

---

WATER & ICE RESEARCH LAB



# **SigLib Documentation**

***Release 2.8***

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

August 08, 2019

<b>1</b>	<b>Overview of the Project</b>	<b>1</b>
1.1	Introduction . . . . .	1
1.2	Acknowledgements . . . . .	2
<b>2</b>	<b>Overview of SigLib.py and its Dependencies</b>	<b>3</b>
2.1	Dependencies . . . . .	3
2.2	Modules . . . . .	3
2.3	Config File . . . . .	3
2.4	Using a Config in an IDE . . . . .	5
2.5	Dimgname Convention . . . . .	5
2.6	ROI.shp format . . . . .	5
2.7	A Note on Projections: . . . . .	6
2.8	Example workflow: . . . . .	6
<b>3</b>	<b>TODO</b>	<b>8</b>
3.1	SigLib.py . . . . .	8
3.2	Metadata.py . . . . .	8
3.3	Image.py . . . . .	8
3.4	Util.py . . . . .	8
<b>4</b>	<b>Using SigLib</b>	<b>9</b>
4.1	Basic SigLib Setup . . . . .	9
4.2	Example #1: Basic Radarsat2 Image Calibration using SigLib . . . . .	9
4.3	Example #2: Discover Radarsat metadata and upload to a geodatabase . . . . .	10
4.4	Example #3: Scientific Mode! . . . . .	11
4.5	Example #4: Polarimetric Mode! . . . . .	12
4.6	Conclusion . . . . .	13
<b>5</b>	<b>SigLib API</b>	<b>14</b>
5.1	SigLib . . . . .	14
5.2	Metadata . . . . .	16
5.3	Image . . . . .	20
5.4	Database . . . . .	25
5.5	Utilities . . . . .	31
	<b>Python Module Index</b>	<b>35</b>
	<b>Index</b>	<b>36</b>

---

## OVERVIEW OF THE PROJECT

### 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 geodatabase. 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 or **ROI shapefile** with a specific format as input to delineate 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.
3. A **Scientific Mode** where remote sensing scenes can be converted to sigma, beta, or gamma nought. Image data (from each band) is then subsampled by way of an **ROIshapefile** that references every image and specific polygon to be analyzed. These polygons represent sampling regions that are known (a priori) or are hand digitized from Exploratory mode images. Data can be stored in a table in a geodatabase for further processing.
4. A **Polarimetric Mode** where quad-pol scenes are converted to sigma0, cropped to tracking beacon instances, have polarimetric matrices generated, polarimetric filtering, and polarimetric decompositions generated.

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). 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 (workstudy student & computer programmer) 2012, Sougal Bouh-Ali (workstudy student & computer programmer) 2013-2016, and Cameron Fitzpatrick (computer programmer) 2018-Present. Ron Saper, Anna Crawford and Greg Lewis-Paley helped out as well (indirectly).

## 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, psycpg2, matplotlib, datetime... It's recommended you install the *Anaconda Python Distribution* as it contains pretty well everything required. [www.anaconda.com](http://www.anaconda.com)
- gdal/ogr libraries [www.osgeo.org](http://www.osgeo.org)
- snap (sentinel-1 toolbox): <https://step.esa.int/main/download/snap-download/> Make sure when downloading to configure python api for snapPy!
- PostgreSQL/PostGIS (could be on another computer) [www.postgresql.org](http://www.postgresql.org) / [postgis.net](http://postgis.net)
- It is highly recommended that you have access to QGIS or ArcGIS Pro to manipulate shapefiles

### 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. **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:

*Directories*

- scanDir = path to where you want SigLib to look for SAR image zip files to work with
- tmpDir = a working directory to extract zip files to (A folder for temporary files that will only be used during the running of the code, then deleted)
- projDir = where projection definition files are found in well-known text format (wkt)
- 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
- logDir = where logs are created
- archDir = where CIS archive data are found
- 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
- host = database hostname
- create\_tblmetadata = 0 for append, = 1 for overwrite/create a metadata table.
- uploadROI = if the ROI file needs to be uploaded to the database. 0 if no, 1 if yes
- table = name of table in database that Database.py will query against to find imagery

