Siglib Documentation

Release 2.8

D. Mueller, C. Lopes, S. Bouh-Ali, C. Fitzpatrick

July 27, 2018

CONTENTS

CHAPTER

ONE

OVERVIEW OF THE PROJECT

Intro

SigLib stands for Signature Library and is a suite of tools to query, manipulate and process remote sensing imagery (primarily Synthetic Aperture Radar (SAR) imagery) and store the data in a geodatabse. It uses open source libraries and can be run on Windows or Linux.

There are 4 main *modes* that it can run in (or combinations of these)

1. A data **Discovery Mode** where remote sensing scenes are discovered by ingesting a copy of the Canadian Ice Service archive (or other geodatabase containing metadata, with tweaks), or by crawling through a hard drive and extracting metadata from zipped SAR scenes, or by querying a table in a local database that contains geospatial metadata. Queries take a Region Of Interest – **ROI shapefile** with a specific format as input. The region of interest delineates the spatial and temporal search boundaries. The required attribute fields and formats for the ROI are elaborated upon in a section below.
2. An **Exploratory Mode** where remote sensing scenes are made ready for viewing. This includes opening zip files, converting imagery (including Single Look Complex), geographical projection, cropping, masking, image stretching, renaming, and pyramid generation. The user must supply the name of a single zip file that contains the SAR imagery, a directory where a batch of zip files to be prepared resides, or a query that selects a list of zip files to be processed (functionality to come).
3. A **Scientific Mode** where remote sensing scenes can be converted to either calibrated (sigma0), noise level, or incidence angle images. Image data (from each band) can be subsampled by way of an **ROI shapefile** that references every image and specific polygon you want to analyze. These polygons represent sampling regions that you know about (a priori) or they are hand digitized from Exploratory mode images. Data can be stored in a table in a geodatabase for further processing.
4. An **Analysis Mode** where data that was stored in the geodatabase is retrieved and plotted [Note, this is essentially depreciated since it hasn't been used for over 5 years]

These modes are brought together to work in harmony by **SigLib.py** the recommended way to interact with the software. This program reads in a configuration file that provides all the parameters required to do various jobs. However, this is only one way to go... Anyone can call the modules identified above from a custom made python script to do what they wish, using the SigLib API

In addition, there are different ways to process *input* through SigLib.py that can be changed for these modes. You can input based on a recursive **scan** of a directory for files that match a pattern; you can input one **file** at a time (useful for parallelization, when many processes are spawned by gnu parallel) and; you can input an SQL **query** and run the resulting matching files through SigLib (note that query input is not yet enabled, but it wouldn't take long).

Acknowledgements

This software was conceived and advanced initially by Derek Mueller (while he was a Visiting Fellow at the Canadian Ice Service). Some code was derived from from Defence Research and Development Canada (DRDC). I benefited from discussions with Ron Saper, Angela Cheng and my salary was provided via a CSA GRIP project (PI Roger De Abreu).

At Carleton this code was modified further and others have worked to improve it since the early days at CIS: Cindy Lopes (workstudy student & computer programmer) 2012, Sougal Bouh-Ali (workstudy student & computer programmer) 2013-2016, and Cameron Fitzpatrick 2018. Ron Saper, Anna Crawford and Greg Lewis-Paley helped out as well (indirectly).

CHAPTER

TWO

OVERVIEW OF SIGLIB.PY AND ITS DEPENDENCIES

Dependencies

You will need a computer running linux or windows

- Python 2 (not 3), along with several scientific libraries - numpy, pandas, psycopg2, matplotlib, datetime... Recommend you install the anaconda package as these contain pretty well everything you will need.
- gdal/ogr libraries
- PostrgreSQL/PostGIS (could be on another computer)

Nice to haves/future...

- It is highly recommended that you have access to QGIS or ArcGIS to manipulate shapefiles
- Also, if you want to work with ASF CEOS Files, you will need ASF MapReady (some functionality)
- Eventually, there will be a push to integrate other remote sensing tools - SNAP(replaces NEST,PolSARPro), CP Simulator, MADGIC, etc.

Modules

There are several modules that are organized according to core functionality.

1. **Util.py** - a bunch of utilities for manipulating files, shapefiles, etc
2. **Metadata.py** - used to discover and extract metadata from image files
3. **Database.py** - used to interface between the PostGIS database for storage and retrieval of information
4. **Image.py** - used to manipulate images, project, calibrate, crop, etc.
5. **LogConcat.py** - used to combine individual log files into one master .txt file, and separate log files containing errors for analysis (Mainly for use after large runs in parallel)

SigLib.py is the front-end of the software. It calls the modules listed above and is in turn controlled by a configuration file. To run, simply edit the *.cfg file with the paths and inputs you want and then run siglib.py.

However, you can also code your own script to access the functionality of the modules if you wish.

Config File

The ***.cfg** file is how you interface with siglib. It needs to be edited properly so that the job you want done will happen! Leave entry blank if you are not sure. There are several categories of parameters and these are:

Directories

- scanDir = path to where you want siglib to look for SAR image zip files to work with
- tmpDir = a working directory for extracting zip files to (Basically, a folder for temporary files that will only be used during the running of the code, then deleted, in scratch folder)
- projDir = where projection definition piles are found in well-known text format (/tank/ice/data/proj)
- vectDir = where vector layers are found (ROI shapefiles or masking layers)
- dataDir = /tank/path2folder
- imgDir = a working directory for storing image processing intermediate files and final output files, in scratch folder
- logDir = where logs are placed
- archDir = where CIS archive data are found (/tank/ice/data/vector/CIS_Archive)
- errorDir = Where logConcat will send .log files with errors (For proper review of bad zips at end of run)

Database

- db = the name of the database you want to connect to
- create_tblmetadata = 0 for append, 1 for overwrite/create

Input

Note that these are mutually exclusive options - sum of 'Input' options must = 1

- path = 1 for scan a certain path and operate on all files within; 0 otherwise
- query = 1 for scan over the results of a query and operate on all files returned; 0 otherwise
- file = 1 for run process on a certain file, which is passed as a command line argument (note this enables parallelized code); 0 otherwise
- scanFor = a file pattern to search for (eg. *.zip) - use when path=1
- sql = define a custom query here for selecting data to process - use when query=1. eg. SELECT location FROM tblmetadata WHERE granule = 'B0558007.img'

Process

- data2db = 1 when you want to upload metadata to the metadata table in the database
- data2img = 1 when you want to manipulate images (as per specs below)

