

Automatic Interferogram System

System architecture Configuration and Operation Manual

Project Sponsored by: NOA – IAASARS

Project team

Member	Task	Contact
Dr Harris Kontoes – NOA Research Director	Sponsor, Management	kontoes@space.noa.gr
Dr Giannis Papoutsis – NOA Reasearcher	Management, Design, Testing	ipapoutsis@noa.gr
Alex Apostolakis – Electrical and Computer Engineer, MSc SSTA	System Design, Development, Testing	a.apostolakis@yahoo.gr alex.apostolakis@technoesis.gr tel: +306974045959
Andreas Al Saer	Development, testing	adreas.saer@gmail.com

Document History

Date	Version	Author	Reviewer
	1.0	Alex Apostolakis	

Contents

1. INTRODUCTION	5
2. SYSTEM ARCHITECTURE.....	6
3. INSTALLATION	7
3.1. PRODUCTS COLLECTION AND TASK SCHEDULING SERVICE	7
3.2. ENVI IDL	7
3.3. POSTGRESQL SERVICE DATABASE	7
4. SERVICE BASIC FLOWCHART	8
5. SERVICE OPERATION	9
5.1. START, STOP SERVICE	9
5.2. THE FOLDER STRUCTURE AND FILE NAMING	9
5.2.1. Configuration folder	9
5.2.2. Orbits folder	9
5.2.3. Logs folder	10
5.2.4. Events folder	10
5.2.5. Sentinel-1 folder	10
5.3. NEW SERVICE REQUEST	11
5.4. MONITOR PROCESSING.....	11
5.4.1. View view_service_output	13
5.4.2. View steps_exec	14
5.5. CONTROL EXECUTION OF SERVICE REQUESTS	14
5.5.1. Reset a service request.....	14
5.5.2. Re-execute a step	14
5.5.3. Stop execution and Cancel a step.....	15
5.6. LOGGING	15
5.6.1. Main application Log	15
5.6.2. Monitor service status Log	15
5.6.3. Service request Log	15
5.6.4. The STEPS_LOG table	15
6. SERVICE CONFIGURATION	16
6.1. MAIN SERVICE CONFIGURATION FILE.....	16
6.2. THE STEPS TABLE	20
6.3. THE INTERFEROGRAM “IFG” SERVICE STEPS IMPLEMENTATION	21
6.4. THE USERS TABLE.....	22
6.5. THE SARSCAPE, IDL SCRIPTS CONFIGURATION FILES	22
6.5.1. IDL scripts configuration file.....	22
6.5.2. The xml ENVI - SARscript profiles	23
A. DATABASE SCHEMA.....	25
A.1. SATELLITE_INPUT TABLE	25
A.2. SERVICE_OUTPUT TABLE	25
A.3. SERVICE_REQUESTS TABLE.....	25
A.4. STEPS_EXECUTION TABLE	25
A.5. STEPS TABLE	26
A.6. STEPS_LOG TABLE	26
A.7. USERS TABLE	26
B. PYTHON CLASS REFERENCE.....	27

B.1. AUTOIFGSRV.PY	27
B.2. SERVICEEVENT.PY	29
B.3. SEARCHIMAGES.PY	32
B.4. SERVICEPROCESS.PY	35
B.5. EVENTDETECTION.PY	37
B.6. GEOMTOOLS.PY	38
B.7. STEP_UTILS.PY	38
C. REFERENCES	40

1. Introduction

The Automatic Interferogram System has as purpose to automatically create all the SAR interferograms around the location of a geological hazard event like earthquake, volcano eruption etc. The interferograms production is triggered by the input of an area of interest or point and a timestamp. The timestamp represents the time of the event being studied and the area of interest a polygon of an area around the event or the point the point of the event.

After the system is triggered by an event input it automatically scans the Copernicus hubs to find the appropriate Sentinel-1 satellite data, downloads the data and executes the tasks needed to produce the interferograms. For the interferogram creation the system uses ENVI - SARscape commands.[1]

The service users are notified about the progress and the status of processing steps automatically by email or consulting information tables of the service metadata database.

2. System Architecture

The main components are:

- The “Products collection and Task scheduling service” written in python. This program initiates the request processing, finds and downloads the necessary satellite inputs and controls the output production tasks.
- The above system is assisted by a service metadata database (postgresql) that contains processing information, input and output metadata
- The ENVI SARscape module is the interferogram production engine. The “Products collection and Task scheduling service” execute commands from ENVI SARscape module using IDL scripts in order to create the interferograms
- The IDL scripts that execute the main interferogram tasks calling the SARscape commands.

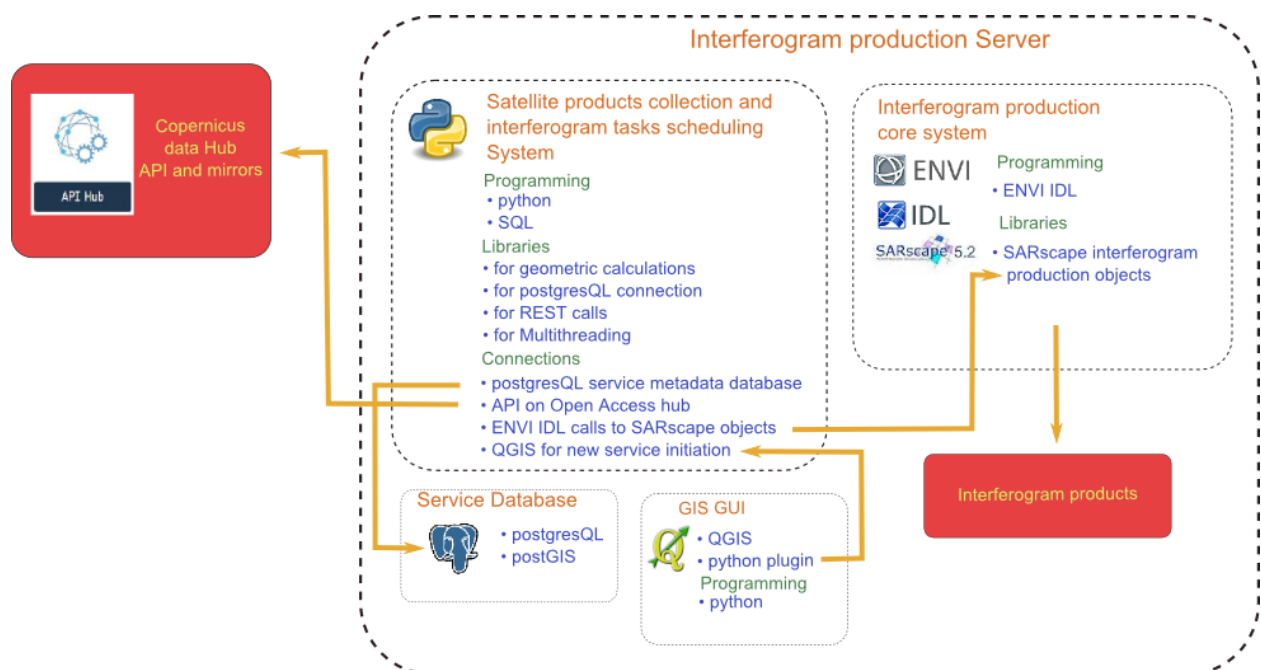


Figure 1: Technologies and Architecture

3. Installation

3.1. Products collection and Task scheduling Service

The “Products collection and Task scheduling Service” is a python program that consists of several python modules. The main .py module that starts the service is the `autoifgsrv.py`. The service root folder and database are defined by the `GEOHUBROOT` and `GEOHUBDB` environment variables that have to be set before running the `autoifgsrv.py`.

The modules are written for python 2.7.x, thus Python 2.7 has to be installed on the server and additional python 2.7 libraries for Gdal-osgeo, Postgresql, requests [2][3][4]

If more than one python installations are present the environment variables for `PYTHONHOME` and `PYTHONPATH` have to be set for the right python executables and libraries. [5]

Another point of attention is potential conflict between 32-bit and 64-bit installations of GDAL libraries that may exist simultaneously on the server. In such case the libraries of one installation might have to be completely uninstalled for the other to function properly.

