompt to complete the authentication process.

1.2.1 Google Earth Engine

Google Earth Engine (GEE) is a powerful tool for obtaining and analysing satellite imagery. This directory contains some useful scripts for interacting with the Earth Engine API. The earth engine API is installed automatically as part of the `pyveg` package installation. If you wish to install it separately, you can follow the instructions here.

1.3 Downloading data from GEE with `pyveg`

1.3.1 Downloading data from GEE using the CLI

To run a `pyveg` download job, use

```
pyveg_run_pipeline --config_file <path to config>
```

The download job is fully specified by a configuration file, which you point to using the `--config_file` argument. A sample config file is found at `pyveg/configs/config_all.py`. You can also optionally specify a string to identify the download job using the `--name` argument.

Note that we use the GEE convention for coordinates, i.e. `(longitude,latitude)`.

Generating a download configuration file

To create a configuration file for use in the pyveg pipeline described above, use the command

```
pyveg_generate_config
```

this allows the user to specify various characteristics of the data they want to download via prompts. The list in order is as follows:

- `--configs_dir`: The path to the directory containing the config file, with a default option `pyveg/configs`.
- `--collection_name`: The name of the dataset used in the collection, either Sentinel2, or Landsat 8, 7, 5 or 4.

  - Sentinel2: Available from 2015-06-23 at 10m resolution.
  - Landsat8: Available from 2013-04-11 at 30m resolution.
  - Landsat7: Available from 1999-01-01 at 30m resolution.
  - Landsat5: Available from 1984-03-10 to 2013-01-31 at 60m resolution.
  - Landsat4: Available from 1982-07-16 to 1993-12-14 at 60m resolution.
- `--latitude`: The latitude (in degrees north) of the centre point of the image collection.
- `--longitude`: The longitude (in degrees east) of the centre point of the image collection.
- `--country`: The country (for the file name) can either be entered, or use the specified coordinates to look up the country name from the OpenCage database.
- `--start_date`: The start date in the format 'YYYY-MM-DD', the default is '2015-01-01' (or '2019-01-01' for a test config file).
- `--end_date`: The end date in the format 'YYYY-MM-DD', the default is today's date (or '2019-03-01' for a test config file).
- `--time_per_point`: The option to run the image collection either monthly ('1m') or weekly ('1w'), with the default being monthly.
- `--run_mode`: The option to run time-consuming functions on Azure ('batch') or running locally on your own computer ('local'). The default is local. For info about running on Azure go here.
- `--output_dir`: The option to write the output to a specified directory, with the default being the current directory.
- `--test_mode`: The option to make a test config file, containing fewer months and a subset of sub-images, with a default option to have a normal config file.
  - By choosing the test config file, the start and end dates (see below) are defaulted to cover a smaller time span.
  - It is recommended that the test config option should be used purely to determine if the options specified by the user are correct.
- `--n_threads`: Finally, how many threads the user would like to use for the time-consuming processes, either 4 (default) or 8.

For example:

```
pyveg_generate_config --configs_dir "pyveg/configs" --collection_name "Sentinel2" --
↪latitude 11.58 --longitude 27.94 --start_date "2016-01-01" --end_date "2020-06-30" -
↪-time_per_point "1m" --run_mode "local" --n_threads 4
```

This generates a file named `config_Sentinel2_11.58N_27.94E_Sudan_2016-01-01_2020-06-30_1m_local.py` along with instructions on how to use this configuration file to download data through the pipeline, in this case the following:

```
pyveg_run_pipeline --config_file pyveg/configs/config_Sentinel2_11.58N_27.94E_Sudan_
↪2016-01-01_2020-06-30_1m_local.py
```

Individual options can be specified by the user via prompt. The options for this can be found by typing `pyveg_generate_config --help`.

1.3.2 More Details on Downloading

During the download job, `pyveg` will break up your specified date range into a time series, and download data at each point in the series. Note that by default the vegetation images downloaded from GEE will be split up into 50x50 pixel images, vegetation metrics are then calculated on the sub-image level. Both colour (RGB) and Normalised Difference Vegetation Index (NDVI) images are downloaded and stored. Vegetation metrics include the mean NDVI pixel intensity across sub-images, and also network centrality metrics, discussed in more detail below.

For weather collections e.g. the ERA5, due to coarser resolution, the precipitation and temperature "images" are averaged into a single value at each point in the time series.

1.3.3 Rerunning partially succeeded jobs

The output location of a download job is datestamped with the time that the job was launched. The configuration file used will also be copied and datestamped, to aid reproducibility. For example if you run the job

```
pyveg_run_pipeline --config_file pyveg/configs/my_config.py
```

there will be a copy of `my_config.py` saved as `pyveg/configs/cached_configs/my_config_<datestamp>.py`. This also means that if a job crashes or timeouts partway through, it is possible to rerun, writing to the same output location and skipping parts that are already done by using this cached config file. However, in order to avoid a second datestamp being appended to the output location, use the `--from_cache` argument. So for the above example, the command to rerun the job filling in any failed/incomplete parts would be:

```
pyveg_run_pipeline --config_file pyveg/configs/cached_configs/my_config_<datestamp>.
↪py --from_cache
```

1.3.4 Using Azure for downloading/processing data

If you have access to Microsoft Azure cloud computing facilities, downloading and processing data can be sped up enormously by using batch computing to run many subjobs in parallel. See here for more details.

1.3.5 Downloading data using the API

Although `pyveg` has been mostly designed to be used with the CLI as shown above, we can also use `pyveg` functions through the API. A tutorial of how to download data this way is included in the `notebooks/tutorial_download_and_process_gee_images.ipynb` notebook tutorial.

1.4 Analysing the Downloaded Data with `pyveg`

1.4.1 Analysing the Downloaded Data using the CLI

Once you have downloaded the data from GEE, the `pyveg_gee_analysis` command allows you to process and analyse the output. To run:

```
pyveg_gee_analysis --input_dir <path_to_pyveg_download_output_dir>
```

The analysis code preprocesses the data and produces a number of plots. These will be saved in an `analysis/` subdirectory inside the `<path_to_pyveg_download_output_dir>` directory.

Note that in order to have a meaningful analysis, the dowloaded time series should have at least 4 points (and more thant 12 for an early warning analysis) and not being the result of a "test" config file, in this case the analysis fails.

The commands also allows for other options to be added to the execution of the script (e.g. run analysis from a downloaded data in Azure blob storage, define a different output directly, don't include a time series analysis, etc), which can be displayed by typing:

```
pyveg_gee_analysis --help
```

The analysis script executed with the `pyveg_gee_analysis` command runs the following steps:

Preprocessing

`pyevg` supports the following pre-processing operations:

- Identify and remove outliers from the time series.
- Fill missing values in the time series (based on a seasonal average), or resample the time series using linear interpolation between points.
- Smoothing of the time series using a LOESS smoother.
- Calculation of residuals between the raw and smoothed time series.
- De-seasonalising (using first differencing), and detrending using STL.

Plots

In the `analysis/` subdirectory, `pyveg` creates the following plots:

- Time series plots containing vegetation and precipitation time series (seasonal and de-seasonalised). Plots are labelled with the AR1 of the vegetation time series, and the maximum correlation between the Vegetation and precipitation time series.
- Auto-correlation plots for vegetation and precipitation time series (seasonal and de-seasonalised).
- Vegetation and precipitation cross-correlation scatterplot matrices.
- STL decomposition plots.
- Resilience analysis:
  - `ewstools` resilience plots showing AR1, standard deviation, skewness, and kurtosis using a moving window.
  - Smoothing filter size and moving window size Kendall tau sensitivity plots.
  - Significance test.

1.4.2 Running the analysis using the API

Although `pyveg` has been mostly designed to be used with the CLI as shown above, we can also use `pyveg` functions through the API. A tutorial of how to run the data analysis in this way is included in the `notebooks/tutorial_analyse_gee_data.ipynb` notebook tutorial.

1.5 Other functionalities of `pyveg`

1.5.1 Analysis summary statistics data

The `analyse_pyveg_summary_data.py` functionality processes collections of data produced by the main `pyveg` pipeline described in the section above (download + time series analysis for different locations and time periods) and creates a number of plots of the summary statistics of these time series.

To run this analysis in Python, there is an entrypoint defined. Type:

```
pyveg_analysis_summary_data --input_location  <path_to_directory_with_collection_
↪summary_statistics>
```

if you wish you can also specify the `outpur_dir` where plots will be saved. Type:

```
pyveg_analysis_summary_data --help
```

to see the extra options.

1.5.2 Pattern simulation

The `generate_patterns.py` functionality originates from some Matlab code by Stefan Dekker, Willem Bouten, Maarten Boerlijst and Max Rietkerk (included in the "matlab" directory), implementing the scheme described in:

Rietkerk et al. 2002. Self-organization of vegetation in arid ecosystems. The American Naturalist 160(4): 524-530.

To run this simulation in Python, there is an entrypoint defined. Type:

```
pyveg_gen_pattern --help
```

to see the options. The most useful option is the `--rainfall` parameter which sets a parameter (the rainfall in mm) of the simulation - values between 1.2 and 1.5 seem to give rise to a good range of patterns. Other optional parameters for `generate_patterns.py` allow the generated image to be output as a csv file or a png image. The `--transpose` option rotates the image 90 degrees (this was useful for comparing the Python and Matlab network-modelling code). Other parameters for running the simulation are in the file `patter_gen_config.py`, you are free to change them.

Running the pattern simulation using the API

Although `pyveg` has been mostly designed to be used with the CLI as shown above, we can also use `pyveg` functions through the API. A tutorial of how to run the simulation of the pattern generation in this way is included in here.

1.5.3 Network centrality

There is an entrypoint defined in `setup.py` that runs the *main* function of `calc_euler_characteristic.py`:

```
pyveg_calc_EC --help
```

will show the options.

- `--input_txt` allows you to give the input image as a csv, with one row per row of pixels. Inputs are expected to be "binary", only containing two possible pixel values (typically 0 for black and 255 for white).
- `--input_img` allows you to pass an input image (png or tif work OK). Note again that input images are expected to be "binary", i.e. only have two colours.
- `--sig_threshold` (default value 255) is the value above (or below) which a pixel is counted as signal (or background)
- `--upper_threshold` determines whether the threshold above is an upper or lower threshold (default is to have a lower threshold - pixels are counted as "signal" if their value is greater-than-or-equal-to the threshold value).
- `--use_diagonal_neighbours` when calculating the adjacency matrix, the default is to use "4-neighbours" (i.e. pixels immediately above, below, left, or right). Setting this option will lead to "8-neighbours" (i.e. the four neighbours plus those diagonally adjacent) to be included.
- `--num_quantiles` determines how many elements the output feature vector will have.
- `--do_EC` Calculate the Euler Characteristic to fill the feature vector. Currently this is required, as the alternative approach (looking at the number of connected components) is not fully debugged.

Note that if you use the `-i` flag when running python, you will end up in an interactive python session, and have access to the `feature_vec`, `sel_pixels` and `sc_images` variables.

Examples:

```
pyveg_calc_EC --input_txt ../binary_image.txt --do_EC
>>> sc_images[50].show() # plot the image with the top 50% of pixels (ordered by␣
↪subgraph centrality) highlighted.
>>> plt.plot(list(sel_pixels.keys()), feature_vec, "bo") # plot the feature vector vs␣
↪pixel rank
>>> plt.show()
```

1.5.4 Uploading results to the Zenodo open source repository

See here for more details.

CHAPTER

TWO

CONTRIBUTING

We welcome contributions from anyone who is interested in the project. There are lots of ways to contribute, not just writing code. See our Contributor Guidelines to learn more about how you can contribute and how we work together as a community.

CHAPTER

THREE

LICENCE

This project is licensed under the terms of the MIT software license.

