WATER & ICE RESEARCH LAB

WIRL

SigLib Documentation

Release 2.9

Mueller, Fitzpatrick, Plourde, Romero, Bouh-Ali, Lopes

Oct 08, 2021

CONTENTS

CHAPTER

ONE

OVERVIEW OF THE PROJECT

1.1 Introduction

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, Mac, or Linux.

There are 3 main ''modes" that it can run in (or combinations of these)

1. A data **Query Mode** where remote sensing scenes by querying various SAR imagery sources including EODMS, Copernicus Open-Access Hub, and local harddrives. 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. A **Qualitative 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 csv/txt list of file paths.
3. A **Quantitative Mode** where a quantitative analysis of the radar signature or backscatter (also known as the normalized radar cross section (nrcs) or sigma nought) is performed. It can extract the sigma nought values within imagery that coincides with specific regions of interest (ROIs) and derive statistics on this SAR signature. These statistics can be stored in a database table or CSV file for further analysis.

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 either input based on a recursive **scan** of a directory for files that match a pattern, or you can input one **file** at a time (useful for parallelization, when many processes are spawned by gnu parallel).

1.2 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). At CIS he benefited from discussions with Ron Saper, Angela Cheng and his 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, Sougal Bouh-Ali, Cameron Fitzpatrick, Allison Plourde, and Jazmin Romero. Ron Saper, Anna Crawford and Greg Lewis-Paley helped out as well (indirectly).

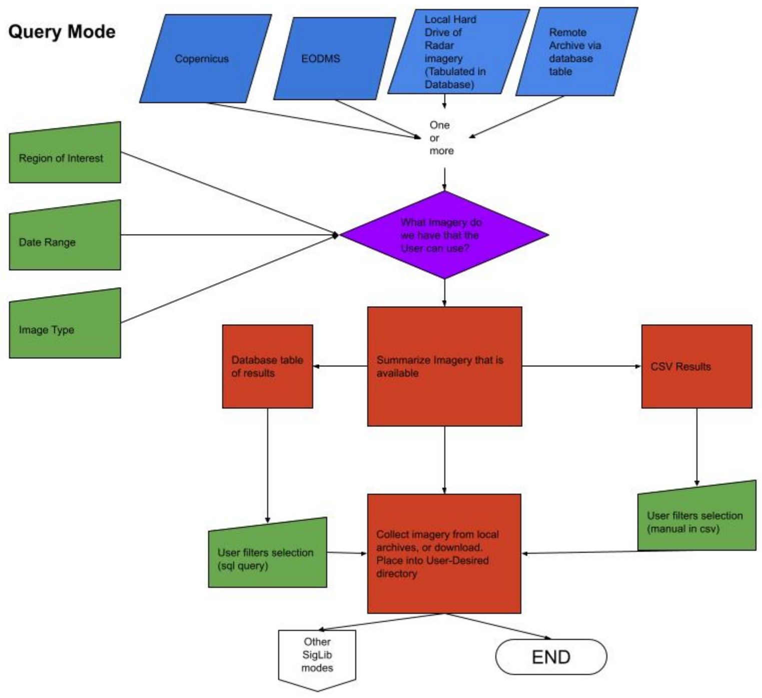

Query Mode
Copernicus
EODMS
Local Hard Drive of Radar imagery (Tabulated in Database)
Remote Archive via database table
One or more
Region of Interest
Date Range
Image Type
What Imagery do we have that the User can use?
Database table of results
Summarize Imagery that is available
CSV Results
User filters selection (sql query)
Collect imagery from local archives, or download. Place into User-Desired directory
User filters selection (manual in csv)
Other SigLib modes
END

Fig. 1: Flowchart depecting the basics of Query Mode (This is the end goal, we are currently not at this stage!)

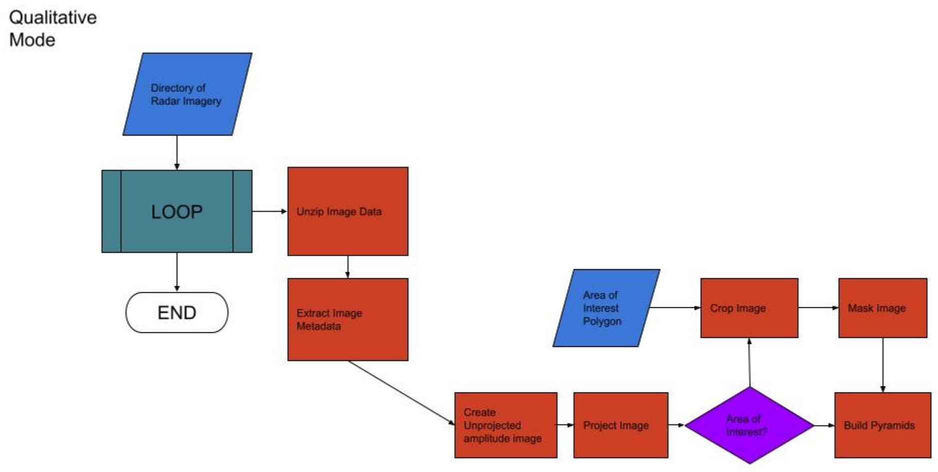


Fig. 2: Flowchart depecting the basics operations performed in Qualitative Mode.

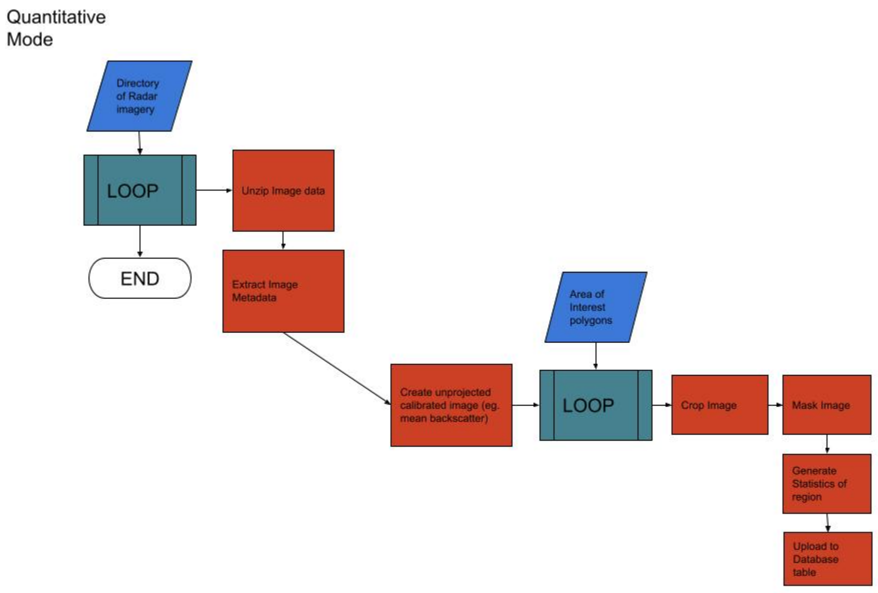

Quantitative Mode
Directory of Radar imagery
LOOP
Unzip Image data
END
Extract Image Metadata
Area of Interest polygons
Create unprojected calibrated image (eg. mean backscatter)
LOOP
Crop Image
Mask Image
Generate Statistics of region
Upload to Database table

Fig. 3: Flowchart depecting the basics operations performed in Quantitative Mode.

CHAPTER

TWO

OVERVIEW OF SIGLIB.PY AND ITS DEPENDENCIES

2.1 Dependencies

You will need a computer running Linux, Windows, or MacOS along with:

Python 2.x or 3.x (preferred). It is recommended you install the Python Anaconda package manager as it contains pretty well everything you will need related to Python. The following libraries are needed, and can be installed individually using your preferred package manager (eg. pip, conda).