#### *Input*

**Note that path, query, and file are mutually exclusive options - sum of 'Input' options must = 1**

- path = 1 to scan a certain path and operate on all files within; 0 otherwise
- query = 1 to scan over the results of a query and operate on all files returned; 0 otherwise
- file = 1 to run process on a certain file, which is passed as a command line argument (*note* this must be enabled to parallelize SigLib); 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)
- scientific = 1 when image manipulation requires Database.py
- polarimetric = 1 when you want to do SAR polarimetry

#### *MISC*

- proj = basename of wkt projection file (eg. lcc) in the projDir
- imgtypes = types of images to process (Amplitude Image - amp, Sigma Nought Image - sigma, Gamma Nought Image - gamma, Beta Nought Image - beta)
- imgformat = file format for output imagery (gdal convention)
- roi = ROI Shapefile for Discovery or Scientific modes, stored in vectDir
- roiproj = projection of the roi
- crop = leave blank 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 (with only one feature) to mask image data with
- 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
```

## Dimname 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 Derek made this problem worse with his own ‘standard image naming convention’ called **dimname**. 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: **dimname 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	nil, utm, lcc, aea	3
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']

## A Note on Projections:

SigLib uses projections in two ways; either as .wkt files during image processing outside the database, or SRID values when using PostgreSQL/PostGIS. For when Database.py is not being used, projections should be downloaded as .wkt files from [spatialreference.org](http://spatialreference.org) and placed into a projection directory. If using Database.py functionality, make sure the spatial\_ref\_sys table is defined in your database. This table has a core of over 3000 spatial reference systems ready to use, but custom projections can be added very easily!

To add a custom spatial reference, download the desired projection in "PostGIS spatial\_ref\_sys INSERT statement" format from [spatialreference.org](http://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.

## 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 dimname 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
IN-STID	String	An iterator for each new row of the same OBJ	0,1,2,3,4	Both
FROM-DATE	String	ISO Date-time denoting the start of the time period of interest	2002-04-15 00:00:00	Discovery
TO-DATE	String	ISO Date-time denoting the end of the time period of interest	2002-09-15 23:59:59	Discovery
IM-GREF	String	dimgname of a specific image known to contain the OBJ polygon (Spaces are underscores)	20020715 135903 r1scwa 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

## TODO

\*# 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)

- 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

- Does scientific work?

## Metadata.py

- get look direction for RSAT2, test against RSAT1

## Image.py

- test image crop and mask - in both modes
- Sigma0 testing (image vs snap vs ZAPro)
- compare snap equivlancy to gdal, if so remove unnessesary gdal code (clean!)

## Util.py

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

## USING SIGLIB

Welcome to the tutorial sections of the SigLib documentation! This section gives a brief overview of how to use the Metadata, Util, Database, and Image functions via SigLib and its config file, or in a custom way via qryDatabase.

### 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 are five Python files (*.py*), a *config file* (*.cfg*), and an extras folder containing some odds and ends (including this very document you are reading!).

A number of folders must be created and referred to 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 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 are given for the Spyder Python IDE; the setup for other IDEs 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 Calibration using SigLib