CHAPTER

FOUR

SCRIPTS OF THE PYVEG PACKAGE

4.1 pyveg.scripts.analyse_gee_data module

This script analyses data previously download with *download_gee_data.py*. First, data is preprocessed using the *analysis_preprocessing.py* module. Plots are produced from the processed data.

pyveg.scripts.analyse_gee_data.**analyse_gee_data**(*input_location*, *input_location_type='local'*, *input_json_file=None*, *output_dir=None*, *do_time_series=True*, *do_spatial=False*, *upload_to_zenodo=False*, *upload_to_zenodo_test=False*)

Run analysis on dowloaded gee data

Parameters

- **input_location** (*str*) – Location of results_summary.json output from pyveg_run_pipeline, OR if input_location_type is *zenodo* or *zenodo_test*, the 2-digit coordinate_id representing the row in *coordinates.py*.
- **input_location_type** (*str*) – Can be 'local', 'azure', 'zenodo', or 'zenodo_test'.
- **input_json** (*str, optional. Full path to the results summary json file.*) –
- **output_dir** (*str,*) – Location for outputs of the analysis. If None, use input_location
- **do_time_series** (*bool*) – Option to run time-series analysis and do plots
- **do_spatial** (*bool*) – Option to run spatial analysis and do plots
- **upload_to_zenodo** (*bool*) – Upload results to the production Zenodo repo
- **upload_to_zenodo_test** (*bool*) – Upload results to the sandbox Zenodo repo

pyveg.scripts.analyse_gee_data.**main**()

CLI interface for gee data analysis.

pyveg.scripts.analyse_gee_data.**run_early_warnings_resilience_analysis**(*filename*, *output_dir*)

Run early warning resilience analysis on time series data. This function can be called on the detrended process data.

Parameters

- **filename** (*str*) – Path to the time series csv to analyse.

- **output_dir** (*str*) – Path to the directory to save plots to.

pyveg.scripts.analyse_gee_data.**run_time_series_analysis**(*filename*, *output_dir*, *detrended=False*)

Make plots for the time series data. This function can be called for the seasonal or detrended process data.

Parameters

- **filename** (*str*) – Path to the time series csv to analyse.
- **output_dir** (*str*) – Path to the directory to save plots to.

4.2 pyveg.scripts.analyse_pyveg_summary_data module

This script analyses summary statistics produced previously with *analyse_gee_data.py* for individual locations.

pyveg.scripts.analyse_pyveg_summary_data.**analyse_pyveg_summary_data**(*input_location*, *output_dir*)

Run analysis on summary statistics data

Parameters **input_location** (*str*) –

Location of summary statistics output files from analyse_gee_data.py. Can be 'zenodo' or 'zenodo_test' for the production or sandbox Zenodo depositions, or the path to the local directory containing the files.

output_dir: str, Location for outputs of the analysis. If None, use input_dir

pyveg.scripts.analyse_pyveg_summary_data.**barplot_plots**(*df*, *output_dir*)

Create barplots of summary data.

Parameters

- **df** (*dataframe*) – Dataframe of summary data.
- **output_dir** (*str*) – Path to the directory to save plots to.

pyveg.scripts.analyse_pyveg_summary_data.**boxplot_plots**(*df*, *output_dir*)

Create boxplots of summary data.

Parameters

- **df** (*dataframe*) – Dataframe of summary data.
- **output_dir** (*str*) – Path to the directory to save plots to.

pyveg.scripts.analyse_pyveg_summary_data.**correlation_plots**(*df*, *output_dir*)

Create correlation plots of summary data.

Parameters

- **df** (*dataframe*) – Dataframe of summary data.
- **output_dir** (*str*) – Path to the directory to save plots to.

pyveg.scripts.analyse_pyveg_summary_data.**main**()

CLI interface for gee data analysis.

pyveg.scripts.analyse_pyveg_summary_data.**process_input_data**(*input_location*)

Read all input summary statistics and transform data into a more analysis-friendly format

Parameters **input_location** (*str*) – Location of summary statistics output files from analyse_gee_data.py. Can be 'zenodo' or 'zenodo_test' to download from the production or sandbox Zenodo depository, or the path to a directory on the local filesystem.

Returns **df**

Return type DataFrame containing all the time-series data concatenated together.

pyveg.scripts.analyse_pyveg_summary_data.**scatter_plots**(*df*, *output_dir*)
Create scatter plots and correlation plots of summary data.

Parameters

- **df** (*dataframe*) – Dataframe of summary data.
- **output_dir** (*str*) – Path to the directory to save plots to.

4.3 pyveg.scripts.calc_euler_characteristic module

script to run the Euler Characteristic calculation on a given input image. The code that does the actual calculations is in src/subgraph_centrality.py

pyveg.scripts.calc_euler_characteristic.**main**()

4.4 pyveg.scripts.create_analysis_report module

pyveg.scripts.create_analysis_report.**add_rgb_images**(*mdFile*, *rgb_location*, *rgb_location_type*, *fig_count*)

pyveg.scripts.create_analysis_report.**add_time_series_plots**(*mdFile*, *analysis_plots_location*, *analysis_plots_location_type*, *satellite_suffix*)

pyveg.scripts.create_analysis_report.**create_markdown_pdf_report**(*analysis_plots_location*, *analysis_plots_location_type*, *rgb_location*, *rgb_location_type*, *do_timeseries*, *output_dir=None*, *collection_name=None*, *metadata=None*)

pyveg.scripts.create_analysis_report.**get_collection_and_suffix**(*collection_name*)
Lookup collection and suffix based on the name of the collection as used by GEE

Parameters **collection_name** (*str, GEE name of the collection, eg. 'COPERNICUS/S2'*) –

Returns

- **collection** (*str, user-friendly name of the collection, e.g. 'Sentinel2'*)

- **suffix** (*str, contraction of collection name, used in the filenames of plots*)

pyveg.scripts.create_analysis_report.**main**()
CLI interface for gee data analysis.

4.5 pyveg.scripts.crop_and_convert_images module

Use some of the functionality in src/image_utils.py as command-line script.

pyveg.scripts.crop_and_convert_images.**main**()
use command line arguments to specify input and output directories, and parameters for doing the cropping and conversion.

4.6 pyveg.scripts.download_from_azure module

4.7 pyveg.scripts.generate_config_file module

Generate a config file pyveg/configs/<config_filename> for use when running download and processing jobs with pyveg_run_pipeline –config_file pyveg/configs/<config_filename>

User specifies: * Coordinates OR id of location in coordinates.py * Date range * time per point * Satellite collection name (e.g. “Sentinel2”, “Landsat8”) * run mode (“local” or “batch”) * whether to run in ‘test’ mode (fewer dates, and only a few sub-images).

These can be given directly as command-line arguments, or the user will be prompted for them.

4.7.1 Usage

pyveg_generate_config

then respond to prompts, or

pyveg_generate_config –help

to see a list of command line options. (Note that command line options and prompted inputs can be mixed-and-matched).

pyveg.scripts.generate_config_file.**get_template_text**()

pyveg.scripts.generate_config_file.**main**()

pyveg.scripts.generate_config_file.**make_filename**(*configs_dir*, *test_mode*, *longitude*, *latitude*, *country*, *pattern_type*, *start_date*, *end_date*, *time_per_point*, *region_size*, *collection_name*, *run_mode*, *coords_id*)
Construct a filename from the specified parameters.

pyveg.scripts.generate_config_file.**make_output_location**(*coords_id*, *collection_name*, *latitude*, *longitude*, *country*)

pyveg.scripts.generate_config_file.**write_file**(*configs_dir*, *output_location*, *longitude*, *latitude*, *country*, *pattern_type*, *start_date*, *end_date*, *time_per_point*, *region_size*, *collection_name*, *run_mode*, *n_threads*, *test_mode=False*, *coords_id=None*)

Take the arguments, construct a filename, and write contents

4.8 pyveg.scripts.generate_pattern module

Translation of Matlab code to model patterned vegetation in semi-arid landscapes.

pyveg.scripts.generate_pattern.**main**()

4.9 pyveg.scripts.plot_feature_vectors module

4.10 pyveg.scripts.run_pyveg_module module

Build and run a pyveg pipeline based on a configuration json file.

pyveg.scripts.run_pyveg_module.**build_module**(*config_file*)

Load json config and instantiate modules

pyveg.scripts.run_pyveg_module.**configure_and_run_module**(*module*)

Call configure() run() on all the module

pyveg.scripts.run_pyveg_module.**main**()

4.11 pyveg.scripts.run_pyveg_pipeline module

Build and run a pyveg pipeline based on a configuration json file.

pyveg.scripts.run_pyveg_pipeline.**build_pipeline**(*config_file*, *from_cache=False*)

Load json config and instantiate modules

pyveg.scripts.run_pyveg_pipeline.**configure_and_run_pipeline**(*pipeline*)

Call configure() run() on all sequences in the pipeline.

pyveg.scripts.run_pyveg_pipeline.**main**()

4.12 pyveg.scripts.upload_to_zenodo module

Upload the results_summary.json or the outputs of the time-series analysis to the Zenodo open source platform for data www.zenodo.org.

Will create a zipfile, with a name based upon the coordinates and satellite collection, and upload it to a "deposition" via the zenodo API.

In addition to the main production zenodo API, there is also a "sandbox" for testing. Use the '–test_api' argument to use this.

pyveg.scripts.upload_to_zenodo.**create_new_deposition**(*use_sandbox=False*)

Create a new deposition, and populate it with the metadata from zenodo_config.py.

Parameters **use_sandbox** (*bool, if True use the "sandbox" API rather than the production one*) –

Returns **deposition_id** – is then copied into zenodo_config.py

Return type int, the ID of the newly created deposition. Recommended that this

pyveg.scripts.upload_to_zenodo.**main**()

pyveg.scripts.upload_to_zenodo.**upload_results_summary**(*json_location*, *json_location_type*, *use_test_api*)

Upload the results summary json from running pyveg pipeline to download and process data from GEE.

pyveg.scripts.upload_to_zenodo.**upload_summary_stats**(*csv_filepath*, *use_test_api*)

Typically called by the analyse_gee_data script, upload the results summary csv file.

4.13 Module contents

CHAPTER

FIVE

SOURCE CODE OF THE PYVEG PACKAGE

5.1 pyveg.src.analysis_preprocessing module

This module consists of methods to process downloaded GEE data. The starting point is a json file written out at the end of the downloading step. This module cleans, resamples, and reformats the data to make it ready for analysis.

pyveg.src.analysis_preprocessing.**detrend_data**(*dfs*, *period='MS'*)

Loop over each sub image time series DataFrames and remove time series seasonality by subtracting the previous year. Remove seasonality from precipitation data in the same way.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **period** (*str, optional*) –
- **Resample time series to this frequency and then infer** (`) – lag to use for deseasonalizing.

Returns Time series data for multiple sub-image with seasonality removed.

Return type dict of DataFrame

pyveg.src.analysis_preprocessing.**detrend_df**(*df*, *period='MS'*)

Remove seasonality from a DataFrame containing the time series for a single sub-image.

Parameters

- **df** (*DataFrame*) – Time series data for a single sub-image location.
- **period** (*str, optional*) –
- **Resample time series to this frequency and then infer** (`) – lag to use for deseasonalizing.

Returns Input with seasonality removed from time series columns.

Return type DataFrame

pyveg.src.analysis_preprocessing.**drop_veg_outliers**(*dfs*, *column='offset50'*, *sigmas=3.0*)

Loop over vegetation DataFrames and drop points in the time series that a significantly far away from the mean of the time series. Such points are assumed to be unphysical.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **column** (*str*) – Name of the column to drop outliers on.

- **sigmas** (*float*) – Number of standard deviations a data point has to be from the mean to be labelled as an outlier and dropped.

Returns Time series data for multiple sub-image locations with some values in *column* potentially set to NaN.

Return type dict of DataFrame

pyveg.src.analysis_preprocessing.**fill_veg_gaps**(*dfs*, *missing*)

Loop through sub-image time series and replace any gaps with mean value of the same month in other years.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **missing** (*dict of array*) – Missing time points where no sub-images were analyse for each veg dataframe in *dfs*.

pyveg.src.analysis_preprocessing.**get_missing_time_points**(*dfs*)

Find missing time points for each vegetation dataframe in *dfs*, and return a dict, with the same key as in *dfs*, but with values corresponding to missing dates.

Parameters **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.

Returns Missing time points for each vegetation df.