- future
- psycopg2
- GDAL
- numpy
- scipy
- pandas
- geopandas
- configparser
- func-timeout

If using Query mode, the following additional libraries are needed:

- eodms_api_client
- sentinelsat

Other requirements include:

- gdal/ogr libraries - (https://gdal.org/)
- PostrgreSQL/PostGIS (could be on another computer) (https://www.postgresql.org/ and https://postgis.net/)

Note that it is possible to run the software without PosgresSQL/PostGIS but functionality will be limited.

Nice to have:

- It is highly recommended that you have access to QGIS or ArcGIS to manipulate shapefiles
- You should have a good Python Integrated Development Environment (IDE) - for example: Spyder, which can be installed via Anadonda
- To work with ASF CEOS Files, you will need ASF MapReady software (https://asf.alaska.edu/how-to/data-tools/data-tools/)

- For Linux users, GNU Parallel (https://www.gnu.org/software/parallel/) can be used to drastically reduce runtimes by running SigLib on multiple computer cores.

Details on how to install and set up these dependencies is out of scope for this manual.

2.2 Setup

Once all the dependencies are met you can set up the SigLib software

2.2.1 SigLib

Depending on your level of experience with coding and, in particular, Python, this portion of the the setup should take about an hour for those who are familiar with setting up code repositories. If you are a novice programmer you may want to set aside more time then that.

- Download or clone the latest version of SigLib from Github (https://github.com/wirl-ice/SigLib - N.B. link is private to WIRL members for now)
- Install Python Libraries: The ideal method for doing this is via conda to create a new Python environment with the SigLib dependancies. Read up on Anaconda to learn how this is done.
- Setup Directories: you will need to create a set of directories (folders) that SigLib will access through the config.cfg file. The contents of each folder will be explained later on, for now you just need to create empty folders. They should be named to reflect the associated variable in the config.cfg file, for example, create a folder named 'ScanDirectory' to link to the 'scanDir' variable.
- Config File Setup: Enter the paths to the directories you just created into your config file in the **Directories** section. If you know the name and host of the database you would like to use, enter these now into the **Database** section. If you are creating a new database, then refer to the *PostGIS* section.

2.2.2 Postgres/PostGIS

Whether you are accessing an external or local PostGIS database, you will need to take steps to set up your PostGIS database in such a way that SigLib.py can connect to it. The following provides an overview on how to add new users, create a new database, and add new projections. For those who are familiar with PostGres/PostGIS this setup should only take about an hour; if you are new to PostGres/PostGIS you will likely want to set aside a few hours.

- Setup/modify users in **PGAdmin** (Postgres GUI) or using **psql** (the command line utility)
- Ideally, the username should be the same as your username (or another user) on that computer

PGAdmin - Enter the Login/Group Roles dialog under Server - Create a user that can login and create databases. Ideally, the username should be the same as the username on that computer.

psql This method is for Linux users only, if you are using Windows see the above steps for adding new users in PGAdmin.

- At the command line type (where newuser is the new username) to create a user that can create databases:

createuser -d newuser

- Enter psql by specifying the database you want (use default database 'postgres' if you have not created one yet):

psql -d postgres

- Give the user a password like so:

password username

Once a user is set up, they can be automatically logged in when connecting to the Postgres server if you follow these steps (recommended). If not, the user will either have to type in credentials or store them hardcoded in the Python scripts (bad idea!).

Windows The PostgreSQL server needs to have access to the users password so that SigLib can access the database. This achieved through the pgpass.conf file, which you will need to create. - Navigate to the Application data subdirectory:

cd %APPDATA%

- Create a directory called postgresql and enter it:

mkdir postgresql cd postgresql

- Create a plain text file called pgpass.conf:

notepad pgpass.conf

- Enter the following information separated by colons –host:port:database:username:password – for example the following gives user *person* access to the postgres server on the localhost to all databases (*). The port number 5432 is standard:

localhost:5432:*:person:passwordforperson

- Save the file

Linux - Make a file called .pgpass in your home directory and edit it to include host:port:database:username:password (see above for details and example) - Save the file then type the following to make this info private: chmod 600 .pgpass

Permissions

If you are the first or only user on the postgres server then you can create databases and will have full permissions. Otherwise you will have read access to the databases that you connect to (typically). To get full permissions (recommended for SigLib) to an existing database do the following (to give user 'username' full permissions on database 'databasename'):

- **PGAdmin** – Under Tools, select Query tool, type the following and execute - lightning icon or F5:
- **psql** – At the pqsl prompt, type the following and press enter:

GRANT ALL PRIVILEGES ON DATABASE databasename TO username;

Creating a New Database

To create a new database you will need to have PostGIS installed on your machine. If you are using Windows it is recommended you install the PGAdmin GUI (this should be included with your installation of PostGIS). - Open a server in PGAdmin and create a new database. Set the '"db"' variable in the config file to the name of your new database. - Set the *host* variable in the config file to the 'Owner' of the database, this is typically your username for a local database setup. - Check that the 'spatial_ref_sys' table has been automatically created under '"Schemas|Tables"'. This table contains thousands of default projections; additionally new projections can be added (See *A Note on Projections*. If the table has yet to been created, you will have to add it manually. Under Tools in PGAdmin, select Query Tool, type the following and execute:

CREATE EXTENSION postgis;

- In the SigLib config file, set the *create_tblmetadata* variable to *1*. Leave all options in [Process] equal to 0.
- Save your config file with these changes and run SigLib.py

'"python /path_to_script/SigLib.py /path_to_file/config_file.cfg"'

- You will be prompted in the terminal to create/overwrite **tblMetadata**. Select yes to create a new metadata table.

2.2.3 Modules

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

1. **Util.py** - Several 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. **Query.py** - Used to discover and accumulate desired SAR imagery for a project.

SigLib.py is the front-end of the software. It calls the modules listed above and is 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. An example of this is included:

1. **Polarimetry.py** - An independant script used generate polarimetric variables for SAR imagery using SNAP-ESA..

2.2.4 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. Do not add comments or any additional text to the config file as this will prevent the program from interpreting the contents. Only update the variables as suggested in their descriptions. 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 files are found in well-known text (.wkt) format. This folder should be populated with any projection files that you plan to use in your analysis.
- vectDir = where vector layers are found (ROI shapefiles or masking layers)
- imgDir = a working directory for storing image processing intermediate files and final output files, in scratch folder
- logDir = where logs are placed
- outDir = where csv results from QueryMode are placed

Database

- db = the name of the database you want to connect to
- host = hostname for PostGIS server
- create_tblmetadata = 0 for append, 1 for overwrite/create. Must initially be set to 1 to initialize a new database.
- uploadROI = 1 if ROI file listed should be uploaded to the database
- metatable_name = database table containing image information that Database.py will query against

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
- 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, *.csv, or *.txt), use when path is 1

Process

- metaUpload = 1 when you want to upload image metadata to the metadata table in the database
- qualitative = 1 when you want to manipulate images (as per specs below) (Qualitative Mode)
- quanitative = 1 when you want to do image manipulation involving the database (Quantitative Mode)
- query = 1 when you want to find and retrieve SAR imagery

MISC

- proj = basename of wkt projection file (eg. lcc)
- projSRID = SRID # of wkt projection file
- imgtypes = The image type of the results (amp or sigma)
- imgformat = File format for output imagery (gdal convention)
- roi = name of ROI Shapefile for Discovery or Scientific modes, stored in your ''vectDir" folder
- roiprojSRID = Projection of ROI as an SRID for use by PostgreSQL (see *A Note on Projections|A Note on Projections]* for instructions on finding your SRID and ensuring it is available within your PostGIS database)
- mask = a polygon shapefile (one feature) to mask image data with.
- 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
- spatialrel = ST_Contains (Search for images that fully contain the roi polygon) or ST_Intersects (Search for images that merely intersect with the roi)
- elevationCorrection = the desired elevation (in meters) to georeference the tie-points. Enter an integer value (eg, 0, 100, 500). For example, when studying coastlines, the elevation of the study region is **0**. Leave blank to use the default georeferencing scheme (using average elevation of tie-points).
- uploadResults = 1 to upload descriptive statistics of subscenes generated by Quanitative mode to database

