GeSS

Release 0.6.0

Cec M

Mar 26, 2023

CONTENTS

GeSS (Generate Siril Scripts) is a package built upon pySiril, a Python wrapper for Siril astronomical processing software.

Its main purpose is to batch-process astrophotography sessions, somewhat like using Siril scripts or SiriLic, with following features:

- gess can be used to generate on-the-fly new scripts by passing keywords or ticking a few boxes to modify your usual processing workflow,
- gess can name Siril sequences based on info contained in your images headers, so that you resulting stacks have meaningful names,
- gess can locate automatically masters from libraries based on the values in the FITS header of frames to be calibrated,
- gess can crawl all your folders from the night before and capture anything that needs to be calibrated, registered and stacked,
- gess can build all your master libraries if the clouds rolled in and you decided not to loose the night,
- gess can gather calibrated lights from multiple sessions, and then register/stack them.

Gess is based on fetching info from image headers and parsing them to recognize string patterns in your files and folders names. As such, it is primarily intended for users with a repeatable acquisition process. Most likely using an imaging software that nicely arrange frames in folders with a stable naming convention. Please also have a look at the section regarding *folders conventions*.

Many options/preferences on how you want to process your files can be specified through configuration files. A disclaimer here: No fancy GUI except for settings! GeSS is intended for users who have some kind of drive towards automation and command line. It you do not recognize yourself in these words, then probably GeSS will be more of a burden than a help.

Knowing Python is not necessary, though it can prove useful if you want to re-use some modules of the package to build your own scripts. After installing the package, just type *gess* in a shell to get some help on how to use it (or much better, keep on reading these pages just a little bit longer).

CHAPTER

ONE

WHICH MODULE SHOULD YOU USE?

- Single sessions
  - If you want to process a single imaging session, you should have a look at using GeSS in *interactive* or *command-line* modes. Probably a good idea to follow the steps shown in the *Jump Start* section if this is your first use.
  - After you have processed a few sessions on the same target, you will want to gather and process them with the *multisession* mode.
- Batch processing
  - If you need to batch process multiple sessions from the same night, with multiple targets, filters or build master libraries, you need to head to *loop mode* section.
  - Once multiple sessions are batch-processed, you can reorganize the calibrated lights using *copyback* utility.

1.1 Before you start

1.1.1 Prerequisites

Before installing GeSS, you will need to install:

- a recent version of **Python**, 3.6 or above. Please note that versions 2.x are not supported.
- the latest version of **Siril** (1.0.0) or above.

> **Warning:** **For Windows users**: Python and its scripts folder need to be in your PATH to be able to use GeSS. Remember to enable this when installing Python for Windows (*Add Python to your PATH*). And please please, install Python for all users when prompted to do so, to avoid any installation in obscure hidden folders only Windows would know of.

The following Python packages will be installed/updated if not already present:

- **pySiril** (>=0.0.12)
- **pySimpleGUI**
- **astropy**
- **pyexiv2**

Warning: **For Windows users**: enable symbolic links by switching Windows to developper mode: head to Windows 10 Settings > Update & Security > For Developers and select "Developer mode".

1.1.2 Installation

If you want the latest release version:

- download the wheel from release page
- in a shell, type:

```
pip install gess-x.y.z-py3-none-any.whl
```

x.y.z being the version number of the release.

If you want to build from the sources using pip (requires setuptools and wheel packages):

```
pip install https://gitlab.com/free-astro/gess/-/archive/maindev/gess-master.tar.gz  ␣
↪ # stable version
pip install https://gitlab.com/free-astro/gess/-/archive/maindev/gess-maindev.tar.gz  ␣
↪ # dev version
```

1.1.3 Important: Folders and files

GeSS is built around the same philosophy as Siril standard scripts. Consequently, it follows the same convention. An imaging session consists of a working directory (as you would set the home directory in Siril), containing at least a subfolder with lights. It can also contain other subfolders with darks, flats and/or biases. The name you choose for all these subfolders is up to you (or the imaging software you use) and can be configured through options. But this filing architecture is mandatory for GeSS to work.

Example with NINA as imaging software:

A valid filename specifier could be
$$TARGETNAME$$\$$DATEMINUS12$$**$$IMAGETYPE$$**\$$IMAGETYPE$$_$$FRAMENR$$
returning a file tree like this:

```
MyAstroPics
└── M31
    ├── 2021-01-01
    │   ├── FLAT
    │   │   ├── FLAT_0001.FITS
    │   │   └── FLAT_0002.FITS
    │   └── LIGHT
    │       ├── LIGHT_0001.FITS
    │       └── LIGHT_0002.FITS
    └── 2021-01-02
```

The **LIGHT** folder containing the light frames is a subfolder to MyAstroPics\M31\2021-01-01 which is the working directory.

On the contrary, using this string specifier produces a non-valid file tree:
$$TARGETNAME$$**$$IMAGETYPE$$**\$$DATEMINUS12$$\$$IMAGETYPE$$_$$FRAMENR$$

```
MyAstroPics
└── M31
    ├── FLAT
    │   ├── 2021-01-01
    │   │   ├── FLAT_0001.FITS
    │   │   └── FLAT_0002.FITS
    │   └── 2021-01-02
    └── LIGHT
        ├── 2021-01-01
        │   ├── LIGHT_0001.FITS
        │   └── LIGHT_0002.FITS
        └── 2021-01-02
```

The **LIGHT** folder does not contain the light frames. There are located further down the tree inside the 2021-01-01 subfolder.

If you are already used to working with Siril, this should not be any news to you. And if you are new to Siril and want to unleash the power of batch processing with GeSS, then, you will need to comply with this convention…

1.2 Jump Start

1.3 First Steps

1.3.1 Processing worflow of an imaging session

The most important module of GeSS is called *gess.gessengine*.

It goes through the whole processing workflow of an imaging session:

1. Locate the master frames to be applied to calibrate the light frames, whether there are in a library or need to be processed first.
2. Start Siril and apply some user preferences (bitdepth, compression)
3. Prepare the masterframes if required or copy the masters from your masters libraries.
4. Preprocess the light frames (and possibly extract the Ha/OIII layers - OSC only).
5. Extract background.
6. Register.
7. Stack.

This workflow is the complete set of actions that gessengine can run. However, you don't need to go through all the steps. What actions are to be taken is fully configurable through a bunch of options that are passed to the module. Say, if you want to stop just after lights are calibrated, just set the options accordingly and gessengine will stop there.

gessengine is optimized to avoid repeating steps, should you want to apply different processings to the same session. For instance, it will store masters already stacked in *masters* subfolder and re-use them (see also the section about *storage folders*). It will also try to avoid conversion and preprocessing steps if they have already been performed previously.

Warning: If you are trying different versions of masters for some testing of your liking, don't forget to delete *masters* and *processxx* folders before trying to process again your session!

Note: Important: Before trying to run GeSS in any kind of way, you need to have set *gess default options*.

1.3.2 Interactive mode

The simplest way is to process a single imaging session is to start gessengine in interactive mode:

From a shell:

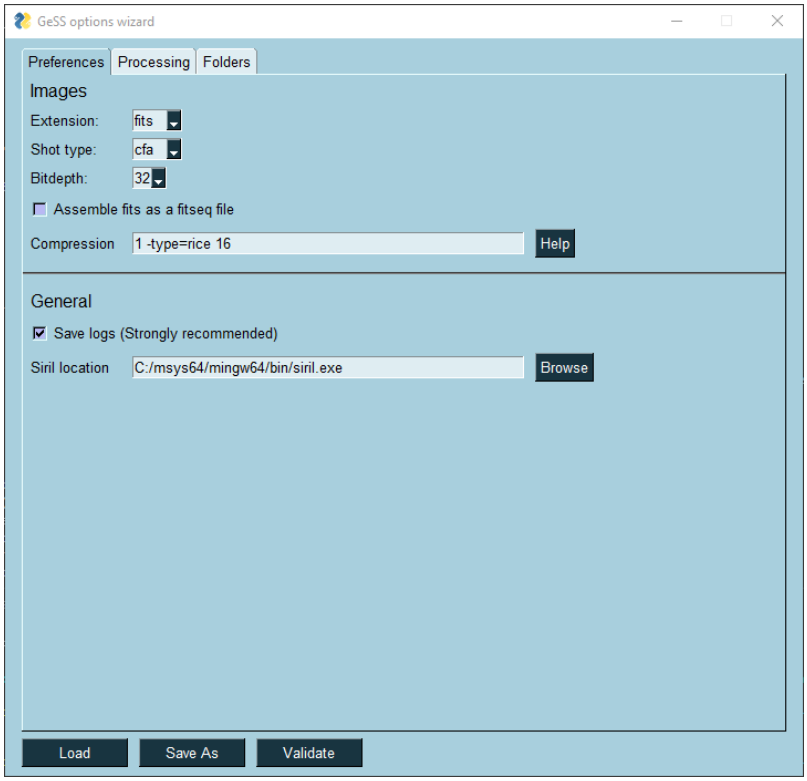
```
gess -i
```

From Python:

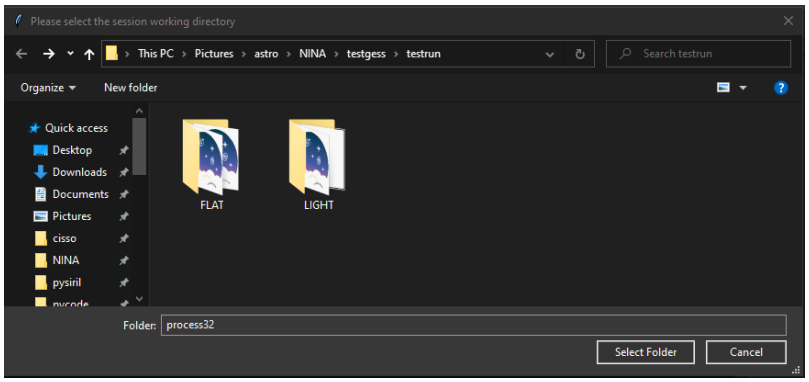
```
from gess import gessi
success,res=gessi.Run()
```

This will:

- start a GUI to review the default options and to modify some of them if you wish. Press Validate when done.