3.2. ENVI IDL

ENVI and SARscape have to be installed on the server of course with their usage license.[6][7]

In order to run the IDL scripts and the SARscape module routines the environment variables `IDL_PATH`, `IDL_DLM_PATH` [8] have to be set like below.

```
IDL_DLM_PATH=<IDL_DEFAULT>
```

```
IDL_PATH=<IDL_DEFAULT>;service source folder\idl_code;
```

3.3. PostgreSQL Service database

The Products collection and Task scheduling Service is assisted by a PostgreSQL database that is used as storage for the service requests data and metadata. Thus on the server a postgresql installation has to be present along with the PostGIS add-on.[9]

A copy of the database in SQL format can be found in the source folder of the service program. In order to perform new installation of the service a new PostgreSQL database has to be created from that copy.[10]

4. Service basic flowchart

The service is constantly scanning for new trigger input to start processing a new request. If a new event request is detected the processing flow starts. A basic flowchart of the functionality is shown below.

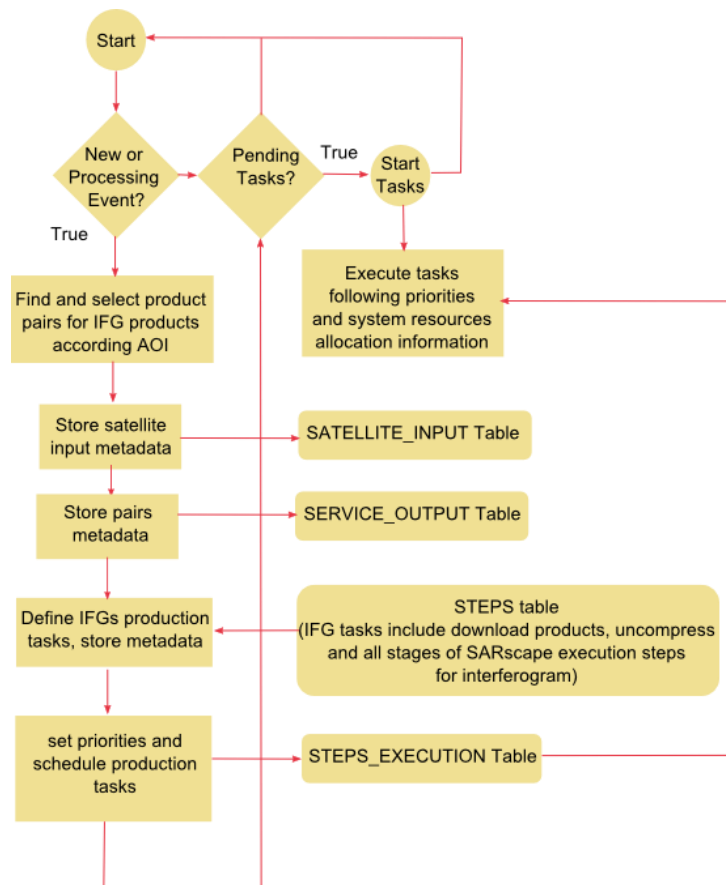


Figure 2: Basic flowchart

When an event trigger is detected the service starts the search for product pairs to produce the respective interferograms relevant to the event. The pairs searched are two kinds: the pre-seismic and co-seismic pairs.

Pre-seismic pairs consist of two images on the same orbit and direction. The “master image is the most close before the event and the “slave” is the next more recent before the event.

Co-seismic pairs consist again of two images on the same orbit and direction. The “master image is the most close before the event and the “slave” is the more timely close after the event.

First “master” products are searched. The service selects as “master” products all the timely closer to the event products in different orbits that their footprints intersect the AOI.

For each master product the service then run a search to find pairs for co-seismic and pre-seismic products. The condition to accept a pair is configurable and it is defined by a parameter in the main configuration file (see 6.1 paircondition:).

Of course co-seismic product pairs may not be available the time of the request. The service constantly searches the hubs until those products become available.

Each “approved” product pair, from which an interferogram output will derive, is stored to the SERVICE_OUTPUT table. In this table the “master” and “slave” pair is recorded in the “inputs” column.

After defining the interferogram outputs the service sets priorities according which co-seismic pair is the most likely to be produced first. Then it stores all the tasks (steps) for each output in the STEPS_EXECUTION table and starts the execution according priorities, prerequisites and resources limitations. In order to form the records of the STEPS_EXECUTION table the service looks at the STEPS table (see 6.2, 6.3) where the tasks their sequence, prerequisites and parameters of necessary steps for producing the interferogram (“ifg”) type output are stored.

5. Service Operation

5.1. Start, Stop service

The start stop operation is done by windows command (or batch) files.

To start the service run:

start_autoifg.bat

The service starts and then checks the status constantly.

If the start command window is closed, to check the status of the service run:

check_autoifg.bat

To stop the service run:

stop_autoifg.bat

If in a command window (start or check) you get the message "Server has abnormally stopped or hung" check with Task manager for IDL processes or other high CPU consuming processes running (after the server is stopped). Kill those processes and then run:

clean_autoifg.bat

before starting the server again.

5.2. The folder structure and file naming

Under the root folder (see 3.1, 6.1 rootpath:) of the service there are the following folders:

5.2.1. Configuration folder

The Configuration folder is named by configpath: parameter (see 6.1 configpath:)

Under the configuration folder we find:

- the processed folder (see 6.1 processed:) where the event trigger files are moved
- the “profiles” folder where the XML ENVI - SARscript profiles are located (see 6.5.2)
- the event trigger files are sought in this folder by the service (see 5.3)

5.2.2. Orbits folder

The Orbits folder is named by orbits: parameter (see 6.1)

Under the orbits folder the orbits files are stored in the following way:

<Orbits_folder>/<mission (S1A or S1B)>/<Year>/<Month>/<orbit file name>

5.2.3. Logs folder

The Logs folder is named by logs: parameter (see 6.1)

Under the logs folder the system logs are located (see 5.6)

5.2.4. Events folder

The events folder is named by datapath: parameter (see 6.1) and it is the main location where output files from processing a service request are created. Under the event folder the output is organized like below:

- <event folder> the event folder name consists of event name and event date time as they were given in the trigger file of the event.
 - A log file specific to the event is located in this folder (see 6.1 processlog:, 5.6).
 - <Output folders> are located under <event folder>. They contain the output of the interferogram processing for the output in the “ifg” folder under this folder. The output folders are named like below:

<master sensing time>_<slave sensing time>_<relative orbit>_<direction>

 - Interferogram (“ifg”) folder is located under each output folder and contains all the interferogram process output files
 - Under “ifg” except of the output files there are working directories for each process that contain logs of the ENVI – SARscape command execution. The working directory names are defined by the IDL scripts configuration file. (see 6.5.1)

5.2.5. Sentinel-1 folder

The sentinel1 folder contains all the downloaded products and the output processing of individual products (not pairs) like for example the ENVI import (ingestion) or the DEM extraction. It is named by the sentinel1path: parameter (see 6.1)

Under the sentinel-1 folder the product folders are organized like below:

- <Year>/<Month>/<sensing time>_<relative orbit>_<direction>
 - Product file compressed
 - Product quick look image
 - <uncompressed> folder that contains the product data in uncompressed form
 - Ingestion folder contains the output processing data of ENVI SARscape import process (see 6.5.1 import_dir:)
 - DEM folder contains the output processing data of ENVI SARscape import process (see 6.5.1 DEM_dir:)
 - Under ingestion and DEM folders except of the output files there are working directories for each process that contain logs of the ENVI – SARscape command execution. The working directory names are defined by the IDL scripts configuration file. (see 6.5.1)

5.3. New service request

5.3.1. New service initiation based on trigger file from qgis python interface

To initiate and start processing a new service request the system has to detect an input trigger. This trigger is in fact a file that contains the necessary input to form a new service request.

This file can be created automatically by the QGIS add on “Hazard Pro” or it can be created manually by the user. The file name is defined by the parameter “eventfile:” in the main configuration file (see 6.1). If a file with the right specs is present in the configuration folder the service loads it and initiate a new request.

The file format is:

```
Event Name
YYYY-MM-DD HH:MM:SS
(long,lat)
(long,lat)
Profile XML file full path name
```

The first line is the event name, the second line is the date, the third and fourth lines are the diagonal coordinates of the vertices of a rectangle that its sides are parallel to meridians and parallels of latitude. The last line is the full path name of an XML file that contains the parameters needed as input to SARscape commands for creating interferogram (see 6.5.2).