2.2.5 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
```

2.3 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 we have made this problem worse with our 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_band_data_proj.ext

Example: 20080630_225541_r1_scwa__hh_s_lcc.tif

Table: **dimgname fields**

Position	Meaning	Example	Chars
Date	year month day	20080630	8
Time	hour min sec	225541	6
Sat	satellite/platform/sensor	r1,r2,e1,en	2
Beam	beam for SAR, band combo for optical	st1__,scwa_,fqw20_,134__	5
Band	pol for SAR, meaning of beam for optical (tc = true colour)	hh, hx, vx, vv, hv, qp	2
Data	what is represented (implies a datatype to some extent)	a= amplitude, s=sigma, t=incidence,n=NESZ, o=optical	1
Proj	projection (not present in database)	nil, utm, lcc, aea	3
Ext	file extension	tif, rrd, aux, img	3

2.4 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*.

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’	Query
TODATE	String	ISO Date-time denoting the end of the time period of interest	‘2002-09-15 23:59:59’	Query
IMGREF	String	dimgname of a specific image known to contain the OBJ polygon (Spaces are underscores)	‘20020715 135903 r1 scwa hh s’	Quanitative
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
Notes	String	Comment field to explain the OBJ	‘Georeferencing may be slightly off here?’	Optional

- See folder ROISamples for example ROIs

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’]

2.5 A Note on Projections:

SigLib accepts projections in two ways; either as .wkt files or as SRID codes. An SRID value is required when working with PosgresSQL/PostGIS, but otherwise either is accepted. Within the config, there is an option to specify the name of a projection file (minus the path or extension) in the PROJ directory (*proj*), and an option to specify an SRID (*projSRID*). Warning! If both options are presented, the .wkt projection will supersede the SRID for all image manipulation! The only time both options are nessesary is if you are performing image manipuation with a .wkt projection, and need to specify the SRID for PostGIS. If this is the case, your projection is most-likely a custom one with an SRID not recognised by GDAL/OGR. If so, make sure it has been added to the spatial_ref_sys table in the database!

To add a custom spatial reference, download the desired projection in “PostGIS spatial_ref_sys INSERT statement” format from spatialreference.org. This option is an sql executable that can be run within PostgreSQL to add the desired projection into the spatial_ref_sys table.

2.6 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 *Qualitative* 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 *Quanitative* format. In this case, the principles are the same as in the *Qualitative* 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 *Qualitative* 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 *Quanitative* OBJ should change when you are hand digitizing a specific subsample from each OBJ from the *Qualitative* 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 *Quanitative* 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).

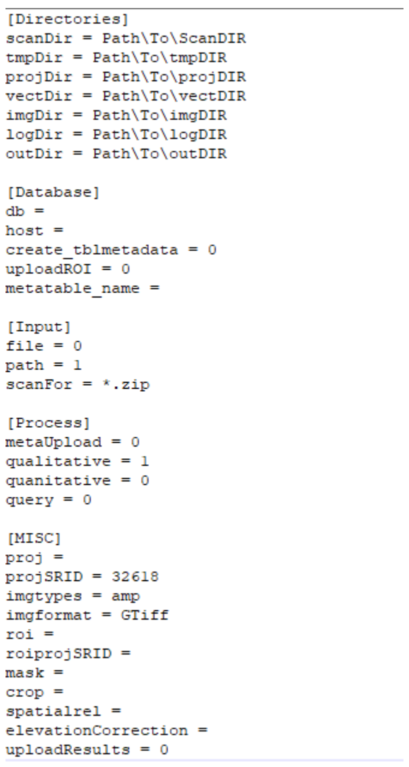
CHAPTER THREE

USING SIGLIB

This section gives a brief overview of how to use the Metadata, Util, Database, and Image functions via SigLib and its config file. These example assume the preseeding documentation has been read in detail, and proper setup has been performed as per the **Setup** section of this document. For examples on how to use the Query mode, please see the proceeding section.

3.1 Example #1: Basic Radarsat2 Image Calibration using SigLib (Qualitative Mode)

Qualitative mode can be utilized to create projected, byte-scale amplitude imagery from Radarsat-1, Radarsat-2, and Sentinel-1 products. The basic config settings needed to run this mode in its default configuration can be found in the figure below.



```
[Directories]
scanDir = Path\To\ScanDIR
tmpDir = Path\To\tmpDIR
projDir = Path\To\projDIR
vectDir = Path\To\vectDIR
imgDir = Path\To\imgDIR
logDir = Path\To\logDIR
outDir = Path\To\outDIR

[Database]
db =
host =
create_tblmetadata = 0
uploadROI = 0
metatable_name =

[Input]
file = 0
path = 1
scanFor = *.zip

[Process]
metaUpload = 0
qualitative = 1
quanitative = 0
query = 0

[MISC]
proj =
projSRID = 32618
imgtypes = amp
imgformat = GTiff
roi =
roiprojSRID =
mask =
crop =
spatialrel =
elevationCorrection =
uploadResults = 0
```

Fig. 1: A basic config file for this task

In its default settings, full size imagery will be produced, however, a simple crop or mask can be conducted in this mode if desired. If cropping is desired, the upper-left and lower-right coordinates of the crop need to be specified in the *crop* config option, the coordinates themselves in the image projection specified (IE crop = ULX ULY LRX LRY). For masking, place a shapefile of the desired mask into the vect directory, and specify the filename (minus the extension) in the *mask* config option. If a mask is provided, a crop zone does not need to be provided, as the image will automatically be cropped to the bounding area of the mask before masking.

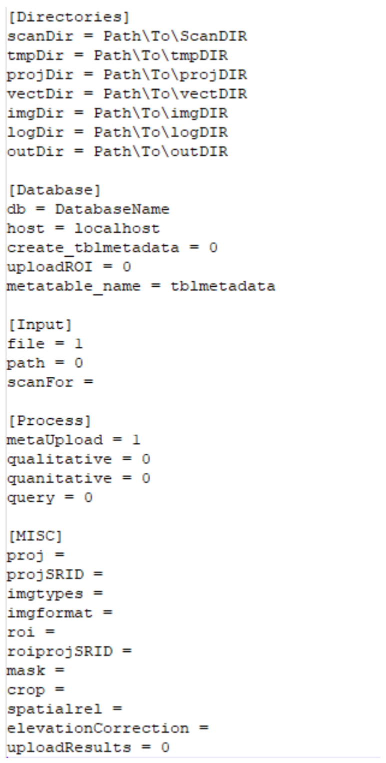
This config is using the *scanFor* option to find '.zip' files within the specified *SCANDIR* to process. The *scanFor* option can also be either '.csv' or '.txt' to look for these files within the *SCANDIR*. '.csv' or '.txt' files should contain a list of image filepaths for SigLib to process, which is a preferred option to '.zip' if the zipfiles are scattered around your computer.

3.2 Example #2: Accumulating Image Metadata in a local geodatabase (and intro to parallel processing)

The metaUpload function of SigLib is used to collect and upload metadata for any SAR imagery found locally to a geodatabase. A database table of local imagery can be useful as an archive for any accumulated products, and is also potentially nessesary for using SigLib's Quantitative mode. This example will also demonstrate how to run a SigLib mode in parallel using the Linux library GNU Parallel (https://www.gnu.org/software/parallel). GNU Parallel allows the user to run a common command in tandom on different computer cores, thus allowing for processing time of repetative tasks to decrease dramatically.

The basic config settings needed to run this job in parallel are presented in the figure below.

A review of the settings needed for this particular example can be seen in the figure below.



