SigLib Documentation

Release 2.8

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

Jun 13, 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 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 are discovered by ingesting a copy of the Canadian Ice Service archive (or other geodatabase containing metadata, with tweaks), or by crawling through a hard drive and extracting metadata from zipped SAR scenes, or by querying a table in a local database that contains geospatial metadata. Queries take a Region Of Interest – ***ROI shapefile*** with a specific format as input. The region of interest delineates the spatial and temporal search boundaries. The required attribute fields and formats for the ROI are elaborated upon in a section below. New functionality will include the ability to query online resources such as EODMS.
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 query that selects a list of zip files to be processed (functionality to come).
3. A **Quantitative Mode** where remote sensing scenes can be converted to either calibrated (sigma0), noise level, or incidence angle images. Image data (from each band) can be subsampled by way of an **ROI shapefile** that references every image and specific polygon you want to analyze. These polygons represent sampling regions that you know about (a priori) or they are hand digitized from Qualitative mode images. Data is output to a .csv file or can be stored in a table in a geodatabase for further processing (functionality to come).

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).

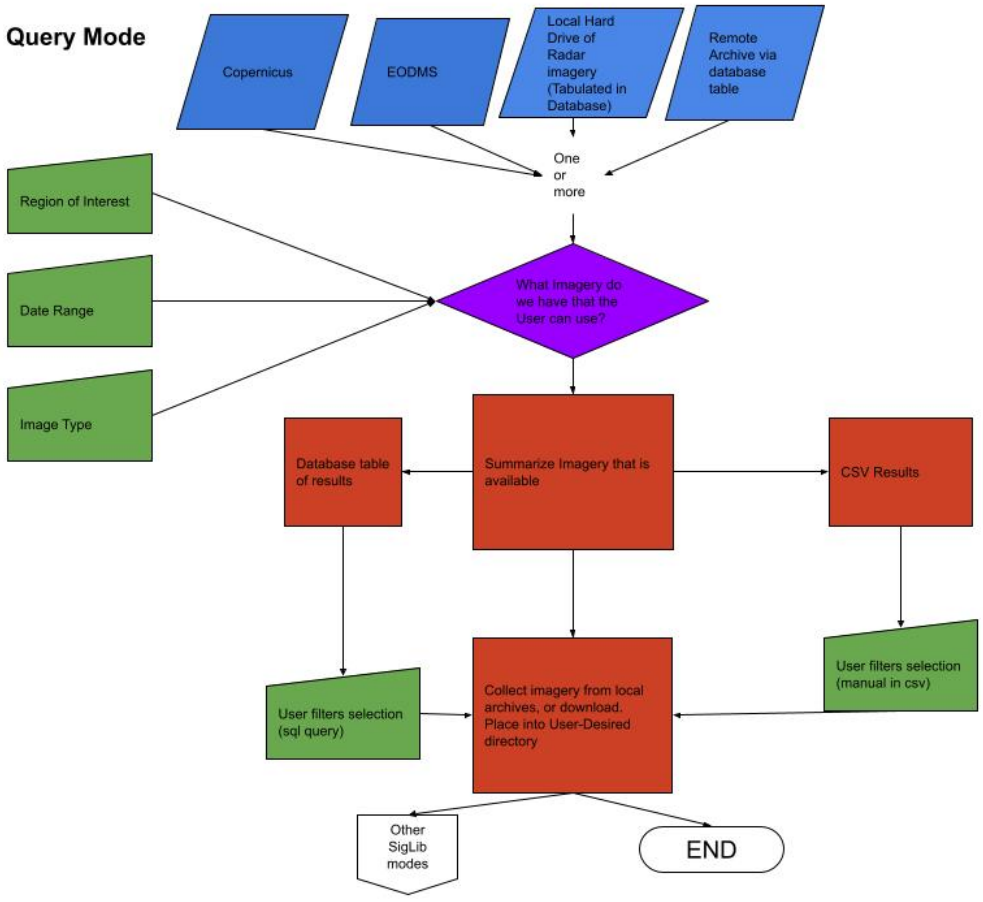

Fig. 1: Flowchart depecting the basics of Query Mode (This is the end goal, we are currently not at this stage!)