IMGMode

- proj = basename of wkt projection file (eg. lcc)
- imgtypes = types of images to process (Amplitude Image - amp, Sigma Nought Image - sigma, Incidence Angle image - theta, Noise Floor Image - nes2?)
- crop = nothing for no cropping, or four space-delimited numbers, upper-left and lower-right corners (in proj above) that denote a crop area: ul_x ul_y lr_x lr_y
- maskshp = a polygon shapefile (one feature) to mask image data with (eg. /tank/ice/data/vector/CIS_Vectors/coast_poly.shp)
- roiproj = Projection of roi
- imgformat = File format for output imagery (gdal convention)
- roi = ROI Shapefile for Discovery or Scientific modes, stored in same directory as python files

- spatialrel = ST_Contains (Search for images that fully contain the roi polygon) or ST_Intersects (Search for images that merely intersect with the roi)

Using a Config in an IDE

You can run SigLib inside an integrated development environment (Spyder, IDLE, etc) or at the command line. In either case you must specify the configuration file you wish to use:

```
python /path_to_script/SigLib.py /path_to_file/config_file.cfg
```

Dimgname Convention

"The nice thing about standards is that there are so many to chose from" (A. Tannenbaum), but this gets annoying when you pull data from MDA, CSA, CIS, PDC, ASF and they all use different file naming conventions. So I made this problem worse with my own 'standard image naming convention called **dimgname**. All files processed by SigLib get named as follows, which is good for:

- sorting on date (that is the most important characteristic of an image besides where the image is - and good luck conveying that simply in a file name).
- viewing in a list (because date is first, underscores keep the names tidy in a list - you can look down to see the different beams, satellites, etc.)
- extensibility - you can add on to the file name as needed - add a subscene or whatever on the end, it will sort and view the same as before.
- extracting metadata from the name (in a program or spreadsheet just parse on "_")

Template: date_time_sat_beam_data_proj.ext

Example: 20080630_225541_r1_scwa__hh_s_lcc.tif

Table: **dimgname fields**

Position	Meaning	Example	Chars
1 Date	year month day	20080630	8
1 Time	hour min sec	225541	6
1 Sat	satellite/platform/sensor	r1,r2,e1,en	2
1 Beam	Beam for SAR, band combo for optical	st1__,scwa_,fqw20_,134__	5
1 Band	pol for SAR, meaning of beam for optical (tc = true colour)	hh, hx, vx, vv, hv, qp	2
1 Data	what is represented (implies a datatype to some extent	a= amplitude, s=sigma, t=incidence,n=NESZ, o=optical	1
1 Proj	projection	nil, utm, lcc, aea	3
1 Ext	file extension	tif, rrd, aux, img	3

ROI.shp format

The ROI.shp or Region Of Interest shapefile is what you need to extract data. Basically it denotes *where* and *when* you want information. It has to have certain fields to work properly. There are two basic formats, based on whether you are using the **Discovery** or **Scientific** mode. If you are interested in 1) finding out what scenes/images might be available to cover an area or 2) generating images over a given area then use the *Discovery* format. If you have examined the

images already and have digitized polygons of areas that you want to analyze (find statistics), then make sure those polygons are stored in a shapefile using the *Scientific* format. In either case you must have the fields that are required for *Both* formats in the table below. You can add whatever other fields you wish and some suggestions are listed below as *Optional*.

The two fields which are required for both Discovery or Scientific mode use may be confusing, so here are some further details with examples.

- OBJ - this is a unique identifier for a given area or object (polygon) that you are interested in getting data for.
- INSTID - A way to track OBJ that is repeatedly observed over time (moving ice island, a lake during fall every year for 5 years). [If it doesn't repeat just put '0']

Example workflow:

You could be interested in lake freeze-up in the Yukon, drifting ice islands, or soil moisture in southern Ontario farm fields. First you will want to find out what data are available, retrieve zip files and generate imagery to look at. In this case use the *Discovery* format. Each lake, region that ice islands drift through or agricultural area that you want to study would be given a unique OBJ. If you have only one time period in mind for each, then INSTID would be '0' in all cases. If however, you want to look at each lake during several autumns, ice islands as they drift or farm fields after rain events, then each OBJ will have several rows in your shapefile with a different FROMDATE and TODATE. Then for each new row with the same OBJ, you must modify the INSTID such that a string that is composed of OBJ+INSTID is unique across your shapefile. This is what is done internally by SigLib and a new field is generated called INST (in the PostGIS database). Note that the FROMDATE and TODATE will typically be different for each OBJ+INSTID combination.

If you know what imagery is available already, or if you have digitized specific areas corresponding where you want to quantify backscatter (or image noise, incidence angle, etc), then you should use the *Scientific* format. In this case, the principles are the same as in the *Discovery* mode but your concept of what an OBJ might be, will be different. Depending on the study goals, you may want backscatter from the entire lake, in which case your OBJ would be the same as in *Discovery* mode, however, the INSTID must be modified such that there is a unique OBJ+INSTID for each image (or image acquisition time) you want to retrieve data for. The scientific OBJ should change when you are hand digitizing a specific subsample from each OBJ from the *Discovery* mode. For example:

- within each agricultural area you may want to digitize particular fields;
- instead of vast areas to look for ice islands you have actually digitized each one at a precise location and time

Build your *Scientific* ROI shapefile with the field IMGREF for each unique OBJ+INSTID instead of the FROMDATE and TODATE. By placing the dimgname of each image you want to look at in the IMGREF field, SigLib can pull out the date and time and populate the DATEFROM and DATETO fields automatically. Hint: the INSTID could be IMGREF if you wished (since there is no way an OBJ would be in the same image twice).

Once you complete your ROI.shp you can name it whatever you like (just don't put spaces in the filename, since that causes problems).

Table: **ROI.shp fields**

Field	Var. Type	Description	Example	ROI Format
OBJ	String	A unique identifier for each polygon object you are interested in	00001, 00002	Both
INSTID	String	An iterator for each new row of the same OBJ	0,1,2,3,4	Both
FROMDATE	String	ISO Date-time denoting the start of the time period of interest	2002-04-15 00:00:00	Discovery
TODATE	String	ISO Date-time denoting the end of the time period of interest	2002-09-15 23:59:59	Discovery
IMGREF	String	dimgname of a specific image known to contain the OBJ polygon (Spaces are underscores)	20020715 135903 r1 scwa hh s lcc.tif	Scientific
Name	String	A name for the OBJ is nice to have	Ward Hunt, Milne, Ayles	Optional
Area	Float	You can calculate the Area of each polygon and put it here (choose whatever units you want)	23.42452	Optional

| Area | Float | You can calculate the Area of each polygon and put it here (choose whatever units you want) | 23.42452 | Optional |

- See folder ROISamples for example ROIs - Discovery and Scientific mode

CHAPTER

THREE

TODO

*# capture stdout and stderr from spawned processes

*# Make sure there is process/output testing and error trapping at every major step.