```
[Directories]
scanDir = Path\To\ScanDIR
tmpDir = Path\To\tmpDIR
projDir = Path\To\projDIR
vectDir = Path\To\vectDIR
imgDir = Path\To\imgDIR
logDir = Path\To\logDIR
outDir = Path\To\outDIR

[Database]
db = DatabaseName
host = localhost
create_tblmetadata = 0
uploadROI = 0
metatable_name = tblmetadata

[Input]
file = 1
path = 0
scanFor =

[Process]
metaUpload = 1
qualitative = 0
quanitative = 0
query = 0

[MISC]
proj =
projSRID =
imgtypes =
imgformat =
roi =
roiprojSRID =
mask =
crop =
spatialrel =
elevationCorrection =
uploadResults = 0
```

Fig. 2: Config file settings for discovering and uploading metadata.

In the case of the above figure, it is assumed that the metadata table has already been created, thus the *create_tblmetadata* option is set to 0. A metadata table created by SigLib is nessesary for this process, as it contains all the required metadata fields. If you have not done so, create tblmetadata as specified in **Setup** before continuing with this example!

The main parameter that differentiates parallel processing from standard processing is the *file* option in the **Input** section. This option configures SigLib to process SAR files via commandline input instead of via directory scan or csv/txt list. This means that SigLib is configured to process only a single file per run of the script, which is required for parallel processing to prevent the same file from being processed multiple times by multiple instances.

To start the parallel job:

1. Open a terminal

2. cd into the directory containing all your 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 /path/to/SigLib.py/ /path/to/config.cfg/** Where -j is the number of cores to use, and –nice is how nice the process will be to other processes).

The find command is looking for zipped SAR files in a directory we specify, then piping what's found to the parallel command one at a time. The parallel command manages our 16 cores, each of which runs an instance of SigLib processing a single zipfile from the find command. When processing of a file is completed by a core, and the script ends, parallel automatically starts a new instance of SigLib on that core with a zipfile from the find command. This is repeated until there is no further input from find.

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

Remember: this SigLib mode can be run without parallel by simply using the *path* option instead of the *file* option, and specifying a scan directory and filetype.

If you are on a Windows machine, and wish to run SigLib in parallel, it is possible to use the Multiprocessing Library in Python to call instances of SigLib. An example on how to do this will not be covered.

3.3 Example #3: Quantitative Mode

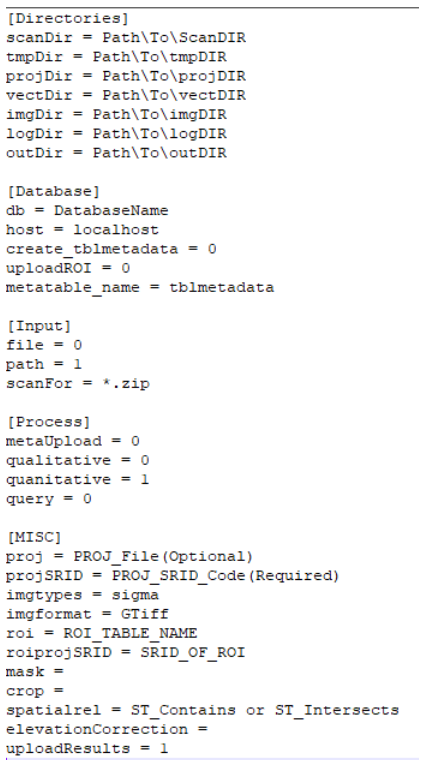
Quantitative mode is used for a quantitative analysis of the radar signature or backscatter (also known as the normalized radar cross section (nrcs) or sigma nought). It can extract the sigma nought values within imagery that coincides with specific regions of interest (ROIs) and derive statistics on this SAR signature. These statistics can be stored in a database table or CSV file for further analysis. The ROI should contain a series of polygons representing regions of interest in specific timeframes, and be uploaded to the database as a new table. This can be accomplished by running SigLib with the *uploadROI* option set to 1, and the name of an ROI shapefile located in the *VECTDIR* specified in *roi*. As per the creation of tblmetadata in the previous example, uploading the roi should be done before continuing with this example.

NOTE: This config setting **MUST** be 0 if running in parallel, or else the ROI will constantly be overwritten.

This mode also requires a database table with SAR image metadata to reference, like the one made in the previous example. This is needed to determine which polygons in the ROI overlap an image being processed, thus, any image being processed in this mode must have its metadata in the metatable.

An example config file for this mode is depicted in the figure below.

For this example, the *uploadResults* parameter is set to 1, meaning the image statistics for each ROI polygon will be uploaded to a results table in the database. If uploading results, you must create the results table called 'tblbanddata'. This can be done by running the "CreateResultsTable.sql" script located in the Extras folder in PGAdmin. Changing this parameter to 0 will instead output cropped and masked image data for each ROI polygon to the *IMGDIR*. Functionality to export image statistics as a csv is a work in progress.



```
[Directories]
scanDir = Path\To\ScanDIR
tmpDir = Path\To\tmpDIR
projDir = Path\To\projDIR
vectDir = Path\To\vectDIR
imgDir = Path\To\imgDIR
logDir = Path\To\logDIR
outDir = Path\To\outDIR

[Database]
db = DatabaseName
host = localhost
create_tblmetadata = 0
uploadROI = 0
metatable_name = tblmetadata

[Input]
file = 0
path = 1
scanFor = *.zip

[Process]
metaUpload = 0
qualitative = 0
quanitative = 1
query = 0

[MISC]
proj = PROJ_File(Optional)
projSRID = PROJ_SRID_Code(Required)
imgtypes = sigma
imgformat = GTiff
roi = ROI_TABLE_NAME
roiprojSRID = SRID_OF_ROI
mask =
crop =
spatialrel = ST_Contains or ST_Intersects
elevationCorrection =
uploadResults = 1
```

Fig. 3: Example config file settings for Quanitative Mode.

CHAPTER

FOUR

TUTORIAL ON HOW TO RUN SIGLIB/QUERY.PY OR STANDALONE SCRIPT QUERY_IMGS.PY.

To be able to run the SigLib on query mode, the first step consists of changing the config file (config.cfg) with query = 1 under Process section. After that simply type:

‘”python SigLib.py config.cfg”’

On the other hand, the standalone script can be run as follows:

‘”python query_imgs.py config.cfg”’

If the scripts successfully run, the following menu should appear on the screen:

```
Available Query Methods:

        1: tblmetadata: Query

        2: tblmetadata: Download

        3: CIS Archive (WIRL users only)