- ask for the working directory,



- run the workflow.

1.3.3 Command line mode

If you prefer a more command line approach, you can call gessengine directly.

From a shell:

```
gess -e /MyAstroPics/M31/2021-01-01
gess -e /MyAstroPics/M31/2021-01-01 gess_altopt.cfg
gess -e /MyAstroPics/M31/2021-01-01 doHO=1 dostack=0
gess -e /MyAstroPics/M31/2021-01-01 gess_altopt.cfg doHO=1 dostack=0
```

From Python:

```
from gess import gessengine
from gess.common.options import options

#success,res=gessengine.Run(workdir,opt=None,app=None,dryrun=False)

success,res=gessengine.Run('/MyAstroPics/M31/2021-01-01')

opt=options(optiontype='gess',addcfgfile='gess_altopt.cfg').getoptions()
success,res=gessengine.Run('/MyAstroPics/M31/2021-01-01',opt)

myopts={
    doHO = True,
    dostack = False
}
opt=options(optiontype='gess',dictcfg=myopts).getoptions()
success,res=gessengine.Run('/MyAstroPics/M31/2021-01-01',opt)

opt=options(optiontype='gess',addcfgfile='gess_altopt.cfg',dictcfg=myopts).getoptions()
success,res=gessengine.Run('/MyAstroPics/M31/2021-01-01',opt)
```

- **gess -e /MyAstroPics/M31/2021-01-01** runs gessengine in the folder /MyAstroPics/M31/2021-01-01. The workflow is executed as per your default options.
- **gess -e /MyAstroPics/M31/2021-01-01 gess_altopt.cfg** does the same but updates the default options with the values specified in *gess_altopt.cfg*. In the Python call, opt is a *DictX* instance containing additional options. Such object can be easily built using the *options* class.

This additional option file specifies some (or all of the) values to be updated. This can be handy if you want, for instance, to modify the names of some folders. The example below shows what *gess_altopt.cfg* could modify if your default options are set for imaging with NINA but you also shoot with APT. APT does have the same names as NINA for light/flat/dark/bias folders, nor uses the same FITS extension.

```
{
    "ext": "fit",
    "lights": "LIGHTS",
    "flats": "FLATS",
    "darks": "DARKS",
    "biases": "BIASES",
}
```

Other examples that come to mind are:

- specifying the location of master libraries if you use another camera. Default options could specify paths for camera#1 and *gess_cam2.cfg* updates for camera#2,
- specifying the location of flats library if you use another telescope,
- specifying Ha/OIII extraction for sessions where you've used a dual narrowband filter, though this is easier done with the 3rd syntax shown right below.

- **gess -e /MyAstroPics/M31/2021-01-01 doHO=1 dostack=0** runs gessengine with updated options specified by an '=' sign. You can specify as many as you want though it is probably useful if you only change just a few numerical values. Otherwise, the syntax shown above with an addtional file is probably more practical.
- **gess -e /MyAstroPics/M31/2021-01-01 gess_altopt.cfg doHO=1 dostack=0** does the same, but updates first with the additional file *gess_altopt.cfg* then with the values specified with an '=' sign.

1.3.4 Multisession mode

Multisession mode is a "manual" mode for gathering preprocessed lights from multiple sessions. "Manual" means that the user need to select one by one the folders to be merged. This can be useful is you have only a couple of sessions on a target and not that many filters. If you want to process many sessions with many filters, you should probably turn to *Copyback mode*.

To run multisession, from a shell:

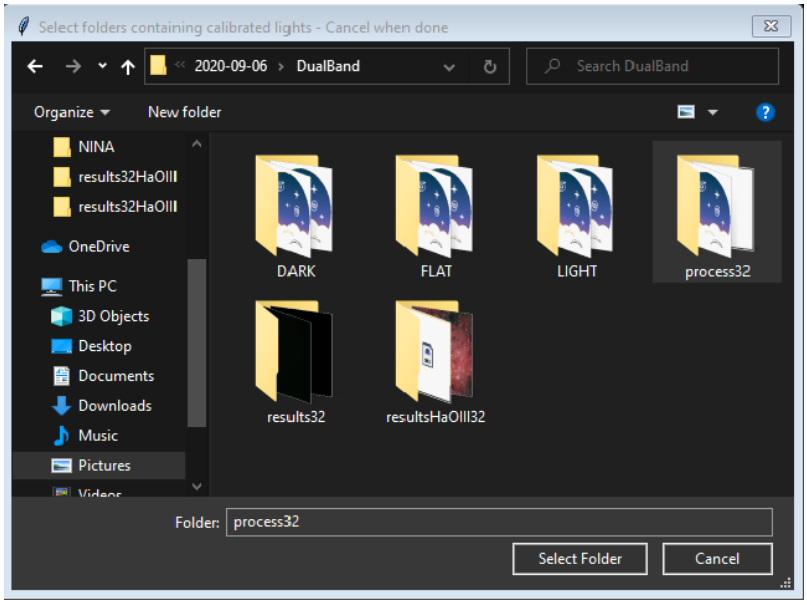
```
gess -m
```

From Python:

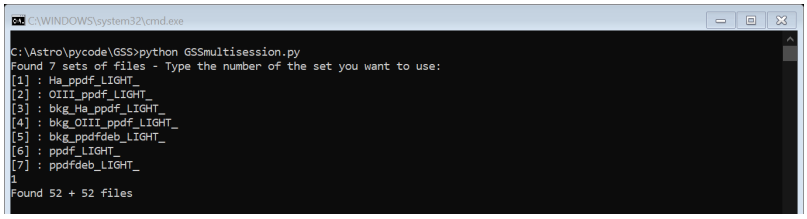
```
from gess import gessmultisession

success,res=gessmultisession.Run()
```

The steps are as follows:

- Select a folder containing calibrated lights (if you have calibrated them using GeSS, the folder should be named process16 or process32)

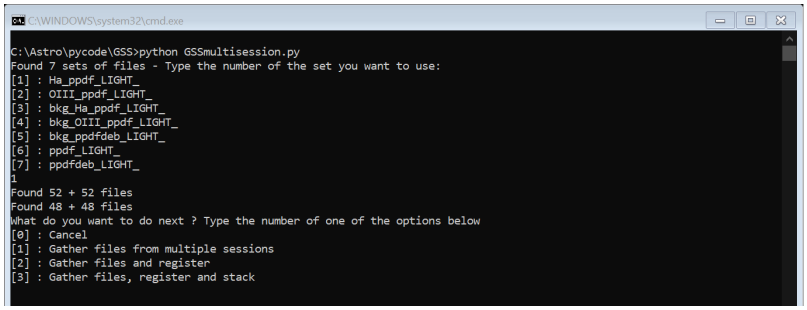


- Select the sequence that you want to use by entering its number:



In the example above, 52+52 files have been found because an Ha sequence from an Ha/OIII extraction was selected. Multisession will automatically collect the OIII sequence as well, hence why there are 52+52 files.

- Indicate what you want to do with these images:



Multisession will then ask for a directory where to store the newly created sequence (cannot be one of the folders you have already selected). It will proceed with copying the files and re-numbering them as required, and finally applying the processing specified.

1.4 Advanced Use

This section presents two more advanced modes. It is recommended to understand how gessengine works from the First Steps *section* before using them.

1.4.1 Loop mode

Loop mode is intended for more advanced users. You will probably need to have processed sucessfully a few individual sessions before jumping to this usage.

Note: Important: Before trying to run GeSS in loop mode, you need to have set *loop default options*.

Loop crawls your imaging folder and process all the FOLDERS (not files) with a specified string in their path. By default, it will search for the date of the day before but you can also specify any string to search for (a previous date, an object name etc…).

Loop will do the following:

- First identify all the folders containing calibration frames (darks, biases, flats) and light frames,
- Prepare masterframes if required in that order: biases, flats, darks
- Store them in your libraries (if you are working with libraries for some or all masters),
- Process all the light frames folders.

With this mode, you can therefore assemble all your masters libraries, process multiple sessions with multiple targets, filters, exposures etc… with one single line of command.

In NINA (and most probably in other imaging software), you can specify this as the script to be executed at the end of the session. You wake up the next morning to a handfull of processed images to review with your morning cup of coffee.

To run loop, from a shell:

```
gess -l
gess -l gess_setup2.cfg
gess -l gess_setup2.cfg loop_setup2.cfg
gess -l searchstr=M31
```

From Python:

```
from gess import gessloop

# success,res=gessloop.Run(addgesscfg='',addloopcfg='',searchstr='')

success,res=gessloop.Run()
success,res=gessloop.Run('gess_setup2.cfg')
success,res=gessloop.Run('gess_setup2.cfg','loop_setup2.cfg')
success,res=gessloop.Run(searchstr='M31')
```

- **gess -l** runs gessloop using default options from gess.cfg and loop.cfg, in the folders containing the date of the day before.
- **gess -l gess_setup2.cfg** runs gessloop using default options from gess.cfg with addtional options from *gess_setup2.cfg*.