In this example we will be using SigLib to produce Tiff images from Amplitude Radarsat2 image files. Before any work begins in Python, the config file must be configured for this type of job, see the figure below for the required settings. Place a few Radarsat2 zip files in your scanDir, 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. Image.py will create an initial Tiff image via GDAL or SNAPPY, and saved to the output directory. The image will then be reprojected and stretched into a byte-scaled Tiff file. All intermediate files will then be cleaned and Siglib will move onto the next zipfile, until all the files in the scanDir are converted.

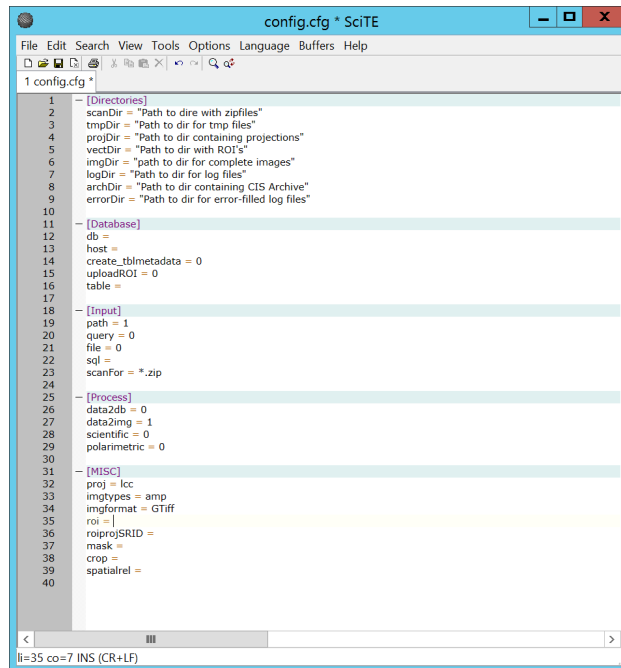


Fig. 4.1: A basic config file for this task

## 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. Also, since we are running this example in parallel, the input must be **File** not **Path**.

In this case, we need the metadata table in our geodatabase to already exist. If this table has not been created yet, run SigLib with “create\_tblmetadata” equal to 1, with all modes under **Process** equal to 0 before continuing with the rest of this example.

A review of the settings needed for this particular example can be seen in the figure 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 /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 (I.E. A lower -nice level gives this job a higher priority over other processes). The first directory is the location of your version 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.

Once started, Parallel will begin to step through your selected directory looking for .zip files. Once one is found, it will

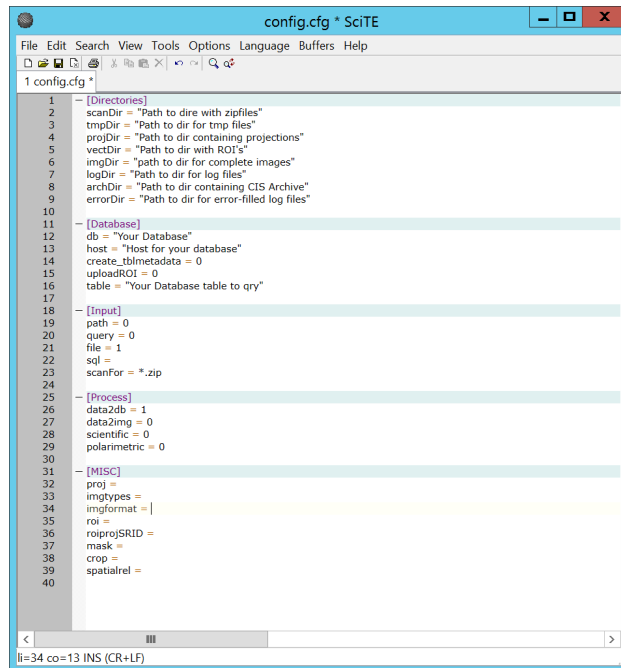


Fig. 4.2: Config file settings for discovering and uploading metadata.

pass it to one of the 16 available (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 upload 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.

Most 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 scanDir, and using the same settings as in the above figure, except in the *Input* section, **File** must be set to 0, and **Path** must be set to 1.

## Example #3: Scientific Mode!

In this 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, beta0, and gamma0) followed by cropping and masking them into small pieces via a scientific ROI. The ROI should contain a series of polygons representing regions of interest for different scenes. For example, the polygons could be individual farmers fields, or individual icebergs. The ROI created must be uploaded to the geodatabase for querying by SigLib. To upload the ROI specified in the config, set 'uploadROI' equal to 1, as seen below in the example config. **NOTE:** This config setting **MUST** be 0 if running in parallel, or else the ROI will constantly be overwritten. This case also requires a database with SAR image footprints, like the one made in the previous example!

Once begun, this mode takes a SAR image in the scanDir, and calibrates it to the selected image type. Once completed, database.py is used to query the ROI against the image footprint to find which polygons in the ROI are within the scene being processed. Each of these hits is then processed one at a time, beginning with a bounding-box crop around the instance, followed by a mask using the ROI polygon (both queried via Database.py). At this point, each instance is projected and turned into its own TIFF file for delivery, or the image data for the instances is uploaded to a database

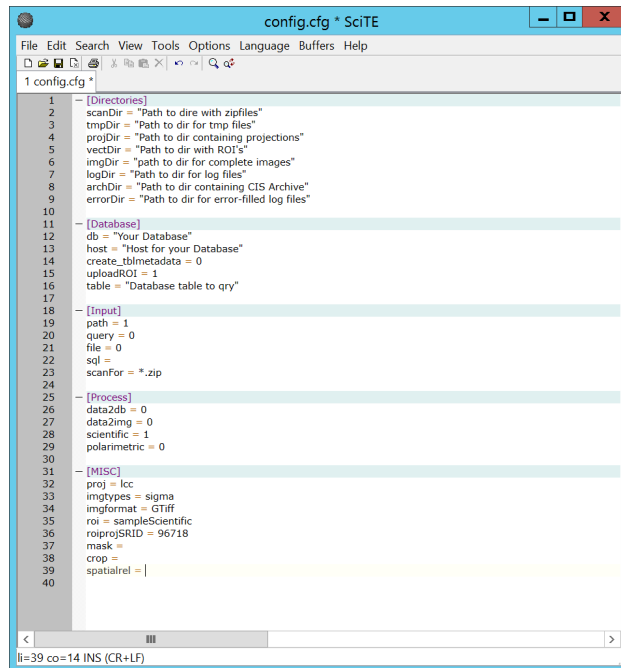


Fig. 4.3: Config file settings for scientific mode. Note that we are uploading an ROI in this example. The first time scientific is run with a new ROI, this setting will be necessary, otherwise it can be set equal to 0