5.3.2. New service initiation based on file from automated event detection system

The service constantly monitors earthquakes web services like USGS and EMSC and collects data for events according the user preferences. The service each time it collects an event creates a file with name :

event_<id>.json

The file format is:

```
{
  "depth": 20.0,
  "magnitude": 5.6,
  "name": "NEAR N COAST OF PAPUA, INDONESIA",
  "epicenter": "POINT (139.8 -1.84)",
  "time": "2018-04-26 16:56:00"
}
```

All files of the above format are loaded into the service_request table and have as status ‘ev_detected’. The user must change the status from ‘ev_detected’ to ‘ev_trigger’ to start processing the event (see 6.1). That kind of file can also be created manually by the user in order to initiate a service request.

5.4. Monitor Processing

The tool to monitor the process and steps execution is the pgadmin III tool of the PostgreSQL database.

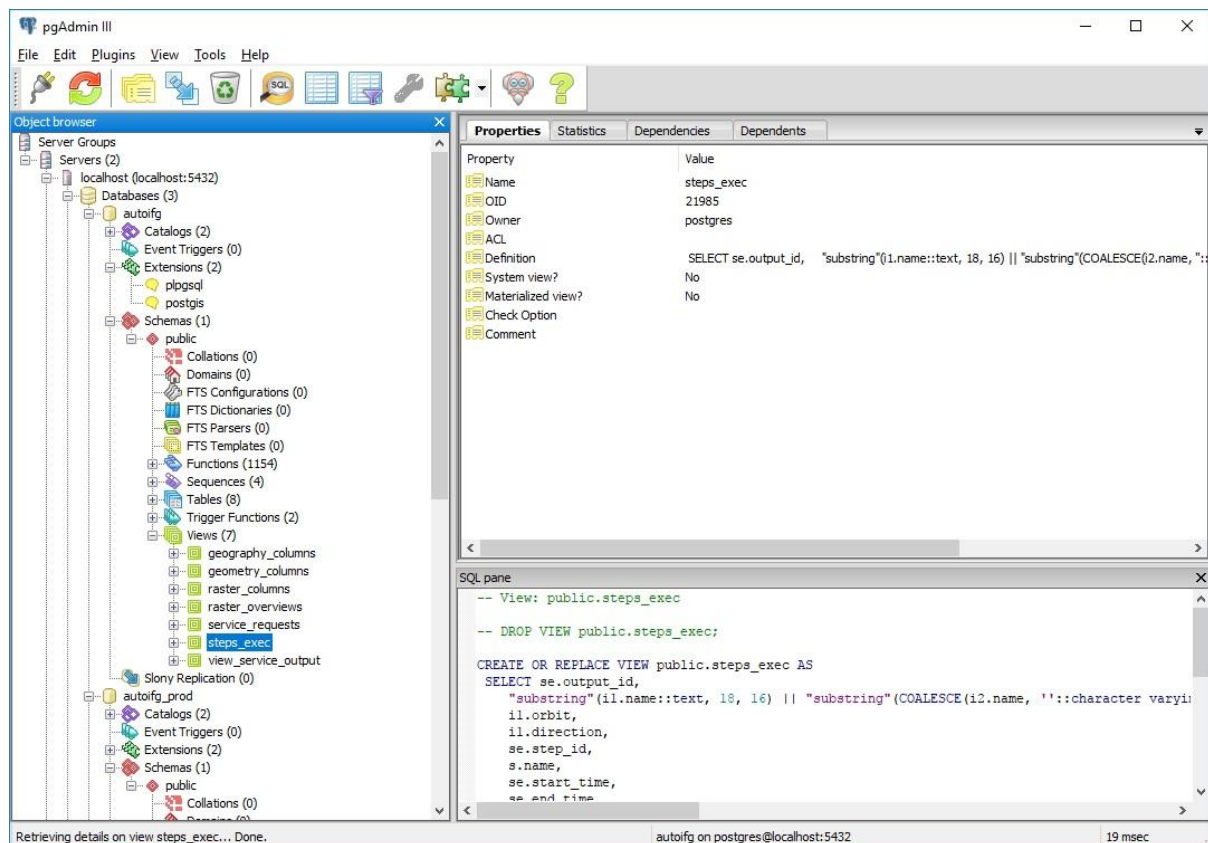




Figure 3: the PgAdmin III tool

In order to open a database view we start the pgadmin III tool, we select the localhost server and double click to connect. We select from the databases the one that has the name defined by the environment variable GEOHUBDB in the start_autoifg.bat command window and we open the tree. From the schemas we select public and then the “views” tree (in case we want to open a table we select the “tables” tree). We select the desired view and then we click on the table icon ().

After opening a view we can apply a filter for any of the columns of the view or a combination of them. To apply a filter we click on the filter button (). The conditions syntax is like the “WHERE” part of an SQL sentence. For example if we want to see only the rows where the output_id column has values over 309 we write: output_id>309.

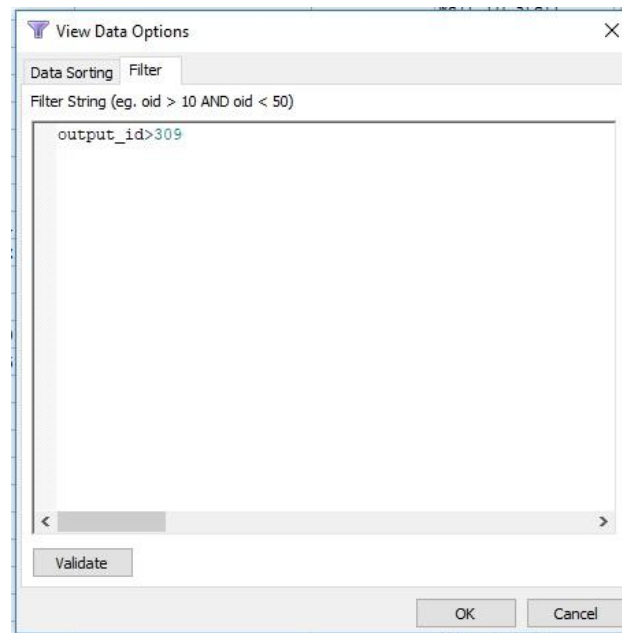



Figure 4: Filter the View

The database views “view_service_output” and “steps_exec” contain all the necessary information about the processing of the outputs.

To refresh the contents of the view click on  or press F5

5.4.1. View view_service_output

The view_service_output view contains all the service outputs that are processed, processing or wait to be processed.

Table 1: The service output view

Column	Description
output_id	the output id
event_location	The event name (location usually)
event_date	the event date and time
Master	the master product name
Slave	the slave product name
Priority	The processing priority of the output
output_status	Status can be requested, processing or finished
first_step	The first step-task id in the processing seunce
out_start	Start date-time of processing
last_step	The last step-task id in the processing seunce
out_end	End date-time of processing
out_est_end	Estimated end time of processing

5.4.2. View steps_exec

The steps_exec view contains in its rows in detail all the service steps of all outputs that are processed, processing or wait to be processed.

Table 2: The STEPS_EXEC View

Column	Description
Outname	The output name consist of master product sensing time and slave product sensing time if slave is available
Orbit	the relative orbit number
Direction	the direction
step_id	the step ID
Name	the step name
start_time	The start time of the step
end_time	The end time of the step
Duration	the duration
Status	The status can be waiting, processing, failed or cancelled. A step is cancelled after it fails to finish successfully a number of times.
estimate_end	the estimation end time for the step
Progress	The progress of the step execution. This column has a value only when the step is in processing status. Each step gets input from the process running about its progress. In the case of external OS processes called (like IDL scripts) the service uses an assigned log parser to parse the appropriate information log of the OS process.

5.5. Control execution of Service requests

5.5.1. Reset a service request

You may want to recreate the outputs of a service request in case for example you change the parameters of the interferogram production. To force the recalculation of a service request open the database table “SERVICE_REQUEST”. To open a database table, follow the procedure described in 5.4 but instead of opening the “views” tree open the “tables” tree.