- **gess -l gess_setup2.cfg loop_setup2.cfg** runs gessloop using default options from gess.cfg with addtional options from *gess_setup2.cfg* and loop.cfg updated from *loop_setup2.cfg*.
- **gess -l searchstr=M31** runs gessloop using default options from gess.cfg and loop.cfg, in the folders containing the string *M31* (so not necessarily a date).

Loop is set to go on through the different folders it should process and run till the end. If, along the way, some folder processing fails, it will show in the summary and go on with processing the next folder. So please carefully review the summary shown at the end of execution to make sure everything went as expected.

1.4.2 Copyback mode

Note: Important: Before trying to run copyback mode, you need to have set *copyback default options*.

This mode can be used to gather calibrated lights from multiple sessions in an automated way. Once you are done imaging a target, all the calibrated lights are ready for final registration and stacking.

It is intended for users imaging sessions on the same targets overmultiple nights, with mutliple filters ans most importantly with a repeatable acquisition process. If you want to gather sessions occasionally, you are better off using *Multisession mode*.

To run copyback, from a shell:

```
gess -c
gess -c gess_setup2.cfg
gess -c gess_setup2.cfg loop_setup2.cfg 2021-01-02 copyback_setup2.cfg
gess -c searchstr=M31
```

From Python:

```
from gess import gesscopyback

# success,res=copyback.Run(addgesscfg='',addloopcfg='',searchstr='',addcopybackcfg='')

success,res=gesscopyback.Run()
success,res=gesscopyback.Run('gess_setup2.cfg')
success,res=gesscopyback.Run('gess_setup2.cfg','loop_setup2.cfg','2021-01-02','copyback_
↪setup2.cfg')
success,res=gesscopyback.Run(searchstr='M31')
```

- **gess -c** runs copyback using default options from gess.cfg, loop.cfg and copyback.cfg, in the folders containing the date of the day before.
- **gess -c gess_setup2.cfg** runs copyback using default options from gess.cfg with addtional options from *gess_setup2.cfg*.
- **gess -c gess_setup2.cfg loop_setup2.cfg 2021-01-02 copyback_setup2.cfg** runs copyback using default options from gess.cfg, loop.cfg and copyback.cfg, complemented with options in *gess_setup2.cfg*, *loop_setup2.cfg* and *copyback_setup2.cfg*. It searches the folders containing string "2021-01-02".
- **gess -c searchstr=M31** runs copyback using default options from gess.cfg, loop.cfg and copyback.cfg, in the folders containing the string *M31* (so not necessarily a date).

Note: Before copying a calibrated light in the destination folder, copyback will check if a file with the same *DATE-OBS* FITS header key is present to avoid copying twice (or more) a file. It will also deal with incrementing the sequential

number to avoid copying over an existing file.

1.5 Setting Options

At the very beginning, GeSS was a pure command line program. Over the time, efforts have been made to make it a bit more user-friendly, though it is not intended to build a GUI to operate it. Since the point is to automate processing a lot to run it mostly unattended, it would be stupid to ask the user to click on buttons all along the process.

That being said, GeSS works with options files, three of them for now. Setting these options can be a bit tedious, in particular since it is going to be the first thing you will need to do before running anything. In order to ease the process, small GUIs have been developped to assist you.

This section is an attempt at explaining in the clearest possible way what these options files need to specify and how.

So sit back, relax and keep on reading…

1.5.1 GeSS options files

There are three default options files that are mandatory:

- **gess.cfg**, which specifies preferences for your usual image processing *worflow*. This is required for processing a session in *Interactive mode* or *Command line mode*, as well as gathering sessions with *Multisession mode*.
- **loop.cfg**, passing additional options required by *Loop mode*.
- **copyback.cfg**, passing additional options required by *Copyback mode*.

A clean version of these files is created the first time the package is loaded. The location where they are stored depends on your OS:

- Windows: %USERPROFILE%\gess (copy this in Windows Search bar)
- Linux and Mac OS: /usr/home/gess

These files are written in JSON format, a human-readable format, editable in any text editor.

If you do not feel comfortable editing these files manually, you can open a graphical editor for each of these, by typing one of the commands below:

```
gess -o         # edit gess.cfg
gess -ol        # edit loop.cfg
gess -oc        # edit copyback.cfg
```

These are the three default files. But you can write as many as you want and store them in the same location.

The default files are supposed to reflect your typical worflow and preferences, the ones you would like to execute most times. But if you are using multiple setups or multiple imaging software, you could create as many as required and call them to process your different sessions. All the modules that use the default cfg files also accept additional cfg files. Additional cfg can specify only a few values that differ from your default cfg or the whole set of options.

1.5.2 Setting gess.cfg

This is the most important option file, as it defines your preferred processing worflow. This section lists all its keys and values.

If you've chosen to edit via the GUI, you will need to type:

```
gess -o        # edit gess.cfg
```

You will see that the different parameters are organized in 3 tabs:

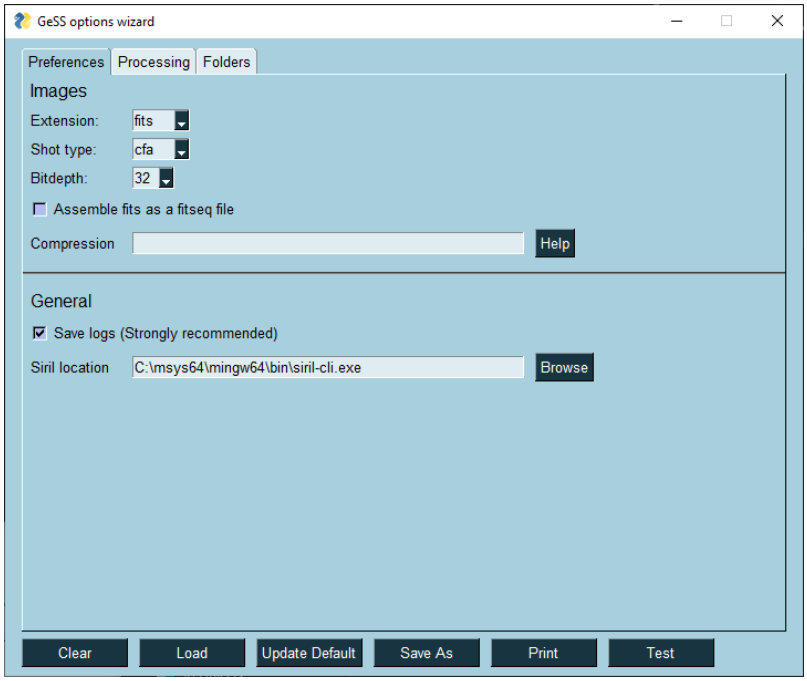
- Preferences
- Processing
- Folders

as shown below.

Hovering above any of the fields will display tooltips.

Preferences

The Preferences tab gathers all the options regarding general preferences.



The values for each key is detailed in this table, in the order shown in the image above.

Key	Type	Value	Comment
ext	*str*	fit/[fits]/fts/raw	The extension of your FITS file as saved by your imaging software. Special case for raw, see below.
shottype	*str*	cfa/mono	The type of images, either cfa (color) or mono.
bitdepth	*int*	16/32	The bitdepth to use in Siril, either 16b integer or 32b float.
seqasfitseq	*bool*	true/false	If true, the fits are converted to a single multi-image container, known as fitseq.
compression	*str*		Leave blank to use current settings from Siril. Otherwise, this should be the string passed after setcompress command.
debug	*bool*	true/false	If true, each module will save a *.log file keeping trace of Siril execution and module messages.
sirilexe	*str*		Leave blank to use Siril installed version. Specify a path to use a development version instead.

Note:

Setting **ext** key to *raw* can be used for handling one special case: in case your imaging software also saves *.jpg or other compressed format along with the raws for faster preview. In that case you must specify *raw* so that gess sends a convertraw command instead of convert. Only raw files are selected for conversion.

When the raws have been converted, the FITS files will used a *.*fit* extension (no choice here).
Apart from this case, you can use one of the three FITS extensions even if you are shooting with a DSLR. The universal convert command will handle this perfectly.

Processing

The Processing tab sets all the options regarding how you want to preprocess your images and which processing steps you want to execute after calibration.

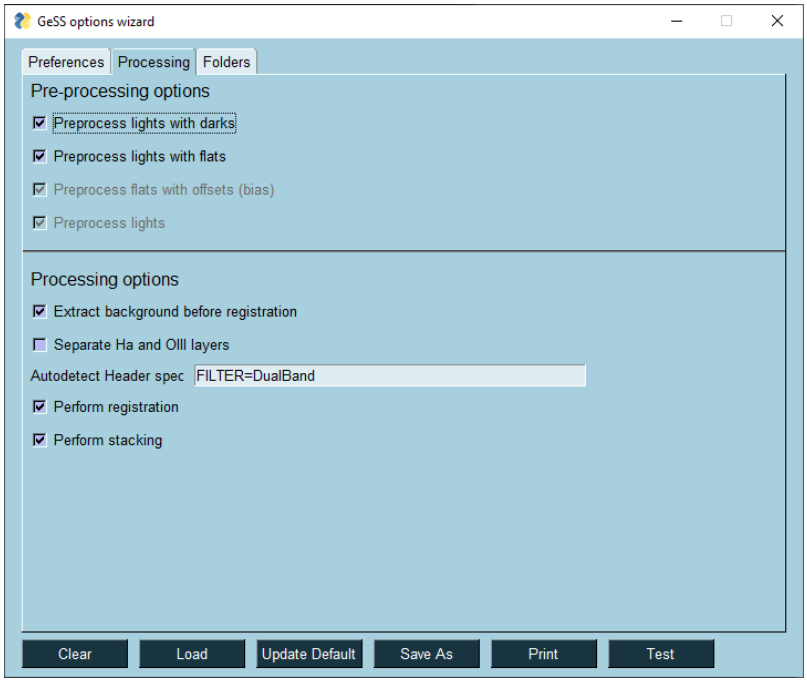


Table below gives the values and types for the different keys, in the order shown in the image above. Further explanation is given right after.