table made to store data from this run.

## Example #4: Polarimetric Mode!

In this final example, we will look at using SigLib's Polarimetric mode. Polarimetric mode uses the SNAP python library SnapPy to perform SAR polarimetry for quad-pol scenes containing tracking beacon instances. Three different polarimetric operations are conducted in this mode: Matrix Generation, Polarimetric Speckle Filtering (Using Refined Lee Filter), and Polarimetric Decomposition Generation. All the matrices and decompositions available in SnapPy that are designed for quad-pol imagery are stored in lists in *polarimetric* in **SigLib.py**, these can be edited to contain only desired options. No terrain-correction/reprojections are done in this mode.

This example requires a database table containing tracking beacon instances and one containing quad-pol SAR image footprints. The necessary columns for the tracking beacons are geom, latitude, longitude, beaconid, and time (in UTC). The geom and time columns will be compared to similar columns in the SAR footprints table. The sql statement to compare these two tables will need to be edited to match your column names, see *beaconIntersections* in **Image.py** to make appropriate edits.

Once started, this mode goes through the images in the scan directory, processing one image at a time. The program checks to make sure the images are quad-pol, and then queries the geodatabase to see if any beacon pings are contained in the image within a 91 minute buffered timeframe of the SAR data being collected. If there are beacons in the image, the ID's, latitudes, and longitudes of the beacons are collected. The scene is then calibrated to sigma0. At this point, image processing begins per beacon instance in this scene. First, both polarimetric matrices, C3 and T3, are applied (two separate files are created, one with each). Each of these files is then speckle-filtered with a Refined-Lee Filter. After, each decomposition type specified is generated upon each matrix type (again, separate file for each). For the Touzi and H-A-Alpha Decompositions, they are generated four times per matrix, each with a different set of bands. Max conditions will generate 28 GeoTiff files per beacon instance, if both matrices and all snap decompositions are utilized.

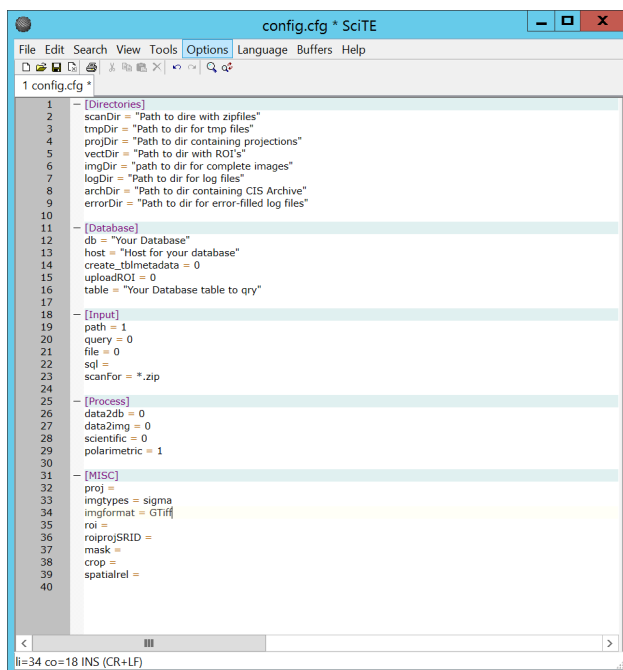


Fig. 4.4: Config file settings for polarimetry mode

## 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 an overview, the parameters, and the outputs, of each function in the main five modules.

## SIGLIB API

## SigLib

**SigLib.py**

This script is the margin that brings together all the SigLib modules with a config script to query, manipulate 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 data2img 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 extension but no path

*imgname* : image filename without extension

*granule* : unique name of an image in string format

**class** SigLib.**SigLib**

**createLog** (*zipfile=None*)

Creates log file that will be used to report progress and errors **Parameters**

*zipfile*

**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 careful)

**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*

*imgname*

*zipname*



*sattype* : satellite platform

*granule*

*zipfile*

*sar\_meta* : instance of the Metadata class

*unzipdir* : directory zipfiles were unzipped into

**handler** (*signum, frame*)

Handles exceptions - most notably time out errors

**polarimetric** (*db, fname, imgname, zipname, sattype, granule, zipfile, sar\_meta, unzipdir*)

This function will take full FQ images and perform desired polarimetric matrix generation(s), speckle filtering, and decomposition(s). Combinations can be set in the config file. BE CAREFULL, if no combination is specified, then all possible combinations will be performed, so be prepared for a long run-time and large amounts of data!