        4: EODMS: Query

        5: EODMS: Order

        6: EODMS: Download

        7: Copernicus: Query

        8: Copernicus: Download

        9: Execute Raw Sql Query

        0: Exit

> Please select the desired query method (0,1,2,3,4,5,6,7,8,9):
```

4.1 Setup the config file to run the query mode

The first step to run the query mode is to update the config file (config.cfg) as follows. Under the section *MISC*, update the parameters *roi*, *roiprojSRID*, and *spatialrel* with the name of the ROI shapefile (as it is found in the database), the projection, and the spatial relation (either *ST_Intersects* or *ST_Overlaps*).

In addition, under the section *Database*, the parameter *table* should contain the name of the local table to be queried (this only applies when querying local tables). Finally, an additional parameter should be added to the config file is *outDir* under the section *Directories*. This parameter should contain the directory where the images will be downloaded. Notice that such directory should exists on disk.

4.2 Querying a ROI shapefile on a local table

We clarify that the ROI shapefile and the table to be queried should already exists on the local database. For details about how to upload a ROI and how to create a local table on the local database please consult **Setup**.

Let us suppose we would like to query a ROI shapefile called ArcticBay with roiprojSRID=4326 and ST_Intersects as spatialrel. The local table that we want to query is tblmetadata. After updating the config file as explained in the initial section of this tutorial, the config file should look as follows:

[Database]

…

table = tblmetadata

…

[MISC]

…

roi = ArcticBay

roiprojSRID = 4326

…

spatialrel = ST_Intersects

After that, we run '"python query_imgs.py config.cfg"' or '"python SigLib.py config.cfg"', and we select from the menu option '1: tblmetadata: Query'.

We expect the following output, if there were no errors when executing the query:

```
ROI start date: 2010-05-01  end date: 2015-12-31

Query completed. Write query results to:

1: CSV file

2: Local table

3: Both

> Enter your option (1,2,3):
```

With this menu, the user can select either to output the results to a CSV file or to a local table.

Suppose we choose option *1: CSV file*, the program will output a standard filename. The user can accept that name or type one of its own. Notice that only the filename with extension should be provided (no path).:

```
Filename will be: ArcticBay_tblmetadata_24-08-2021_13-44.csv

> Press [Enter] to accept this name or Introduce a new filename (add .csv extension):
```

Suppose for example, we keep the provided name. Afterwards, the program will ask if the images should be downloaded:

```
Results saved to /home/outDir/ArcticBay_tblmetadata_24-08-2021_13-44.csv

> Download 2 images to output directory [Y/N]? N
```

Notice that if choosing *Y* all the images in the query result will be downloaded to the output directory (outDir).

4.3 Downloading images from a local table

The user can either download the images directly after performing a query to a local table, or alternatively, he can filter the results in the CSV file or in the results table to narrow the images to download. After doing such filtering, the images could be download with the option ''2: tblmetadata: Download".

Suppose for example, we filter the previous file '/home/outDir/ArcticBay_tblmetadata_24-08-2021_13-44.csv', and now we select option 2. After that, we introduce to the program the location of the csv file.:

```
> Enter CSV filename or tablename of images to download: /home/outDir/ArcticBay_
↪tblmetadata_24-08-2021_13-44.csv
```

After that, the script will output the list of download images to a txt file. The rest of the query execution could follow like this:

```
Filename will be: ArcticBay_download_metadata_PATHS_24-08-2021_14-45.txt
> Press [Enter] to accept this name or Introduce a new filename (add .txt extension)

List of downloaded images: /home/outDir/ArcticBay_download_metadata_PATHS_24-08-2021_14-
↪45.txt
```

In this example, ArcticBay_download_metadata_PATHS_24-08-2021_14-45.txt, will contain the list of the paths of the download images.

4.4 Querying EODMS

To query EODMS (as well as Sentinel), we must place all the ROI shapefiles (5 files) on the *vectDir* directory. The config file must be modified as was explained at the beginning of this Section.

To query the EODMS database (https://www.eodms-sgdot.nrcan-rncan.gc.ca/index-en.html), we select option 4 from the Query menu. After that, a series of questions would be asked by the program:

```
Enter collection to query or press enter for default "Radarsat1":

```

(continues on next page)

(continued from previous page)

```
ROI start date 2010-05-01 end date 2015-12-31

> Enter EODMS username: eodms_user

> Enter EODMS password: ******

Querying EODMS...

2021-08-24 15:42:20 | eodmsapi.main | WARNING | Number of search results (150) equals␣
↪query limit (150)

Fetching result metadata: 100%| 296/296 [01:03<00:00,  4.65item/s]

Query completed. Write query results to:

1: CSV file

2: Local table

3: Both

> Enter your option (1,2,3): 1

Filename will be: ArcticBay_EODMS_24-08-2021_15-51.csv

Press [Enter] to accept this name or Introduce a new filename (add .csv extension):

Results saved to /home/outDir/ArcticBay_EODMS_24-08-2021_15-44.csv

> Would you like to order the images? [Y/N] N

The user could order directly all the images returned by EODMS or alternatively he could␣
↪filter the results first and order them in a separate step.
```

4.5 Order Images to EODMS

Let us suppose we filtered the CSV file obtained in the previous example '/home/outDir/ArcticBay_EODMS_24-08-2021_15-44.csv'. To order the images from EODMS, we select the option '5: EODMS: Order'. The program will ask the location and name of the file or alternatively table with the images to order.:

```
> Enter CSV filename or tablename of images to order: /home/outDir/ArcticBay_EODMS_24-08-
↪2021_15-51.csv.

Ordering 5 images:

> Would you like to order 5 images? [Y/N] Y

> Enter collection to order or press enter for default "Radarsat1":

> Enter EODMS username: eodms_user
```

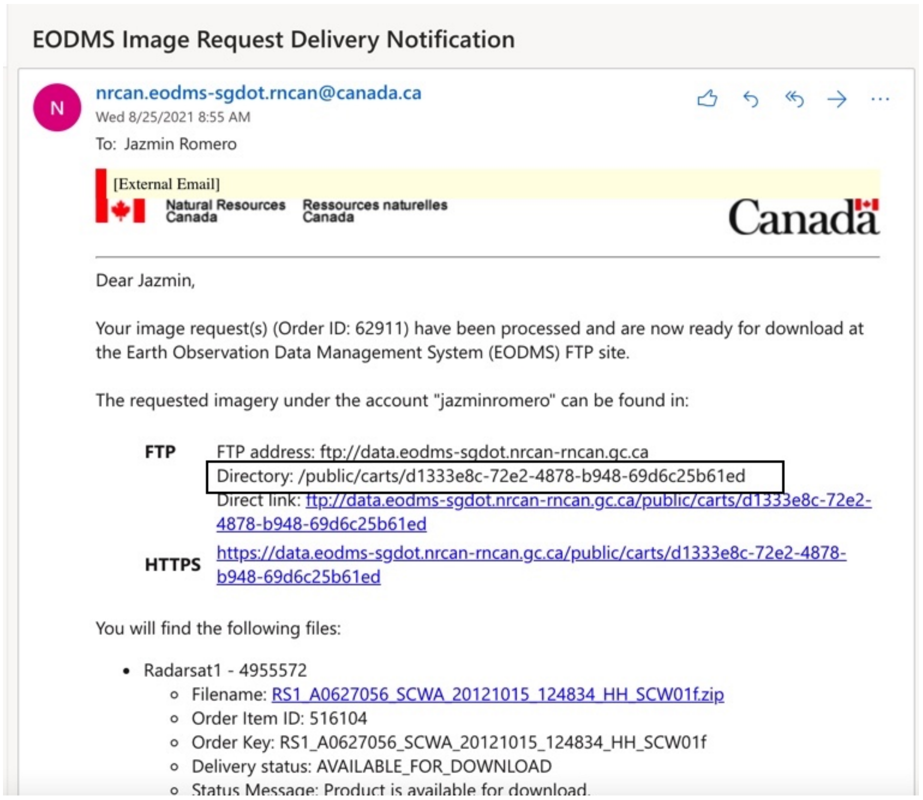
(continues on next page)

(continued from previous page)

```
> Enter EODMS password: ******

2021-08-25 08:53:20 | eodmsapi.main | INFO | Submitting order for 5 items

Order [516096, 516097, 516093, 516094, 516095] submitted. Wait for confirmation email.