Return type dict

pyveg.src.analysis_preprocessing.**make_time_series**(*dfs*)

Given a dictionary of DataFrames which may contian many rows per time point (corresponding to the network centrality values of different sub-locations), collapse this into a time series by calculating the mean and std of the different sub- locations at each date.

Parameters **dfs** (*dict of DataFrame*) – Input DataFrame read by *read_json_to_dataframes*.

Returns **ts_list** – The time-series results averaged over sub-locations. First entry will be main dataframe of vegetation and weather. Second one (if present) will be historical weather.

Return type list of DataFrames

pyveg.src.analysis_preprocessing.**preprocess_data**(*input_json*, *output_basedir*, *drop_outliers=True*, *fill_missing=True*, *resample=True*, *smoothing=True*, *detrend=True*, *n_smooth=4*, *period='MS'*)

This function reads and process data downloaded by GEE. Processing can be configured by the function arguments. Processed data is written to csv.

Parameters

- **input_json** (*dict*) – JSON data created during a GEE download job.
- **output_basedir** (*str,*) – Directory where time-series csv will be put.
- **drop_outliers** (*bool, optional*) – Remove outliers in sub-image time series.
- **fill_missing** (*bool, optional*) – Fill missing points in the time series.
- **resample** (*bool, optional*) – Resample the time series using linear interpolation.
- **smoothing** (*bool, optional*) – Smooth the time series using LOESS smoothing.
- **detrend** (*bool, optional*) – Remove seasonal component by subtracting previous year.

- **n_smooth** (*int, optional*) – Number of time points to use for the smoothing window size.
- **period** (*str, optional*) – Pandas DateOffset string describing sampling frequency.

Returns

- **output_dir** (*str*) – Path to the csv file containing processed data.
- **defs** (*dict*) – Dictionary of dataframes.

pyveg.src.analysis_preprocessing.**read_json_to_dataframes**(*data*)
convert json data to a dict of DataFrame. :param data: :type data: dict, json data output from run_pyveg_pipeline

Returns A dict of the saved results in a DataFrame format. Keys are names of collections and the values are DataFrame of results for that collection.

Return type dict

pyveg.src.analysis_preprocessing.**read_results_summary**(*input_location*, *input_filename='results_summary.json'*, *input_location_type='local'*)
Read the results_summary.json, either from local storage, Azure blob storage, or zenodo.

Parameters

- **input_location** (*str, directory or container with results_summary.json in,*) – or coords_id if reading from zenodo
- **input_filename** (*str, name of json file, default is "results_summary.json"*) –
- **input_location_type** (*str: 'local' or 'azure' or 'zenodo' or 'zenodo_test'*) –

Returns **json_data**

Return type dict, the contents of results_summary.json

pyveg.src.analysis_preprocessing.**resample_data**(*dfs*, *period='MS'*)
Resample vegetation and rainfall DataFrames. Vegetation DataFrames are resampled at the sub-image level.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **period** (*string*) – Period for resampling.

Returns Resampled data.

Return type dict of DataFrame

pyveg.src.analysis_preprocessing.**resample_dataframe**(*df*, *columns*, *period='MS'*)
Resample and interpolate a time series dataframe so we have one row per time period.

Parameters

- **df** (*DataFrame*) – Dataframe with date as index.
- **columns** (*list*) – List of column names to resample. Should contain numeric data.
- **period** (*string*) – Period for resampling.

Returns DataFrame with resample time series in *columns*.

Return type DataFrame

pyveg.src.analysis_preprocessing.**resample_time_series**(*series*, *period='MS'*)
Resample and interpolate a time series dataframe so we have one row per time period (useful for FFT)

Parameters

- **df** (*DataFrame*) – Dataframe with date as index
- **col_name** (*string,*) – Identifying the column we will pull out
- **period** (*string*) – Period for resampling

Returns pandas Series with datetime index, and one column, one row per day

Return type Series

pyveg.src.analysis_preprocessing.**save_ts_summary_stats**(*ts_dirname*, *output_dir*, *metadata*)
Given a time series DataFrames (constructed with *make_time_series*), give summary statistics of all the avalaible time series.

Parameters

- **ts_dirname** (*str*) – Directory where the time series are saved.
- **output_dir** (*str*) – Directory to save the plots in.
- **metadata** (*dict*) – Dictionary with metadata from location

pyveg.src.analysis_preprocessing.**smooth_all_sub_images**(*df*, *column='offset50'*, *n=4*, *it=3*)
Perform LOWESS (Locally Weighted Scatterplot Smoothing) on the time series of a set of sub-images.

Parameters

- **df** (*DataFrame*) – DataFrame containing time series results for all sub-images, with multiple rows per time point and (lat,long) point.
- **column** (*string, optional*) – Name of the column in df to smooth.
- **n** (*int, optional*) – Size of smoothing window.
- **it** (*int, optional*) – Number of iterations of LOESS smoothing to perform.

Returns DataFrame of results with a new column containing a LOESS smoothed version of the column *column*.

Return type Dataframe

pyveg.src.analysis_preprocessing.**smooth_subimage**(*df*, *column='offset50'*, *n=4*, *it=3*)
Perform LOWESS (Locally Weighted Scatterplot Smoothing) on the time series of a single sub-image.

Parameters

- **df** (*DataFrame*) – Input DataFrame containing the time series for a single sub-image.
- **column** (*string, optional*) – Name of the column in df to smooth.
- **n** (*int, optional*) – Size of smoothing window.
- **it** (*int, optional*) – Number of iterations of LOESS smoothing to perform.

Returns The time-series DataFrame with a new column containing the smoothed results.

Return type DataFrame

pyveg.src.analysis_preprocessing.**smooth_veg_data**(*dfs*, *column='offset50'*, *n=4*)
Loop over vegetation DataFrames and perform LOESS smoothing on the time series of each sub-image.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **column** (*str*) – Name of the column to drop outliers and smooth.
- **n** (*int*) – Number of neighbouring point to use in smoothing

Returns Time series data for multiple sub-image locations with new column for smoothed data and ci.

Return type dict of DataFrame

pyveg.src.analysis_preprocessing.**store_feature_vectors**(*dfs*, *output_dir*)
Write out all feature vector information to a csv file, to be read later by the feature vector plotting script.

Parameters

- **dfs** (*dict of DataFrame*) – Time series data for multiple sub-image locations.
- **output_dir** (*str*) – Path to directory to save the csv.

5.2 pyveg.src.azure_utils module

pyveg.src.azure_utils.**check_blob_exists**(*blob_name*, *container_name*, *bbs=None*)
See if a blob already exists for this account name.

pyveg.src.azure_utils.**check_container_exists**(*container_name*, *bbs=None*)
See if a container already exists for this account name.

pyveg.src.azure_utils.**create_container**(*container_name*, *bbs=None*)

pyveg.src.azure_utils.**delete_blob**(*blob_name*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**download_rgb**(*container*, *rgb_dir*)

Parameters

- **container** (*str, the container name*) –
- **rgb_dir** (*str, directory into which to put image files.*) –

pyveg.src.azure_utils.**download_summary_json**(*container*, *json_dir*)

Parameters

- **container** (*str, the container name*) –
- **json_dir** (*str, temporary directory into which to put json file.*) –

pyveg.src.azure_utils.**get_blob_to_tempfile**(*filename*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**get_sas_token**(*container_name*, *token_duration=1*, *permissions='READ'*, *bbs=None*)

pyveg.src.azure_utils.**list_directory**(*path*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**read_image**(*blob_name*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**read_json**(*blob_name*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**remove_container_name_from_blob_path**(*blob_path*, *container_name*)
Get the bit of the filepath after the container name.

pyveg.src.azure_utils.**retrieve_blob**(*blob_name*, *container_name*, *destination='/tmp/'*, *bbs=None*)
use the BlockBlobService to retrieve file from Azure, and place in destination folder.

pyveg.src.azure_utils.**sanitize_container_name**(*orig_name*)
only allowed alphanumeric characters and dashes.

pyveg.src.azure_utils.**save_image**(*image*, *output_location*, *output_filename*, *container_name*, *format='png'*, *bbs=None*)
Given a PIL.Image (list of pixel values), save to requested filename - note that the file extension will determine the output file type, can be .png, .tif, probably others...

pyveg.src.azure_utils.**save_json**(*data*, *blob_path*, *filename*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**write_file_to_blob**(*file_path*, *blob_name*, *container_name*, *bbs=None*)

pyveg.src.azure_utils.**write_files_to_blob**(*path*, *container_name*, *blob_path=None*, *file_endings=[]*, *bbs=None*)
Upload a whole directory structure to blob storage. If we are given 'blob_path' we use that - if not we preserve the given file path structure. In both cases we take care to remove the container name from the start of the blob path

5.3 pyveg.src.batch_utils module

Functions for submitting batch jobs. Currently only support Azure Batch. Largely taken from https://github.com/Azure-Samples/batch-python-quickstart

pyveg.src.batch_utils.**add_task**(*task_id*, *job_name*, *input_script*, *input_config*, *input_azure_config*, *task_dependencies*, *batch_service_client=None*)
add the batch task to the job.

Parameters

- **task_id** (*str, unique ID within this job for the task*) –
- **job_name** (*str, name for the job - usually Sequence name + timestamp*) –
- **input_script** (*ResourceFile corresponding to bash script uploaded to blob storage*) –
- **input_config** (*ResourceFile corresponding to json config for this task uploaded to blob storage*) –
- **input_azure_config** (*ResourceFile corresponding to azure config, uploaded to blob storage*) –
- **task_dependencies** (*list of str, task_ids of any tasks that this one depends on*) –
- **batch_service_client** (*BatchServiceClient*) –

pyveg.src.batch_utils.**check_task_failed_dependencies**(*task*, *job_id*, *batch_service_client=None*)
If a task depends on other task(s), and those have failed, the job will not be able to run.

Parameters

- **task** (*azure.batch.models.CloudTask, the task we will look at dependencies for*) –

- **job_id**(*str, the unique ID of the Job.*)–
- **batch_service_client** (*BatchServiceClient – will create if not provided.*)–

Returns

- *True if the job depends on other tasks that have failed (or those* – tasks depend on failed tasks)
- *False otherwise*

pyveg.src.batch_utils.**check_tasks_status**(*job_id*, *task_name_prefix=''*, *batch_service_client=None*)

For a given job, query the status of all the tasks.

Returns task_status – num_success: int, successfully completed num_failed: int, completed but with non-zero exit code num_running: int, currently running num_waiting: int, in "active" state num_cannot_run: int, in "active" state, but with dependent tasks that failed.

Return type dict, containing the following keys/values:

pyveg.src.batch_utils.**create_batch_client**()

pyveg.src.batch_utils.**create_job**(*job_id*, *pool_id=None*, *batch_service_client=None*)

Creates a job with the specified ID, associated with the specified pool.

Parameters

- **job_id** (*str, ID for the job – will typically be module or sequence name +timestamp*)–
- **pool_id** (*str, ID for the pool. If not provided, use the one from azure_config.py*)–
- **batch_service_client** (*BatchServiceClient instance. Create one if not provided.*)–

pyveg.src.batch_utils.**create_pool**(*pool_id*, *batch_service_client=None*)

Creates a pool of compute nodes.

Parameters

- **pool_id**(*str, identifier for the pool*)–
- **batch_service_client** (*azure.batch.BatchServiceClient, A Batch service client.*)–

pyveg.src.batch_utils.**delete_job**(*job_id*, *batch_service_client=None*)

Removes a job, and associated tasks.

pyveg.src.batch_utils.**delete_pool**(*pool_id=None*, *batch_service_client=None*)

Removes a pool of batch nodes

pyveg.src.batch_utils.**prepare_for_task_submission**(*job_name*, *config_container_name*, *batch_service_client*, *blob_client*)

Create pool and job if not already existing, and upload the azure config file and the bash script used to run the batch job.

Parameters

- **job_name**(*str, ID of the job*)–
- **batch_service_client** (*BatchServiceClient to interact with Azure batch.*)–

Returns **input_azure_config, input_script** – and batch_commands.sh scripts, uploaded to blob storage.

Return type ResourceFiles corresponding to the azure_config.py

pyveg.src.batch_utils.**print_task_output**(*batch_service_client*, *job_id*, *encoding=None*)
Prints the stdout.txt file for each task in the job.