#### Parameters

*db* : an instance of the Database class

*fname*

*imgname*

*zipname*

*sattype* : Satellite platform

*granule*

*zipfile*

*sar\_meta* : an instance of the Metadata class

*unzipdir* : location zipfiles were unzipped into

**proc\_Dir** (*path, pattern*)

Locates satellite 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 be parallelizable

#### Parameters

*zipfile*

**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*

**scientific** (*db, fname, imgname, zipname, sattype, granule, zipfile, sar\_meta, unzipdir*)

**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 OR uploading img data to database

#### Parameters

*db* : instance of the Database class  
*fname*  
*zipname*  
*sattype* : satellite platform  
*granule*  
*zipfile*  
*sar\_meta* : instance of the Metadata class  
*unzipdir* : directory zipfile was unzipped into

## Metadata

**class** SigLib.**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 extension  
*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

#### 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 extension  
*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 dictionary 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 extension, of the ASF meta file

**getASFProductType** (*ASFName*)

Description needed!

**Parameters**

*ASFName* : Name with extension, 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)

**getDimname ()**

**Create a filename that conforms to the dimname standard naming convention:**

yyyymmdd\_HHmss\_sat\_beam\_pol...

See *Dimname Convention* in SigLib Documentation for more information

**Returns**

*dimname* : 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 guarantee this is to set n\_gcps = 6

**Parameters**

*n\_gcps* : # of GCP's to return

**Returns**

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

**getRS2metadata ()**

Open a Radarsat2 product.xml 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 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 conversion

**Returns**

*doy* : Day of year

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 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 formatted date 2009-05-31T14:43:17.184550Z and converts it to python datetime. Satttype needed due to differing date conversions between RS2 and CDPF.

**Parameters**

*date* : Date in Rsat2 format

*sattype* : Satellite type (RS2, CDPF)

**Returns**

Date in python datetime conversion

## 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 equivalents **@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 extension but no path

*imgname* : image filename without extension

*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, loghandler=None, pol=False*)

This is the Img class for each image. RSAT2, RSAT1 (CDPF)

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*)

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](http://en.wikipedia.org/wiki/Histogram_equalization)

#### Parameters

*stats* : Array of stats for a band, in array: 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

#### **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...

**decomposition\_generation** (*decomposition*, *matrix=None*, *input=None*, *dir=None*, *beaconid=None*, *outputType=0*, *amp=False*)

Generate chosen decomposition on a fully polarimetric product, if looking to create an amplitude image with quad-pol data, set amp to True and send either a Pauli Decomposition or Sinclair Decomposition

#### Parameters

*decomposition* : decomposition to be generated. options include: Sinclair Decomposition, Pauli Decomposition, Freeman-Durden Decomposition, Yamagushi Decomposition, van Zyl Decomposition, H-A-Alpha Quad Pol Decomposition, Cloude Decomposition, or Touzi Decomposition

*input* : filename with path of snap raster (.dim) to have decomposition generated upon (must be calibrated and have a matrix generated)

