vison Documentation

Release 0.9+374.gd7313d7

Ruyman Azzollini

Jun 23, 2020

CONTENTS

Contents:

CHAPTER

ONE

README

vison Euclid VIS Ground Calibration Pipeline

Documentation: https://ruymanengithub.github.io/vison/

Author Ruyman Azzollini

Contact r.azzollini_at_ucl.ac.uk

issue 0.9+374.gd7313d7

version 0.9+374.gd7313d7

date Jun 23, 2020

This Python package "vison" is the software pipeline that has been used at MSSL for ground calibration of the VIS detection chains (12 + 2 spares), including one ROE, one RPSU and three CCDs each.

CHAPTER

TWO

INSTALLATION

The package is distributed via github. The repository is hosted at:

https://github.com/ruymanengithub/vison

Detailed instructions:

2.1 Installation

2.1.1 Cloning *vison* from the repository using *git*

If you don't have *git* installed in your system, please follow this link first.

Here we will follow these instructions to clone the repository to your own computer. Follow the link for instructions in other operative systems.

Step-by-step:

- Go to https://github.com/ruymanengithub/vison.
- Click on the green "Clone or download" button.
- In the Clone with HTTPs section, click to copy the clone URL for the repository.
- Open a Terminal.
- Change the current working directory to the location where you want the cloned directory to be made.
- Type `git clone`, and then paste the URL you copied in Step 1.

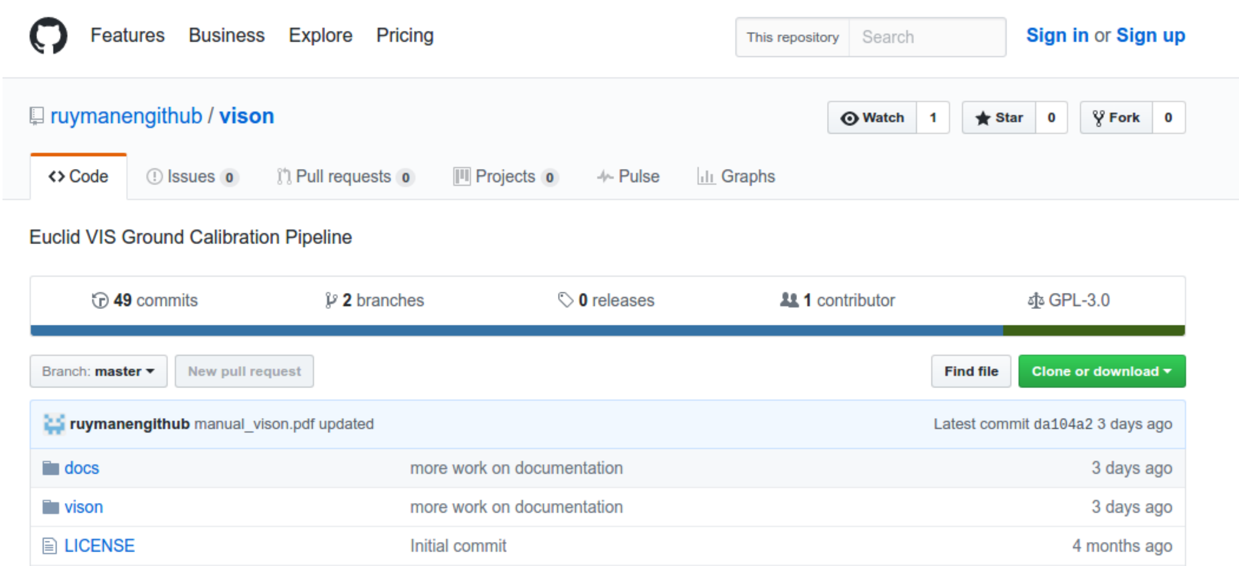
```
~$ git clone https://github.com/ruymanengithub/vison
```

- Press Enter. Your local clone will be created.

2.1.2 Installation

We recommend installing the code through a *conda* environment, with a specific list of packages, so you can be sure you have all the needed dependencies.

First, if you don't have *conda* already installed in your system already, follow the instructions in this link.



Installing conda and creating *vison* environment

Once you have succesfully installed conda, we will create an environment that will allow you to install the pipeline and meet all its dependencies (save for SAO DS9, which is only used in real-time monitoring, optionally).

Step-by-Step:

- change directory to your copy of the vison repository:

```
~$ cd vison
```

- Under the 'conda' sub-folder, you will find several text files:

```
~$ cd conda
~$ ls
env-conda_vison_linux.txt  env-pip_vison.txt
```

- Then execute the following command to create a new conda environment, *vison*.

Use the OS version that may correspond in your case (by now, only linux-64 bits version available).