Parameters

- **batch_client** (*batchserviceclient.BatchServiceClient*) – The batch client to use.
- **job_id** (*str*) – The id of the job with task output files to print.

pyveg.src.batch_utils.**submit_tasks**(*task_dicts*, *job_name*)
Submit batch jobs to Azure batch.

task_dicts: list of dicts, [{ "task_id": <task_id>, "config": <config_dict>, "depends_on": [<task_ids>]

}]

job_name: str, should identify the sequence generating the jobs

pyveg.src.batch_utils.**upload_file_to_container**(*block_blob_client*, *container_name*, *file_path*)
Uploads a local file to an Azure Blob storage container.

Parameters

- **block_blob_client** (*azure.storage.blob.BlockBlobService*) – A blob service client.
- **container_name** (*str*) – The name of the Azure Blob storage container.
- **file_path** (*str*) – The local path to the file.

Return type *azure.batch.models.ResourceFile*

Returns A ResourceFile initialized with a SAS URL appropriate for Batch

tasks.

pyveg.src.batch_utils.**wait_for_tasks_to_complete**(*job_id*, *timeout=60*, *batch_service_client=None*)
Returns when all tasks in the specified job reach the Completed state.

Parameters

- **batch_service_client** (*azure.batch.BatchServiceClient*) – A Batch service client.
- **job_id** (*str*) – The id of the job whose tasks should be to monitored.
- **timeout** (*timedelta*) – The duration to wait for task completion. If all

tasks in the specified job do not reach Completed state within this time period, an exception will be raised.

5.4 pyveg.src.combiner_modules module

Modules that can consolidate inputs from different sources and produce combined output file (typically JSON).

class pyveg.src.combiner_modules.**CombinerModule**(*name=None*)
Bases: *pyveg.src.pyveg_pipeline.BaseModule*

class pyveg.src.combiner_modules.**VegAndWeatherJsonCombiner**(*name=None*)
Bases: *pyveg.src.combiner_modules.CombinerModule*

Expect directory structures like: <something>/<input_veg_location>/<date>/network_centralities.json <something>/<input_weather_location>/RESULTS/weather_data.json

check_output_dict(*output_dict*)
For all the keys (i.e. dates) in the vegetation time-series, count how many have data for both veg and weather

combine_json_lists(*json_lists*)
If for example we have json files from the NetworkCentrality and NDVI calculators, all containing lists of dicts for sub-images, combine them here by matching by coordinate.

get_metadata()
Fill a dictionary with info about this job - coords, date range etc.

get_veg_time_series()
Combine contents of JSON files written by the NetworkCentrality and NDVI calculator Modules. If we are running in a Pipeline, get the expected set of date strings from the vegetation sequence we depend on, and if there is no data for a particular date, make a null entry in the output.

get_weather_time_series()

run()

set_default_parameters()
See if we can set our input directories from the output directories of previous Sequences in the pipeline. The pipeline (if there is one) will be a grandparent, i.e. self.parent.parent and the names of the Sequences we will want to combine should be in the variable self.depends_on.

5.5 pyveg.src.coordinate_utils module

Collection of utility functions for manipulating coordinates and their string representations.,

pyveg.src.coordinate_utils.**coords_dict_to_coords_string**(*coords*)
Given a dict of long/lat values, return a string, rounding to 2 decimal places.

pyveg.src.coordinate_utils.**coords_list_to_coords_string**(*coords*)
Given a list or tuple of [long, lat], return a string, rounding to 2 decimal places.

pyveg.src.coordinate_utils.**find_coords_string**(*file_path*)
Parse a file path using a regular expresion to find a substring that looks like a set of coordinates, and return that.

pyveg.src.coordinate_utils.**get_region_string**(*coords*, *region_size*)
Given a set of (long,lat) coordinates, and the size of a square region in long,lat space, return a string in the format expected by GEE.

Parameters

- **coords** (*list of floats, [longitude,latitude]*) –
- **region_size** (*float, size of each side of the region, in degrees*) –

Returns **region_string** – representing four corners of the region.

Return type str, string representation of list of four coordinates,

pyveg.src.coordinate_utils.**get_sub_image_coords**(*coords*, *region_size*, *x_parts*, *y_parts*)
If an image is divided into sub_images, return a list of coordinates for all the sub-images.

Parameters

- **coords** (*list of floats, [long,lat]*) –

- **region_size** (*float, size of square image in degrees long,loat*) –
- **x_parts** (*int, number of sub-images in x-direction*) –
- **y_parts** (*int, number of sub-images in y-direction*) –

Returns sub_image_coords

Return type list, of lists of floats [[long,lat],..]

pyveg.src.coordinate_utils.**lookup_country**(*latitude*, *longitude*)
Use the OpenCage API to do reverse geocoding

5.6 pyveg.src.data_analysis_utils module

Data analysis code including functions to read the .json results file, and functions analyse and plot the data.

pyveg.src.data_analysis_utils.**ar1_moving_average_time_series**(*series*, *length=1*)
Calculate an AR1 time series using a moving average

Parameters

- **series** (*pandas Series*) – Time series observations.
- **length** (*int*) – Length of the moving window in number of observations.

Returns pandas Series with datetime index, and one column, one row per date

Return type pandas Series

pyveg.src.data_analysis_utils.**calculate_ci**(*data*, *ci_level=0.99*)
Calculate the confidence interval on the mean for a set of data. :param data: Series of data to calculate the confidence interval of the mean. :type data: Series :param ci_level: Size of the confidence interval to calculate :type ci_level: float, optional

Returns Confidence interval value where the CI is [mu - h, mu + h], where mu is the mean.

Return type float

pyveg.src.data_analysis_utils.**cball**(*x=range(1, 13)*, *alpha=1.5*, *n=150.0*, *xbar=8.0*, *sigma=2.0*)
Calculates the Crystal Ball pdf on the values 1 to 12 by default (i.e. monthly) Default parameter values give a fit close to those we would expect from offset50 time series :param x: Index values going from 1 to the length of the annual time series :type x: Time series :param alpha: Parameters used in Crystal Ball pdf calculation :type alpha: Model parameters, int :param n: Parameters used in Crystal Ball pdf calculation :type n: Model parameters, int :param xbar: Parameters used in Crystal Ball pdf calculation :type xbar: Model parameters, int :param sigma: Parameters used in Crystal Ball pdf calculation :type sigma: Model parameters, int

Returns The values of the Crystal Ball pdf for each index of x

Return type ndarray

pyveg.src.data_analysis_utils.**cball_parfit**(*p0*, *timeseries*, *plot_name='CB_fit.png'*, *output_dir=''*)
Uses least squares regression to optimise the parameters in cball to fit the timeseries supplied. The supplied time series should be the original series as this function finds the mean annual ts and reverses and normalises it :param p0: A list a parameters (alpha, n, xbar, sigma) to use in the Crystal Ball calculation as an initial estimate :type p0: Initial parameters, list :param timeseries: Original time series to calculate mean annual time series on, reverse and normalise

and then use to optimise the parameters on

Parameters

- **plot_name** (*string*) – Name for the data/fit comparison plot
- **output_dir** (*str*) – Directory to save the plots in.

Returns

- *ndarray* – A list of optimised parameters (alpha, n, xbar, sigma)
- *int* – A indication that the optimisation works (if output is 1,2,3 or 4 then ok)
- *float* – The residuals from the best CB fit

pyveg.src.data_analysis_utils.**coarse_dataframe**(*geodf*, *side_square*)

Coarse the granularity of a dataframe by grouping lat,long points that are close to each other in a square of L = size_square :param geodf: Input dataframe :type geodf: Dataframe :param side_square: Side of the square :type side_square: integer

Returns A coarser dataframe

Return type A dataframe

pyveg.src.data_analysis_utils.**convert_to_geopandas**(*df*)

Given a pandas DatFrame with *lat* and *long* columns, convert to geopandas DataFrame. :param df: Pandas DatFrame with *lat* and *long* columns. :type df: DataFrame

Returns

Return type geopandas DataFrame

pyveg.src.data_analysis_utils.**create_lat_long_metric_figures**(*geodf*, *metric*, *output_dir*)

From input data-frame with processed network metrics create 2D gird figure for each date available using Geopandas. :param geodf: Input dataframe :type geodf: GeoDataframe :param metric: Variable to plot :type metric: string :param output_dir: Directory to save the figures

Returns

Parameters **----------** –

pyveg.src.data_analysis_utils.**decay_rate**(*x*, *resolution=12*, *method='basic'*)

Calculates the decay rate between the max and min values of a time series. :param x: Time series to calculate decay rate on. mean_annual_ts is calculated

on this series within this function so raw time series is expected.

Parameters

- **resolution** (*int*) – Number of values each year in a time series (12 is monthly for example)
- **method** (*'basic' (default) or 'adjusted'*) – A choice on whether to calculate the decay rate on the mean annual time series calculated within the function or to adjust the time series such that the min value is set to 1 by substracting the minimum plus 1 of the mean annual time series (useful for offset50 values)

Returns The decay rate value

Return type float

pyveg.src.data_analysis_utils.**early_warnings_null_hypothesis**(*series*, *indicators=['var', 'ac']*, *roll_window=0.4*, *smooth='Lowess'*, *span=0.1*, *band_width=0.2*, *lag_times=[1]*, *n_simulations=1000*)

Function to estimate the significance of the early warnings analysis by performing a null hypothesis test. The function estimate distributions of trends in early warning indicators from different surrogate timeseries generated after fitting an ARMA(p,q) model on the original data. The trends are estimated by the nonparametric Kendall tau correlation coefficient and can be compared to the trends estimated in the original timeseries to produce probabilities of false positives. The function returns a dataframe that contains the Kendall tau rank correlation estimates for orignal data and surrogates. :param series: Time series observations. :type series: pandas Series :param indicators: The statistics (leading indicator) selected for which the sensitivity analysis is perfomed. :type indicators: list of strings :param roll_window: Rolling window size as a proportion of the length of the time-series

data.

Parameters

- **smooth** (*string*) – Type of detrending. It can be {'Gaussian', 'Lowess', 'None'}.
- **span** (*float*) – Span of time-series data used for Lowess filtering. Taken as a proportion of time-series length if in (0,1), otherwise taken as absolute.
- **band_width** (*float*) – Bandwidth of Gaussian kernel. Taken as a proportion of time-series length if in (0,1), otherwise taken as absolute.
- **lag_times** (*list of int*) – List of lag times at which to compute autocorrelation.
- **n_simulations** (*int*) – The number of surrogate data. Default is 1000.

Returns A dataframe that contains the Kendall tau rank correlation estimates for each indicator estimated on each surrogate dataset.

Return type DataFrame

pyveg.src.data_analysis_utils.**early_warnings_sensitivity_analysis**(*series*, *indicators=['var', 'ac']*, *winsizerange=[0.1, 0.8]*, *incrwinsize=0.1*, *smooth='Gaussian'*, *bandwidthrange=[0.05, 1.0]*, *spanrange=[0.05, 1.1]*, *incrbandwidth=0.2*, *incrspanrange=0.1*)

Function to estimate the sensitivity of the early warnings analysis to the smoothing and windowsize used. The function returns a dataframe that contains the Kendall tau rank correlation estimates for the rolling window sizes (winsize variable) and bandwidths or span sizes depending on the de-trending (smooth variable). This function is inspired in the sensitivity_ews.R function from Vasilis Dakos, Leo Lahti in the early-warnings-R package: https://github.com/earlywarningtoolbox/earlywarnings-R. :param series: Time series observations. :type series: pandas Series :param indicators: The statistics (leading indicator) selected for which the sensitivity analysis is perfomed. :type indicators: list of strings :param winsizerange: Range of the rolling window sizes expressed as ratio of the timeseries length (must be numeric between 0 and 1). Default is 0.25 - 0.75. :type winsizerange: list of float :param incrwinsize: Increments the rolling window size (must be numeric between 0 and 1). Default is 0.25. :type incrwinsize: float :param smooth: Type of detrending. It can be {'Gaussian', 'Lowess', 'None'}. :type smooth: string :param bandwidthrange: Range of the bandwidth used for the Gaussian kernel when gaussian filtering is selected. It is expressed as percentage of the timeseries length (must be numeric between 0 and 100). Default is 5% - 100%. :type bandwidthrange: list of float :param spanrange: Parameter that controls the degree of Lowess smoothing (numeric between 0 and 1). Default is 0.05 - 1. :type spanrange: list of float :param incrbandwidth: Size to increment the bandwidth used for the Gaussian kernel when gaussian filtering is applied. It is expressed as percentage of the timeseries length (must be numeric between 0 and 1). Default is 0.2. :type incrbandwidth: float :param incrspanrange: Size to increment the the span used for the Lowess smoothing :type incrspanrange: float

Returns A dataframe that contains the Kendall tau rank correlation estimates for the rolling window sizes (winsize variable) and bandwidths or span sizes depending on the de-trending (smooth variable).