Images ordered to EODMS. Wait for confirmation email.
```

4.6 Download EODMS

Two emails are received from EODMS, the first one is a confirmation that the order was received, and the second one has the ftp location of the images. It should look something like the email below:



EODMS Image Request Delivery Notification

nrcan.eodms-sgdot.rncan@canada.ca
Wed 8/25/2021 8:55 AM
To: Jazmin Romero

[External Email]
Natural Resources Canada Ressources naturelles Canada Canada

Dear Jazmin,

Your image request(s) (Order ID: 62911) have been processed and are now ready for download at the Earth Observation Data Management System (EODMS) FTP site.

The requested imagery under the account "jazminromero" can be found in:

FTP FTP address: ftp://data.eodms-sgdot.nrcan-rncan.gc.ca
Directory: /public/carts/d1333e8c-72e2-4878-b948-69d6c25b61ed
Direct link: ftp://data.eodms-sgdot.nrcan-rncan.gc.ca/public/carts/d1333e8c-72e2-4878-b948-69d6c25b61ed

HTTPS https://data.eodms-sgdot.nrcan-rncan.gc.ca/public/carts/d1333e8c-72e2-4878-b948-69d6c25b61ed

You will find the following files:

- Radarsat1 - 4955572
  - Filename: RS1_A0627056_SCWA_20121015_124834_HH_SCW01f.zip
  - Order Item ID: 516104
  - Order Key: RS1_A0627056_SCWA_20121015_124834_HH_SCW01f
  - Delivery status: AVAILABLE_FOR_DOWNLOAD
  - Status Message: Product is available for download.

To download all the ordered images, we select option '6: EODMS: Download'. This option prompt us to enter the cart directory. This directory corresponds to the "Directory" information under FTP section. If the cart is still available, all the files will be downloaded to the output directory.:

```
> Enter cart directory: /public/carts/d1333e8c-72e2-4878-b948-69d6c25b61ed

Downloading...RS1_A0627056_SCWA_20121015_124834_HH_SCW01f.zip
All files downloaded for 181s
Downloading...RS1_A0627101_SCWA_20121005_124016_HH_SCW01f.zip
All files downloaded for 155s
```

(continues on next page)

(continued from previous page)

```
Downloading...RS1_A0626939_SCWA_20121011_130504_HH_SCW01f.zip
All files downloaded for 169s
Downloading...RS1_A0627215_SCWA_20121009_122350_HH_SCW01f.zip
All files downloaded for 123s
Downloading...RS1_A0627116_SCWA_20121016_221354_HH_SCW01f.zip
All files downloaded for 39s

Filename will be: _eodms_PATHS_25-08-2021_09-27.txt
> Press [Enter] to accept this name or Introduce a new filename (add .txt extension):

List of downloaded images: /home/outDir/_eodms_PATHS_25-08-2021_09-27.txt
The file /_eodms_PATHS_25-08-2021_09-27.txt contains the list of the filenames␣
↪downloaded.
```

4.7 Query to Copernicus

Similar as with EODMS, we must place all the ROI shapefiles (5 files) on the *vectDir* location. The config file must be modified as was explained in Section 1.

After that we select ‘’7: Copernicus: Query”. Here is a sample of the execution:

```
> Enter satellite to query or press enter for default "Sentinel-1":

> Enter product type or press enter for "None":

> Enter sensor type ("SM", "IW", "EW", "WV") or press enter for None:

ROI start date 2010-05-01 end date 2015-12-31

> Enter your Copernicus username: copernicus_user

> Enter your Copernicus password: ******

Querying products: 100% 371/371 [00:06<00:00, 42.13 products/s]

Query completed. Write query results to:

1: CSV file

2: Local table

3: Both

> Enter your option (1,2,3): 1

Filename will be: ArcticBay_SENTINEL_25-08-2021_10-01.csv

> Press [Enter] to accept this name or Introduce a new filename (add .csv extension):

File saved to /home/outDir/ArcticBay_SENTINEL_25-08-2021_10-01.csv
```

(continues on next page)

(continued from previous page)

```
> Would you like to download 371 images? [Y/N] N
```

4.8 Download Copernicus

To download images from Copernicus, we select option ‘8: Copernicus: Download’ from the menu. Let us suppose we filtered the CSV file from the previous example. Below is the execution and interaction with the program for downloading images:

```
> Enter CSV filename or tablename of images to download: /home /outDir/ArcticBay_
↪SENTINEL_25-08-2021_10-01.csv

> Do you want to download 5 images [Y/N] Y

> Enter your Copernicus username: copernicus_user

> Enter your Copernicus password: ******

2A_MSIL2A_20201013T175311_N0214_R141_T16XEG_20201017T092049 is offline - Retry later.

Try next file.

S2A_MSIL2A_20201012T182321_N0214_R127_T16XEG_20201012T224404 is offline - Retry later.

Try next file.

S2B_MSIL2A_20201011T180259_N0214_R041_T16XEG_20201011T201358 is offline - Retry later.

Try next file.

S2A_MSIL2A_20190302T182251_N0211_R127_T16XEG_20190302T225351 is offline - Retry later.

Try next file.

S2B_MSIL2A_20190226T175309_N0211_R141_T16XEG_20190226T214409 is offline - Retry later.

Try next file.

There were 5 offline images. Retry to download them later.
```

For this example, all the images were offline and cannot be downloaded. However, since we attempted to download the images, some hours later they will be available to download.

Notice that even if some of the images were online, we don’t need to remove those records from the CSV file to attempt to download the offline images again. Images that are already downloaded will not be downloaded again (unless they are deleted from the directory *outDir*).

4.9 Execute SQL query

This option of the program simply executes blindly an SQL query. To execute an example select ‘’option 9: Execute Raw Sql Query” from the menu. Below is an example of the execution:

```
> Enter file with SQL query: /home/outDir/rawsql_.sql

Filename will be: _SQL_25-08-2021_13-20.csv

> Press [Enter] to accept this name or Introduce a new filename (add .csv extension):