Key	Type	Value	Comment
ppdark	*bool*	true/false	If true, preprocess lights with darks
ppflat	*bool*	true/false	If true, preprocess lights with flats
ppoffset	*bool*	true/false	If true, preprocess flats with offsets
pplight	*bool*	true/false	If true, preprocess lights
dobkg	*bool*	true/false	If true, perform background extraction on preprocessed lights
doHO	*bool*	true/false	If true, perform Ha/OIII extraction before registration
autoHOspec	*str*		A specifier string, such as FILTER=DualBand to automatically trigger HO extraction
doregister	*bool*	true/false	If true, perform registration
dostack	*bool*	true/false	If true, perform stacking

- **ppdark**, if true, will instruct gessengine to do mainly two things:
  - if you shoot darks along with each imaging session, it will stack them into a masterdark. This step is skipped if you use a darks library.
  - it instructs gessengine to preprocess the lights with the masterdark.
- Same applies to **ppflat** and **ppoffset**. Before stacking the flats, they will be calibrated by the masteroffset. As substracting a masteroffset from flats is mandatory in the processing workflow, the GUI does not leave a choice whether to set **ppoffset** to true or false. The value is determined from **ppflat** and is exposed for information only.
- **pplight** is always set to true via the GUI.
- **dobkg**, if true, will instruct gessengine to perform background extraction with a polynom of degree 1 after lights calibration.
- **doHO**, if true, will instruct gessengine to perform Ha/OIII extraction. After this operation, two sequences are passed to the next step.
- **autoHOspec** is a string specifier that will instruct gessengine to perform Ha/OIII extraction if the condition is true. The specifier is formed as KEY=VALUE, where KEY is a valid FITS header key and VALUE is the value that will trigger the extraction. The most obvious example would be *FILTER=DualBand*, meaning that if the lights header contains the keyword FILTER and that it is set to *DualBand* then, the condition becomes true and the Ha/OIII extraction is triggered. Warning: This specifier overides the **doHO** value defined above. Leave empty to conform to the doHO switch.
- **doregister**, if true, will instruct gessengine to perform global registration.
- **dostack**, if true, will instruct gessengine to perform stacking. The parameters for this operation are the same as in the standard Siril scripts. After stacking, if **doHO** is activated, the Ha and OIII stacked layers are linear matched.

Folders

The Folders tab sets all the options regarding your filing system and conventions.

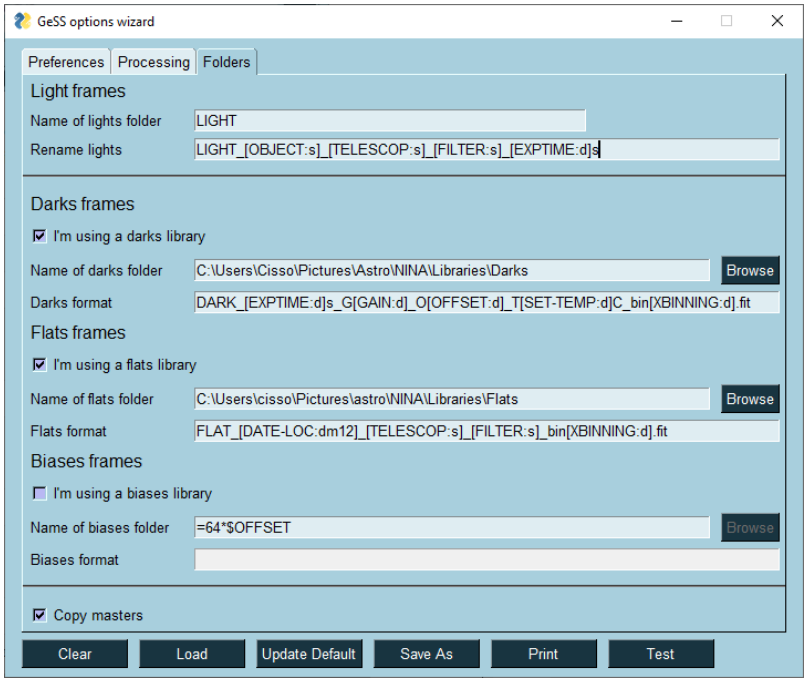


Table below gives the values and types for the different keys, in the order shown in the image above. Further explanation is given right after.

Key	Type	Value	Comment
lights	*str*	subfolder	Name of subfolder containing light frames, e.g. light or lights or raw etc....
lightsfmt	*str*		Optional name for the converted sequence
darks	*str*	subfolder or path	Name of folder containing dark frames: - either name of subfolder if you shoot darks with every session - or full path to your masterdarks library
darksfmt	*str*		Name convention for masterdarks (for library only)
biases	*str*	subfolder or path	Name of folder containing bias frames: - either name of subfolder if you shoot biases with every session - or an algebraic expression to use synthetic offsets - or full path to your masterbiases library
biasesfmt	*str*		Name convention for masterbiases (for library only)
copymasters	*bool*	true/[false]	If true, hardcopy the masters in the masters subfolder

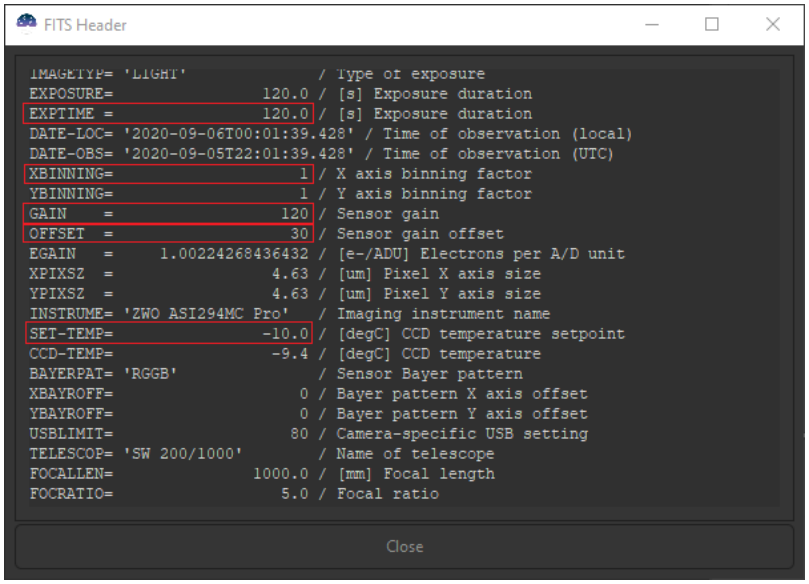
- **lights** is pretty much self-explanatory. You should give here the name convention imposed by your imaging software for the folder where the lights are stored, whether it is *light* or *lights* or *raw* etc...
- **lightsfmt** is the name that you want to give to the lights sequence (without extension). If left blank, the lights will be converted to a sequence named after **lights** field. For instance, LIGHT_00001.fits, LIGHT_00002.fits etc...
  Now you could want to use some info contained in your lights header to parse a more meaningful name. In this case, you can define here a format specifier. The full details on naming conventions are given in *darksfmt*.

  Example:
  A specifier like *LIGHT_[OBJECT:s]_[DATE-OBS:dm12]_[TELESCOP:s]_[FILTER:s]_[EXPTIME:d]s* would result in your lights being converted to:
  LIGHT_M31_2021-02-01_NEWT200F5_DualBand_300s_00001.fits etc
  As this initial sequence name is then passed on to all the following sequences, the resulting stacked image will

also have this string embedded.

Warning: It can be handy to tag your sessions as shown above with a date field ([DATE-OBS:dm12]) if you are shooting a target over one single night. But beware, if you are imaging over multiple nights, you will end up with calibrated frames over different nights belonging to sequences with different names. This would prevent *multisession* or *copyback* modes to properly gather shots from different nights.

- For **darks** and all the other calibration frames, the same philosophy applies:
  - If you shoot them with every session, **darks** is a string giving the subfolder name, say "darks", "dark", etc…as defined by your imaging software.
  - If you want to use a darks library, tick the *I'm using a darks library* box. This will enable the *Browse* button. Click on it and navigate to the folder containing your masters, confirm to save the location.
    If you opt-in for the library method, you will need to define as well the **darksfmt** field, see below.
- For **darksfmt** and all other *fmt* fields, you need to specify the name convention to identify the masterdarks adequate to process your lights. In the figure above, you can see for example:
  DARK_[EXPTIME:d]s_G[GAIN:d]_O[OFFSET:d]_T[SET-TEMP:d]C_bin[XBINNING:d].fit
  - all the terms between brackets are formed as follows: [*KEY*:*fmt*]
  - *KEY* is any (valid) key from the lights FITS header
  - *fmt* is a format specifier, such as *d* for integer, *f* for float or *s* for string.
    For instance, [EXPTIME:d] will be parsed to *60* if the lights have been exposed for 60s. But would be parsed to *60.0* if you specify [EXPTIME:0.1f].
  - you can find all the keys of a FITS using Siril. Open a FITS file (a light for this example) and head to Menu->Image Information-> FITS Header. It should display a window like this:



    The framed values are the ones used in the *fmt* string: DARK_[EXPTIME:d]s_G[GAIN:d]_O[OFFSET:d]_T[SET-TEMP:d]C_bin[XBINNING:d].fit
    The corresponding masterdark name would be DARK_**60**s_G**120**_O**30**_T-**10**C_bin**1**.fit
  - Table below recalls which FITS headers are read to parse the name of which master:

Master	reads header from
darks	lights
flats	lights
biases	flats

  - In order to parse a date from a date-time header key, you can use the special non-standard formatter *dm12*, which means date minus 12h.
    For instance, if in a header, the key DATE-LOC has a value of '2021-01-01T00:01:01.000', [DATE-LOC:dm12] would convert to '2020-12-31', which was the date at the start of the night.
    You can also use special formatter *dm0* which will just parse the date, without substracting 12h.
  - In order to parse RA and DEC info from OBJCTRA and OBJCTDEC header keys, you can use the special non-standard formatter *ra* and *dec*.
    For instance, if in a header, the keys OBJCTRA and OBJCTDEC have a value of '02 34 30 and '+61 23 07' respectively, [OBJCTRA:ra]_[OBJCTDEC:dec] would convert to '02h34m30s_+61d23m07s'.
  - There is no such thing as a FITS header for RAW images from a DSLR. However, some fields are still extracted from exif data and converted to standard FITS header keys for you to tag your images:
    * INSTRUME: the name of the camera
    * ISOSPEED: the iso setting of the shot
    * EXPTIME: the exposure time in s
    * DATE-OBS: the date and time of the exposure

- **biases** are the only masters that accept a special string, *in lieu* of a path, to trigger the use of synthetic offsets. More details about the values and formats to be used can be found in this tutorial.
- During the processing of a session, gess will create a masters folder within the working directory. This provides a handy way to keep the masters once you are done processing a session. You can then simply delete the process folder altogether and keep your working directory tidy.
  copymasters is an option to make a hard copy, not a symlink, of each master coming from libraries in this masters folder.
  In case you want to process the same session later, all the masters are here, stored together the lights to be calibrated, with the same versions as when the session was shot.
  Note: For masters which are stacked during the processing of the session, so not from libraries, there is no choice, they are hard-copied in the masters folder.

General notes on masters:

- You do not need to settle for an all-library or all-subfolder approach. Each calibration frame type can have its own specification. The example above specifies libraries for darks and biases and subfolder for flats.
- All these values are read (or not) depending on the keys given for preprocessing (*ppdark*, *ppflat* and *ppoffset*). For instance, if *ppdark* = false, it does not matter if there is something in **darks** and **darksfmt**. These values are just ignored.
  On the contrary, if *ppdark* = true, then there must be something specified at least in **darks** and possibly in **darksfmt**.

For advanced users: using wildcards in mastersfmt

It could be that you want to use some key value in your masters name that do not match the key value in the frames to be calibrated. With an example, it may be a bit clearer:
Say, you want, in your masterflats names, to keep record of their exposure time. Something like:
FLAT_1.32s_Halpha_G120_O30.fit.
If you put a field [EXPTIME:0.2f] in **flatsfmt**, it will end up with an error. Because the EXPTIME key will be read from a light frame, not a flat…

There are 2 ways to deal with this situation:

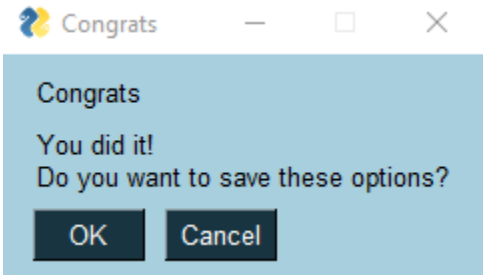
- If you do not plan on using *Loop mode* to build your masters libraries, then just replace the token to be ignored by a wildcard sign: *. In the flats example above, **flatsfmt** would become:
  FLAT_*s_F[FILTER:s]_G[GAIN:d]_O[OFFSET:d].fit
- If you plan on using *Loop mode* for building libraries, then you cannot just ignore this altogether, as loop will need this information to name your masters. You still need to make use of a wildcard character, but this time at the start of the token.
  In the flats example above, **flatsfmt** would become:
  FLAT_[*EXPTIME:0.2f]s_F[FILTER:s]_G[GAIN:d]_O[OFFSET:d].fit

GUI buttons

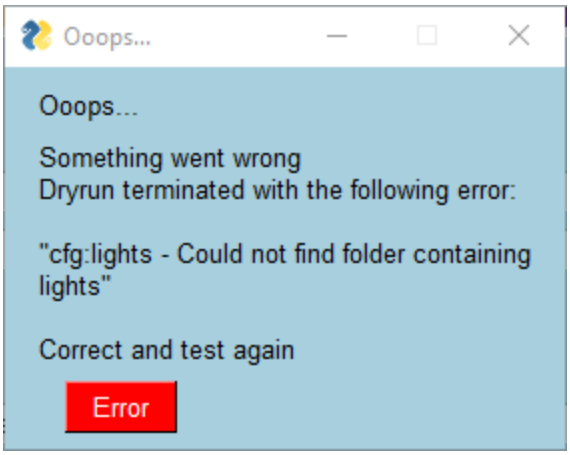
The buttons at the bottom of the interface have the following effects:

- Clear Clear all the entries for all the keys and restore defaults (mostly blank strings so beware! You should get a warning before everything is cleared if case you have not saved before).
- Load Load an existing cfg file. You can use this either to load gess.cfg (the default file) or to load another one to rework it.
- Update Default Save the current options as your new default. This will overwrite the current gess.cfg file. As this one is particularly important, you will get a warning just to confirm this is indeed what you intend to do.
- Save As Save the current options as another cfg file. This can be useful if you use different setups or want to test alternative options without modifying your default. It is recommended to name it gess*.cfg to recognise its type easily as there will be other types of option files in the same folder.
- Print Press print to display all the current values to the terminal. This is useful to find out the names of the different options (you will not use the GUI forever and better start learning their names…).
- Test Run a test with the options as currently set in the interface. This will call gessengine in *dryrun* mode. Meaning that only the folders checking and preferences passing is done. Gessengine will not go into all the processing of the images but if this test is succesful, it could mean that you are almost acertained that this set is valid.

If the test is validated, you will get a window proposing to save this set of options (so that you cannot forget). Save it as gess.cfg to make it the new default or to an alternate name like with the *Save As* button.



If the test fails, you will get a window telling you where it went bad. Try to fix this re-reading this section and test again.



gess.cfg JSON File

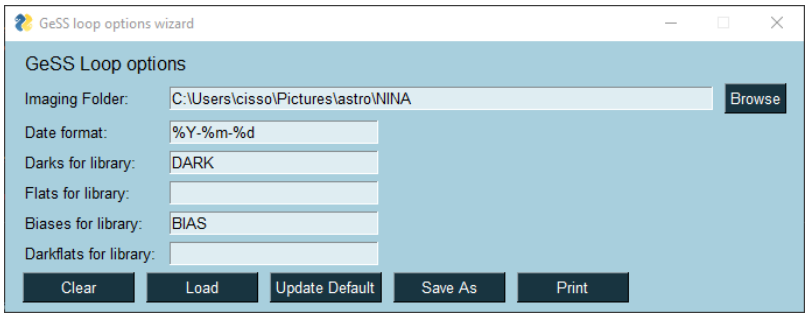
Once you've set these options through the graphical interface, you can always go to the *options location* and review gess.cfg. (You could also modify it from there…) The file created by the GUI with the values shown in the figures above will look like this:

```
{
  "ext": "fits",
  "shottype": "cfa",
  "bitdepth": 32,
  "seqasfitseq": false,
  "compression": "",
  "debug": true,
  "sirilexe": "C:\\msys64\\mingw64\\bin\\siril-cli.exe",
  "ppdark": true,
  "ppflat": true,
  "ppoffset": true,
  "pplight": true,
  "dobkg": true,
  "doHO": false,
  "autoHOspec": "FILTER=DualBand",
  "doregister": true,
  "dostack": true,
  "lightsfmt": "LIGHT_[OBJECT:s]_[TELESCOP:s]_[FILTER:s]_[EXPTIME:d]s",
  "lights": "LIGHT",
  "darks": "C:\\Users\\Cisso\\Pictures\\Astro\\NINA\\Libraries\\Darks",
  "darksfmt": "DARK_[EXPTIME:d]s_G[GAIN:d]_O[OFFSET:d]_T[SET-TEMP:d]C_bin[XBINNING:d].fit
↪",
  "flats": "C:\\Users\\cisso\\Pictures\\astro\\NINA\\Libraries\\Flats",
  "flatsfmt": "FLAT_[DATE-LOC:dm12]_[TELESCOP:s]_[FILTER:s]_bin[XBINNING:d].fit",
  "biases": "=64*$OFFSET",
  "biasesfmt": "",
  "copymasters": true
}
```

1.5.3 Setting loop.cfg

Setting this option file is required to use GeSS *Loop mode*. This section lists all its keys and values.
If you want to edit via the GUI, you can launch it with:

```
gess -ol        # edit loop.cfg
```



Hovering above any of the fields will display tooltips.

Table below gives the values and types for the different keys, in the order shown in the image above. Further explanation is given right after.

Key	Type	Value	Comment
imagingfolder	*str*		the path where all your images are stored. Usually the same as defined in your imaging software.
datefmt	*str*	%Y-%m-%d	a format specifier to parse the date to be searched for.
darks2lib	*str*		If blank, uses gess.cfg darks settings. If subfolder is passed, all subfolders containing this string will be used to build new masterdarks
flats2lib	*str*		If blank, uses gess.cfg flats settings. If subfolder is passed, all subfolders containing this string will be used to build new masterflats
biases2lib	*str*		If blank, uses gess.cfg biases settings. If subfolder is passed, all subfolders containing this string will be used to build new masterbiases

- **datefmt** is used to parse the string that will be searched for in **imagingfolder**. Check the configuration from your favorite imaging software.
  This is used if you do not pass an argument *searchstr=* to gessloop. The date is then taken as the current date minus 12h to find back the date at the start of last night session. And the date string is built according to **datefmt**.