Locate the row of the service request you want to recalculate, double click on the “status” column of that row and change the status to the value defined by “ev_reset:” in the main configuration file (see 6.1). Save the table (File->Save or click the “save” icon). In a few seconds all the steps running will stop and the outputs creation steps of this service request will be erased. To start processing the service request again change the status from ‘detected’ to ‘requested’. If you want to erase files created from this service_request you have to do it manually

5.5.2. Re-execute a step

You may want to restart a step that was cancelled or because you changed some execution parameters. To restart a step open the database table “STEPS_EXECUTION” (see 5.4)

Locate the row of the step using columns step_id and output_id. You can use the Filter options to make your search easier. Double click on the “status” column of the right step record row and change the status to the value defined by “step_reset:” parameter in the main configuration file (see 6.1). Save the table (File->Save or click the “save” icon). The step will

be first terminated (if running) and will be scheduled to run again from the service according priorities, prerequisites and resource allocation.

5.5.3. Stop execution and Cancel a step

You may want to avoid the execution of a step and those that are depended on that step. To do that open the database table “STEPS_EXECUTION” (see 5.4)

Locate the row of the step using columns `step_id` and `output_id`. You can use the Filter options to make your search easier. Double click on the “status” column of the right step record row and change the status to the value defined by “step_kill:” parameter in the main configuration file (see 6.1). Save the table (File->Save or click the “save” icon). The step task will be terminated if running and it will turn to cancel status, thus it will be ignored by the step running process.

5.6. Logging

5.6.1. Main application Log

The main log is located under the “Logs” folder (see 6.1 logs: parameter) and is named `system.log`. It records the starting and stopping of the service and errors that may happen in the core functionality of the service.

Search for “error”, “warning” or “traceback” keywords in the log to locate problems in execution”.

5.6.2. Monitor service status Log

This file, under the Logs folder (see 6.1 runfile: parameter) is constantly updated by the server with a timestamp in order to show activity. If that file exists and the service is stopped an abnormal termination has probably happened. To restart the service this file has to be deleted. Additionally the task manager has to be checked for OS processes running that may have been called by the service before abnormal termination.

5.6.3. Service request Log

This log is located under each service request folder (see 5.2.4). It contains information about the seeking of products in the Copernicus hub for this service request and any warning or error may occur during the initiation and definition of the request processing tasks.

Search for “error”, “warning” or “traceback” keywords in the log to locate problems in execution”.

5.6.4. The STEPS_LOG table

For each step-task that is defined uniquely by the step ID and the output ID a history is kept. This table contains the task execution history.

Filter with values of parameters “step_failed:”, “step_processing:” or “step_cancel:” (see 6.1) the status to locate problems in execution of steps.

Table 3: The STEPS_LOG Table

Column	Description
<code>output_id</code>	Output ID
<code>step_id</code>	Step ID
Logtime	Log Time
Message	Message
Status	Status of the step the time of the logging

6. Service Configuration

6.1. Main service configuration file

This file is called ifgconfig.ini and it is the main configuration file that defines the service parameters. It is located in the python source folder. The parameters are described below:

Table 4: Main configuration table

Parameter	Description
[Paths]	Service Folders
rootpath:	The root path of the service. All folders in the configuration are relative to the root path. This parameter is overridden by the environment variable GEOHUBROOT
logs:	The logs path of the service
processlog:	The log filename for each event. It is stored under each event folder
configpath:	The configuration path. In that folder the trigger event file is stored.
datapath:	In that folder all the interferogram data for each event are stored
sentinellpath:	Downloaded Sentinel-1 data folder. Additionally ingestion and DEM processing files are stored in that folder,
sentinelunzip:	Folder to store the uncompressed data of the downloaded sentinel-1 product
processed:	Folder to store trigger files uncompressed
orbits:	Folder to store orbits files
[Filenames]	Service File names
eventfile:	The trigger file name for a new event. This file is moved to "processed folder" after it is processed
stopfile:	The information file that stops the service
runfile:	The information file used to check the service is running
idl_pathconfig:	The file used to pass parameters to idl scripts (deprecated)
idl_configini:	the initial IDL scripts configuration in IDL source code folder
idl_python_config:	The merged configuration file for IDL scripts in the configuration folder. This file is automatically created before running an IDL command (if not present) by merging the initial IDL configuration file with the ifgconfig.ini. In case of change of the initial IDL configuration file or the ifgconfig.ini this file has to be deleted in order to be recreated with the changes applied.
[Copernicushubs]	Copernicus hubs

hubs1..n:	Each entry (hubsN where N integer from 1 to n) represent a Copernicus hub where the service is searching for products
orbitshub:	The URL where the orbits data are stored
[dbconnection]	Database connection
pg_dbname: autoifg	Database name. The database name is overridden by the environment variable GEOHUBDB
pg_user:	Database user
pg_password:	Database password
pg_host:	Database host
pg_port:	Database liscening port
[Status]	Status definitions
ev_trigger:	New event, it will automatically start processing
ev_process:	Processing event
ev_ready:	Finished processing event
ev_reset:	Request to reset event and re-process it
ev_detected:	New event that will not automatically start processing. Processing will start after changing the status to ev_trigger:
inp_new:	New satellite input
inp_searching:	Searching for Input
inp_downloading:	Downloading input
inp_available:	Input is locally available (downloaded)
out_new:	out_new: requested
out_searching:	searching output
out_downloading:	downloading output
out_processing:	processing output
out_ready:	finished processing output
out_archiving	The output files are currently copied to the archive storage location
out_archived	The output files are moved to the archive storage location
step_wait:	Step is waiting for its turn to be started by the service
step_processing:	step is running
step_completed:	step is completed successfully
step_failed:	step has failed
step_cancel:	step is cancelled probably after a number of failed process
step_archiving	The output files are currently copied to the archive storage location
step_archived	The output files are moved to the archive storage location
[Types]	Output types

out_sentinel:	Sentinel download
out_ifg: ifg	interferogram production
[IFG service]	[IFG service]
pastperiod:	Period to check for first product in the past of the event date. Example: If "repassing" period is 6 days and pastperiod is 1 the service will look first in the range[event date-6 days, event date] to find products. If pastperiod is 2 the service will look first in the range[event date-12 days, event date-6 days] to find products
repassing:	Repassing satellite period in days
searchperiods:	How many periods in the past (of the event) to look for products
tilesearchrange:	Range to search before and after a specific sensing time in minutes when the service is searching based on sensing time. This type of search is happening when the service is searching for the "slave" products. For example if tilesearchrange is 2 the searching range is [sensing time-2 minutes,sensing time+2 minutes]
filters:	Main filters to apply in search for interferogram product pairs according Full text search Copernicus hub specifications. Separate filters with "," Example value: platformname:Sentinel-1, producttype: SLC
inpsplit:	Character used to separate output inputs in inputs field of service output table
paircondition:	Condition to check in order to decide if a "slave" product is eligible to form an interferogram pair. For example if paircondition is: pcarea>0.7 or (pcarea>0.3 and not f2roiintersect.IsEmpty()) this means that the service is accepting pairs where master and slave products either have an area overlap greater than 70% or the overlap is greater than 30% and the slave's area intersects the given area of interest
checkslow:	Re-check time in minutes for new products when no new product is expected soon
checkfast:	Re-check time in minutes for new products when a new product is expected soon
fastsearchperiod:	Time range in minutes to consider a new product is expected soon after the sensing time has past
# IFG service parameters	Parameter aliases for calling IFG processing steps
pathmaster:	Alias for Path to master product
pathslave:	Alias for Path to slave product
pathorbitmaster:	Alias for Path to orbit file of master product
fileorbitmaster:	Alias for Orbit file name of master product
pathorbitslave:	Alias for Path to orbit file of master product
fileorbitslave:	Alias for Orbit file name of slave product
pathdem:	Alias for Path to DEM output

pathifg:	Alias for Path to IFG output
configxml:	Alias for XML configuration file and path
masterid:	Alias for Id of master product
slaveid:	Alias for Id of slave product
outputid:	Alias for Id of processing output
#request params	Parameters aliases for service request
xmlprofile: XML profile	Parameter alias for service request XML profile
[Resources]	Resources definition
procnum:	We assume that each execution step of the service acquire a virtual resource. Thus, limits are defined to the number of parallel steps (or processes-threads) that can use those resources according step priority. For example if procnum is {"server": {"1":1, "2":2}, "internet": {"1":1, "2":2} } that means that the service is using two types of resources ("internet", "server") and processes with priority "1" can run up to one process for both types while processes with lower priority ("2" and above) can run up to two processes for both types again. Of course lower priority steps can not acquire a resource if higher priority steps have consumed the total of available resources for their higher priority.
stepretries:	Number of retries after failed to characterize a step "cancelled"
bestdownloadbytes:	Number of bytes to download for hub speed test
slowdownloadbytes:	Number of bytes to download to determine if a download is very slow
slowdownloadtime:	Time limit (sec) to download 'slowdownloadbytes'
mediumdownloadbytes:	Number of bytes to download to determine if a download is slow and search for faster hub
Mediumdownloadtime:	Time limit (sec) to download 'mediumdownloadbytes'
[Notifications]	Notifications mail server
smtphost:	smtp host
smtpuser:	smtp user
smtppass:	smtp password
[Event]	Automated event detection parameters
Minmagnitude:	Minimum magnitude to collect events.
minmagnitudegreece:	Minimum magnitude to collect events from Greece area.
minmagnitudeworld:	Minimum magnitude to collect events from all over the world.
eventpastrange:	Past minutes from now to collect events
eventfileprefix:	Event file name prefix
rectcornerdist:	Rectangle corner distance in meters from event point to in order to form the search AOI