Return type DataFrame

pyveg.src.data_analysis_utils.**err_func**(*params*, *ts*)

Calculates the difference between the cball function with supplied params and a supplied time series of the same length. err_func is used within cball_parfit function below where full time series needs to be supplied :param params: Parameters used in Crystal Ball pdf calculation

alpha, n, xbar, sigma

Parameters **ts** (*Time series*) – Time series to compare output of cball function to

Returns Residuals/differences between Crytal Ball pdf and supplied time series

Return type ndarray

pyveg.src.data_analysis_utils.**exp_model_fit**(*x*, *resolution=12*, *method='basic'*)

Fits an exponential model from the maximum to the minimum of the mean annual time series. A raw time series is expected as an input. :param x: Time series to calculate decay rate on. mean_annual_ts is calculated

on this series within this function so raw time series is expected.

Parameters

- **resolution** (*int*) – Number of values each year in a time series (12 is monthly for example)
- **method** (*'basic' (default) or 'adjusted'*) – A choice on whether to fit the expoenential model on the mean annual time series calculated within the function or to adjust the time series such that the min value is set to 1 by substracting the minimum plus 1 of the mean annual time series (useful for offset50 values)

Returns The coefficient values from the exponential model fit

Return type ndarray

pyveg.src.data_analysis_utils.**fft_series**(*time_series*)
Perform Fast Fourier Transform on an input series (assume one row per day). :param time_series: :type time_series: a pandas Series with one row per day, and datetime index (which we'll ignore)

Returns **xvals, yvals** – Ready to be plotted directly in a matplotlib plot.

Return type np.arrays of frequencies (1/day) and strengths in frequency space.

pyveg.src.data_analysis_utils.**get_AR1_parameter_estimate**(*ys*)
Fit an AR(1) model to the time series data and return the associated parameter of the model. :param ys: Input time series data. :type ys: array

Returns

- *float* – The parameter value of the AR(1) model..
- *float* – The parameter standard error

pyveg.src.data_analysis_utils.**get_ar1_var_timeseries_df**(*series*, *window_size=0.5*)
Given a time series calculate AR1 and variance using a moving window. Put the two resulting time series into a new DataFrame and return the result. :param series: Time series observations. :type series: pandas Series :param window_size: Size of the moving window as a fraction of the time series length. :type window_size: float (optional)

Returns The AR1 and variance results in a time series dataframe.

Return type DataFrame

pyveg.src.data_analysis_utils.**get_confidence_intervals**(*df*, *column*, *ci_level=0.99*)
Calculate the confidence interval at each time point of a DataFrame containing data for a large image. :param df: Time series data for multiple sub-image locations. :type df: DataFrame :param column: Name of the column to calculate the CI of. :type column: str :param ci_level: Size of the confidence interval to calculate :type ci_level: float, optional

Returns Time series data for multiple sub-image locations with added column for the ci.

Return type DataFrame

pyveg.src.data_analysis_utils.**get_correlation_lag_ts**(*series_A*, *series_B*, *window_size=0.5*)
Given two time series and a lag betweent them, calculate the lagged correlation between the two time series using a moving window. Additionally calculate the lag of the maximum precipitation using the moving window.. :param series_A: Observations of the first time series. :type series_A: pandas Series :param series_B: Observations of the second time series. :type series_B: pandas Series :param window_size: Size of the moving window as a fraction of the time series length. :type window_size: float (optional)

Returns Lagged corrleation and lag which maximises the correlation time series.s

Return type DataFrame

pyveg.src.data_analysis_utils.**get_corrs_by_lag**(*series_A*, *series_B*)

pyveg.src.data_analysis_utils.**get_datetime_xs**(*df*)
Return the date column of *df* as datetime objects.

pyveg.src.data_analysis_utils.**get_kendell_tau**(*ys*)
Kendall's tau gives information about the trend of the time series. It is just a rank correlation test with one variable being time (or the vector 1 to the length of the time series), and the other variable being the data itself. A tau value of 1 means that the time series is always increasing, whereas -1 mean always decreasing, and 0 signifies no overall trend. :param ys: Input time series data. :type ys: array

Returns

- *float* – The value of tau.

- *float* – The p value of the rank correlation test.

pyveg.src.data_analysis_utils.**get_max_lagged_cor**(*dirname*, *veg_prefix*)

Convenience function which returns the maximum correlation as a function of lag (using a file saved earlier). :param dirname: Path to the *analysis/* directory of the current analysis job. :type dirname: str :param veg_prefix: Compact representation of the satellite collection name used to

obtain vegetation data.

Returns Max correlation, and lag, for smoothed and unsmoothed vegetation time series.

Return type tuple

pyveg.src.data_analysis_utils.**mean_annual_ts**(*x*, *resolution=12*)

Calculate mean annual time series from time series. Also fills in missing values by linear interpolation. NB Fails if there is missing value at the start or end. :param x: Time series to calculate mean annual time series for :type x: Time series :param resolution: Number of values each year in a time series (12 is monthly for example) :type resolution: float

Returns Array of length equal to resolution that is the mean annual time series

Return type ndarray

pyveg.src.data_analysis_utils.**moving_window_analysis**(*df*, *output_dir*, *window_size=0.5*)

Run moving window AR1 and variance calculations for several input time series time series. :param df: Input time series DataFrame containing several time series. :type df: DataFrame :param output_dir: Path output plotting directory. :type output_dir: str :param window_size: Size of the moving window as a fraction of the time series length. :type window_size: float (optional)

Returns AR1 and variance time-series for each of the input time series.

Return type DataFrame

pyveg.src.data_analysis_utils.**network_figure**(*df*, *date*, *metric*, *vmin*, *vmax*, *output_dir*)

Make 2D heatmap plot with network centrality measures :param df: Input dataframe :type df: Dataframe :param date: Date to be plot :type date: String :param metric: Which metric is going to be plot :type metric: string :param vmin: Colorbar minimum values :type vmin: int :param vmax: Colorbar max values :type vmax: int :param output_dir: Directory where to save the plots :type output_dir: string

pyveg.src.data_analysis_utils.**reverse_normalise_ts**(*x*)

Takes what is expected to be a mean annual time series (from mean_annual_ts), arranges it so the first value is the last, reverses it and then normalises it. It is to be used within cball function below. :param x: Time series reverse and normalise. Assumed this is from mean_annual_ts output :type x: time series

Returns The reversed and normalised time series

Return type ndarray

pyveg.src.data_analysis_utils.**stl_decomposition**(*series*, *period=12*)

Run STL decomposition on a pandas Series object. :param series: The observations to be deseasonalised. :type series: Series object :param period: Length of the seasonal period in observations. :type period: int (optional)

pyveg.src.data_analysis_utils.**variance_moving_average_time_series**(*series*, *length*)

Calculate a variance time series using a moving average :param series: Time series observations. :type series: pandas Series :param length: Length of the moving window in number of observations. :type length: int

Returns pandas Series with datetime index, and one column, one row per date.

Return type pandas Series

pyveg.src.data_analysis_utils.**write_slimmed_csv**(*dfs*, *output_dir*, *filename_suffix=''*)

pyveg.src.data_analysis_utils.**write_to_json**(*filename*, *out_dict*)
Create or append the contents of *out_dict* to json file *filename*. :param filename: Output json filename. :type filename: array :param out_dict: Information to save. :type out_dict: dict

5.7 pyveg.src.date_utils module

Useful functions for manipulating dates and date strings, e.g. splitting a period into sub-periods.

When dealing with date strings, ALWAYS use the ISO format YYYY-MM-DD

pyveg.src.date_utils.**assign_dates_to_tasks**(*date_list*, *n_tasks*)
For batch jobs, will want to split dates as evenly as possible over some number of tasks.

pyveg.src.date_utils.**find_mid_period**(*start_date*, *end_date*)
Given two strings in the format YYYY-MM-DD return a string in the same format representing the middle (to the nearest day)

Parameters

- **start_date** (*str, date in format YYYY-MM-DD*) –
- **end_date** (*str, date in format YYYY-MM-DD*) –

Returns **mid_date**

Return type str, mid point of those dates, format YYYY-MM-DD

pyveg.src.date_utils.**get_date_range_for_collection**(*date_range*, *coll_dict*)
Return the intersection of the date range asked for by the user, and the min and max dates for that collection.

Parameters

- **date_range** (*list or tuple of strings, format YYYY-MM-DD*) –
- **coll_dict** (*dictionary containing min_date and max_date kyes*) –

Returns

Return type tuple of strings, format YYYY-MM-DD

pyveg.src.date_utils.**get_date_strings_for_time_period**(*start_date*, *end_date*, *period_length*)
Use the two functions above to slice a time period into sub-periods, then find the mid-date of each of these.

Parameters

- **start_date** (*str, format YYYY-MM-DD*) –
- **end_date** (*str, format YYYY-MM-DD*) –
- **period_length** (*str, format '<integer><d|w|m|y>', e.g. 30d*) –

Returns **periods** – each of which is the mid-point of a sub-period

Return type list of strings in format YYYY-MM-DD,

pyveg.src.date_utils.**get_num_n_day_slices**(*start_date*, *end_date*, *days_per_chunk*)
Divide the full period between the start_date and end_date into n equal-length (to the nearest day) chunks. The size of the chunk is defined by days_per_chunk. Takes start_date and end_date as strings 'YYYY-MM-DD'. Returns an integer with the number of possible points avalaible in that time period]

pyveg.src.date_utils.**get_time_diff**(*date1*, *date2*, *units='years'*)
calculate the time difference between two dates, :param date1: :type date1: str, dates in format YYYY-MM-DD

:param date2: :type date2: str, dates in format YYYY-MM-DD :param units: :type units: str, can be "years", "months", "days"

Returns **time_diff**

Return type int, difference in times, in specified units

pyveg.src.date_utils.**slice_time_period**(*start_date*, *end_date*, *period_length*)
Slice a time period into chunks, whose length is determined by the period_length, which will be e.g. '30d' for 30 days, or '1m' for one month.

Parameters

- **start_date** (*str, format YYYY-MM-DD*) –
- **end_date** (*str, format YYYY-MM-DD*) –
- **period_length** (*str, format '<integer><d|w|m|y>', e.g. 30d*) –

Returns **periods** – each of which is the start and end of a sub-period

Return type list of lists of strings in format YYYY-MM-DD,

pyveg.src.date_utils.**slice_time_period_into_n**(*start_date*, *end_date*, *n*)
Divide the full period between the start_date and end_date into n equal-length (to the nearest day) chunks. Takes start_date and end_date as strings 'YYYY-MM-DD'. Returns a list of tuples [(chunk0_start,chunk0_end),. . .]

5.8 pyveg.src.download_modules module

5.9 pyveg.src.file_utils module

pyveg.src.file_utils.**consolidate_json_to_list**(*json_dir*, *output_dir=None*, *output_filename=None*)
Load all the json files (e.g. from individual sub-images), and return a list of dictionaries, to be written out into one json file.

Parameters

- **json_dir** (*str, full path to directory containing temporary json files*) –
- **output_dir** (*str, full path to desired output directory.*) – Can be None, in which case no output written to disk.
- **output_filename** (*str, name of the output json file.*) – Can be None, in which case no output written to disk.