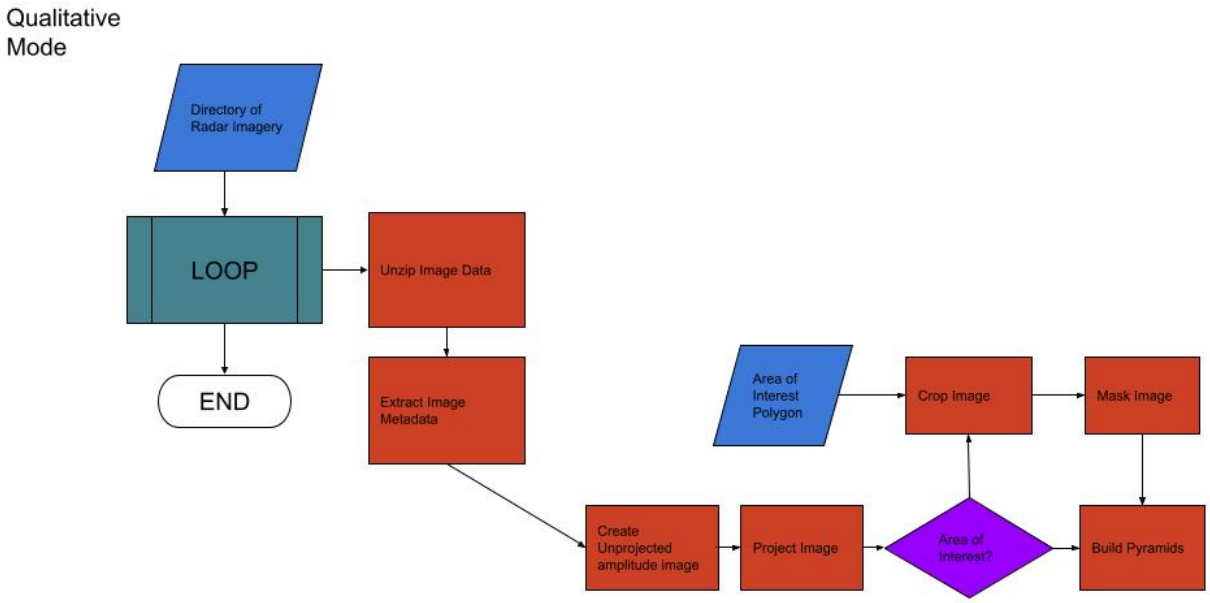

Fig. 2: Flowchart depecting the basics operations performed in Qualitative Mode.