checkinterval:	Minutes interval for querying API for new event
[Archive]	Automated archiving parameters
freespacetrigger:	Start archiving when production storage free space is under this number of GBs
freespacelimit:	Do not archive if archive location free space is under this number of GBs
oldnesstrigger:	Archive outputs that have finished more than this number of days ago.
archiveroot:	Path to archive location
eventfileprefix:	Event file name prefix
rectcornerdist:	Rectangle corner distance in meters from event point to in order to form the search AOI
checkinterval:	Minutes interval for querying API for new event

6.2. The STEPS table

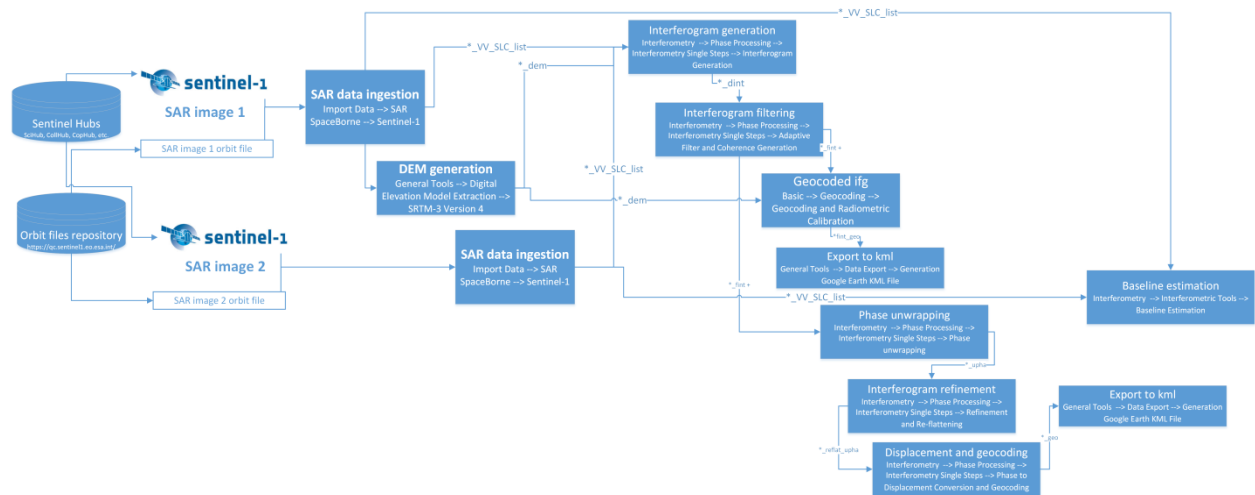
The steps table of the service database contains the necessary tasks information and the flow to execute in order to produce a specific type of a service output. The tasks schedule in the steps_execution table is created based on the information in that configuration table. Except for the meantime which is automatically calculated from the past execution times all the other fields have to manually be filled by a system's advanced user.

Table 5: The STEPS Table

Column	Description
id	The step ID
name	The step name
command	The command that execute the step. It can be a python function or an OS command
params	<p>The parameters and configuration of the command in JSON format.</p> <p>Example 1, step that calls a python function: <code>{"system": "python", "dyn_params": "#SlaveId#" }</code> The command name is in this case a python function and the parameters to be passed in the actual command is the slave product ID</p> <p>Example 2, step that calls an external OS function: <code>{ "system": "os", "dyn_params": "-e sarscape_script_EXPORTING_TO_KML -args #Path IFG#", "parser": "parseIDLscriptout" }</code> The command in this case is an external OS process (the IDL CLI) and additionally the parameters to be passed we add a "parser" name which is a python function that monitors the progress and successful or failed completion of the external OS process by parsing its logs.</p>
type	The type of the service output ("ifg" for interferogram)
prereq_steps	Prerequisite steps that have to be successfully completed before running this step
resource	Virtual resource acquired by this task
activated	If false the step is ignored when creating the tasks of a new output
meantime	Execution mean time of the step

6.3. The interferogram “ifg” service steps implementation

The steps for the Interferogram (“ifg”) output type are already filled based on the following flow:



In the steps table the following records are filled that execute the flow in the right order. The commands in the steps correspond to the python functions and IDL scripts that execute the respective tasks.

Table 6: STEPS table contents for Interferogram output type

ID	Name	Command	params	type	Prereq steps	resource
10	Download master	searchimages.ProductDownloader	{ "system": "python", "dyn_params": "#MasterId#" }	ifg		internet
15	Uncompress master	searchimages.ProductUnzipper	{ "system": "python", "dyn_params": "#MasterId#" }	ifg	10	server
20	Download slave	searchimages.ProductDownloader	{ "system": "python", "dyn_params": "#SlaveId#" }	ifg		internet
25	Uncompress slave	searchimages.ProductUnzipper	{ "system": "python", "dyn_params": "#SlaveId#" }	ifg	20	server
30	Ingestion master	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e SARscape_script_import_Sentinel_1 -args #Path Master# #Path Orbit Master# #File Orbit Master#", "parser": "parseIDLscriptout" }	ifg	15	server
35	DEM creation	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e SARscape_script_dem_extraction -args #Path Master#", "parser": "parseIDLscriptout" }	ifg	30	server
40	Ingestion slave	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e SARscape_script_import_Sentinel_1 -args #Path Slave# #Path Orbit Slave# #File Orbit Slave#", "parser": "parseIDLscriptout" }	ifg	25	server
50	Interferogram creation	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_interferogram -args #Path Master# #Path Slave# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout" }	ifg	35,40	server
60	Baseline estimation	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_baseline_estimation -args #Path Master# #Path Slave# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout" }	ifg	30,40	server
70	Interferogram filtering	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e SARscape_script_adapt_filt_coh_gen -args #Path Master# #Path Slave# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout" }	ifg	50	server
80	Interferogram	C:\\Program Files\\Exelis\\IDL85\\bin	{ "system": "os", "dyn_params": "-e sarscape_script_geocoding_rad_cal -args #Path	ifg	70	server

	geocoding	\\bin.x86_64\\idl.exe	Master# #Path Slave# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout")			
90	Interferogram export to kml	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_EXPORTING_TO_KML -args #Path IFG#", "parser": "parseIDLscriptout"}	ifg	80	server
100	Phase unwrapping	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_phase_unwrapping -args #Path IFG#", "parser": "parseIDLscriptout"}	ifg	70	server
105	Automatic GCP computation	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_automatic_gcp_computation -args #Path Master# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout"}	ifg	100	server
110	Interferogram refinement	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_ifg_refinement -args #Path Master# #Path Slave# #Path IFG# #XML Configuration#", "parser": "parseIDLscriptout"}	ifg	105	server
120	Displacement and geocoding	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_displacement_and_geocoding -args #Path Master# #Path IFG#", "parser": "parseIDLscriptout"}	ifg	110	server
130	Phase unwrapping Export to kml	C:\\Program Files\\Exelis\\IDL85\\bin\\bin.x86_64\\idl.exe	{ "system": "os", "dyn_params": "-e sarscape_script_phase_export_to_kml -args #Path IFG#", "parser": "parseIDLscriptout"}	ifg	120	server

6.4. The USERS table

The users table is the table that contains the users who should be notified for the different phases of the operation of the system and the service requests processing. Each user can be notified for different types of event notifications

Table 7: the USERS table

Column	Description
name	The user name
Email	The user email
Notifications	A json format string that contains the types of notifications the user should receive. Example: { "Step started": ["10", "20", "40", "30", "50", "100"], "Step finished": ["10", "20", "40", "30", "80", "120"], "output": ["new"], "event": ["new"], "error": ["hub"]} This user should receive be notified for notifications of type "Step started" for steps with id 10, 20, 40, 30, 50, 100, for notifications of type "Step finished" for steps with id 10, 20, 40, 30, 80, 120, for notifications of type "output" for "new" output, for notifications of type "event" in case of "new" event and finally for notifications of type "error" in case of "hub" error
Registered	The notification is sent to the user only in case this field is true

6.5. The SARscape, IDL scripts configuration files

The interferogram production is based on the ENVI – SARscape commands for that purpose. IDL scripts have been developed to execute the necessary tasks that are in their turn executed from the service environment.