*# Develop a test suite of imagery for the project - R2 and R1 images that are in different beam modes, orbit directions, even bad images to test siglib. (imagery with no EULA so it can be shared)

- version control (github? bitbucket?) - both software and version identification and tracking changes for users
- Continue documentation
    1. overarching documentation important too
    2. UML diagram for visual
    3. example scripts/configs
    4. example ROI.shp
- add local? [Not sure exactly what this is]
- investigate compatibility with python 3

SigLib.py

- add 'modes' to this - so that siglib can do what is described above.
- add qryDatabase stuff or at least some of it (part of discovery mode)
- update config.cfg accordingly

Metadata.py

- get look direction for RSAT2, test against RSAT1

Image.py

- test Pauli decomp and write in a switch for this - so users can choose?
- test image crop and mask - in both modes

Util.py

- deltree needs work (or can it be removed?)

Sphinx

- Run this to put wiki info in. Must save info as .wik file: pandoc -s -S -f mediawiki intro.wik -t rst -o intro.rst
- Using Siglib!

CHAPTER

FOUR

USING SIGLIB

Welcome to the tutorial sections of the Siglib documentation! This section will be used to give a brief overview of how to use the Metadata, Util, Database, and Image functions via Siglib and its config, or in a custom way via qryDatabase. Examples will be given on how to do: basic radarsat image processing via Siglib and config, extracting and uploading metadata via Siglib, config, and the Linux parallel library, a customizable Discovery mode via qryDatabase, and Scientific mode via Siglib and config.

Basic Siglib Setup

Before Siglib and its dependencies can be used for the first time, some basic setup must first be completed. In the downloaded Siglib file, there should be five Python files, a config file, and an extras folder containing some odds and ends (including this very document you are reading!).

Inside the main Siglib folder alongside the five main Python files and the config, a number of directory folders must be created and associated with the config file. Please see the config section of the documentation above for the required folders. These directories are used to keep the various input, temporary, and output files organized. Once created, the full path to each file must be added to the config file alongside the directory it is set to represent. The config file contains example path listings for example purposes.

Now, for Siglib.py to recognise and use the config file properly, your Python IDE must be set up for running via the command line. The following instructions on how to set this up will be given for the Spyder Python IDE by Anaconda, the setup for other IDE's may vary.

1. Go to Run -> Configuration per file... (Ctrl + f6)
2. **Under General Settings, check the box labeled** *Command Line Options:*
3. **In the box to the right, put the full path to the config** file, including the config file itself and its extension.
4. **Press the OK button to save the setting and close the** window

Example #1: Basic Radarsat2 Image Processing using Siglib

Now that the basic setup steps are complete, we can being our first example! In this example we will be doing a basic xml - to - Tiff image processing job for a couple of Radarsat2 image files, via Siglib. Before any work beings in Python, the config file must be configured for this type of job. (Add picture with desired settings!!!!). Place a few radarsat2 zip files in your testzips directory, then open your IDE configured for command line running, and run Siglib.

What will happen is as follows: The zipfile will be extracted to the temp directory via Util.py. The metadata will then be extracted and saved to the output directory, via Metadata.py. An initial Tiff image will then be created via GDAL (soon to be snappy), and saved to the output directory. The image will then be reprojected and the stats produced into

a new Tiff file, the old one being deleted, all this via Image.py. The temp directory will then be cleaned and Siglib will move onto the next zipfile, if more than one are being processed.

Example #2: Discover Radarsat metadata and upload to a geodatabase

This example will be the first introduction to Database.py and PGAdmin. In this example we will be uploading the metadata of radarsat scenes to a geodatabase for later reference (and for use in later examples). This process will be done using the parallel library on linux. See https://www.gnu.org/software/parallel for documentation and downloads for the parallel library. **NOTE:** This example only works on *linux* machines, how the results of this example can be replicated on other machines will be explained afterwards.

This job will be done via the data2db process of Siglib, as seen in the config under *Process*, so change that setting to 1, and any other setting under *Process* to 0. Also, in order for parallel running to work properly, the input must be **File** not **Path**, so that setting must be switched in the config under *Input*. A review of the settings needed for this particular example can be seen in the image below.

To start the parallel job:

1. Open a terminal
2. **cd into the directory containing all your radarsat images (They can be in multiple** directories, just make sure they are below the one you cd into, or they will not be found)
3. **Type in the terminal: find . -name '*.zip' -type f | parallel -j 16 –nice 15 –progress python /tank/SCRATCH/cfitzpatrick/SigLib.py /tank/SCRATCH/cfitzpatrick/config_linux.cfg**

   Where -j is the number of cores to use, and –nice is how nice the process will be to other processes (I.E. A lower –nice level gives this job a higher priority over other processes). The first directory is the location of your verison of Siglib.py, the second is the location of the associated config file.

NOTE: ALWAYS test parallel on a small batch before doing a major run, to make sure everything is running correctly.

What happens once you hit *ENTER* is Parallel will step though your selected directory looking for .zip files, once one is found, it will pass it to one of the 16 availible (or however many cores you set) openings of Siglib.py. Siglib will unzip the file via Util.py, grab the metadata via Metadata.py, then connect to your desired database, and unpload this retrieved metadata to the relational table *tblmetadata* (which will have to be created by running createTblMetadata() in Database.py before parallelizing) via Database.py. This will repeat until parallel has fully stepped through your selected directory.

Any Siglib process can be parallelized, as long as the correct config parameters are set, and the above steps on starting a parallel job are followed.

The same results for this example can be achieved for non-linux machines by putting all the zip files containing metadata for upload into your testzips directory, and using the config settings as seen in the image below, similar to the first example.

Example #3: Discovery Mode via qryDatabase.py

For this example, we are diverging away from the set rules of Siglib.py and config files, and getting our hands dirty so to speak. Siglib.py and its associated config file is a neat, organized way to do certain tasks in large batches. However, Metadata.py, Util.py, Image.py, and Database.py contain an extrordinary number of customizable functions, some of which are not even covered by Siglib.py! Other times, Siglib can just be too complex for small batch runs. This is why the function qryDatabase.py was created: to show off how to to use Siglibs' dependencies in custom ways!

For this example, qryDatabase.py is configured to search through your relational table tblmetadata (made in example #2) based on an uploaded ROI shapefile, and find images that overlap this ROI. The results can then be sent to anothet table, exported to a csv, made into real images, or whatever you like! This is the beauty of customization.

To begin, open qryDatabase.py, Metadata.py, Util.py, and Database.py in your desired IDE. Unlike in example #1, your IDE **should not** be configured for running via command line! The config file is not used by this script, all the settings are set in program. At the top of the script is a parameters selction with the various directories, and a few other settings needed. An ROI (Region of Interest) is a shapefile containing polygons of land areas of interest, you can look at **Sample_DiscoveryROI_lcc.shp** in the extras folder as an example before making one of your own.