Returns **results**

Return type list of dicts.

pyveg.src.file_utils.**construct_filename_from_metadata**(*metadata*, *suffix*)
Given a dictionary of metadata, construct a filename. Will be used for the results summary json, and the summary stats csv as they are uploaded to Zenodo.

pyveg.src.file_utils.**construct_image_savepath**(*output_dir*, *collection_name*, *coords*, *date_range*, *image_type*)
Function to abstract output image filename construction. Current approach is to create a new dir inside *output_dir* for the satellite, and then save date and coordinate stamped images in this dir.

pyveg.src.file_utils.**download_and_unzip**(*url*, *output_tmpdir*)
Given a URL from GEE, download it (will be a zipfile) to a temporary directory, then extract archive to that same dir. Then find the base filename of the resulting .tif files (there should be one-file-per-band) and return that.

Parameters

- **url** (*str, URL of zipfile on GEE server.*) –
- **output_tmpdir** (*str, full path of directory into which to unpack zipfile.*) –

Returns **tif_filenames**

Return type list of strings, the full paths to unpacked tif files.

pyveg.src.file_utils.**get_filepath_after_directory**(*path*, *dirname*, *include_dirname=False*)
Return part of a filepath from a certain point onwards. e.g. if we have path /a/b/c/d/e/f and we say dirname=c, then this will return d/e/f if include_dirname==False, or c/d/e/f if it is True.

Parameters

- **path** (*str, full filepath*) –
- **dirname** (*str, delimeter, from where we will take the remaining filepath*) –
- **include_dirname** (*bool, if True, the returned path will have dirname as its root.*) –

pyveg.src.file_utils.**get_tag**()
Get the git tag currently checked out.

pyveg.src.file_utils.**save_image**(*image*, *output_dir*, *output_filename*, *verbose=False*)
Given a PIL.Image (list of pixel values), save to requested filename - note that the file extension will determine the output file type, can be .png, .tif, probably others...

pyveg.src.file_utils.**save_json**(*out_dict*, *output_dir*, *output_filename*, *verbose=False*)
Given a dictionary, save to requested filename -

pyveg.src.file_utils.**split_filepath**(*path*)

5.10 pyveg.src.gee_interface module

5.11 pyveg.src.image_utils module

Modify, and slice up tif and png images using Python Image Library Needs a relatively recent version of pillow (fork of PIL): ` pip install --upgrade pillow `

pyveg.src.image_utils.**adaptive_threshold**(*img*)
Threshold a grayscale image using the mean pixel value of a local area to set the threshold at each pixel location. At the moment set above average brightness pixels to the max (255) and vice versa for below average brightness pixels.

@param img 2D numpy array representing a grayscale image @return thresholded image

pyveg.src.image_utils.**check_image_ok**(*rgb_image*, *black_pix_threshold=0.05*)
Check the quality of an RGB image. Currently checking if we have > X% pixels being masked. This indicates problems with cloud masking in previous steps.

Parameters **rgb_image** (*Pillow.Image*) – Input image to check the quality of

Returns *True* if image passes quality requirements, else *False*.

Return type bool

pyveg.src.image_utils.**combine_tif**(*band_dict*)
Read tif files - one per specified band, and rescale and combine pixel values to r,g,b values betweek 0 and 255 in a combined output image.

Parameters **band_dict** (*dict, format {'<r|g|b>': {'band': <band_name>, 'filename': <filename>}}*) –

Returns **new_img**

Return type PIL Image, 8-bit rgb image.

pyveg.src.image_utils.**compare_binary_image_files**(*filename1*, *filename2*)
Wrapper for compare_binary_images that opens and closes the image files.

pyveg.src.image_utils.**compare_binary_images**(*image1*, *image2*)
Return the fraction of pixels that are the same in the two images.

pyveg.src.image_utils.**convert_to_bw**(*input_image*, *threshold*, *invert=False*)
Given an RGB input, apply a threshold to each pixel. If pix(r,g,b)>threshold, set to 255,255,255, if <threshold, set to 0,0,0

pyveg.src.image_utils.**convert_to_rgb**(*band_dict*)
If we are given three or more bands, interpret the first as red, the second as green, the third as blue, and scale them to be between 0 and 255 using the combine_tif function. If we are only given one band, use the scale_tif function to scale the range of input values to between 0 and 255 then apply this to all of r,g,b

Parameters **band_dict** (*dict, format {'<r|g|b|rgb>': {'band': <band_name>, 'filename': <filename>}}*) –

pyveg.src.image_utils.**create_gif_from_images**(*directory_path*, *output_name*, *string_in_filename=''*)
Loop through a directory and convert all images in it into a gif chronologically

Parameters

- **directory_path** – directory where all the files are.
- **output_name** – name to be given to the output gif
- **string_in_filename** – select only files that containsa particular string, default is "" which implies all in directory files are selected

Returns

pyveg.src.image_utils.**crop_and_convert_all**(*input_dir*, *output_dir*, *threshold=470*, *num_x=50*, *num_y=50*)
Loop through a whole directory and crop and convert to black+white all files within it.

pyveg.src.image_utils.**crop_and_convert_to_bw**(*input_filename*, *output_dir*, *threshold=470*, *num_x=50*, *num_y=50*)
Open an image file, convert to monochrome, and crop into sub-images.

pyveg.src.image_utils.**crop_image_nparts**(*input_image*, *n_parts_x*, *n_parts_y=None*)
Divide an image into n_parts_x*n_parts_y equal smaller sub-images.

pyveg.src.image_utils.**crop_image_npix**(*input_image*, *n_pix_x*, *n_pix_y=None*, *region_size=None*, *coords=None*)
Divide an image into smaller sub-images with fixed pixel size. If region_size and coordinates are provided, we want to return the coordinates of the sub-images along with the sub-images themselves.

pyveg.src.image_utils.**hist_eq**(*img*, *clip_limit=2*)
Perform contrast limited local histogram equalisation on an imput image.

@param img 2D numpy array representing a grayscale image @param clip_limit controls the strength of the equalisation @return 2D numpy array representing the equalised image

pyveg.src.image_utils.**image_all_same_colour**(*image*, *colour=(255, 255, 255)*, *threshold=0.99*)
Return true if all (or nearly all) pixels are same colour

pyveg.src.image_utils.**image_file_all_same_colour**(*image_filename*, *colour=(255, 255, 255)*, *threshold=0.99*)
Wrapper for image_all_same_colour that opens and closes the image file

pyveg.src.image_utils.**image_file_to_array**(*input_filename*)
Read an image file and convert to a 2D numpy array, with values 0 for background pixels and 255 for signal. Assume that the input image has only two colours, and take the one with higher sum(r,g,b) to be "signal".

pyveg.src.image_utils.**image_from_array**(*input_array*, *output_size=None*, *sel_val=200*)
Convert a 2D numpy array of values into an image where each pixel has r,g,b set to the corresponding value in the array. If an output size is specified, rescale to this size.

pyveg.src.image_utils.**invert_binary_image**(*image*)
Swap (255,255,255) with (0,0,0) for all pixels

pyveg.src.image_utils.**median_filter**(*img*, *r=3*)
Convolve a median filter over the image.

@param img 2D numpy array representing a grayscale image @param r the size of the grid to convolve @return 2D numpy array representing the smoothed image

pyveg.src.image_utils.**numpy_to_pillow**(*numpy_image*)
Convert a 2D numpy array to a PIL Image object.

@param img 2D numpy array to convert @return PIL Image object

pyveg.src.image_utils.**pillow_to_numpy**(*pil_image*)
Convert a PIL Image object to a numpy array (used by openCV).

@param img PIL Image object to convert @return 2D or 3D numpy array (depending on input image)

pyveg.src.image_utils.**plot_band_values**(*input_filebase*, *bands=['B4', 'B3', 'B2']*)
Plot histograms of the values in the chosen bands of the input image

pyveg.src.image_utils.**process_and_threshold**(*img*, *r=3*)
Perform histogram equalisation, adaptive thresholding, and median filtering on an input PIL Image. Return the result converted back to a PIL Image.

@param img input PIL Image object @return processed PIL Image

pyveg.src.image_utils.**scale_tif**(*input_filename*)
Given only a single band, scale to range 0,255 and apply this value to all of r,g,b

Parameters **input_filename**(*str, location of input image*) –

Returns **new_img**

Return type pillow Image.

5.12 pyveg.src.pattern_generation module

Translation of Matlab code to model patterned vegetation in semi-arid landscapes.

class pyveg.src.pattern_generation.**PatternGenerator**
Bases: object

Class that can generate simulated veget ation patterns, optionally from a loaded starting pattern, and propagate through time according to various amounts of rainfall and/or surface and soil water density.

static calc_plant_change(*plant_biomass*, *soil_water*, *uptake*, *uptake_saturation*, *growth_constant*, *senescence*, *grazing_loss*)
Change in plant biomass as a function of available soil water and various constants.

static calc_soil_water_change(*soil_water*, *surface_water*, *plant_biomass*, *frac_surface_water_available*, *bare_soil_infilt*, *infilt_saturation*, *plant_growth*, *soil_water_evap*, *uptake_saturation*)
Change in soil water as a function of surface water, plant_biomass, and various constants.

static calc_surface_water_change(*surface_water*, *plant_biomass*, *rainfall*, *frac_surface_water_available*, *bare_soil_infilt*, *infilt_saturation*)
Change in surface water as a function of rainfall, plant_biomass, and various constants.

configure()
Set initial parameters, loaded from JSON.

evolve_pattern(*steps=10000*, *dt=1*)
Run the code to converge on a vegetation pattern

initial_conditions()
Set initial arrays of soil and surface water.

initialize()
Set initial values to zero, and boundary conditions.

load_config(*config_filename*)
Load a set of configuration parameters from a JSON file

make_binary(*threshold=None*)
if not given a threshold to use, look at the (max+min)/2 value - for anything below, set to zero, for anything above, set to 1

plot_image()
Display the current pattern.

print_config()

save_as_csv(*filename*)
Save the image as a csv file

save_as_matlab(*filename*)
Save the image as a matlab file

save_as_png(*filename*)
Save the image as a png file

set_rainfall(*rainfall*)
Rainfall in mm

set_random_starting_pattern()
Use the frac from config file to randomly cover some fraction of cells.

set_starting_pattern_from_file(*filename*)
Takes full path to a CSV file containing m rows of m comma-separated values, which are zero (bare soil) or not-zero (vegetation covered).

5.13 pyveg.src.plotting module

Plotting code.

pyveg.src.plotting.**kendall_tau_histograms**(*series_name*, *df*, *output_dir*)
Produce histograms with kendall tau distribution from surrogates for significance analysis

Parameters

- **series_name** (*str*) – String containing data collection and time series variable.
- **df** (*Dataframe*) – The output dataframe from the sensitivity analysis function.
- **output_dir** – Path to the directory to save the produced figures

pyveg.src.plotting.**plot_autocorrelation_function**(*df*, *output_dir*, *filename_suffix=''*)
Given a time series DataFrames (constructed with *make_time_series*), plot the autocorrelation function relevant columns.

Parameters

- **df** (*DataFrame*) – Time series DataFrame.
- **output_dir** (*str*) – Directory to save the plots in.

pyveg.src.plotting.**plot_correlation_mwa**(*df*, *output_dir*, *filename_suffix=''*)
Given a moving window time series DataFrame, plot the time series of veg-precip correlation.

Parameters

- **df** (*DataFrame*) – The time-series results for veg-precip correlation coeff and lag.
- **output_dir** (*str*) – Directory to save the plot in.
- **filename_suffix** (*str*) – Add suffix string to file name

pyveg.src.plotting.**plot_cross_correlations**(*df*, *output_dir*)
Plot a scatterplot matrix showing correlations between vegetation and precipitation time series, with different lags. Additionally write out the correlations as a function of the lag for later use.

Parameters

- **df** (*DataFrame*) – Time-series data.
- **output_dir** (*str*) – Directory to save the plot in.

pyveg.src.plotting.**plot_ews_resiliance**(*series_name*, *EWSmetrics_df*, *Kendalltau_df*, *dates*, *output_dir*)
Make early warning signals resiliance plots using the output from the ewstools package.