  Example:

  Let's assume that you have the following keys in loop.cfg and your imaging folder tree looks like this:

```
{
  "imagingfolder": "C:\MyAstropics",
  "datefmt": "%Y-%m-%d"
}
```

```
MyAstroPics
├── CaliforniaNebula
│   └── 2021-01-03
├── RosetteNebula
│   └── 03-01-2021
```

(continues on next page)

(continued from previous page)

```
├── M31_2021-01-02
└── M31_2021-01-03
```

Note: Hopefully your imaging folder will never look like this mess!

Now say you launch gessloop (*gess -l*) on Jan 4th in the morning.
The following folders will be crawled, searching for frames:

- *MyAstroPics\CaliforniaNebula\2021-01-03*
- *MyAstroPics\M31_2021-01-03*

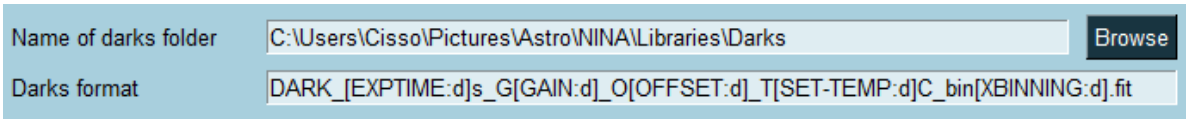
If you want to process *MyAstroPics\M31_2021-01-02* instead, you should use the command: *gess -l searchstr=2021-01-02*

Finally, if you want to process *MyAstroPics\RosetteNebula\03-01-2021*, you should change *datefmt* to "%d-%m-%Y".

- **darks2lib** and all other *2lib* keys are used to build automatically masters libraries. You can leave them blank if you do not want to use this feature, although it is recommended to set these fields correctly in case you change your mind later.
  How it works: if you have a path to a darks library specified in gess.cfg (*darks*), you need to specify here the name of the subfolder where individual darks are stored by your imaging software. While crawling the folders, gessloop will detect these subfolders, stack the darks to new masterdarks and copy them to your darks library.

  Example: with the following settings

  - in gess.cfg (3rd tab - Processing):



  - in loop.cfg:

  All the darks found in subfolders *DARK* will be stacked and saved to:
  C:\Users\cisso\Pictures\astro\NINA\Libraries\Darks.
  The masterdarks are named according to the convention given in *darksfmt*:
  DARK_[EXPTIME:d]s_G[GAIN:d]_O[OFFSET:d]_T[SET-TEMP:d]C_bin[XBINNING:d].fit

---

Note: If a version of the same masterdark (with same name) is already present in your library, the old version is saved in subfolder *./previous* with date and time appended. So that in case your new masterdark does not come out as expected, you still have a back-up of the previous version to copy back in your library.

- the keyword *lightsfmt* from gess.cfg file, if defined, is used to create sets of lights, in case you can have lights with different characteristics in the same folder (different exposures, different filters etc…).
  For instance, if you have set **lightsfmt** to
  "LIGHT_F[FILTER:s]_[EXPTIME:d]s_G[GAIN:d]_bin[XBINNING:d]"
  gessloop will find subsets of lights based on this naming convention and create as many folders as required to symlink copy the different kinds of lights before processing. This field is particularly useful to keep your processed files organized if you plan to use *Copyback* on sessions with multiple filters.

  Note: if you are working with masters libraries, the same applies to calibration files (darks, flats etc) without you specifying anything. So if you plan to store more than one set of calibration frames in the same folder when you

shoot them (i.e. shooting multiple darks with different exposures during the same night and storing them in the same folder), you should be using a library approach. Even if it is a temporary one.

loop.cfg JSON File

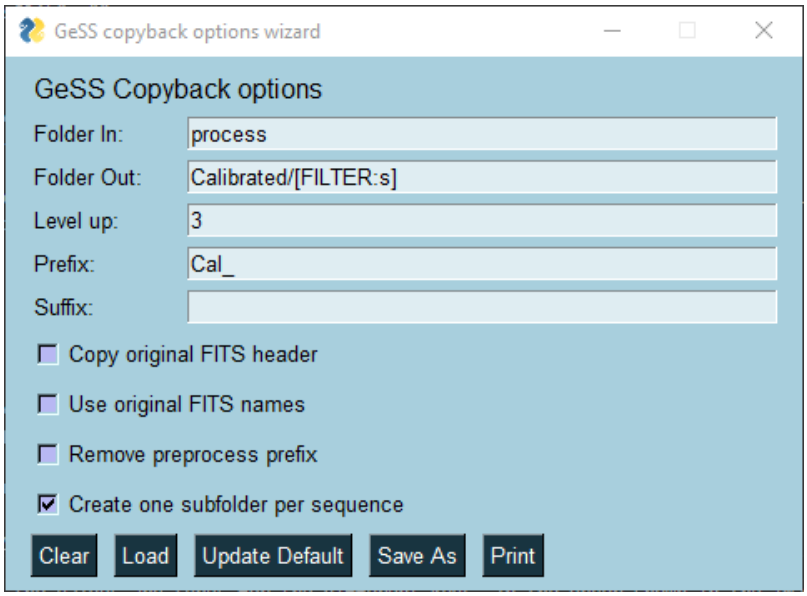
Once you've set these options through the graphical interface, you can always go to the *options location* and review loop.cfg. (You could also modify it from there…) The file created by the GUI with the values shown in the figures above will look like this:

```
{
    "imagingfolder": "C:\\Users\\cisso\\Pictures\\astro\\NINA",
    "datefmt": "%Y-%m-%d",
    "darks2lib": "DARK",
    "flats2lib": "",
    "biases2lib": "BIAS",
}
```

1.5.4 Setting copyback.cfg

Setting this option file is required to use GeSS *Copyback mode*. This section lists all its keys and values.
If you want to edit via the GUI, you can launch it with:

```
gess -oc        # edit copyback.cfg
```



Hovering above any of the fields will display tooltips.

Table below gives the values and types for the different keys, in the order shown in the image above. Further explanation is given right after.

Key	Type	Value	Comment
folderin	*str*	process	The string to identify the folders to be searched for. Leave the default value *process* if your sessions have bee processed with GeSS.
folderout	*str*		The string to build folders names. They can contain FITS headers keys to be parsed (see below)
levelup	*int*	>0	The number of levels up to store your files (see below)
prefix	*str*		Any prefix you would like to add to the calibrated lights names
suffix	*str*		Any suffix you would like to add to the calibrated lights names
copyheader	*bool*	false	If true, it will find the original uncalibrated light frame and copy its full header to the calibrated frame.
useoriginal-FITSname	*bool*	false	If true, it will replace the calibrated light frame name with that of the orginal uncalibrated frame.
removepp	*bool*	false	If true, it will remove the all the preprocessing prefixes (pp_,ppd_ etc…) from the calibrated frame name.
onefolder-perseq	*bool*	true	If true, it will create one subfolder per sequence in *folderout*

- **folderin** is used to search your *imaging folder* as defined in loop.cfg file, for folders containing calibrated light frames. Normally, you should leave the default value, *i.e. process*, as this is the folder base name into which gessengine will store calibrated frames.
- **folderout** is used to generate a filing architecture from the top level folder where you want to store calibrated frames. This can be a multiple level folder naming convention, mixing regular strings and FITS headers keys, which will be parsed based on values found in the frames to copy.
- **levelup** is used to determine how many levels up it should go relative to the folder where the calibrated frames are stored. Consider the following example:

```
MyPics
└── M31
    └── NEWT200F5
        ├── 2021-03-29
        │   └── DualBand
        │       ├── LIGHT
        │       ├── LIGHT_M31_NEWT200F5_DualBand_180s
        │       ├── masters
        │       └── process32
        ├── 2021-03-30
        │   └── DualBand
        │       ├── LIGHT
        │       ├── LIGHT_M31_NEWT200F5_DualBand_180s
        │       ├── masters
        │       └── process32
        ├── 2021-03-31
        │   └── DualBand
        │       ├── LIGHT
        │       ├── LIGHT_M31_NEWT200F5_DualBand_180s
        │       ├── masters
        │       └── process32
        └── Calibrated
            ├── ppdfdeb_LIGHT_M31_NEWT200F5_DualBand_180s
            └── ppdf_LIGHT_M31_NEWT200F5_DualBand_180s
```

The same target has been imaged over 3 nights and all the sessions have been processed with *loop*. Now, the aim is to gather all the calibrated lights contained in each *process32* folder to process all the lights together.
A sensible location would be at the same level as the dates folders. Considering the calibrated lights are in a process32 folder, one needs to go up 3 levels to land into *NEWT200F5* folder. Then, if **folderout** is specified as "Calibrated" and option *onefolderperseq* is set to True, copyback will:

- generate the *Calibrated* folder into *NEWT200F5* directory.
- create one folder per calibrated sequence (calibrated sequences start with prefixes pp)
- copy the calibrated lights into the right folders.
- **prefix** and **suffix** are used to complement the names of the files copied. Values from FITS headers, if specified, are parsed.
- **copyheader** is used to force the copy of the original (uncalibrated) light file in place of the header written by Siril. Can be useful if you need to retain all the orginal info contained in the initial header.
- **useoriginalFITSname** is used to copy each calibrated light using the original filename instead of the (wonderful) name generated by gess.
- **removepp** is used to tell copyback to remove the *pp*_* prefix from the calibrated file names.
- **onefolderperseq** is used to generate one folder per sequence (see the example above).