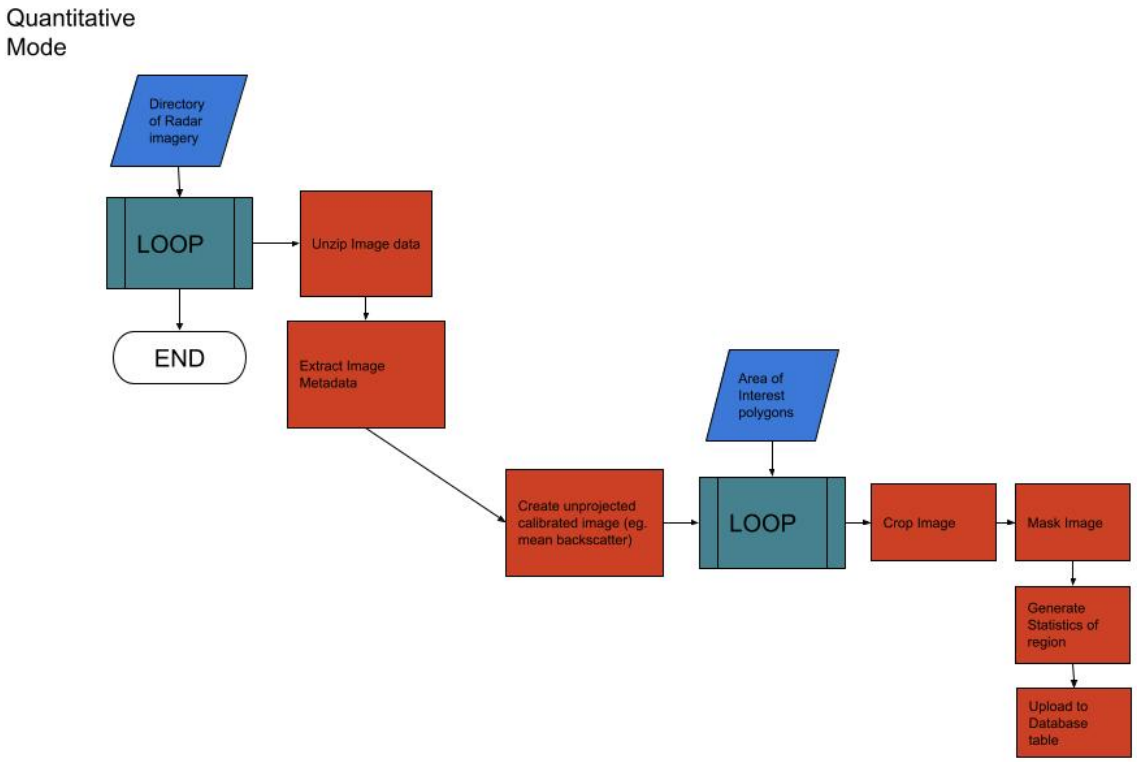

Fig. 3: Flowchart depecting the basics operations performed in Quantitative Mode.

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, and Allison Plourde. Ron Saper, Anna Crawford and Greg Lewis-Paley helped out as well (indirectly).

CHAPTER

TWO

OVERVIEW OF SIGLIB.PY AND ITS DEPENDENCIES

2.1 Dependencies

You will need a computer running linux or windows 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); alternatively, or you can use the provided Requirements.txt file to install them all at once.

- future
- psycopg2
- GDAL
- numpy
- scipy
- pandas
- configparser

Other requirements include:

- gdal/ogr libraries - (https://gdal.org/)
- PostrgreSQL/PostGIS (could be on another computer) (https://www.postgresql.org/ and https://postgis.net/)
- SNAP (Sentinel Application Platform from ESA) (https://step.esa.int/main/toolboxes/snap/) - installation will require you to know where your python.exe file is located, if you are using Anaconda, this can be found by entering <where anaconda> into the Anaconda Command Prompt.

Note that it is possible to run the software without SNAP or PosgresSQL/PostGIS but functionality will be ''very" 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
- If you want to take advantage of multiple cores on your **linux** machine to greatly enhance processing speed you will need GNU Parallel

Sorry, but 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

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: in your SigLib folder, there is a file named Requirements.txt that contains all the necessary Python Libraries. The libraries can be installed all at once by entering the following command in your terminal:

```
pip install -r /path/to/Requirements.txt
```

- Setup Directories: you will need to create a set of directories (folders) that SigLib will access through the [[SigLib#Config File|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.
- You will need to add projections to the folder you created for the ''projDir". Please refer to the *A Note on Projections* for more information. Adding at least one projection file into your projection directory may be a necessary step in order to run SigLib functions that operate outside of a PostGIS database.

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:password_person

- 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;

2.2.1 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 config file, set the *create_tblmetadata* variable to *1*
- 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.

2.3 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 (work in progress).

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. An examples of this are included:

1. **Polarimetry.py** - An independant script used generate polarimetric variables for SAR imagery using SNAP-ESA (Work in Progress).

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. 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
- 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 = 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
- table = 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
- query = 1 for scan over the results of a query and operate on all files returned; 0 otherwise
- file = 1 for run process on a certain file, which is passed as a command line argument (note this enables parallelized code); 0 otherwise
- scanFor = a file pattern to search for (eg. *.zip) - use when path=1
- uploadData = 1 to upload descriptive statistics of subscenes generated by Scientific mode to database

Process

- data2db = 1 when you want to upload metadata to the metadata table in the database (Discovery Mode, outdated)
- data2img = 1 when you want to manipulate images (as per specs below) (Qualitative Mode)
- scientific = 1 when you want to do image manipulation involving the database (Quantitative Mode)

IMGMode

- proj = basename of wkt projection file (eg. lcc)
- projSRID = SRID # of wkt projection file
- imgtypes = types of images to process
- 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).