Discovery Mode in its simplest form only takes five lines, these are:

1. **db = Database.Database('')** where '' is your desired database
2. db.updateROI(roi, roiProj, homedir, True) #loads (overwrites) ROI file into dbase
3. **copylist, instimg = db.qrySelectFromAvailable(roi, selectFrom, spatialrel, roiProj)** #Spatial query to find overlapping images
4. **db.instimg2db(roi, spatialrel, instimg, mode='create')** #this uploads the instimg to a relational table in the database
5. db.removeHandler()

These five lines connect you to your database, upload the ROI to the database, runs a query to find what images overlap the roi, uploads the results to another table, and cancels the logging system.

This is a very basic example, but feel free to modify qryDatabase to suit your specific needs! More information on all the availible functions Siglib and its dependencies have to offer can be seen further on in this documentation in the *Siglib API* section of this documentation.

Example #4: Scientific Mode!

In this final example, we will dive into the depths of Siglibs' Scientific Mode! Scientific Mode (as described in an earlier section of this documentation) is a way of taking normal radarsat images and converting them to a new image type (sigma0, noise level, or incidense angle) as well as cutting them up and masking them into small pieces via a scientific ROI of many small polygons of study areas.

In Siglib.py, there is a spcific function for this mode, which can be configured via the config file.

(Work on the Siglib version of this functionality, and finish documentation once complete)

Conclusion

This is the conclusion to the *Using Siglib* section of this documentation. For additional help in using Siglib.py and its dependencies, please refer to the next section of this documentation, *Siglib API*. This section gives and overview, the parameters, and the outputs, of each function in the main five scripts.

CHAPTER

FIVE

SIGLIB API

SigLib

SigLib.py

This script will bring together all the SigLib modules with a config script to

Created on Mon Oct 7 20:27:19 2013 **@author:** Sougal Bouh Ali **Modified on** Wed May 23 11:37:40 2018 **@reason:** Sent instance of Metadata to data2img instead of calling Metadata again **@author:** Cameron Fitzpatrick

class SigLib.**SigLib**

createLog(*zipfile=None*)
Creates log file that will be used to report progress and errors **Parameters**

zipfile : a valid zipfile name with full path (optional) for file input

data2db(*meta*, *db*, *zipfile*)
Adds the image file metadata to tblmetadata table in the specified database. Will create/overwrite the table tblmetadata if prompted (be carefull)

Parameters

meta : A metadata instance from Metadata.py

db : database connection

data2img(*fname*, *imgname*, *zipname*, *sattype*, *granule*, *zipfile*, *sar_meta*, *unzipdir*)
Opens an image file and converts it to the format given in the config file

Parameters

fname : image filename (i.e. R1_980705_114117.img OR product.xml)

imgname : image name (i.e. R1_980705_114117)

zipname : zipname

sattype : satelite platform

granule : granule name

zipfile : zipfile

sar_meta : instance of the Metadata class

unzipdir : unzip directory

handler(*signum*, *frame*)
Handles exceptions - most notably time out errors

proc_Dir(*path*, *pattern*)
Locates satelite image raw data files (zipfiles) using a *pattern* in *path* search method, and then calls createImg() to process the data into image.

Parameters

path : directory tree to scan

pattern : file pattern to discover

proc_File(*zipfile*)
Locates a single satellite image zip file and processes it according to the config file. Note this cannot be nested in proc_dir since the logging structure and other elements must parallelizable

Parameters

zipfile : a valid zipfile name with full path

retrieve(*zipfile*)
Given a zip file name this function will: find out what satellite it is, unzip it, get instance of metadata, then dependant on the config, save metadata in a file and/or one of the following: Process to image or process to database.

Parameters

zipfile : A valid zipfile name with full extention

scientific(*sar_img*, *granule*, *zipname*)

Process images 'Scientifically', based on an ROI in the database, and per zipfile: -Qry to find what polygons in the ROI overlap this image -Process one polygon at a time (Project, crop, and mask), saving each as its own img file

Parameters

sar_img : instance of the Image class

granule : granule name

zipname : name of zipfile, no path

Metadata

class SigLib.**Metadata**(*granule*, *imgname*, *path*, *zipfile*, *sattype*, *loghandler=None*)
This is the metadata class for each image RSAT2, RSAT1 (ASF and CDPF)

__init__(*granule*, *imgname*, *path*, *zipfile*, *sattype*, *loghandler=None*)
This initializes the class based on input

Parameters

granule : unique name of the image (String)

imgname : name of the file to open, representing the image (String)

path : path to the image (String)

Zipfile : A valid zipfile name with full path (String)

sattype : type of data (String)

loghandler : A valid pre-set loghandler (Optional)

Returns

An instance of Meta

Metadata.py

Created on Jan 1, 2009 **@author:** Derek Mueller

This module creates an instance of class Meta and contains functions to query raw data files for metadata which is standardized and packaged for later use, output to file, upload to database, etc.

This source code to extract metadata from CEOS-format RADARSAT-1 data was developed by Defence Research and Development Canada [Used with permission]

Modified on Wed May 23 11:37:40 2018 **@reason:** Sent instance of Metadata to data2img instead of calling Metadata again **@author:** Cameron Fitzpatrick

class Metadata.**Metadata**(*granule*, *imgname*, *path*, *zipfile*, *sattype*, *loghandler=None*)

This is the metadata class for each image RSAT2, RSAT1 (ASF and CDPF)

clean_metaASF(*result*)

Takes meta data from origmeta and checks it for completeness, coerces data types splits values, if required and puts it all into a standard format

NOT TESTED!!

Parameters

result : Dictionary of metadata

clean_metaCDPF(*result*)

Takes meta data from origmeta and checks it for completeness, coerces data types splits values, if required and puts it all into a standard format

Parameters

result : A dictonary of metadata

createMetaDict()

Creates a dictionary of all the metadata fields for an image this can be written to file or sent to database

Note that the long boring metadata fields are not included

Returns

metadict : Dictionary containing all the metadata fields

extractGCPs(*interval*)

Description needed!

Parameters

interval :

Returns

tuple(gcps) : GCP's returned in tuple format

getASFMetaCorners(*ASFName*)

Use ASF Mapready to generate the metadata

Parameters

ASFName : ?

getASFProductType(*ASFName*)

Description needed!

Parameters

ASFName : ?

getCEOSmetafile()
Get the filenames for metadata

getCornerPoints()
Given a set of geopts, calculate the corner coords to the nearest 1/2 pixel. Assumes that the corners are among the GCPs (not randomly placed)

getDimgname()

Create a filename that conforms to my own standard naming convention:
yyyymmdd_HHmmss_sat_beam_pol...

Returns

dimgname : Name for image file conforming to standards above

getMoreGCPs(*n_gcps*)
If you have a CDPF RSat1 image, gdal only has 15 GCPs Perhaps you want more? If so, use this function. It will grab all the GCPs available (3 on each line) and subselect n_gcps of these to return.