6.5.1. IDL scripts configuration file

Some configuration items like folder and file names have to be common to all IDL scripts and in the same time accessible to the main service program. For that purpose the system uses

the “autoifg_idl.ini” file in JSON format located in the IDL source folder (idl_code/ folder under python source folder). This file is merged with some parameters of the main configuration file (see 6.1) and is copied to the configuration folder under the service root folder. The configuration parameters are described below:

Table 8: the IDL scripts configuration file

Parameter	Description
import_dir:	The ingestion folder of ENVI SARscape import process. It is located under the product download folder
import_work_dir:	The work folder of the ingestion task. It is located under the import folder
dem_dir:	The DEM folder
DEM_file:	The DEM file
dem_work_dir:	The work folder of the DEM task
ifg_out:	interferogram output files prefix
interferogram_work_dir:	The work folder of the interferogram task
interferogram_xml_params:	Parameter list to import from xml profile for interferogram task
baseline_work_dir:	The baseline task work folder
baseline_out1:	Baseline task output file name
baseline_out2:	Baseline task output file
adapt_filt_work_dir:	Interferogram filtering work folder
filtering_out:	Interferogram filtering output file
geocoding_work_dir:	Interferogram geocoding work folder
geocoding_out:	Interferogram geocoding output file suffix
exporting_work_dir:	Interferogram exporting work folder
exportifg_out:	Interferogram exporting output file
phase_work_dir:	Phase unwrapping work folder
phase_out:	Phase unwrapping output file
gcp_work_dir:	Automatic GCP computation work folder
autogcp_out:	Automatic GCP computation output file
ifg_refinement_work_dir:	Interferogram refinement work folder
refinement_out:	Interferogram refinement output file
displacement_work_dir:	Displacement and geocoding work folder
disbandgeo_out:	Displacement and geocoding output suffix
phase_export_work_dir:	Phase unwrapping export work folder
exportphase_out:	Phase unwrapping export output file

6.5.2. The xml ENVI - SARscript profiles

SARscape commands can take as input a significant number of configuration parameters defining the processing methods, inputs and results. All these parameters are included in an xml file located under the “profiles” folder which is under the service’s configuration folder (see 5.1)

Each service request can be bound to a different xml file providing as input to the SARscape commands the calculation parameters of that specific xml.

A. Database Schema

A.1. SATELLITE_INPUT Table

Column	Description
id	The ID given by GEOHUB database
product_id	The ID product ID from Copernicus hubs
sensing_start	Sensing start time (UTC)
sensing_stop	Sensing stop time (UTC)
direction	Orbit direction (Ascending- Descending)
orbit	The relative orbit number
footprint	The footprint polygon of the product in "Well Known Text" format
orbit_file	The file name of that includes orbit information retrieved from: https://qc.sentinel1.eo.esa.int/aux_resorb/
status	the status of the product. It can be requested, available or downloading
name	The name of the product
params	The parameters of the product in JSON format retrieved from query to Copernicus hub

A.2. SERVICE_OUTPUT Table

Column	Description
service_id	The ID of the service request that created the output
inputs	The product input ids (master, slave) of the output
id	The ID of the output
status	the status can be requested, processing or finished (see 6.1)
type	The output type. "ifg" for interferogram
priority	The output Priority. Outputs will be processed in the sequence defined by this number

A.3. SERVICE_REQUESTS Table

Column	Description
id	The ID of the service request
name	The name of the request, usually the event location
date	The event date
status	It can be requested, processing or ready (see 6.1)
request_date	The of the request initiation
search_poly	The polygon of the Area of interest in binary format
last_check	The date time of the last search for products relevant to the request
request_params	The parameters the specific to the service request in JSON format. (eg SARscape xml parameters file)

A.4. STEPS_EXECUTION Table

Column	Description
output_id	The output ID for that step
step_id	The step ID
start_time	Date time the task started

end_time	Date time the task ended
status	it can be waiting, processing, failed, completed or cancelled (see 6.1)
estimate_end	An estimation of the date time the task processing will finish
dyn_params	The specific parameters for that task that are passed to the task command
Progress	A message that is updated every few seconds when the task is in processing status and contains information about the progress.
enabled	If false the task can' t start

[A.5. STEPS Table](#)

See 6.2

[A.6. STEPS_LOG Table](#)

See 5.6.4

[A.7. USERS Table](#)

See 6.4

B. Python Class reference

B.1. autoifgsrv.py

DESCRIPTION

This is the file that contains the class of the main service loop and starts the service and controls the running of the steps.

CLASSES

Looper

```
class Looper(__builtin__.object)
|   Main Service Loop
|
|   Methods defined here:
|
|   __init__(self)
|       Loads Environment
|   archive(self)
|       Moves outputs to an archive storage when there is limited space on production storage
|       or there are "old" outputs present in production storage
|
|   check_already_running(self)
|       Check if service already running
|
|   check_archived_status(self, archive_not_running)
|       Checks if the archived outputs are really moved from production storage
|
|   check_archiving_status(self, archivekeys)
|       Checks if the archiving has finished for an output
|
|   check_steps_status(self)
|       Checks steps metadata for abnormal states, restart failed steps, handles step kill or reset
commands
|
|   check_stop_status(self)
|       Check if stop command activated (if stop file trigger exists)
|
|   cleantreads(self)
|       Cleans stopped threads and checks for reset or cancel commands for a step.
|       Updates metadata database with steps logging and status information.
|
|   done_prereq(self, step)
|       Check if prerequisite steps of a step are executed
|
|   execute_step(self, step)
|       Execute a service step
|
|   get_freeSpace(self, drive)
|       Return the free space of a drive (Windows command)
```

```

|
| get_notif_data(self, step)
|     Get data to form Notification
|
| has_processing_conflict(self, step_id, output_id, steprec)
|     Checks a step before starting it for processing conflicts with steps running.
|     Avoids to run the same step twice and checks if a step accesses the same input files with
running steps
|
| inputinuse(self, input_id)
|     Checks if a running step uses a specific input file
|
| is_resource_available(self, resource, step_priority, terminate=True, allpriorities=True)
|     Check if a 'resource' is available in order to run a step
|     In case of priority 1 step it terminates all non priority 1 to free resources
|
| log_except_hook(self, *exc_info)
|     Log Unhandled exceptions to system log
|
| loopforever(self)
|     Run main Loop
|
| print_instructions(self)
|
| process_count(self, resource, priority=None)
|     Count all threads. If 'priority' if specified it counts threads of the specific priority
|
| step_runner(self)
|     Collects the steps that should run in the right order and starts a step if conditions are met
|
| stop_on_escape(self)
|
| stop_on_stopstatus(self)
|     Stop server if stop command activated
|
| store_process_time(self)
|     Store average process time to metadata database
|
| terminate_all(self, resource)
|     Terminates all threads or a specific resource
|
| terminate_all_non_p1(self, resource)
|     Terminates all non priority 1 threads
|
| terminate_service_threads(self, sid)
|     Terminates all threads or a specific service
|
| update_steps(self)
|     Updates metadata database with information collected from threads about running steps.
|     Steps progress and logging information are updated
|

```

```
| validate_history(self, step)
|     Check if a step with same parameters with the one specified by 'step' is running or is
completed
```

B.2. serviceevent.py

DESCRIPTION

This file contain the class that initiates the service and searches for the Interferogram pairs

The service requests are started based on the event information files that are located in the configuration folder.

There are of two possible formats: The automatied event detection system or the python-qgis interface.

The user in order to create a custom service request that does not come from either the detection system or the Qgis interface can manually create an event information file of either format.