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
```

2.6 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 Derek made this problem worse with his own 'standard image naming convention' called **dimgname**. All files processed by SigLib get named as follows, which is good for:

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

Template: date_time_sat_beam_data_proj.ext

Example: 20080630_225541_r1_scwa__hh_s_lcc.tif

Table: **dimgname fields**

Position	Meaning	Example	Chars
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

2.7 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']

2.8 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 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. This option is an sql executable that can be run within PostgreSQL to add the desired projection into the spatial_ref_sys table.

2.9 Example workflow:

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

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

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

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

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

Table: **ROI.shp fields**

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

- See folder ROISamples for example ROIs

CHAPTER

THREE

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.

NOTE: These tutorials are out of date and will need to be updated in the future! Please do not rely on them.

3.1 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 refered 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

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

In this example we will be using SigLib to produce Tiff images from Amplitude Radarsat2 image files. Before any work beings 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.

```
[Directories]
scanDir = "Path to dir with zipfiles"
tmpDir = "Path to dir for temporary files"
projDir = "Path to dir containing projection files"
vectDir = "Path to dir with ROI shapefile(s)"
imgDir = "Dir completed products will be stored
logDir = "Dir logs are stored in"
archDir = "Image archive directory"
errorDir = "Dir to store error logs"

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

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

[Process]
data2db = 0
data2img = 1
scientific = 0
polarimetric = 0

[MISC]
proj = lcc
projSRID = 96718
imgtypes = amp
imgformat = GTiff
roi =
roiprojSRID =
mask =
crop =
spatialrel =
```

Fig. 1: A basic config file for this task

3.3 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 verison of SigLib.py, the second is the location of the associated config file.

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

```
[Directories]
scanDir = "Path to dir with zipfiles"
tmpDir = "Path to dir for temporary files"
projDir = "Path to dir containing projection files"
vectDir = "Path to dir with ROI shapefile(s)"
imgDir = "Dir completed products will be stored
logDir = "Dir logs are stored in"
archDir = "Image archive directory"
errorDir = "Dir to store error logs"

[Database]
db = "Your Database"
host = "Host for your Database"
create_tblmetadata = 0
uploadROI = 0
table = "Database table to query"

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

[Process]
data2db = 1
data2img = 0
scientific = 0
polarimetric = 1

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

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

Once started, Parallel will begin to step though your selected directory looking for .zip files. Once one is found, it will pass it to one of the 16 availible (or however many cores you set) openings of Siglib.py. SigLib will unzip the file via Util.py, grab the metadata via Metadata.py, then connect to your desired database, and 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.

3.4 Example #3: Quantitative Mode Basics

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

```
[Directories]
scanDir = "Path to dir with zipfiles"
tmpDir = "Path to dir for temporary files"
projDir = "Path to dir containing projection files"
vectDir = "Path to dir with ROI shapefile(s)"
imgDir = "Dir completed products will be stored
logDir = "Dir logs are stored in"
archDir = "Image archive directory"
errorDir = "Dir to store error logs"

[Database]
db = "Your Database"
host = "Host for your Database"
create_tblmetadata = 0
uploadROI = 1
table = "Database table to query"

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

[Process]
data2db = 0
data2img = 0
scientific = 1
polarimetric = 0

[MISC]
proj = lcc
projSRID = 96718
imgtypes = sigma
imgformat = GTiff
roi = sampleScientific
roiprojSRID = 96718
mask =
crop =
spatialrel =
```

Fig. 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 nessesary, otherwise it can be set equal to 0

projected and turned into its own TIFF file for delivery, or the image data for the instances is uploaded to a database table made to store data from this run.

3.5 Conclusion

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

CHAPTER

FOUR

SIGLIB API