The GCPs will not necessarily be on the 'bottom corners' since the gcps will be spaced evenly to get n_gcps (or more if not divisible by 3) If you want corners the only way to guarentee this is to set n_gcps = 6

Parameters

n_gcps : # of GCP's to return

Returns

tuple(gcps) : # of GCP's specified returned in tuple format

getRS2metadata()
Open a Radarsat2 file and get all the required metadata

get_ceos_metadata(**file_names*)
Take file names as input and return a dictionary of metadata file_names is a list of strings or a string (with one filename)

This source code to extract metadata from CEOS-format RADARSAT-1 data was developed by Defence Research and Development Canada [Used with Permission]

Parameters

**file_names* : List of strings

Returns

result : Dictionary of Metadata

getgdalmeta()
Open file with gdal and get metadata

Returns

gdal_meta : Metadata found by gdal

saveMetaFile(*dir=''*)
Makes a text file with the metadata

Metadata.**byte2int**(*byte*)
Reads a byte and converts to integer

Metadata.**date2doy**(*date*, *string=False*, *float=False*)
Give a python datetime and get an integer or string doy fractional doy returned if float=True

Metadata.**datetime2iso**(*datetimeobj*)
Return iso string from a python datetime

Metadata.**getEarthRadius**(*ellip_maj*, *ellip_min*, *plat_lat*)
Calculates the earth radius at the latitude of the satellite from the ellipsoid params

Metadata.**getGroundRange**(*slantRange*, *radius*, *sat_alt*)
Finds the ground range from nadir which corresponds to a given slant range must be an slc image, must have calculated the slantRange first

Metadata.**getSlantRange**(*gsr*, *pixelSpacing*, *n_cols*, *order_Rg*, *groundRangeOrigin=0.0*)

gsr = ground to slant range coefficients -a list of 6 floats pixelSpacing - the img. res., n_cols - how many pixels in range ground range orig - for RSat2 (seems to be zero always)

Valid for SLC as well as SGF

Metadata.**getThetaPixel**(*RS*, *r*, *h*)
Calc the incidence angle at a given pixel

Metadata.**getThetaVector**(*n_cols*, *slantRange*, *radius*, *sat_alt*)
Make a vector of incidence angles in range direction

Metadata.**get_data_block**(*fp*, *offset*, *length*)
gets a block of data from file

Metadata.**get_field_value**(*data*, *field_type*, *length*, *offset*)
Description needed!

Parameters

data :

field_type :

length :

offset :

Returns

converted data_str

Metadata.**readdate**(*date*, *sattype*)
Takes a rsat2 formated date 2009-05-31T14:43:17.184550Z and converts it to python datetime

Image Processing

imgProcess.py

Created on ??? Jul ? ??:??:?? 2009 **@author:** Derek Mueller

This module creates an instance of class Img and opens a file to return a gdal dataset to be processed into an amplitude, calibrated, noise or theta (incidence angle) image, etc. This image can be subsequently projected, cropped, masked, stretched, etc.

Modified on ??? Feb ? ??:??:?? 2012 **@reason:** Repackaged for r2convert **@author:** Derek Mueller **Modified on** 23 May 14:43:40 2018 **@reason:** Added logging functionality **@author:** Cameron Fitzpatrick

class Image.**Image**(*fname*, *path*, *meta*, *imgType*, *imgFormat*, *zipname*, *imgDir*, *loghandler=None*)
This is the Img class for each image. RSAT2, RSAT1 (ASF and CDPF)

Opens the file specified by fname, passes reference to the meta class and declares the imgType of interest.

Parameters

fname : filename

path : full directory path of the filename

meta : reference to the meta class

imgType : amp, sigma, noise, theta...

imgFormat : gdal format code gtiff, vrt

zipname : name of the image's zipfile

applyStretch(*stats*, *procedure='std'*, *sd=3*, *bitDepth=8*, *sep='tog'*)
Given stats... will stretch a multiband image to the dataType based on procedure (either sd for standard deviation, with +ve int in keyword sd, or min-max, also a linear stretch).

!!A nodata value of 0 is used in all cases!!

!!For now, dataType is byte and that's it!!

Note: gdal_translate -scale does not honour nodata values See: http://trac.osgeo.org/gdal/ticket/3085

Have to run this one under the imgWrite code. The raster bands must be integer, float? or byte and int data assumed to be only positive. Won't work very well for dB scaled data (obviously) it is important that noData is set to 0 and is meaningful.

sep = separate: applies individual stretches to each band (=better visualization/contrast)

tog = together: applies the same stretch to all bands (looks for the band with the greatest dynamic range) (=more 'correct')

For further ideas see: http://en.wikipedia.org/wiki/Histogram_equalization

cleanFiles(*levels=['crop']*)
Removes files that have been written.

Input a list of items to delete: raw, nil, proj,crop

Parameters

levels : a list of different types of files to delete

combineTif(*imgdir*, *zipname*, *mergedir*)
Takes a set of tif images made in scientific mode and combines them into a single tif image

Parameters

imgdir : directory containing images to combine

*zipname : name of zipfile (no path)

mergedir : directory containing gdal_merge.py

cropBig(*llur*, *subscene*)
Here we have a way to crop that will expand the area of an image. However, this uses gdalwarp - and resampling/offsetting could skew result - by a fraction of a pixel obviously, but still..

Parameters

llur : list/tuple of tuples in projected units

subscene : the name of a subscene

cropImg(*ullr*, *subscene*)
Given the cropping coordinates, this function tries to crop in a straight-forward way. If this cannot be accomplished (likely because the corner coordinates of an image are not known to a sufficient precision) then gdalwarp (cropBig) will do the job.

Parameters

ullr : upper left and lower right coordinates

subscene : the name of a subscene

cropSmall(*urll*, *subscene*)
This is a better way to crop b/c no potential for warping... However, this will only work if the region falls completely within the image.

Parameters

urll : list/tuple of tuples in projected units

subscene : the name of a subscene

decomp(*format='imgFormat'*)
Takes an input ds of a fully polarimetric image and writes an image of the data using a decomposition - could be 1) pauli

TODO: 2) freeman 3) cloude

Differs from imgWrite b/c it ingests all bands at once...

fnameGenerate(*projout=None*, *subset=None*, *band=None*)
Decide on some parameters based on self.imgType we want...