Warning: Copyback options have been intentionally made very flexible so that you can virtually do whatever you like when moving your calibrated files around. But with great flexibility comes great responsibility. Not all combinations will lead to a filing system that can then be used to finalize the preocessing of your sessions.
For instance, refrain from specifying dates that will end up with non valid sequence names for Siril to process afterwards. Please, think your naming conventions through to make sure files from different sources with same sequence names do not end up mixed in a single folder.
One good practice is surely to use *lightsfmt* in gess.cfg with sufficient details to make sure you get unique sequence names.

copyback.cfg JSON File

Once you've set these options through the graphical interface, you can always go to the *options location* and review copyback.cfg. (You could also modify it from there...) The file created by the GUI with the values shown in the figures above will look like this:

```
{
    "folderin": "process",
    "folderout": "Calibrated\[FILTER:s]",
    "levelup": 3,
    "prefix": "Cal_",
    "suffix": "",
    "copyheader": false,
    "useoriginalFITSnames": false,
    "removepp": false,
    "onefolderperseq": true
}
```

1.6 Files and Folders naming

GeSS uses different names for sequences so that you can see at a glance which pre- and processing steps have been applied.

1.6.1 Preprocessed files

After preprocessing, your calibrated lights (and sequence) should be named like this:

pp(d)(f)(deb)_(lights)_

- (d) is for darks substracted.
- (f) is for flats applied.
- (deb) is for debayering applied.
- (lights) is the name of the subfolder into which the lights are stored or the name specified by *lightsfmt* if you have specified one.

Examples:

- *ppdfdeb_light_.seq* is the name of a sequence with both darks/flats preprocessing and debayering applied.
- *ppddeb_light_.seq* is same as above except you have asked to skip preprocessing with flats.

1.6.2 Processed files

As names of calibrated sequences are passed on to the next steps (background extraction, registration, stacking), you can have different versions of processing on the same base sequence that coexist in the same folder.

At the end at the processing, result files are named as follows, again depending on the processing steps you have chosen to perform:

(r_)(layer_)(bkg_)(preprocessedlight)_stacked.(ext)

- (r_) is for global resgistration applied.
- (layer_) is either Ha or OIII if layer extraction is applied (color shots only).
- (bkg_) is for background extrcation (linear gradient removed) applied.
- (preprocessedlights) is the string reflecting your preprocessing steps, see above.
- (ext) is the extension of your FITS files.

Examples:

r_Ha_bkg_ppdf_light_stacked.fits is the stacked output of a lights sequence, preprocessed with dark and flats, with Ha layer and background, after being registered and stacked.

1.6.3 Storage folders

All along the process, gess will create the following folders to store intermediate and results files:

- masters: where all the masters are stored, whether thay have been copied from libraries or stacked on the spot. Have a look at *copymasters* if you want to store a hard copy of masters coming from libraries.
- process(bd): storing intermediate files. *bd* is either 16 or 32 depending on chosen bitdepth.
- results(layer)(bd): storing the stacked files. *layer* is none for normal processing and HaOIII is you have asked for Ha/OIII layers extraction.

Once you are happy with the single or multiple processings of your session, you can discard the *process* folder.

If you have also discarded the original calibration folders (darks, flats etc...) to save some space but have kept the *masters* folder, it will act as a local master library if you come back later and want to process again the same session.

1.7 Use cases examples

1.8 Python reference

1.8.1 gess

gessengine

This module is the core module of GeSS, called by most other modules. It takes as input a working directory path (similar to the working directory for launching a script in Siril) and a dictionnary specifying the options you want to apply.

`gess.gessengine.`**Run**(*workdir*, *opt=None*, *app=None*, *dryrun=False*)

Parameters

- **workdir** (*str*) – the path to the working directory.
- **opt** (*DictX, optional*) – a DictX instance containing all the options for the processing workflow described in the *First Steps* section, defaults to None. Can be formed as the options member of an *Options* instance, with optiontype='gess'. Have a look at *gess.cfg settings* for a full description of all the options keys.
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None.
- **dryrun** (*bool, optional*) – flag to stop engine just after starting Siril, if set to True. Used mainly for checking that the options are passed correctly and that the masters can be found, defaults to False.

Returns

(True if successful, dict with keys specifying inputs and outputs)

Return type

(bool,dict)

The bool is set to True is the processing was successful.

The dictionnary contains the following keys (if successful):

- 'dark': the masterdark that was used (if any)
- 'flat': the masterflat that was used (if any)
- 'offset': the masteroffset that was used (if any)
- 'workdir': the path to the working directory
- 'options': the options which were passed
- 'version': gess version number
- 'log': the path to the session log file if debug was on

Otherwise, if something wrong happened, the dictionnary contains a single key 'messsage' detailing where the process failed.

gessi

This module is used to call interactively *gess.gessengine*. More help on its usage can be found in *interactive mode* section.

It takes no input and returns the (bool,dict) tuple output by gessengine.

gessmultisession

This module is used to gather calibrated lights from multiple sessions, already processed with *gess.gessengine*. More help on its usage can be found in *multisession mode* section.

It can take as input an additional gess-type cfg file (if you have used one for the lights calibration) and returns the (bool,dict) tuple output by gessengine.

`gess.gessmultisession.Run`(*addcfgfile=''*)

[summary]

Parameters
addcfgfile (*str, optional*) – the path to a gess-style additional cfg file, if one was used to calibrate your lights, defaults to ''.

Returns
(True if successful, dict with keys specifying inputs and outputs)

Return type
(bool,dict)

see *gess.gessengine* for details on returned values.

gessloop

This module is used to batch-process multiple sessions by calling iteratively *gess.gessengine*. More help on its usage can be found in *loop mode* section.

`gess.gessloop.Run`(*addgesscfg=''*, *addloopcfg=''*, *searchstr=''*)

main function of gessloop.

Parameters

- **addgesscfg** – the path to a gess-style additional cfg file, defaults to ''.
- **addloopcfg** (*str, optional*) – the path to a loop-style additional cfg file, defaults to ''.

- **searchstr** (*str, optional*) – the string to be searched for in your imaging folder, defaults to ‘’

Returns
(True if successful, dict with ‘message’ key summarizing the different folders processed)

Return type
(bool,dict)

gesscopyback

This module is used to copy calibrated lights around, most likely after having processed them with *gess.gessloop*. More help on its usage can be found in *copyback mode* section.

gess.gesscopyback.**Run**(*addgesscfg=''*, *addloopcfg=''*, *searchstr=''*, *addcopybackcfg=''*)

main function of gesscopyback.

Parameters

- **addgesscfg** – the path to a gess-style additional cfg file, defaults to ‘’.
- **addloopcfg** (*str, optional*) – the path to a loop-style additional cfg file, defaults to ‘’.
- **searchstr** (*str, optional*) – the string to be searched for in your imaging folder, defaults to ‘’
- **addcopybackcfg** (*str, optional*) – the path to a copyback-style additional cfg file, defaults to ‘’

Returns
(True if successful, dict with ‘log’,’in’ and ‘out’ keys listing files which have been copied)

Return type
(bool,dict)

1.8.2 gess.common

helpers

gess.common.helpers.**checkcalibexists**(*calibtype*, *calibfolder*, *calibfmt*, *workdir*, *frames*, *ext*, *app=None*)

Checks if a calibration library of a folder containing calibration frames to stack exists

Parameters

- **calibtype** (*str*) – the name of calibration frames, i.e. dark,flat or bias
- **calibfolder** (*str*) – either the path to the calibration frames library or the subfolder name containing calibration frames to stack
- **calibfmt** (*str*) – the string to parse master frame name, containing str and FITS header keys with format specifier e.g: “BIAS_O[OFFSET:d]_bin[XBINNING:d].fit”
- **workdir** (*str*) – the path to the working directory containing the session
- **frames** (*str*) – the string to identify the frames to be calibrated, typ. “light” or “flat”
- **ext** (*str*) – the extension of the frames to be calibrated
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None

Returns
(True if successful,True if the masters are in a library/False if they need to be stacked from the session,full path to the masterframe if found in a library)

Return type
(bool,bool,str)

gess.common.helpers.**checklocalcalibexists**(*calibtype*, *calibfolder*, *calibfmt*, *frames*, *ext*, *app=None*)

Checks if a local copy of the master exists in the working directory (in the masters subfolder)

Parameters

- **calibtype** (*str*) – the name of calibration frames, i.e. dark,flat or bias
- **calibfolder** (*str*) – the path to the local calibration frames library
- **calibfmt** (*str*) – the string to parse master frame name, containing str and FITS header keys with format specifier e.g: "BIAS_O[OFFSET:d]_bin[XBINNING:d].fit"
- **frames** (*str*) – the string to identify the frames to be calibrated, typ. "light" or "flat"
- **ext** (*str*) – the extension of the frames to be calibrated
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None

Returns
(True if master exists, full path to the masterframe if found)

Return type
(bool,str)

gess.common.helpers.**getfirstfile**(*folder*)

Finds the first file of a folder with given extension

Parameters
folder (*str*) – the path to search

Returns
the path to the first file of the folder if one was found, an empty string otherwise

Return type
str

gess.common.helpers.**parsemasterformat**(*masterfmt*, *hdr*, *refframe*, *mode='r'*, *rmspace=True*)

Parses the name of a master based on its formatting string and the header of the frame to calibrate

Parameters
masterfmt (*str*) – a string containing the format specification to parse the name.

All the string tokens between brackets will be parsed. The tokens are formed with: - HEADERKEY: a valid key of the FITS header - fmt: a format specifier. Most of the time d, f or s should do.