CLASSES

servicerequest

```
class servicerequest
| It initiates the service, searches for the Interferogram pairs, creates service metadata
|
| Methods defined here:
|
| __init__(self, env)
|     initiates environment and class scope variables
|
| check_update_output(self, output)
|     Automatic update of output status
|
| check_update_service(self)
|     Automatically updates service request status
|
| clean_steps_output(self, output)
|     Clean steps in steps_execution table that are inactive or deleted in steps table
|
| closelog(self)
|     Closes the service request log
|
| create_output_notif(self, output, est_start, estim_start_inter, prev_est_end,
prev_est_end_inter)
|     Prepares notification information for new output
|
| estimate_ingestion_delay(self, master_id, method=1)
|     Estimates the delay of imagery availability on sentinel hubs after satellite passing using
different methods
|
| estimate_output_end(self, output, enabled)
|     estimates output end time
```

```

|
| estimate_output_start(self, output, prev_out_estim_end, prev_out_estim_end_inter)
|     estimates output start time
|
| estimate_step_end(self, est_start, prevstep, freshstep, step)
|     Estimates step's end time
|
| filter_masters(self, entries)
|     Filter master entries older than existing in same orbit and masters that have small
intersection with roi
|
| filter_slaves(self, entry, candidatepairs, preco)
|     Filter slave entries according intersection with master and ROI
|     and according sensing time if others exist on same orbit
|
| find_masters(self, ptracer)
|     Find 'master' imagery that intersects the AOI for a number of sentinel re-passing periods in
the past
|     Product Orbits found in more recent periods are filtered out
|
| find_outputs(self)
|     Searches, finds and stores outputs and steps for service in metadata
|
| find_probable_repassing(self, inp1)
|     Estimates and returns most probable re-passing time of the Sentinel-1 for specific product
|
| find_probable_repassing_event(self)
|     Estimates and returns most probable re-passing time of the Sentinel-1 for all products for the
specific service
|
| find_slaves(self, masterentry, ptracer)
|     Find 'slave' imagery for co-seismic and pre-seismic pairs.
|     Searches in the estimated repassing time range for a number of sentinel re-passing periods
after the master
|
| get_detected_AOI(self, wktpoint)
|     Creates a rectangle around the event point
|
| get_eventfolder(self, eventregion, eventdate)
|     Returns service request processing folder for IFGs
|
| get_eventname(self, eventregion, eventdate)
|     Returns the service request safe name for folder.
|
| get_eventparams(self)
|     Get parameters from qgis interface file
|
| get_orbits_sensing(self, dtype='masters')
|     Creates a list with newest or oldest sensing time per relative orbit
|     in masters, pre-seismic slaves or co-seismic slaves for the service output
|

```

```

| get_output(self, output_id)
|     Returns output properties of specific output and sets to the class the service properties of
this output
|
| get_output_config(self, output_id)
|     Returns configuration properties of specific output in a dictionary
|
| get_output_inputs(self, ifg_output)
|     Returns input files of specific output
|
| get_output_name(self, output_id)
|     Forms and returns output name for notification
|
| get_service_ids(self, status)
|     Returns a list with the service request ids of a specific status
|
| init_detected_event(self, eventfname, systemlog)
|     Initiates a service in 'detected' status based on automated event detection files
|
| init_service(self, systemlog)
|     Initiates service requests in case event information files exist in configuration location
|
| init_trigger(self, systemlog)
|     Initiates a service in 'requested' status from trigger file from python-qgis interface
|
| insert_event(self, eventregion, eventdate, status, rect_wkt2D, request_params, systemlog,
magnitude=None, epicenter=None, depth=None)
|     inserts event in service_request table
|
| masters_seeker(self, ptracer, period, period_from, period_to, orbitfilter, roi)
|     Seeks 'master' imagery in the AOI for a specific period in all specified sentinel hubs
|
| move_trigger(self, eventpath, eventfile, eventregion, eventdate, systemlog)
|     moves trigger file to processed location
|
| next_product_time(self, sid=None)
|     Estimate and return availability time of next satellite imagery product of service (not
available)
|
| openlog(self, eventfolder)
|     Creates or initiates the service request log
|
| reset_service(self, sid=None)
|     Reset service by stopping all running service steps and erasing all outputs and steps metadata
|     Files created on file system must be deleted manually
|
| sat_input_orbit_file(self, entry)
|     Store satellite imagery files information to satellite_input table
|
| set_priorities(self, sid=None)
|     Sets the processing priority of the for the outputs of the service request

```

```

|
| set_service(self, status, rs_id=None, initlog=True)
|     Sets the service attributes of the service event class reading them from the service_request
table
|
| slaves_seeker(self, master_entry, ptracer, orbitfilter, preco, searchfrom, searchto)
|     Seeks 'slave' imagery for a specific time period in all specified sentinel hubs
|
| sql_get_output(self)
|     Forms an SQL join to retrieve all properties needed for an output
|
| sql_get_repassing(self, inp1=None)
|     Forms SQL for find most probable re-passing time of the Sentinel-1
|
| step_params(self, output, step)
|     Forms and returns step's dynamic parameters for steps_execution table
|
| store_idl_config(self)
|     Updates IDL configuration file
|
| store_output(self, masterentry, pair=None)
|     Store output to service_output table
|
| store_output_config(self, output_id)
|     Store configuration file for IDL steps input (used for testing IDL code)
|
| store_sat_input(self, entry)
|     Store satellite imagery files information to satellite_input table
|
| store_steps(self, sid=None)
|     Stores steps in steps_execution table for all outputs of the event
|
| store_steps_output(self, output, est_start=datetime.datetime(2018, 6, 20, 13, 51, 23, 940000),
est_start_inter=datetime.datetime(2018, 6, 20, 13, 51, 23, 940000))
|     Store output steps in steps_execution table
|
| update_service(self, value, field='status')
|     Updates service request fields
|
| utc_to_local(self, utc_datetime)
|     Converts UTC to local time

```

B.3. [searchimages.py](#)

DESCRIPTION

Library for searching and download sentinel products with the use of metadata database

CLASSES

Downloader
 OrbitFileDownloader
 ProductDownloader
 ProductTracer
 ProductUnzipper
 dbProduct

```
class Downloader(__builtin__.object)
```

```
| Methods for file download
```

```
| Methods defined here:
```

```
| __init__(self, procstatus=None, env=None, log=None)
```

```
| check_faster_hub(self, bytes_read, product, url)
```

```
|     Check for faster download or stop download if too slow
```

```
| download_speed(self, speeds, response, *args, **kwargs)
```

```
|     Measure download speed
```

```
| downloader(self, dest, product, response, *args, **kwargs)
```

```
|     Downloads and stores a file in storage destination 'dest'
```

```
|     In case of slow download searches for better download sources
```

```
|     Information on download progress is send to parent processes
```

```
| find_best_host(self, product, exclude_hosts_url=[])
```

```
|     Find best hub according download speed
```

```
class OrbitFileDownloader(__builtin__.object)
```

```
| class to download orbit file
```

```
| Methods defined here:
```

```
| __init__(self, orbitpath, orbitfile, procstatus=None, env=None, log=None)
```

```
|     Initiates class and downloads orbit file
```

```
| orbit_download(self)
```

```
|     Downloads orbit file
```

```
| validate_orbitfile(self)
```

```
|     Validates orbit file
```

```
class ProductDownloader(__builtin__.object)
```

```
| Class contains methods for downloading a sentinel-1 product and update metadata information
```

```
| Methods defined here:
```

```
| __init__(self, value, procstatus=None, field='product_id', env=None, log=None)
```

```
|     Initiates class, retrieves product metadata information and downloads product
```

```

| decide_download(self, uri, filename, status, knownsize=0, knownchecksum="", product=None)
|     Decides to download a product from start, not download at all or continue download
|     based on existing files and metadata information
|
| file_as_blockiter(self, afile, blocksize=65536)
|     Iterates through a file with a certain 'blocksize'
|
| find_host_session(self)
|     Finds a working hub that contains the product
|
| hash_bytestr_iter(self, bytesiter, hasher, ashexstr=False)
|     calculates MD5 checksum
|
| product_download(self)
|     Downloads Quick look icon and full product
|
| set_host_session(self, hub, checksum, httpsize)
|     Prepare the class properties to download from a hub host that contains the product
|
| set_product_dest(self, product)
|     Sets the product destination
|
class ProductTracer(__builtin__.object)
|     Methods to search Copernicus hub for sentinel products
|
|     Methods defined here:
|
|     __init__(self, env, log=None)
|
|     checkhubs(self)
|
|     format_query(self, fromdate, todate, filters_st, roi, datesearch)
|         Form hub query
|
|     get_manifest(self, uri, name)
|         Retrieves the product manifest from hub
|
|     get_property(self, uri, _property, sess=None)
|         Retrieves a product property from hub
|
|     get_size_http(self, uri, sess=None)
|         Retrieves product size from hub
|
|     get_uri(self, host, product_id)
|         Forms and Returns a sentinel product uri for download
|
|     product_seeker(self, fromdate, todate, filters_st, roi="", datesearch='beginposition', start=0,
rows=100, hubs=None)
|         Seek products that meet hub query