Returns

bands : Integer

dataType : GDal data type

outname : String

getAmp(*datachunk*)
return the amplitude, given the amplitude... but make room for the nodata value by clipping the highest value...

getBandData(*band*)
opens an img file and reads in data from a given band assume that the dataset is small enough to fit into memory all at once

Returns

imgData : data from the given band

xSpacing :

ySpacing :

getImgStats()
Opens a raster and calculates (approx) the stats returns an array - 1 row per band cols: band, dynamicRange, dataType, nodata value, min, max, mean, std

Returns

stats : stats from raster returned in an array of 1 row per band

getMag(*datachunk*)
return the magnitude of the complex number

getNoise(*n_lines*)
For making an image with the noise floor as data

getPhase(*datachunk*)
Return the phase (in radians) of the data (must be complex/SLC)

getSigma(*datachunk*, *n_lines*)
Calibrate data to Sigma Nought values (linear scale)

getTheta(*n_lines*)
For making an image with the incidence angle as data

imgWrite(*format='imgFormat'*, *stretchVals=None*)
Takes an input ds and writes an image.

self.imgType could be 1) amp, 2) sigma, 3) noise, 4) theta

all bands are output (amp, sigma)

Also used to scale an integer img to byte with stretch, if stretchVals are included

Note there is a parameter called chunk_size hard coded here that could be changed If you are running with lots of RAM

makePyramids()
Uses gdaladdo to make pyramids aux style

maskImg(*mask*, *vectdir*, *side*, *imgType*)
Masks all bands with gdal_rasterize using the 'layer'

side = 'inside' burns 0 inside the vector, 'outside' burns outside the vector

Note: make sure that the vector shapefile is in the same proj as img (Use reprojSHP from ingestutil)

Parameters

mask : a shapefile used to mask the image(s) in question

vectdir : directory where the mask shapefile is

side : 'inside' or 'outside' depending on desired mask result

imgType : the image type

openDataset(*fname*, *path=''*)
Opens a dataset with gdal

Parameters

fname : filename

projectImg(*projout*, *projdir*, *format=None*, *resample='bilinear'*, *clobber=True*)
Looks for a file, already created and projects it to a vrt file.

Parameters

projout : projection base name

projdir : path to the projection

format : the image format, defaults to VRT

resample : resample method (as per gdalwarp)

clobber : True/False should old output be overwritten?

NOTE THE PIXEL SIZE IS NOT PROSCRIBED! (it will be the smallest possible)

reduceImg(*xfactor*, *yfactor*)
Uses gdal to reduce the image by a given factor (i.e, factor 2 is 50% smaller or half the # of pixels) and saves as a temporary file and then overwrites.

Parameters

xfactor : float

yfactor : float

stretchLinear(*datachunk*, *scaleRange*, *dynRange*, *minVal*, *offset=0*)
Simple linear rescale: where min (max) can be the actual min/max or mean+/- n*std or any other cutoff

Note: make sure min/max don't exceed the natural limits of dataType takes a numpy array datachunk the range to scale to, the range to scale from, the minVal to start from and an offset required for some stretches (see applyStretch keyword sep/tog)

Parameters

datachunk : array

scaleRange : Range to scale to

dynRange : Range to scale from

minVal : strating min value

Returns

stretchData : datachunk, now linearly rescaled

vrt2RealImg(*subset=None*)
When it is time to convert a vrt to a tiff (or even img, etc) use this

Database

Database.py

Created on Tue Feb 12 23:12:13 2013 **@author:** Cindy Lopes

This module creates an instance of class Database and connects to a database to create, update and query tables.

Tables of note include:

tblmetadata - a table that contains metadata that is gleaned by a directory scan **roi_tbl** - a table with a region of interest (could be named something else) **trel_roiinst_con** or _int - a relational table that results from a spatial query **tblArchive** - a copy of the metadata from the CIS image archive

Other tables could contain data from drifting beacons or other data

Modified on 23 May 14:43:40 2018 **@reason:** Added logging functionality **@author:** Cameron Fitzpatrick

class Database.**Database**(*dbname*, *loghandler=None*, *user=None*, *password=None*, *port='5432'*, *host='localhost'*)
This is the Database class for each database connection.

Creates a connection to the specified database. You can connect as a specific user or default to your own username, which assumes you have privileges and a password stored in your ~/.pgpass file

Note: if you have any issues with a bad query, send a rollback to the database to reset the connection. >>>>db.connection.rollback()

Parameters

dbname : database name

user : user with sudo access on the specified databse (i.e. postgres user has sudo access to all databses)

password : password to go with user

port : server port (i.e. 5432)

host : hostname (i.e. localhost)

alterTimestamp(*shpTable*)
Takes a shape file and converts the gps_time from character type to timestamp time.

Parameters

shpTable :

beaconShapefilesToTables(*dirName*)
Takes a directory containing beacon shape files and converts them to tables and inserts them into the database appending *beacon_* before the name

Parameters

dirName :

bothArchiveandMetadata()
Finds all the results in both the archive and tblmetadata.

checkTblArchiveOverLapsTblMetadata(*filename*)
Check if a file name from tblArchive is in the overlap table.

Parameters

filename :

Returns

dictionary :

copyfiles(*copylist*, *wrkdir*)
Copies files from cisarchive. If file could not be found, check that the drive mapping is correct (above).

Parameters

copylist : a list of images + inst

wrkdir : working directory

copylistExport(*copylist*, *fname*)
Saves the copylist as a text file named fname.txt in the current dir.

Parameters

copylist : a list of images - catalog ids or files

fname : filename to write copylist to

copylistImport(*fname*)
Reads the copylist text file named fname.txt in the current dir.

Parameters

fname : filename to read copylist from

Returns

new_copylist : a list of images + inst

createTblMetadata()
Creates a metadata table called *tblmetadata*. It overwrites if *tblmetadata* already exist.

customizedQuery(*attributeList*, *roi*, *spatialrel*, *proj*)
Customizable query that takes an list of attributes to search for, a roi, a spatialrel, and a proj and returns a dictionary with all the requested attributes for the results that matched the query

Parameters

attributeList :

roi : region of interest file

spatialrel : Spatial relationship (ST_Contains or ST_Intersects)

proj : Desired image projection

Returns

copylist : a list of image catalog ids

instimg : a list of each instance and the images that correspond

dbProj(*proj*)
Relates *proj* the name (ie. proj.wkt) to *proj* the number (i.e. srid #). from Metadata import Metadata
Parameters

proj : projection name

Returns

srid : spatial reference id number of that projection

exportToCSV(*qryOutput*, *outputName*)
Given a dictionary of results from the database and a filename puts all the results into a csv with the filename outputName

Parameters

qryOutput : output from a query - needs to be a tupple - numpy data and list of column names

outputName : the file name

imgData2db(*imgData*, *xSpacing*, *ySpacing*, *bandName*, *inst*, *dimgname*, *granule*)
Here are the data in an array... upload to database need the imgData, the imgType, the bandName, the inst, dimgname and granule

will compute the count, mean, std, min, max for non-zero elements and send them to db as well

Parameters

imgData :

xSpacing :

ySpacing :

bandName :

inst : instance id (i.e. a 5-digit string)

dimgname : Derek's image name

granule : granule name

instimg2db(*roi*, *spatialrel*, *instimg*, *mode='refresh'*)

There can be several relational tables that contain the name of an image and the feature that it relates to: For example: a table that shows what images intersect with general areas or a table that lists images that contain ROI polygons...