Parameters

- **series_name** (*str*) – String containing data collection and time series variable.
- **EWSmetrics_df** (*DataFrame*) – DataFrame from ewstools containing ews time series.
- **Kendalltau_df** (*DataFrame*) – DataFrame from ewstools containing Kendall tau values for EWSmetrics_df time series
- **output_dir** (*str*) – Output dir to save plot in.

pyveg.src.plotting.**plot_feature_vector**(*output_dir*)
Read feature vectors from csv (if they exist) and then make feature vector plots.

Parameters **output_dir** (*str*) – Directory to save the plot in.

pyveg.src.plotting.**plot_moving_window_analysis**(*df*, *output_dir*, *filename_suffix=''*)
Given a moving window time series DataFrame, plot the time series of AR1 and Variance.

Parameters

- **df** (*DataFrame*) – The time-series results for variance and AR1.
- **output_dir** (*str*) – Directory to save the plot in.
- **filename_suffix** (*str*) – Add suffix string to file name

pyveg.src.plotting.**plot_ndvi_time_series**(*df*, *output_dir*)

pyveg.src.plotting.**plot_sensitivity_heatmap**(*series_name*, *df*, *output_dir*)
Produce heatmap plot for the sensitivy analysis

Parameters

- **df** (*Dataframe*) – The output dataframe from the sensitivity analysis function.
- **output_dir** – Path to the directory to save the produced figures

pyveg.src.plotting.**plot_stl_decomposition**(*df*, *period*, *output_dir*)
Run the STL decomposition and plot the results network centrality and precipitation DataFrames in *df*.

Parameters

- **df** (*DataFrame*) – The time-series results.
- **period** (*float*) – Periodicity to model.
- **output_dir** (*str*) – Directory to save the plot in.

pyveg.src.plotting.**plot_time_series**(*df*, *output_dir*, *plot_smoothed=True*)
Given a time series DataFrames (constructed with *make_time_series*), plot the vegetitation and precipitation time series.

Parameters

- **df** (*DataFrame*) – Time series DataFrame.
- **output_dir** (*str*) – Directory to save the plots in.

5.14 pyveg.src.processor_modules module

Class for holding analysis modules that can be chained together to build a sequence.

class pyveg.src.processor_modules.**NDVICalculator**(*name=None*)
Bases: *pyveg.src.processor_modules.ProcessorModule*

Class to look at NDVI on sub-images images, and return the results as json. Note that the input directory is expected to be the level above the subdirectories for the date sub-ranges.

check_sub_image(*ndvi_filename*, *input_path*)
Check the RGB sub-image corresponding to this NDVI image looks OK.

process_single_date(*date_string*)
Each date will have a subdirectory called 'SPLIT' with ~400 NDVI sub-images.

process_sub_image(*ndvi_filepath*, *date_string*, *coords_string*)
Calculate mean and standard deviation of NDVI in a sub-image, both with and without masking out non-vegetation pixels.

set_default_parameters()
Default values. Note that these can be overridden by parent Sequence or by calling configure().

class pyveg.src.processor_modules.**NetworkCentralityCalculator**(*name=None*)
Bases: *pyveg.src.processor_modules.ProcessorModule*

Class to run network centrality calculation on small black+white images, and return the results as json. Note that the input directory is expected to be the level above the subdirectories for the date sub-ranges.

check_sub_image(*ndvi_filename*, *input_path*)
Check the RGB sub-image corresponding to this NDVI image looks OK.

process_single_date(*date_string*)
Each date will have a subdirectory called 'SPLIT' with ~400 BWNDVI sub-images.

set_default_parameters()
Default values. Note that these can be overridden by parent Sequence or by calling configure().

class pyveg.src.processor_modules.**ProcessorModule**(*name*)
Bases: *pyveg.src.pyveg_pipeline.BaseModule*

check_if_finished()

check_input_data_exists(*date_string*)
Processor modules will look for inputs in <input_location>/<date_string>/<input_location_subdirs> Check that the subdirs exist and are not empty.

Parameters **date_string**(*str, format YYYY-MM-DD*) –

Returns

Return type True if input directories exist and are not empty, False otherwise.

check_output_data_exists(*date_string*)
Processor modules will write output to <output_location>/<date_string>/<output_location_subdirs> Check

Parameters **date_string**(*str, format YYYY-MM-DD*) –

Returns

- *True if expected number of output files are already in output location,* – AND self.replace_existing_files is set to False
- *False otherwise*

check_timeout(*task_status*)
See how long since task_status last changed.

create_task_dict(*task_id*, *date_list*, *dependencies=[]*)

get_dependent_batch_tasks()
When running in batch, we are likely to depend on tasks submitted by the previous Module in the Sequence. This Module should be in the "depends_on" attribute of this one.

Task dependencies will be a dict of format {"task_id": <task_id>, "date_range": [<dates>]}

get_image(*image_location*)

run()

run_batch()
" Write a config json file for each set of dates. If this module depends on another module running in batch, we first get the tasks on which this modules tasks will depend on. If not, we look at the input dates subdirectories and divide them up amongst the number of batch nodes.

We want to create a list of dictionaries [{"task_id": <task_id>, "config": <config_dict>, "depends_on": [<task_ids>]}] to pass to the batch_utils.submit_tasks function.

run_local()
loop over dates and call process_single_date on all of them.

save_image(*image*, *output_location*, *output_filename*, *verbose=True*)

set_default_parameters()
Set some basic defaults. Note that these might get overriden by a parent Sequence, or by calling configure() with a dict of values

class pyveg.src.processor_modules.**VegetationImageProcessor**(*name=None*)
Bases: *pyveg.src.processor_modules.ProcessorModule*

Class to convert tif files downloaded from GEE into png files that can be looked at or used as input to further analysis.

Current default is to output: 1) Full-size RGB image 2) Full-size NDVI image (greyscale) 3) Full-size black+white NDVI image (after processing, thresholding, ...) 4) Many 50x50 pixel sub-images of RGB image 5) Many 50x50 pixel sub-images of black+white NDVI image.

construct_image_savepath(*date_string*, *coords_string*, *image_type='RGB'*)
Function to abstract output image filename construction. Current approach is to create a 'PROCESSED' subdir inside the sub-directory corresponding to the mid-period of the date range for the full-size images and a 'SPLIT' subdirectory for the sub-images.

process_single_date(*date_string*)
For a single set of .tif files corresponding to a date range (normally a sub-range of the full date range for the pipeline), construct RGB, and NDVI greyscale images. Then do processing and thresholding to make black+white NDVI images. Split the RGB and black+white NDVI ones into small (50x50pix) sub-images.

Parameters **date_string**(*str, format YYYY-MM-DD*) –

Returns

Return type True if everything was processed and saved OK, False otherwise.

save_rgb_image(*band_dict*, *date_string*, *coords_string*)
Merge the seperate tif files for the R,G,B bands into one image, and save it.

set_default_parameters()
Set some basic defaults. Note that these might get overriden by a parent Sequence, or by calling configure() with a dict of values

split_and_save_sub_images(*image*, *date_string*, *coords_string*, *image_type*, *npix=50*)
Split the full-size image into lots of small sub-images

image: pillow Image date_string: str, format YYYY-MM-DD coords_string: str, format long_lat image_type: str, typically 'RGB' or 'BWNDVI' npix: dimension in pixels of side of sub-image. Default is 50x50

True if all sub-images saved correctly.

class pyveg.src.processor_modules.**WeatherImageToJSON**(*name=None*)
Bases: *pyveg.src.processor_modules.ProcessorModule*

Read the weather-related tif files downloaded from GEE, and write the temp and precipitation values out as a JSON file.

process_single_date(*date_string*)
Read the tif files downloaded from GEE and extract the values (should be the same for all pixels in the image, so just take mean())

Parameters **date_string**(*str, format "YYYY-MM-DD"*) –

set_default_parameters()
Set some basic defaults. Note that these might get overriden by a parent Sequence, or by calling configure() with a dict of values

pyveg.src.processor_modules.**process_sub_image**(*i*, *input_filepath*, *output_location*, *date_string*, *coords_string*)
Read file and run network centrality

5.15 pyveg.src.pyveg_pipeline module

5.15.1 Definitions:

A PIPELINE is the whole analysis procedure for one set of coordinates. It will likely consist of a couple of SEQUENCES - e.g. one for vegetation data and one for weather data.

A SEQUENCE is composed of one or more MODULES, that each do specific tasks, e.g. download data, process images, calculate quantities from image.

A special type of MODULE may be placed at the end of a PIPELINE to combine the results of the different SEQUENCES into one output file.

class pyveg.src.pyveg_pipeline.**BaseModule**(*name=None*)
Bases: object

A "Module" is a building block of a sequence - takes some input, does something (e.g. Downloads from GEE, processes some images, ...) and produces some output. The working directory for all modules within a sequence will be given by the sequence - modules may write output to subdirectories of this (e.g. for different dates), but what we call "output_location" will be the base directory common to all modules, and will contain info about the image collection name, and the coordinates.

check_config()
Loop through list of parameters, which will each be a tuple (name, [allowed_types]) and check that the parameter exists, and is of the correct type.

check_for_existing_files(*location*, *num_files_expected*)
See if there are already num_files in the specified location. If "replace_existing_files" is set to True, always return False

check_if_finished()

configure(*config_dict=None*)
Order of preference for configuriation: 1) config_dict 2) values held by the parent Sequence 3) default values So we set them in reverse order here, so higher priorities will override.

copy_to_output_location(*tmpdir*, *output_location*, *file_endings=[]*)
Copy contents of a temporary directory to a specified output location.

Parameters

- **tmpdir**(*str, location of temporary directory*) –

- **output_location** (*str, either path to a local directory (if self.output_location_type is "local")*) – or to Azure <container>/<blob_path> if self.output_location_type==”azure”)
- **file_endings** (*list of str, optional. If given, only files with those endings will be copied.*)–

get_config()
Get the configuration of this module as a dict.

get_file(*filename*, *location_type*)
Just return the filename if location _type is “local”. Otherwise return a tempfile with the contents of a blob if the location is “azure”.

get_json(*filepath*, *location_type*)
Read a json file either local or blob storage.

join_path(**path_elements*)
If output_location_type is ‘local’, we will just use os.path.join, which puts a “/” separator in for posix, or “” for windows. However, if output_location_type is ‘azure’, we always want “/”.

Parameters **path_elements** (*list of strings. Directory-like path elements.*)–

Returns **path**

Return type str, the path elements joined by “/” or “”.

list_directory(*directory_path*, *location_type*)
List contents of a directory, either on local file system or Azure blob storage.

prepare_for_run()

print_run_status()
Print out how many jobs succeeded or failed

save_config(*config_location*)
Write out the configuration of this module as a json file.

save_json(*data*, *filename*, *location*, *location_type*)
Save json to local filesystem or blob storage depending on location_type

set_default_parameters()

set_parameters(*config_dict*)

class pyveg.src.pyveg_pipeline.**Pipeline**(*name*)
Bases: object

A Pipeline contains all the Sequences we want to run on a particular set of coordinates and a date range. e.g. there might be one Sequence for vegetation data and one for weather data.

cleanup()
Call cleanup() for all our sequences

configure()
Configure all the sequences in this pipeline.

get(*seq_name*)
Return a sequence object when asked for by name.

print_run_status()

run()
run all the sequences in this pipeline

class pyveg.src.pyveg_pipeline.**Sequence**(*name*)
Bases: object

A Sequence is a collection of Modules where the output of one module is typically the input to the next one. It will typically correspond to a particular data collection, e.g. for vegetation imagery, we might have one module to download the images, one to process them, and one to analyze the processed images.

check_if_finished()
Only relevant when one or more modules are running in batch mode, Sequences that depend on this Sequence will call this function while they wait for all Modules to finish.

cleanup()
If we have batch resources (job/pool), remove them to avoid charges

configure()

create_batch_job_if_needed()
If any modules in this sequence are to be run in batch mode, create a batch job for them.

get(*mod_name*)
Return a module object when asked for by name, or by class name

has_batch_job()
Do any of the Modules in this sequence have run_mode == 'batch'?

join_path(**path_elements*)
If output_location_type is 'local', we will just use os.path.join, which puts a "/" separator in for posix, or "" for windows. However, if output_location_type is 'azure', we always want "/".