```

```

|
|
class ProductUnzipper(__builtin__.object)
| class to unzip downloaded product
|
| Methods defined here:
|
| __init__(self, value, procstatus=None, field='product_id', env=None, log=None)
|     Inits and unzip downloaded product
|
|
class dbProduct(__builtin__.object)
| This class is used to retrieve and access the metadata of the sentinel product imagery
|
| Methods defined here:
|
| __init__(self, env)
|
| get_orbit_dest(self)
|     Form and return the orbit file destination on storage
|
| get_product(self, value, field='product_id')
|     Retrieve the product metadata
|
| get_product_dest(self)
|     Form and return the product files destination on storage
|
| update_product(self, value, field=None)
|     Update product metadata in satellite_input table
|

```

B.4. [serviceprocess.py](#)

DESCRIPTION

Classes to handle multithreading

CLASSES

```

__builtin__.object
osprocess
step_process

class osprocess(__builtin__.object)
| Handles OS processes run for steps
|
| Methods defined here:
|
| __init__(self)
|
| parseIDLscriptout(self, parsetype, out_err=None, parsefile=None)

```

```

|   Parse IDL scripts and ENVI SARscape output files for update status of completion and
progress
|
|   parseIDLscriptout_file(self, varargs)
|       Find ENVI SARscape output file for parsing progress and define time to consider process
frozen
|
|   parsing(self, parsetype, out_err)
|
|   parsing_file(self, varargs)
|
|   run_osprocess(self, procinfo, step, command, varargs=[], environ=None, waitsecs=10)
|       Start and monitor OS process
|
|   terminate_osprocess(self, os_process)
|       Terminate OS process
|
|
class step_process(__builtin__.object)
|   This class handles the execution of a service step either it is a python or OS or IDL process
|
|   Methods defined here:
|
|   __init__(self, env, output, step)
|
|   get_message(self)
|       Retrieve message from process
|
|   get_progress(self)
|       Retrieve progress from process
|
|   get_status(self)
|       Retrieve process status from process
|
|   get_step(self)
|       Retrieve step properties from steps_execution table
|
|   init_osprocess(self, command, varargs=[], environ=None)
|       Initiate class in case of OS process
|
|   init_pyprocess(self, func, varargs)
|       Initiate class in case of python process
|
|   is_alive(self)
|       Check if process running
|
|   terminate(self)
|       Terminate process
|

```

B.5. eventdetection.py

DESCRIPTION

Contains class with methods to retrieve quake events from usgs or emsc APIs

CLASSES

```
__builtin__.object
EventDetection
```

```
class EventDetection(__builtin__.object)
| Methods defined here:
|
| __init__(self, env, log)
|     Initiates environment and log
|
| get_emsc_uri(self)
|     query emsc event API
|
|     uri example:
|     http://www.seismicportal.eu/fdsnws/event/1/query?starttime=2018-03-
18&format=json&minmagnitude=5.5
|
| get_last_quakes(self, uri)
|     Queries uri and returns last quakes
|
| get_last_quakes_emsc(self)
|     Returns last quakes from emsc
|
| get_last_quakes_usgs(self)
|     Returns last quakes from usgs
|
| get_minmag(self)
|     Return minimum magnitude
|
| get_starttime(self)
|     Return start time for searching events
|
| get_usgs_uri(self)
|     query usgs event API
|
|     uri example:
|     https://earthquake.usgs.gov/fdsnws/event/1/query?format=geojson&starttime=2018-02-
21T12:19:58&minmagnitude=5
|
| store_last_quakes(self, quakes, source)
|     Store found quakes from source according conditions
|
| store_last_quakes_emsc(self, quakes)
|     Store quakes from emsc
|
```

```

| store_last_quakes_usgs(self, quakes)
|     Store quakes from usgs
|
| store_quake(self, detected_quake, source)
|     Store quake in json file
|

```

B.6. [geomtools.py](#)

DESCRIPTION

Geometry tools based on osgeo ogr,osr libraries

CLASSES

geomtools

```

class geomtools(__builtin__.object)
| Geometry tools based on osgeo ogr,osr libraries
|
| Static methods defined here:
|
| convert_geom(WKT, inputEPSG, outputEPSG)
|     Convert coordinate system
|
| create_polygon(coords)
|     Create polygon from coordinates list
|
| getGreecePoly()
|     Polygon containing Greece Cyprus and part of Asia Minor
|
| pointGeom(point)
|     Return point geometry from point coordinates
|
| point_in_greecepoly(pt)
|     Check if point is within Greece polygon
|
| point_in_polygon(pt, poly)
|     Check if point is within a polygon
|
| rectangle_coords(p1, p2)
|     Create rectangle coordinates list from two diagonal corners points
|

```

B.7. [step_utils.py](#)

DESCRIPTION

Utilities to retrieve or update steps_log, steps_execution and service_output tables information

CLASSES

steputils

```

class steputils
| Utilities to retrieve or update steps_log, steps_execution and service_ouput tables information
|
| Methods defined here:
|
| __init__(self, env, systemlog=None)
|
| get_output(self, output_id=None, order=None, where_cl=None)
|     Retrieves information from an output based on query criteria
|
| get_output_sql(self, output_id=None, order=None, where_cl=None)
|     Form SQL to retrieve wide range of output information combining steps
|
| get_step(self, output_id, step_id)
|     Retrieves a step information dictionary by query in steps, steps_executionn and
service_output tables
|
| get_steps(self, query)
|     Retrieves a list of steps from steps_execution table
|
| iter_output(self, output_id=None, order=None, where_cl=None)
|     Creates an output iterator based on query criteria
|
| update_output(self, output, value, field='output_status', extra='')
|     Updates service_output table
|
| update_step(self, output, step, value, field='status', extra='')
|     Updates step information in steps_execution table
|
| update_step_log(self, output, step, status, message, connection=None)
|     Inserts a new entry in steps_log table

```

C. References

1. ENVI SARscape. [Online]
<http://www.harrisgeospatial.com/SoftwareTechnology/ENVISARscape.aspx>.
2. Python. [Online] <https://www.python.org/>.
3. Python 2.7 Installation. [Online] <https://www.python.org/download/releases/2.7/>.
4. GDAL - Geospatial Data Abstraction Library. [Online] <http://www.gdal.org/>.
5. Command line and environment. [Online] <https://docs.python.org/2/using/cmdline.html>.
6. Installing SARscape on Windows. [Online]
<http://www.harrisgeospatial.com/Support/SelfHelpTools/HelpArticles/HelpArticles-Detail/TabId/2718/ArtMID/10220/ArticleID/17296/Installing-SARscape-on-Windows.aspx>.
7. ENVI 5.4 / IDL 8.6 Quick Start Install and Licensing Guide. [Online]
<http://www.harrisgeospatial.com/Support/SelfHelpTools/HelpArticles/HelpArticles-Detail/TabId/2718/ArtMID/10220/ArticleID/15146/14988.aspx>.
8. Directory and Search Path Preferences. [Online]
http://www.harrisgeospatial.com/docs/prefs_directory.html.
9. postgresQL. [Online] <https://www.postgresql.org/>.
10. Postgres backup and restore. [Online]
<https://www.postgresql.org/docs/8.1/static/backup.html#BACKUP-DUMP-RESTORE>.
11. Copernicus Open Access Hub. [Online] <https://scihub.copernicus.eu/>.
12. Copernicus Collaborative Node (HNSDMS). [Online] <https://sentinels.space.noa.gr/>.
13. Introducing JSON. [Online] <http://www.json.org/>.
14. PostGIS. [Online] <http://postgis.net/>.