This function runs in create mode or refresh mode Create - Drops and re-creates the table

Refresh - Adds new data (leaves the old stuff intact)

Parameters

roi : region of interest table

spatialrel : spatial relationship (i.e. ST_Contains or ST_Intersect)

instimg : a list of only images of that instance id

copylist : a list of images + inst

mode : create or refresh mode

instimgExport(*instimg*, *fname*)

Saves the instimg listing as a csv file named fname.csv in the current dir.

Parameters

instimg : a list of images and where they cover

fname : filename to write to

meta2db(*metaDict*, *overwrite=False*)

Uploads image metadata to the database as discovered by the meta module. *meta* is a dictionary - no need to upload all the fields (some are not included in the table structure)

Note that granule and dimgname are unique - as a precaution - a first query deletes records that would otherwise be duplicated This assumes that they should be overwritten!

Parameters

metaDict : dictionnary containing the metadata

nameTable(*roi*, *spatialrel*)

Automatically gives a name to a relational table

Parameters

roi : region of interest

spatialrel : spatial relationship (i.e. ST_Contains or ST_Intersect)

Returns

name : name of the table

numpy2sql(*numpyArray*, *dims*)

Converts a 1- or 2-D numpy array to an sql friendly array Do not use with a string array!

Parameters

numpyArray : numpy array to convert

dims : dimension (1 or 2)

Returns

array_sql : an sql friendly array

qryCropZone(*granule*, *roi*, *spatialrel*, *proj*, *inst*, *metaTable*)
Writes a query to fetch the bounding box of the area that the inst polygon and image in question intersect. returns a crop ullr tupple pair in the projection given

Parameters

granule : granule name

roi : region of interest file

spatialrel : spatial relationship (i.e. ST_Contains or ST_Intersect)

proj : projection name

inst : instance id (i.e. a 5-digit string)

metaTable : metadata table containing data of images being worked on

Returns

ullr : upper left, lower right tupple pair in the projection given

qryFromFile(*fname*, *path*, *output=False*)
Runs a query in the current databse by opening a file - adds the path and .sql extension - reading contents to a string and running the query

Note: do not use % in the query b/c it interfers with the pyformat protocol used by psycopg2

Parameters

fname : file name (don't put the sql extension, it's assumed)

path : full path to fname

output : make true if you expect/want the query to return results

qryFromText(*sql*, *output=False*)
Runs a query in the current databse by sending an sql string

Note: do not use % in the query b/c it interfers with the pyformat protocol used by psycopg2; also be sure to triple quote your string to avoid escaping single quotes; IF EVER THE Transaction block fails, just conn.rollback();try to use pyformat for queries - see dbapi2 (PEP); you can format the SQL nicely with an online tool - like SQLinForm

Parametersself.logger.debug('Intermediate file cleanup done')

sql : the sql text that you want to send

output : make true if you expect/want the query to return results

Returns

The result of the query as a tupple containing a numpy array and the column names as a list (if requested and available)

qryGetInstances(*granule*, *roi*, *proj*, *metaTable*)
Writes a quprojectImgery to fetch the instance names that are associated spatially in the relational table.

Parameters

granule : granule name

roi : region of interest file

proj : projection name

metaTable : metadata table containing data of images being worked on

Returns

instances : instances id (unique for entire project, i.e. 5-digit string)

qryMaskZone(*granule*, *roi*, *proj*, *inst*, *metaTable*)

Writes a query to fetch the gml polygon of the area that the inst polygon and image in question intersect from the ROI specified. returns gml text for conversion to shp

Parameters

granule : granule name

roi : region of interest file

proj : projection name

inst : instance id (i.e. a 5-digit string)

metaTable : metadata table containing data of images being worked on

Returns

polytext : gml text

qrySelectFromAvailable(*roi*, *selectFrom*, *spatialrel*, *proj*)

Given a table name (with polygons, from/todates), determine the scenes that cover the area from start (str that looks like iso date) to end (same format).

Eventually include criteria: subtype - a single satellite name: ALOS_AR, RADAR_AR, RSAT2_AR (or ANY) beam - a beam mode

comes back with - a list of images+inst - the bounding box

Parameters

roi : region of interest table in the database

spatialrel : spatial relationship (i.e. ST_Contains or ST_Intersect). Does the image contain the ROI polygon or just intersect with it?

proj : projection name

selectFrom : table in the database to find the scenes

Returns

copylist : a list of image catalog ids

instimg : a list of each instance and the images that correspond

sql2numpy(*sqlArray*, *dtype='float32'*)

Comming from SQL queries, arrays are stored as a list (or list of lists) Defaults to float32

Parameters

sqlArray : an sql friendly array

dtype : default type (float32)

Returns

list : list containing the arrays

updateFromArchive(*archDir*)

Goes to CIS Archive metadata shapefiles and (re)creates and updates tblArchive in the connected database tblArchive then represents all the image files that CIS has (in theory) The first thing this script does is define the table - this is done from an sql file and contains the required SRID Then it uses ogr2ogr to

upload each shp in the archDir The script looks for the *.last files to know which files are the most current (these need to be updated)

Can be extended to import from other archives (In the long term - PDC?)

Parameters

archDir : archive directory

updateROI(*inFile*, *proj*, *wdir*, *ogr=False*)

This function will update an ROI (Region Of Interest) table in the database. It has a prescribed format It will take the shapefile named inFile and update the database with the info

Note that this will overwrite any table named *inFile* in the database

The generated table will include a column *inst* - a unique identifier created by concatenating obj and instid

Parameters

inFile : basename of a shapefile (becomes an roi table name too)

proj : projection name

wdir : Full path to directory containing ROI (NOT path to ROI itself, just the directory containing it)

Required in ROI

obj - the id or name of an object/polygon that defines a region of interest. Very systematic, no spaces.

instid - a number to distinguish repetitions of each obj in time or space. For example an ROI that occurs several summers would have several instids.

fromdate - a valid iso time denoting the start of the ROI - can be blank if imgref is used

todate - a valid iso time denoting the start of the ROI - can be blank if imgref is used

Optional in ROI

imgref - a reference image dimage name (for a given ROI) - this can be provided in place of datefrom and dateto

name - a name for each obj (Area51_1950s, Target7, Ayles)

comment - a comment field

Any other field can be added...

update_NTAI_FLUX_ROI(*inFile*, *path*, *proj*)

Goes to a shapefile named inFile and updates the postGIS database Assumes dbase postgis exists and that outTable does as well - this overwrites!