Results saved to /home/outDir/_SQL_25-08-2021_13-20.csv
```

CHAPTER

FIVE

SIGLIB API

5.1 SigLib

SigLib.py

This script is thr margin that brings together all the SigLib modules with a config script to query, maniputlate and process remote sensing imagery

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 qualitative_mode instead of calling Metadata again @**author:** Cameron Fitzpatrick

Common Parameters of this Module:

zipfile : a valid zipfile name with full path and extension

zipname : zipfile name without path or extension

fname : image filename with extention but no path

imgname : image filename without extention

granule : unique name of an image in string format

5.2 Metadata

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)

Parameters

granule : unique name of an image in string format

imgname : image filename without extention

path : path to the image in string format *zipfile* : a valid zipfile name with full path and extension

sattype : type of data (String)

loghandler : A valid pre-set loghandler (Optional)

Returns

An instance of Metadata

`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 which can be written to file or sent to database

Returns

metadict : Dictionary containing all the metadata fields

`extractGCPs`(*interval*)

Extracts the lat/long and pixel col/row of ground control points in the image

Parameters

interval : line spacing between extractions

Returns

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

`getASFMetaCorners`(*ASFName*)

Use ASF Mapready to generate the metadata

Parameters

ASFName : Name with extention, of the ASF meta file

`getASFProductType`(*ASFName*)

Description needed!

Parameters

ASFName : Name with extention, of the ASF meta file

`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 the dimgname standard naming convention: yyyymmdd_HHmmss_sat_beam_pol…

See *Dimgname Convention* in SigLib Documentation for more information

Returns

dimgname : Name for image file conforming to standards above

`getRS2metadata()`
Open a Radarsat2 product.xml file and get all the required metadata

`getS1metadata()`
Open a S1 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 that it can read. Limited!

Returns

gdal_meta : Metadata found by gdal

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

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

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

Parameters

date : Date in python datetime convension

Returns

doy : Day of year

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

`Metadata.debyte(`*bb*`)`
Convert a byte into int, float, or string

`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 image resolution, 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, angle returned in radians

`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

Parameters

fp : Open file to read data block from

offset : How far down the file to begin the block

length : Length of the data block

`Metadata.`**`get_field_value`**`(`*data*, *field_type*, *length*, *offset*`)`

Take a line of data and convert it to the appropriate data type

Parameters

data : Line of data read from the file

field_type : Datatype line is (ASCII, Integer, Float, Binary)

length : Length of line

offset : How far down the block the line is

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. Sattype needed due to differing date convensions between RS2 and CDPF.

Parameters

date : Date in Rsat2 format

sattype : Satellite type (RS2, CDPF)

Returns

Date in python datetime convension

5.3 Image

imgProcess.py

Created on 14 Jul 9:22:16 2009 @**author:** Derek Mueller

This module creates an instance of class Image. It creates an image to contain Remote Sensing Data as an amplitude, sigma naught, noise or theta (incidence angle) image, etc. This image can be subsequently projected, cropped, masked, stretched, etc.

Modified on 3 Feb 1:52:10 2012 @**reason:** Repackaged for r2convert @**author:** Derek Mueller **Modified on** 23 May 14:43:40 2018 @**reason:** Added logging functionality @**author:** Cameron Fitzpatrick **Modified on** 9 Aug 11:53:42 2019 @**reason:** Added in/replaced functions in this module with snapPy equivilants @**author:** Cameron Fitzpatrick **Modified on** 9 May 13:50:00 2020 @**reason:** Added terrain correction for known elevation @**auther:** Allison Plourde

Common Parameters of this Module:

zipfile : a valid zipfile name with full path and extension

zipname : zipfile name without path or extension

fname : image filename with extention but no path

imgname : image filename without extention

granule : unique name of an image in string format

path : path to the image in string format

`class Image.Image`(*fname*, *path*, *meta*, *imgType*, *imgFormat*, *zipname*, *imgDir*, *tmpDir*, *projDir*, *loghandler=None*, *eCorr=None*, *initOnly=False*)

This is the Img class for each image. RSAT2, RSAT1, S1

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

Parameters

fname

path

meta : reference to the meta class

imgType : amp, sigma, beta or gamma

imgFormat : gdal format code gtiff, vrt

zipname

`applyStretch`(*stats*, *procedure='std'*, *sd=3*, *bitDepth=8*, *sep=False*, *inst=''*)

Given an array of stats per band, will stretch a multiband image to the dataType based on procedure (either std 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)

!sep = 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

Parameters

stats : Array of stats for a band, in arrary: band, range, dtype, nodata, min,max,mean,std

procedure : std or min-max

sd : # of standard deviations

bitDepth : # of bits per pixel

sep : False for same stretch to all bands, True for individual stretches

`cleanFiles`(*levels=['crop']*)

Removes intermediate files that have been written within the workflow.

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

Parameters

levels : a list of different types of files to delete

`compress`()

Use GDAL to LZW compress an image

`correct_known_elevation`()

Transforms image GCPs based on a known elevation (rather than the default average)

`cropBig`(*llur*, *subscene*)

If cropping cannot be done in a straight-forward way (cropSmall), gdalwarp is used instead

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 using cropSmall. If this cannot be accomplished then 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 because there is 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 pauli decomposition.

Differs from imgWrite because it ingests all bands at once…

`fnameGenerate`(*names=False*)
Generate a specific filename for this product.

Returns

bands : Integer

dataType : Gdal data type

outname : New filename

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

Parameters

datachunk : Chunk of data being processed

Returns

outdata : chunk data in amplitude format

`getBandData`(*band*, *inname=None*)
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

Parameters

Band : Array of data for a specific band

Returns

imgData : data from the given band

xSpacing : size of pixel in x direction (decimal degrees)

ySpacing : size of pixel in y direction (decimal degrees)

`getImgStats`(*save_stats=False*)
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)

Parameters

datachunk : chunk of data being processed

n_lines : size of the chunk

Returns

caldata : calibrated chunk

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

`imgWrite`(*format='imgFormat'*, *stretchVals=None*)
Takes an input dataset 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`()
Make image pyramids for fast viewing at different scales (used in GIS)

`maskImg`(*mask*, *vectdir*, *side*, *inname=None*)
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

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

Parameters

fname : filename

`projectImg`(*proj*, *projSRID*, *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 IS NOT prescribed (it will be the smallest possible)

`reduceImg`(*xfactor*, *yfactor*)
Uses gdal to reduce the image by a given factor in x and y(i.e, factor 2 is 50% smaller or half the # of pixels). It overwrites the original.

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*)
Used to convert a vrt to a tiff (or another image format)

5.4 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:

table_to_query - 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

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

class Database.**Database**(*table_to_query*, *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 read or write (as required) access on the specified databse (eg. postgres user has read/write access to all databses)

password : password to go with user

port : server port (i.e. 5432)

host : hostname of postgres server

`createTblMetadata()`

Creates a metadata table of name specified in the cfg file under Table. It overwrites if that table name already exist, be careful!

`create_query_table(`*table_name*, *records*`)`

Creates a table to store results from a query. The structure of the table follows the information from records.

Parameters

table_name : The name of the table to be created. *records*: A list of records from where the table structure (name and data type of columns) will be obtained.

Returns *success* : True if the records were inserted into the table

`create_table_from_dict(`*table_name*, *list*`)`

Creates a table to store results from a query. The structure of the table follows the information from list.

Parameters

table_name : The name of the table to be created. *list*: A list of records from where the table structure (name and data type of columns) will be obtained.

Returns *success* : True if the records were inserted into the table

`execute_raw_sql_query(`*sql_query*`)`

Executes a sql query passed as a parameter.

Parameters

sql_query : The sql_query to be executed

Returns *result* : The result of the sql query or empty string if failed.

`exportDict_to_CSV(`*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

`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

`findInstances(`*roi*`)`

Scans metadata table to determine if any images intersect the given roi

Parameters

roi : roi table name

`imgData2db`(*imgData*, *bandName*, *inst*, *dimgname*, *granule*, *table='tblbanddata'*)

Upload image data as an array to a new database table

What is needed by function: imgData, the imgType, the bandName, the inst, dimgname and granule What is computed to add to upload: count, mean, std, min, max for non-zero elements

Parameters

imgData : Pixel values

bandName : Name of the band being uploaded

inst : instance id

dimgname : Image name

granule : granule name

`insert_query_table`(*table_name*, *records*)

Inserts records in a table_name.

Parameters

table_name : The name of the table in which the records will be inserted.

Returns *success* : True if the records were inserted into the table