You can have a look at https://docs.python.org/3/library/string.html#formatstrings

Special wildcard "*" character before HEADERKEY: If a HEADERKEY is suffixed with a wildcard character, the string returned will change, depending on **mode** value:

- If mode='r': [*HEADERKEY:fmt] is replaced by *
- If mode='w': HEADERKEY is parsed

Parameters

- **hdr** (*dict*) – A dictionnary of **refframe** header keys, as returned by pySiril.Addons::ReadFITSHeader
- **reffframe** (*str*) – the name of the reference frame
- **mode** (*str, optional*) – either 'r' or 'w'. Flag to handle behavior with wildcards, defaults to 'r'
- **mode** – Flag to remove spaces in the output string, defaults to True

Returns
the name of the file with header keys parsed as per specification.

Return type
str

gess.common.helpers.**findmaster**(*workdir*, *master*, *masterfmt*, *reffframe*, *app=None*)

Looks for a suitable master for a given reference frame

Parameters

- **workdir** (*str*) – the path to the working directory containing the session
- **master** (*str*) – either the full path to the masters library or subfolder string wrt. workdir
- **masterfmt** (*str*) – check-out spec in parsemasterformat
- **reffframe** (*str*) – the name of the reference frame
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None

Returns
(True if the master was found,the full path to the master if one was found)

Return type
(bool,str)

gess.common.helpers.**pathhasspace**(*folder*)

Returns True if the path has spaces

Parameters
folder (*str*) – a path

Returns
True if the path has spaces

Return type
bool

gess.common.helpers.**checksubfolder**(*workdir*, *subfolder*)

Checks if a subfolder exists

Parameters

- **workdir** (*str*) – the path to the root folder to be searched
- **subfolder** (*str*) – the subfolder name to be found

Returns
(True if the subfolder exists,The full path to the subfolder)

Return type
(bool,str)

gess.common.helpers.**relativizepath**(*refframe*, *workdir*)

Returns the path of a file relative to a given directory

Parameters

- **refframe** (*str*) – the path to a file
- **workdir** (*str*) – the path to a folder

Returns
the path of refframe relative to workdir, with all separators replaced with "/"

Return type
str

gess.common.helpers.**fast_scandir**(*dirname*)

Returns all the subfolders of a given path

Parameters
dirname (*str*) – the path to a folder

Returns
the list of all subpaths of dirname (full paths)

Return type
list(str)

gess.common.helpers.**checkmastersubs**(*mastertype*, *masters*, *opt*, *subbydate*, *app=None*, *findsubsets=True*, *follownaming=''*)

Returns all the subsets of masters in a path

Parameters

- **mastertype** (*str*) – a name to be used in print outs, giving the type of files being searched
- **masters** (*str*) – the last part of a path corresponding to these masters
- **opt** (DictX) – options member of an Options instance, with optiontype='gess'
- **subbydate** (*list(str)*) – a set of folders to search
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None
- **findsubsets** (*bool, optional*) – True if the headers of the files need to be verified to identify unique sets of frames, defaults to True
- **follownaming** (*str, optional*) – the naming convention to be used to name the subsets, defaults to ''

Returns
the list of all the subsets paths

Return type
list(str)

gess.common.helpers.**checkmastersconfiguration**(*mastertype*, *opt*, *loop*)

Checks the masters configuration as per loop spec

Parameters

- **mastertype** (*str*) – the types of masters being checked
- **opt** (DictX) – options member of an Options instance, with optiontype='gess'

- **loop** (DictX) – options member of an Options instance, with optiontype='loop'

Returns
(True if the check succeeded,True if the masters need to be stacked and copied to a library,the final pathbit of the masters,the format to parse the masters names)

Return type
(bool,bool,str,str,str)

gess.common.helpers.**masterstackingoptions**(*opt*, *mastertype*, *library=False*)

Returns gess options with only one type of masters to be stacked

Parameters

- **opt** (DictX) – options member of an Options instance, with optiontype='gess'
- **mastertype** (*str*) – the type of masters to be activated
- **library** (*bool, optional*) – True if the masters will be copied to a library. Activates 16b with no compression. Defaults to False

Returns
options member of an Options instance, with optiontype='gess', with preprocessing and compression options set to only stack one type of masters

Return type
DictX

gess.common.helpers.**returninifolder**(*inifile*, *boxtitle=None*)

Returns the last session working directory

Parameters

- **inifile** (*str*) – the path to gess.ini file
- **boxtitle** (*str, optional*) – The message to be displayed at the top of the folder chooser, defaults to None

Returns
the path to the selected session folder. gess.ini is updated with the new path

Return type
str

gess.common.helpers.**ReadSirilPrefs**(*app=None*)

Reads Siril Preferences file

Parameters
app (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None, defaults to None

Returns
a dictionnary with Siril configuration

Return type
dict

gess.common.helpers.**ParseSirilBDC**(*prefs=None*, *app=None*)

Parses Siril bitdepth and compression settings

Parameters

- **prefs** (*dict, optional*) – a dictionnary containong Siril preferences. If None, pySiril Addons::GetSirilPrefs() is called.

- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None, defaults to None

Returns
a DictX with compression and bitdepth commands

Return type
DictX

gess.common.helpers.**GetSirilBitDepth**(*prefs=None*, *app=None*)

Returns Siril bitdepth preferences

Parameters

- **prefs** (*dict, optional*) – a dictionnary containong Siril preferences. If None, pySiril Addons::GetSirilPrefs() is called.
- **app** (*Siril, optional*) – a Siril class instance that can be used to call pySiril functions. Pass None if it needs to be started, defaults to None

Returns
"16" or "32"

Return type
str

gess.common.helpers.**GetSirilRawExt**()

Returns Siril list of RAW extensions

Returns
a list of valid extensions for raw format as returned by siril -f

Return type
list(str)

gess.common.helpers.**ReadRawHeader**(*filename*, *validrawexts=None*)

Read raw exif and returns a dictionnary with the following keys:

- INSTRUMEN: the name of the camera
- ISOSPEED: the iso setting of the shot
- EXPTIME: the exposure time in s
- DATE-LOC: the date and time of the exposure

Parameters

- **filename** (*str*) – full path to the raw file to read headers from.
- **validrawexts** (*list(str)*) – a list of valid raw files extensions, defaults to None

Returns
a dictionnary with keys listed above

Return type
dict

DictX

class gess.common.DictX.**DictX**

A overload of dict class that accepts dot assignment

Parameters
dict (*dict*) – a dictionnary

options

class gess.common.options.**options**(*optiontype='gess'*, *addcfgfile=None*, *dictcfg={}*, *updatedefault=False*, *returnclean=False*)

Class to handle options passed to gessengine, gessloop and copyback.

Default cfg file are stored in your user folder at ./gess/

This class is called at initialization of the package to create default cfg files if none is present:

- gess.cfg: the options for your typical worflow
- loop.cfg: additional options to use gessloop module
- copyback.cfg : additional options to use copyback module

Parameters

- **optiontype** (*str, optional*) – the type of options, either 'gess', 'loop' or 'copyback', defaults to 'gess'
- **addcfgfile** (*str, optional*) – an additional cfg file to read more options from on top of default values, defaults to None
- **dictcfg** (*dict, optional*) – a dictionnary to read more options from on top of default values, defaults to {}
- **updatedefault** (*bool, optional*) – Flag to update default cfg with the values passed, defaults to False
- **returnclean** (*bool, optional*) – Flag to return a clean version of the options attribute with all values to default, defaults to False

updateoptions_fromdict(*optdict*)

method to update the options attribute with values from a dictionnary. Checks the type of the entries to make sure they match the types defined in the constructor.

Parameters
optdict (*dict*) – a dictionnary with values to update the options attribute

Returns
True if the update was successful

Return type
bool

updateoptions_fromfile(*addcfg*)

method to update the options attribute with values from a file.

Parameters
addcfg (*str*) – the path to a cfg file. Can be either a full path or a filename. If filename, it is assummed to be located in the dame folder as the default cfgfile.

Returns
True if the update was successful

Return type
bool

`printoptions()`

Print out all the keys and values of the options attribute

`getoptions()`

Return the options attribute

`exportcfgfile(cfgfile='')`

Export options attribute to a cfg file, selected by the user

Parameters
cfgfile (*str, optional*) – optional path to save the cfgfile. If empty, will open a SaveAs window, defaults to ‘’

Returns
True if sucessfull

Return type
bool

`importcfgfile(update=True)`

Import options attribute from a cfg file, selected by the user

Parameters
update (*bool, optional*) – If True, the default cfg file is updated, defaults to True

Returns
True if sucessfull

Return type
bool

Logger

`class gess.common.Logger.Logger(caller='gess_')`

Class to log both to terminal and to .log file

Parameters
object (*object*) – object

1.8.3 gess.utils

SSFy

`gess.utils.SSFy.Run(inputfilename='')`

Analyzes a log generated by gess and returns all the commands that were passed in the form of a Siril ssf file.

Parameters
inputfilename (*str, optional*) – the path to a *.log file, defaults to ‘’.

If an empty string is passed, opens a file chooser to select the input file.

Returns
(True if successful, dict with 'message' key)

Return type
(bool,dict)

CFGy

gess.utils.CFGy.**Run**(*inputfilename=''*)

Analyzes a log generated by gessengine and returns the options in the form of a cfg file

Parameters
inputfilename (*str, optional*) – the path to a *.log file, defaults to ''.

If an empty string is passed, opens a file chooser to select the input file.

Returns
(True if successful, dict with 'message' key)

Return type
(bool,dict)

PYTHON MODULE INDEX

g

INDEX

U