Parameters

inFile : basename of a shapefile

path : full path to inFile

proj : projection name

Required

name, inst, obj, type, fromdate, todate(optional)

fromdate - a valid time_start (ie it is here, when was it here?)

todate - a valid time_end (ie it is here, when was it here?)

inst - an instance id (unique for entire project) - Nominally a 5-digit string

Optional

refimg - a reference image name (ie how do you know it was here)

type - an ice type (ie ice island, ice shelf, mlsi, fyi, myi, epishelf, open water)

subtype - an ice subtype (ie ice island could be iced firn, basement; open water could be calm, windy)

comment - a comment field

name - a name (Target7, Ayles)

obj - an object id tag (to go with name but very systematic: 2342, and if it splits 2342_11 & 2342_12)

Utilities

util.py

This module contains miscellaneous code that helps siglib work with directories, zip files, clean up intermediate files and so on.

Created on Tue Feb 12 20:04:11 2013 **@author:** Cindy Lopes **Modified on** Sat Nov 23 14:49:18 2013 **@reason:** Added writeIssueFile and compareIssueFiles **@author:** Sougal Bouh Ali **Modified on** Sat Nov 30 15:37:22 2013 **@reason:** Redesigned getFilname, getZipRoot and unZip **@author:** Sougal Bouh Ali **Modified on** Wed May 23 14:41:40 2018 **@reason:** Added logging functionality **@author:** Cameron Fitzpatrick

`Util.`**`az`**(*pt1*, *pt2*)

Calculates the great circle initial azimuth between two points in dd.ddd format. This formula assumes a spherical earth. Use Vincenty's formulae for better precision

https://en.wikipedia.org/wiki/Azimuth https://en.wikipedia.org/wiki/Vincenty%27s_formulae

Parameters:

pt1 : point from (tuple of lon and lat)

pt2 : point to (tuple of lon and lat)

Returns

az : azimuth from North in degrees

`Util.`**`cleartree`**(*dirname*)

Delet all files in a certain path

Parameters

dirname :

`Util.`**`compareIssueFiles`**(*file1*, *file2*, *loghandler=None*)

Compares 2 clean issue files generated by writeIssueFile() and generates a separate file containing the list of matched & unmatched files in 2 files.

Parameters

file1 : name of the first text file with extension to be compared (i.e. textfile.txt)

file2 : name of the second text file with extension to be compared (i.e. textfile.txt)

Util.**deltree**(*dirname*)
Delete all the files and sub-directories in a certain path

Parameters

dirname :

Util.**getFilename**(*zipname*, *unzipdir*, *loghandler=None*)
Given the name of a zipfile, return the name of the image, the file name, and the corresponding sensor/platform (satellite).

Parameters

zipname : The basename of the zip file you are working with

unzipdir : Where the zipfile will unzip to (find out with getZipRoot)

Returns

fname : The file name that corresponds to the image

imgname : The name of the image (the basename, sans extension)

sattype : The type of satellite/image format this file represents

Util.**getPowerScale**(*dB*)
Convert a SAR backscatter value from the log dB scale to the linear power scale

Note: dB must be a scalar or an array of scalars

Parameters

dB : backscatter in dB units

Returns

power : backscatter in power units

Util.**getZipRoot**(*zip_file*, *tmpDir*)
Looks into a zipfile and determines if the contents will unzip into a subdirectory (named for the zipfile); or a sub-subdirectory; or no directory at all (loose files)

Run this function to determine where the files will be unzipped to. If the files are in the immediate subfolder, then that is what is required.

Returns the unzipdir (where the files will -or should- go) and zipname (basename of the zipfile)

Parameters

zip_file : full path, name and ext of a zip file

tmpDir : this is the path to the directory where you are working with this file (the path of the zip_file - or wrkdir)

Returns

unzipdir : the directory where the zip file will/should unzip to

zipname : basename of the zip file AND/OR the name of the folder where the image files are

Util.**getdBScale**(*power*)
Convert a SAR backscatter value from the linear power scale to the log dB scale

Note: power must be a scalar or an array of scalars,negative powers will throw back NaN.

Parameters

power : backscatter in power units

Returns

dB : backscatter in dB units

Util.**llur2ullr**(*llur*)
a function that returns: upperleft, lower right when given... lowerleft, upper right a list of tupples [(x,y),(x,y)]

Note - this will disappoint if proj is transformed (before or after)

Parameters

llur : a list of tupples [(x,y),(x,y)] corresponding to lower left, upper right corners of a bounding box

Util.**reprojSHP**(*in_shp*, *vectdir*, *proj*, *projdir*)
Opens a shapefile, saves it as a new shapefile in the same directory that is reprojected to the projection wkt provided.

Note: this could be expanded to get polyline data from polygon data for masking lines (not areas) ogr2ogr -nlt MULTILINESTRING

Parameters

in_shp :

vectdir :

proj :

projdir :

Returns

out_shp : name of the proper shapefile

Util.**ullr2llur**(*ullr*)
a function that returns: lowerleft, upper right when given... upperleft, lower right a list of tupples [(x,y),(x,y)]

Note - this will disappoint if proj is transformed (before or after)

Parameters

ullr : a list of tupples [(x,y),(x,y)] corresponding to upper right, lower left corners of a bounding box

Util.**unZip**(*zip_file*, *unzipdir*, *ext='all'*)
Unzips the zip_file to unzipdir with python's zipfile module.

"ext" is a keyword that defaults to all files, but can be set to just extract a leader file L or xml for example.

Parameters

zip_file : Name of a zip file - with extension

unzipdir : Directory to unzip to

Optional

ext : 'all' or a specific ext as required

Util.**wkt2shp**(*shpname*, *vectdir*, *proj*, *projdir*, *wkt*)
Takes a polygon defined by well-known-text and a projection name and outputs a shapefile into the current directory

Parameters

shpname :

vectdir :

proj :

projdir :

wkt :

Util.**wktpoly2pts**(*wkt*, *bbox=False*)

Converts a Well-known Text string for a polygon into a series of tuples that correspond to the upper left, upper right, lower right and lower left corners

This works with lon/lat rectangles.

If you have a polygon that is not a rectangle, set bbox to True and the bounding box corners will be returned

Note that for rectangles in unprojected coordinates (lon/lat deg), this is slightly different from ullr or llur (elsewhere in this project) which are derived from bounding boxes of projected coordinates

Parameters

wkt : a well-known text string for a polygon

Returns

ul,ur,lr,ll : a list of the four corners

Util.**writeIssueFile**(*fname*, *delimiter*, *loghandler=None*)

Generates a clean list of zipfiles, when given an Issue File written by scripts.

Parameters

fname : name of the text file with extension to be cleaned(i.e. textfile.txt)

delimiter : separator used to split zipfile from unwanted errors (i.e. use most common " / " before zipfile)

Symbols

A

B

C

D

E

F

G

H