`insert_table_from_dict`(*table_name*, *list*)

Inserts records into a table to store results from a query. The structure of the table follows the information from records.

Parameters

table_name : The name of the table to be created. *list*: A list of records from where the table structure (name and data type of columns) will be obtained.

Returns *success* : True if the records were inserted into the table

`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

`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*, *inst*, *metaTable*, *srid=4326*)
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)

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

metaTable : metadata table containing data of images being worked on

srid : srid of desired projection (default WGS84, as this comes out of TC from snap, crop can be done before projection)

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

Parameters

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*)
Writes a query to fetch the instance names that are associated spatially in the relational table.

Parameters

granule : granule name

roi : region of interest file

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*, *srid*, *inst*, *metaTable*)

Writes a query to fetch the area the inst polygon in the roi and the image in question intersect. This polygon will be used to mask the image.

Parameters

granule : granule name

roi : region of interest file

srid : srid of desired projection

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

metaTable : metadata table containing data of images being worked on

idField : name of field containing instance id's

Returns

polytext : gml text

`qrySelectFromAvailable`(*roi*, *selectFrom*, *spatialrel*, *srid*)

Given a roi 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

Returns 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?

srid : srid of desired projecton

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

`relations2db`(*roi*, *spatialrel*, *instimg*, *fname=None*, *mode='refresh'*, *export=False*)

A function to add or update relational tables that contain the name of an image and the features that they are spatially related 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 images that are spatially related to each inst *fname* : csv filename if exporting

mode : create or refresh mode

export : If relations should be exported to a csv or not

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

Coming 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*, *srid*, *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 the ROI table name too)

srid : srid of desired projection

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

Required in ROI (In shp attribute table)

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 (in shp attribute table)

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…

5.5 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.`**`cartesian_to_geographic`**(*x*, *y*, *z*, *a*, *b*)

transforms cartesian space to geographic latitude and longitude

Parameters

x : x values to be transformed (can be single value, an array, or M x N matrix)

y : y values to be transformed (can be single value, an array, or M x N matrix)

z : z values to be transformed (can be single value, an array, or M x N matrix)

a : semi major axis of reference ellipse (float)

b : semi minor axis of reference ellipse (float)

`Util.`**`geographic_to_cartesian`**(*lat*, *lng*, *a*, *b*)

transforms geographic latitude and longitude to cartesian space.

Parameters

lat : latitude values to be transformed (can be single value, an array, or M x N matrix)

lng : longitude values to be transformed (can be single value, an array, or M x N matrix)

a : semi major axis of reference ellipse (float)

b : semi minor axis of reference ellipse (float)

`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.`**`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*, *projFile=False*`)`

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

5.6 Query

class `Query.Query`(*db*, *roi*, *srid*, *roidir*, *scandir*, *localtable*, *spatialrel*, *outputDir*, *method*, *loghandler=None*)
Query class used to search for imagery from a variety of sources

Parameters

roi : name of roi shapefile

srid : srid of roi shapefile

dir : directory that contains roi shapefile

method : user selected query method, ie metadata, cis, EODMS

`buildQuery`(*records*)
builds query to order records

Parameters

records : (list) record and collection ids from EODMS query result

`download_eodms_cart`(*output_directory*, *list_directory*)
Downloads all images from an eodms cart. It also outputs the list of downloaded images into a txt file. One image per line.

Parameters

ouput_directory : The directory where the images will be downloaded..

`download_from_csv_tblmetadata`(*output_dir*, *list_dir*, *csv_filename*, *roi*, *method*)
Download images from a local table The id's of the images are contained in a csv file. It also outputs the list of downloaded images into a txt file. One image per line.

Parameters *db* : An instance of the database class. *output_dir* : The directory where the images will be stored. *csv_filename* : Name of the csv file that contains the images ids. *roi* : The roi shapefile to be queried *method* : query method

`download_from_table_tblmetadata`(*db*, *output_dir*, *list_dir*, *tablename*, *roi*, *method*)
Download images from a local table. The id's of the images are contained in a table. It also outputs the list of downloaded images into a txt file. One image per line.

Parameters *db* : An instance of the database class. *output_dir* : The directory where the images will be stored. *tablename* : Name of the table that contains the images ids. *roi* : The roi shapefile to be queried *method* : query method

`download_images_from_eodms`(*output_dir*)
Downloads images from eodms. It also outputs the list of downloaded images into a txt file. One image per line.

Parameters

output_dir : Directory where the images will be saved

`download_images_from_sentinel`(*db*, *outputDir*, *listDir*)
Call the procedure to download images from Copernicus. The records of images can come either from a CSV or from a table into the database.

Parameters

db : An instance of the Database class. *outputDir* : Directory where the csv results will be saved.

`download_local_table`(*db*, *roi*, *method*)
Download images from a local table. It calls the appropriate function if the ids of the images are in a CSV or a table.

Parameters *db* : An instance of the database class. *output_dir* : The directory where the images will be stored. *csv_filename* : Name of the csv file that contains the images ids. *roi* : The roi shapefile to be queried *method* : query method

`execute_raw_query`(*db*, *outputDir*)
Call database class to execute a raw sql query passed as a parameter. Saves the results of the query into a csv file.

Parameters

db : An instance of the Database class. *outputDir* : Directory where the csv results will be saved.

`getEODMSRecords`(*session*, *start_date*, *end_date*, *coords*)
query EODMS database for records within region and date range

Parameters

session : requests session for HTTP requests

start_date : (datetime) find images after this date

end_date : (datetime) find images before this date

coords : region on interest

`get_EODMS_ids_from_csv`(*filename*)
Obtain the images ids from a CSV file which contains the results of a query to EODMS.

Parameters

filename : The CSV file that contains the images ids.

Returns *record_id* : A list of images id from EODMS

`get_EODMS_ids_from_table`(*db*, *tablename*)
Obtain the images ids from a local table which contains the results of a query to EODMS.

Parameters *db* : An instance of the database class *tablename* : The table that contains the images ids.

Returns *record_id* : A list of images id from EODMS

`get_Sentinel_ids_from_csv`(*filename*)
Obtain the images uuids from a CSV file which contains the results of a query to Sentinel.

Parameters

filename : The CSV file that contains the images ids.

Returns *records* : A list of uuids per image

`get_Sentinel_ids_from_table`(*db*, *tablename*)
Obtain the images uuids from a local table which contains the results of a query to Sentinel.

Parameters *db* : An instance of the database class. *tablename* : The tablename that contains the images ids.

Returns *records* : A list of uuids per image `order_eodms`(*db*)

Order images to eodms

Parameters *db* : An instance of the database class.

`queryEODMS`(*queryParams*)
Query the EODMS database for records matching a given spatial and temperoral region

Parameters

queryParams : (dict) geometrical and temporal parameters to query on

`queryEODMS_MB`(*roiDir*, *roi*, *collection*)
Calls EodmsAPI to download images from copernicus.

Parameters

roiDir : Directory where the shapefile is located *roi*: Name of the shapefile to be queried. *collection* : Collection parameter for EodmsAPI

Returns *records* : Records returned by EodmsAPI

`query_eodms`(*db*, *roi*, *roiDir*, *method*, *outputDir*)
Calls the function _query_eodms which in turns call queryEODMSAPI to download images from EODMS. Saves the results of the query into a CSV file, a local table or both. It also downloads the images from the query, if the user selects that option.

Parameters *db* : An instance of the database class. *roiDir* : Directory where the shapefile is located *roi*: Name of the shapefile to be queried. *method*: Name of the query method (EODMS) *outputDir*: where the CSV will be saved.

`query_local_table`(*db*, *roi*, *tablename*, *spatialrel*, *roiSRID*, *method*)
Query a roi shapefile in a local table.

Parameters *db* : An instance of the database class. *roi* : The roi shape file. *tablename* : The name of the table to query. *spatialrel* : Spatial relation to be used in the query (ST_INTERSECTS OR ST_OVERLAPS) *roiSRID* : Spatial projection of the roi shapefile. *method* : query method

`query_sentinel`(*db*, *roi*, *roiDir*, *method*, *outputDir*)
Calls the function _query_sentinel which in turns call SentinelAPI to download images from copernicus. Saves the results of the query into a CSV file, a local table or both. It also downloads the images from the query, if the user selects that option.

Parameters *db* : An instance of the database class. *roiDir* : Directory where the shapefile is located *roi*: Name of the shapefile to be queried. *method*: Name of the query method (SENTINEL) *outputDir*: where the CSV will be saved.

`readShpFile`(*filename*)
extracts coordinate geometry and fromdate/todate from an ESRI shapefile

Parameters

filename : (string) path+name of roi shpfile

`submit_post`(*query*)
submits order to EODMS

Parameters

query : (dict) query with record and collection ids

PYTHON MODULE INDEX

INDEX