*matrix* : The matrix (string) that was generated on this product

*dir* : directory that finished tifs will be stored in

*outputType* : option to select which set of output parameters that will be used for the Touzi or HAAAlpha (1-8, leave 0 for none)

*amp* : True if looking to generate Pauli or Sinclair decomposition to create quad-pol amp image

#### Returns

*output* : filename of new product

**fnameGenerate** (*projout=None*, *subset=None*, *band=None*, *cal=False*, *TC=False*, *DC=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*)

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 ()**

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

**makeAmp ()**

Use snap bandMaths to create amplitude band for SLC products

**makePyramids ()**

Make image pyramids for fast viewing at different scales (used in GIS)

**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

**matrix\_generation** (*matrix, input*)

Generate chosen matrix for calibrated quad-pol product

**Parameters**

*matrix* : matrix to be generated. options include: C3, C4, T3, or T4

*input* : filename with path of snap raster (.dim) to have matrix generated upon (must be calibrated)

**Returns**

*output* : filename of new product

**openDataset** (*fname, path=''*)

Opens a dataset with gdal

**Parameters**

*fname* : filename

**polarFilter** (*input*)

Apply a speckle filter on a fully polarimetric product

**Parameters**

*input* : filename with path of snap raster (.dim) to filter

**Returns**

*output* : filename of new product

**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 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

**snapCalibration** (*outDataType='sigma', saveInComplex=False*)

This fuction calibrates radarsat images into sigma, beta, or gamma

**Parameters**

*outDataType* : Type of calibration to perform (sigma, beta, or gamma), default is sigma

*saveInComplex* : Output complex sigma data (for polarimetric mode)

**snapDataTypeConv** (*outFormat='GeoTiff-BigTiff'*)

Convert image data to byte format using gdal

#### Parameters

*outFormat* : Format of product this function returns, default is GeoTiff-BigTiff

**snapSubset** (*idNum, lat, longt, ullr=None*)

Using lat long provided, create bounding box 200x200 pixels around it, and subset this (This requires id, lat, and longt) OR Using ul and lr coordinates of bounding box, subset (This requires id and ullr)

#### parameters

*idNum* : id # of subset region (for filenames, each subset region should have unique id)

*lat* : latitude of beacon at centre of subset (Optional)

*longt* : longitude of beacon at centre of subset (Optional)

*ullr* : Coordinates of ul and lr corners of bounding box to subset to (Optional)

**snapTC** (*proj, projDir, existingInput=True, smooth=True, outFormat='BEAM-DIMAP'*)

Perform an Ellipsoid-Correction using snap. This function also projects the product

#### Parameters

*proj* : name of wkt projection file (minus file extension)

*projDir* : directory containing projection file

*existingInput* : If a snap-product has already been created before this function (default is True)

*smooth* : If quantitative data, do not smooth (Smooth using Bilinear resampling for quality)

*outFormat* : Format of product this function returns, default is BEAM-DIMAP

**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)

## 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)

**trcl\_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

**beaconIntersections** (*beacontable, granule*)

Get id, lat and long of beacons that intersect the image within a half hour of the image data being collected

#### Parameters

*beacontable* : database table containing beacon tracks

*granule* : unique identifier of image being analysed

#### Returns

*rows* : all beacon pings that meet requirements. Each row has three columns: beacon id, lat, and long