Parameters **path_elements** (*list of strings. Directory-like path elements.*) –

Returns **path**

Return type str, the path elements joined by "/" or "".

print_run_status()
For all modules in the sequence, print out how many jobs succeeded or failed.

run()
Before we run the Modules in this Sequence, check if there are any other Sequences on which we depend, and if so, wait for them to finish.

set_config(*config_dict*)

set_output_location()

5.16 pyveg.src.subgraph_centrality module

Python version of mao_pollen.m matlab code to look at connectedness of pixels on a binary image, using "Subgraph Centrality" as described in:

Mander et.al. "A morphometric analysis of vegetation patterns in dryland ecosystems", R. Soc. open sci. (2017) https://royalsocietypublishing.org/doi/10.1098/rsos.160443

Mander et.al. "Classification of grass pollen through the quantitative analysis of surface ornamentation and texture", Proc R Soc B 280: 20131905. https://royalsocietypublishing.org/doi/pdf/10.1098/rspb.2013.1905

Estrada et.al. "Subgraph Centrality in Complex Networks" https://arxiv.org/pdf/cond-mat/0504730.pdf

pyveg.src.subgraph_centrality.**calc_adjacency_matrix**(*distance_matrix*, *include_diagonal_neighbours=False*)
Return a symmetric matrix of (n-pixels-over-threshold)x(n-pixel-over-threshold) where each element ij is 0 or 1 depending on whether the distance between pixel i and pixel j is < or > neighbour_threshold.

pyveg.src.subgraph_centrality.**calc_and_sort_sc_indices**(*adjacency_matrix*)
Given an input adjacency matrix, calculate eigenvalues and eigenvectors, calculate the subgraph centrality (ref: <== ADD REF), then sort.

pyveg.src.subgraph_centrality.**calc_distance_matrix**(*signal_coords*)
calculate the distances between all signal pixels in the original image

pyveg.src.subgraph_centrality.**calc_euler_characteristic**(*pix_indices*, *graph*)
Find the edges where both ends are within the pix_indices list

pyveg.src.subgraph_centrality.**crop_image_array**(*input_image*, *x_range*, *y_range*)
return a new image from specified pixel range of input image

pyveg.src.subgraph_centrality.**feature_vector_metrics**(*feature_vector*, *output_csv=None*)
Calculate different metrics for the feature vector

pyveg.src.subgraph_centrality.**fill_feature_vector**(*pix_indices*, *coords*, *adj_matrix*, *num_quantiles=20*)
Given indices and coordinates of signal pixels ordered by SC value, put them into quantiles and calculate an element of a feature vector for each quantile. by using the Euler Characteristic.

Will return: selected_pixels, feature_vector

where selected_pixels is a vector of the pixel coordinates in each quantile, and a feature_vector is either num-connected-components or Euler characteristic, for each quantile.

pyveg.src.subgraph_centrality.**fill_sc_pixels**(*sel_pixels*, *orig_image*, *val=200*)
Given an original 2D array where all the elements are 0 (background) or 255 (signal), fill in a selected subset of signal pixels as 123 (grey).

pyveg.src.subgraph_centrality.**generate_sc_images**(*sel_pixels*, *orig_image*, *val=200*)
Return a dict of images with the selected subsets of signal pixels filled in in cyan.

pyveg.src.subgraph_centrality.**get_signal_pixels**(*input_array*, *threshold=255*, *lower_threshold=True*, *invert_y=False*)
Find coordinates of all pixels within the image that are > or < the threshold (require < threshold if lower_threshold==True) NOTE - if invert_y is set, we make the second coordinate negative, for reasons.

pyveg.src.subgraph_centrality.**invert_y_coord**(*coord_list*)
Convert [(x1,y1),(x2,y2),...] to [(x1,-y1),(x2,-y2),...]

pyveg.src.subgraph_centrality.**make_graph**(*adj_matrix*)
Use igraph to create a graph from our adjacency matrix

pyveg.src.subgraph_centrality.**save_sc_images**(*image_dict*, *file_prefix*)
Saves images from dictionary.

pyveg.src.subgraph_centrality.**subgraph_centrality**(*image*, *use_diagonal_neighbours=False*, *num_quantiles=20*, *threshold=255*, *lower_threshold=True*, *output_csv=None*)
Go through the whole calculation, from input image to output vector of pixels in each SC quantile, and feature vector (either connected-components or Euler characteristic).

pyveg.src.subgraph_centrality.**text_file_to_array**(*input_filename*)
Read a csv-like representation of an image, where each row (representing a row of pixels in the image) is a comma-separated list of pixel values 0 (for black) or 255 (for white).

pyveg.src.subgraph_centrality.**write_csv**(*feature_vec*, *output_filename*)
Write the feature vector to a 1-line csv

pyveg.src.subgraph_centrality.**write_dict_to_csv**(*metrics_dict*, *output_filename*)

5.17 pyveg.src.zenodo_utils module

Use the Zenodo API to deposit or retrieve data.

Needs an API token - to create one: Sign-in or create an account at https://zenodo.org Create an API token by going to this page: https://zenodo.org/account/settings/applications/tokens/new/

tick "deposit:actions" and "deposit:write" in the "Scopes" section

and click Create. Then copy the created token into a file called "zenodo_api_token" in the pyveg/configs/ directory.

OR, to use the "Sandbox" API for testing, follow the same steps but replacing "zenodo.org" with "sandbox.zenodo.org" in the URLs, and put the token into a file named "zenodo_test_api_token" then call the functions in this module with the "test" argument set to True.

pyveg.src.zenodo_utils.**create_deposition**(*test=False*)
Create a new, empty deposition.

Parameters **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns **r**

Return type dict, response from the API with info about the newly created deposition

pyveg.src.zenodo_utils.**delete_file**(*filename*, *deposition_id*, *test=False*)
Delete a file from a deposition.

Parameters

- **filename** (*str, full path to the file to be deleted*) –
- **deposition_id** (*int, ID of the deposition containing this file*) –
- **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns

Return type True if file was deleted OK, False otherwise.

pyveg.src.zenodo_utils.**download_file**(*filename*, *deposition_id*, *destination_path='.'*, *test=False*)
Upload a file to a deposition.

Parameters

- **filename** (*str, full path to the file to be uploaded*) –
- **deposition_id** (*int, ID of the deposition containing this file*) –
- **destination_path** (*str, where to put the downloaded file*) –

- **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns **filepath**

Return type str, location of downloaded file.

pyveg.src.zenodo_utils.**download_results_by_coord_id**(*coords_id*, *json_or_csv='json'*, *destination_path=None*, *deposition_id=None*, *test=False*)

Search the deposition (defined by the deposition_id in zenodo_config.py) for results_summary json or summary_stats csv files beginning with 'coord_id' and download the most recent one.

Parameters

- **coords_id** (*str, two-digit string identifiying the row of the location in coordinates.py*) –
- **json_or_csv** (*str, if "json", download 'results_summary.json', otherwise download 'ts_summary_stats.csv'.*) –
- **destination_path** (*str, directory to download to. If not given, put in temporary dir*) –
- **deposition_id** (*str, deposition ID in Zenodo. If not given, use the one from zenodo_config.py*) –
- **test** (*bool, if True, use the sandbox Zenodo repository*) –

pyveg.src.zenodo_utils.**get_base_url_and_token**(*test=False*)

Get the base URL for the API, and the API token, for use in requests.

Parameters **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns

- **base_url** (*str, the first part of the URL for the API*)
- **api_token** (*str, the personal access token, read from a file.*)

pyveg.src.zenodo_utils.**get_bucket_url**(*deposition_id*, *test=False*)

For a given deposition_id, find the URL needed to upload a file.

Parameters

- **deposition_id** (*int, ID of the deposition.*) –
- **test** (*bool, if True use the sandbox API, if False will use the real one.*) –

Returns **bucket_url**

Return type str, the URL of the bucket for this deposition, or empty string if id not found

pyveg.src.zenodo_utils.**get_deposition_id**(*json_or_csv='json'*, *test=False*)

If we have previously created a deposition, we hopefully stored its ID in the zenodo_config.py file.

pyveg.src.zenodo_utils.**get_deposition_info**(*deposition_id*, *test=False*)

Get the JSON object containing details of a deposition.

Parameters

- **deposition_id** (*int, ID of the deposition.*) –
- **test** (*bool, if True use the sandbox API, if False will use the real one.*) –

Returns **dep_info**

Return type dict, information about the deposition

pyveg.src.zenodo_utils.**get_results_summary_json**(*coords_string*, *collection*, *deposition_id*, *test=False*)
Assuming the zipfile is named following the convention results_<long>_<lat>_<collection>.zip download this from the deposition, and extract the results_summary.json.

pyveg.src.zenodo_utils.**list_depositions**(*test=False*)
List all the depositions created by this account.

Parameters **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns **r**

Return type list of dicts, response from the API with info about the depositions

pyveg.src.zenodo_utils.**list_files**(*deposition_id*, *json_or_csv='json'*, *test=False*)
List all the files in a deposition.

Parameters

- **deposition_id** (*int, ID of the deposition on which to list files*) –
- **json_or_csv** (*str, if 'json', list the deposition containing the results_summary.json*) – otherwise list the one containing ts_summary_stats.csv
- **test** (*bool, True if using the sandbox API, False otherwise*) –

Returns **files**

Return type list[str], list of all filenames in the deposition.

pyveg.src.zenodo_utils.**prepare_results_zipfile**(*collection_name*, *png_location*, *png_location_type='local'*, *json_location=None*, *json_location_type='local'*)
Create a zipfile called <results_long_lat_collection> containing the 'results_summary.json', and the outputs of the analysis.

Parameters

- **collection_name** (*str, typically "Sentinel2" or "Landsat8" or similar*) –
- **base_png_location** (*str, directory containing analysis/subdirectory*) –
- **png_location_type** (*str, either "local" or "azure"*) –
- **base_json_location** (*str, directory containing "results_summary.json.*) – If not specified, assume same as base_png_location
- **json_location_type** (*str, either "local" or "azure"*) –

Returns **zip_filename**

Return type str, location of the produced zipfile

pyveg.src.zenodo_utils.**publish_deposition**(*deposition_id*, *test=False*)
Submit the deposition, so it will be findable on Zenodo and have a DOI.

pyveg.src.zenodo_utils.**unlock_deposition**(*deposition_id*, *test=False*)
Unlock a previously submitted deposition, so we can add to it.

pyveg.src.zenodo_utils.**upload_custom_metadata**(*title*, *upload_type*, *description*, *creators*, *deposition_id*, *test=False*)
Upload a dict to the deposition containing metadata with the format:

{

'metadata': { 'title': 'My first upload', 'upload_type': 'poster', 'description': 'This is my first upload', 'creators': [{'name': 'Doe, John',

'affiliation': 'Zenodo'}]

}

}

title: str, title of the deposition upload_type: str, type of upload, typically "dataset" description: str, description of the deposition creators: dict, format {"name": <str:name>, "affiliation": <str:affiliation>}

Returns **r**

Return type dict, JSON response from the API.

pyveg.src.zenodo_utils.**upload_file**(*filename*, *deposition_id*, *test=False*)
Upload a file to a deposition.

Parameters

- **filename** (*str, full path to the file to be uploaded*) –
- **deposition_id** (*int, ID of the deposition to which we want to upload.*) –
- **test** (*bool, True if we will use the sandbox API, False otherwise*) –

Returns **uploaded_ok**

Return type bool, True if we get status code 200 from the API

pyveg.src.zenodo_utils.**upload_standard_metadata**(*deposition_id*, *json_or_csv='json'*, *test=False*)
Upload the metadata dict defined in zenodo_config.py to the specified deposition ID.Kcontaining metadata with the format:

deposition_id: int, ID of the deposition to which to upload json_or_csv: str, can be either 'json' to upload the metadata for *results_summary.json*

or *csv* to upload the metadata for *ts_summary_stats.csv*

test: if True, use the sandbox API, if False use the production one.

Returns **r**

Return type dict, JSON response from the API.

5.18 Module contents

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

D

E

F

G

N

P

R

S

T

U

V

W