**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* : Directory containing the beacon shapefiles

**convertGPSTime** (*shpTable*)

Takes a shape file and converts the gps\_time from character type to timestamp time.

#### Parameters

*shpTable* : Shapefile table in the database

**createTblMetadata ( )**

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

**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 tuple - numpy data and list of column names

*outputName* : the file name

**imgData2db (imgData, xSpacing, ySpacing, bandName, inst, dimgname, granule)**

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

*xSpacing* :

*ySpacing* :

*bandName* : Name of the band being uploaded

*inst* : instance id

*dimgname* : Image name

*granule* : granule name

**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* : dictionary containing the metadata

**nameRelationTable (roi, spatialrel)**

Automatically gives a name to a relational table in the format: "trel" + roi + "img" + \_int or \_con (in reference to spatialrel)

**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*, *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 tuple 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 tuple pair in the projection given

**qryFromFile** (*fname*, *path*, *output=False*)

Runs a query in the current database 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 interferes with the pyformat protocol used by pycopg2

### 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 database by sending an sql string

**Note:** do not use '%' in the query b/c it interferes with the pyformat protocol used by pycopg2. 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 tuple containing a numpy array and the column names as a list (if requested and available)

**qryGetInstances** (*granule*, *roi*, *metaTable*)

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**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 projection*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...

## 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 getFilename, 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](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*)

Delete all files in a certain path

#### Parameters

*dirname* :

`Util.copyfiles` (*self*, *copylist*, *wkdir*, *fname=None*, *export=False*, *loghandler=None*)

Copies files from a local archive. If file could not be found, check that the drive mapping is correct (above).

#### Parameters

*copylist* : a list of images + inst

*wkdir* : working directory

*fname* : Filename if exporting copylist as txt

*export* : True if want to export copylist as 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 tuples [(x,y),(x,y)]

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

#### Parameters

*llur* : a list of tuples [(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 tuples [(x,y),(x,y)]

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

#### Parameters

*ullr* : a list of tuples [(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

**d**

Database, [25](#)

**i**

Image, [20](#)

**m**

Metadata, [16](#)

**s**

SigLib, [14](#)

**u**

Util, [31](#)

## A

applyStretch() (Image.Image method), 20  
az() (in module Util), 31

## B

beaconIntersections() (Database.Database method), 26  
beaconShapefilesToTables() (Database.Database method), 26  
byte2int() (in module Metadata), 18

## C

clean\_metaASF() (Metadata.Metadata method), 17  
clean\_metaCDPF() (Metadata.Metadata method), 17  
cleanFiles() (Image.Image method), 21  
cleartree() (in module Util), 31  
compress() (Image.Image method), 31  
convertGPSTime() (Database.Database method), 26  
copyfiles() (in module Util), 31  
createLog() (SigLib.SigLib method), 14  
createMetaDict() (Metadata.Metadata method), 17  
createTblMetadata() (Database.Database method), 26  
cropBig() (Image.Image method), 21  
cropImg() (Image.Image method), 21  
cropSmall() (Image.Image method), 21

## D

data2db() (SigLib.SigLib method), 14  
data2img() (SigLib.SigLib method), 14  
Database (class in Database), 26  
Database (module), 25  
date2doy() (in module Metadata), 18  
datetime2iso() (in module Metadata), 19  
decomp() (Image.Image method), 21  
decomposition\_generation() (Image.Image method), 21  
deltree() (in module Util), 31

## E

exportToCSV() (Database.Database method), 27  
extractGCPs() (Metadata.Metadata method), 17

## F

fnameGenerate() (Image.Image method), 22

## G

get\_ceos\_metadata() (Metadata.Metadata method), 18  
get\_data\_block() (in module Metadata), 19  
get\_field\_value() (in module Metadata), 19  
getAmp() (Image.Image method), 22  
getASFMetaCorners() (Metadata.Metadata method), 17  
getASFProductType() (Metadata.Metadata method), 17  
getBandData() (Image.Image method), 22  
getCEOSmetafile() (Metadata.Metadata method), 17  
getCornerPoints() (Metadata.Metadata method), 17  
getdBScale() (in module Util), 32  
getDimgname() (Metadata.Metadata method), 18  
getEarthRadius() (in module Metadata), 19  
getFilename() (in module Util), 31  
getgdalmeta() (Metadata.Metadata method), 18  
getGroundRange() (in module Metadata), 19  
getImgStats() (Image.Image method), 22  
getMag() (Image.Image method), 23  
getMoreGCPs() (Metadata.Metadata method), 18  
getNoise() (Image.Image method), 23  
getPhase() (Image.Image method), 23  
getPowerScale() (in module Util), 32  
getRS2metadata() (Metadata.Metadata method), 18  
getSigma() (Image.Image method), 23  
getSlantRange() (in module Metadata), 19  
getTheta() (Image.Image method), 23  
getThetaPixel() (in module Metadata), 19  
getThetaVector() (in module Metadata), 19  
getZipRoot() (in module Util), 32

## H

handler() (SigLib.SigLib method), 15

## I

Image (class in Image), 20  
Image (module), 20  
imgData2db() (Database.Database method), 27  
imgWrite() (Image.Image method), 23

## L

llur2ullr() (in module Util), 32

## M

makeAmp() (Image.Image method), 23  
 makePyramids() (Image.Image method), 23  
 maskImg() (Image.Image method), 23  
 matrix\_generation() (Image.Image method), 23  
 meta2db() (Database.Database method), 27  
 Metadata (class in Metadata), 16  
 Metadata (class in SigLib), 16  
 Metadata (module), 16

## N

nameRelationTable() (Database.Database method), 27  
 numpy2sql() (Database.Database method), 27

## O

openDataset() (Image.Image method), 24

## P

polarFilter() (Image.Image method), 24  
 polarimetric() (SigLib.SigLib method), 15  
 proc\_Dir() (SigLib.SigLib method), 15  
 proc\_File() (SigLib.SigLib method), 15  
 projectImg() (Image.Image method), 24

## Q

qryCropZone() (Database.Database method), 28  
 qryFromFile() (Database.Database method), 28  
 qryFromText() (Database.Database method), 28  
 qryGetInstances() (Database.Database method), 28  
 qryMaskZone() (Database.Database method), 29  
 qrySelectFromAvailable() (Database.Database method),  
 29

## R

readdate() (in module Metadata), 19  
 reduceImg() (Image.Image method), 24  
 relations2db() (Database.Database method), 29  
 reprojSHP() (in module Util), 33  
 retrieve() (SigLib.SigLib method), 15

## S

saveMetaFile() (Metadata.Metadata method), 18  
 scientific() (SigLib.SigLib method), 15  
 SigLib (class in SigLib), 14  
 SigLib (module), 14  
 snapCalibration() (Image.Image method), 24  
 snapDataTypeConv() (Image.Image method), 24  
 snapSubset() (Image.Image method), 25  
 snapTC() (Image.Image method), 25  
 sql2numpy() (Database.Database method), 30  
 stretchLinear() (Image.Image method), 25

## U

ullr2llur() (in module Util), 33  
 unZip() (in module Util), 33  
 updateFromArchive() (Database.Database method), 30  
 updateROI() (Database.Database method), 30  
 Util (module), 31

## V

vrt2RealImg() (Image.Image method), 25

## W

wkt2shp() (in module Util), 33  
 wktpoly2pts() (in module Util), 33