:: ~$ conda create -n vison –file env-conda_[OS].txt

- When prompted, type "y" and return to install the listed packages.
- Activate the new environment

```
~$ source activate vison
```

- Update pipe

```
~$ pip install --upgrade pip
```

- Install the packages that are accessed via *pip*, within the conda environment:

```
~$ pip install -r env-pip_vison.txt
```

Installing *vison*

Finally, to install the *vison* pipeline itself, we will go back to the folder we downloaded from the github repository:

```
~$ cd ../
~$ ls
conda  docs  LICENSE  manual_vison.pdf  README.md  setup.cfg  setup_distutils.py ␣
↪setup.py  vison
```

Then do the actual installation, via:

~$ python setup.py install

Now the vison package will be accessible from anywhere in your system, whenever you start python from within the *vison* conda environment. For example:

- open a new terminal and go to your home directory

  ```
  ~$ cd
  ```

- activate the vison environment:

  ```
  ~$ source activate vison
  ```

- start the python interpreter and import vison:

  ```
  ~$ source activate vison
  ~$ python
  >>> import vison
  >>> dir(vison)
  ['Eyegore', 'FlatFielding', 'Pipe', 'Report', '__all__', '__builtins__', '__doc__
  ↪', '__file__',
  '__name__', '__package__', '__path__', '__version__', '_version', 'analysis',
  ↪'blocks', 'dark',
  'data', 'datamodel', 'eyegore', 'flat', 'image', 'inject', 'matplotlib', 'ogse',
  ↪'ogse_profiles',
  'other', 'pipe', 'plot', 'point', 'pump', 'stop', 'support']
  ```

2.2 Dependencies

Instructions to acquire a copy of the "conda" environment that provides almost all dependencies is included in the package. See *Installation* instructions for details. The only package that will not be installed by this means is SAO-DS9, which must be installed separetly, in order to be able to use some of the interactive inspection capabilities of Eyegore (data acquisition monitoring).

CHAPTER

THREE

HOW TO USE IT

This is some kind of **"cook-book"**. Written *a bit like when your grandmother tells you to put a pinch of sugar and heat it for a while... that kind of accuracy.*

3.1 vison cook-book

This cook-book is an adaptation of a guide used during the actual calibration campaign. As such, it contains mentions to specific machines in MSSL, which are not really important to the workings of the code, but we have decided to leave the document as close to the original as possible, for it illustrate how the code was used in practice.

This guide was written for users with little acquaintance with Linux. That is why there are so many comments about linux commands and their use throughout the text.

3.1.1 Accessing "vison", the pipeline

The pipeline and all accessory software is installed at:

/disk/euclid_caldata06/data06/SOFTWARE_LITE/

To use it, we have to do a few steps that will be detailed below, but in a nutshell these are:

- ssh-connect to a Linux machine in the MSSL network from where we can access both the software and the data, which will be in any of the euclid_caldata0X drives.
- Change the shell to "bash" and change the system paths to the right values necessary to find the conda environment within which the pipeline is installed.
- Activate the "conda" environment in which the pipeline runs. This environment has all the libraries that the pipeline needs to run.

Then you should be ready to use the pipeline, which can be imported as a Python package, or most likely, you will use via one of its several command line scripts.

Now we explain and show these steps in more detail.

Connecting to the remote machine via ssh

The machines in which we'll run the pipeline to do data "checks" and "analysis" are:

- MSSLAP
- MSSLAN
- MSSLAO

To ssh any of them do, from a terminal:

```
>> ssh -Y mssla?    # ? is here just a wildcard, one of (p,n,o)
Mullard Space Science Laboratory (Distribution 11.0)
```

This computer system is the property of University College London.

Use of this system is limited to authorised individuals only. You are committing a criminal offence under the Computer Misuse Act 1990 sect 1, if you attempt to gain any unauthorised access either to this system or any others at this site.

Communications on or through University College London's computer systems may be monitored or recorded to secure effective system operation and for other lawful purposes.

```
user@mssla?'s password:   # enter here you password
```

The "-Y" in the command allows to have graphical output from the program redirected to your computer screen.

Changing shell to bash and updating system paths

IMPORTANT: once we're connected to the remote machine, mssla?, we change the command shell to "bash":

```
[user@mssla? ~]$ bash   # hit Enter
bash-4.2$
```

Then we have to source a "bashrc" file so that we have the right system "PATHS" to find the pipeline.

```
bash-4.2$ source /disk/euclid_caldata06/data06/SOFTWARE_LITE/bashrc_lite #
↪hit Enter
bash-4.2$
```

Activating the "vison" conda environment

Now we can activate the "vison" conda environment within which the pipeline is installed. If you want to know what "conda" is, check this out https://www.anaconda.com/.

All you have to do to activate the environment is this:

```
bash-4.2$ source activate vison # hit Enter
(vison) bash-4.2$
```

Now we see (vison) preceeding the bash prompt. This indicates we're within the "vison" conda environment, and the pipeline should already be accessible. To check this is the case, do this and check you get the same reply:

```
(vison) bash-4.2$ which vison_run # hit Enter
/disk/euclid_caldata06/data06/SOFTWARE_LITE/anaconda2/envs/vison/bin/vison_
↪run
```

This indicates both that the command "vison_run" which executes the pipeline is accessible, and that we're using the pipeline installed in euclid_caldata06/data06/SOFTWARE_LITE, as we should.

3.1.2 Data Checks / Analysis Work Environment Setup

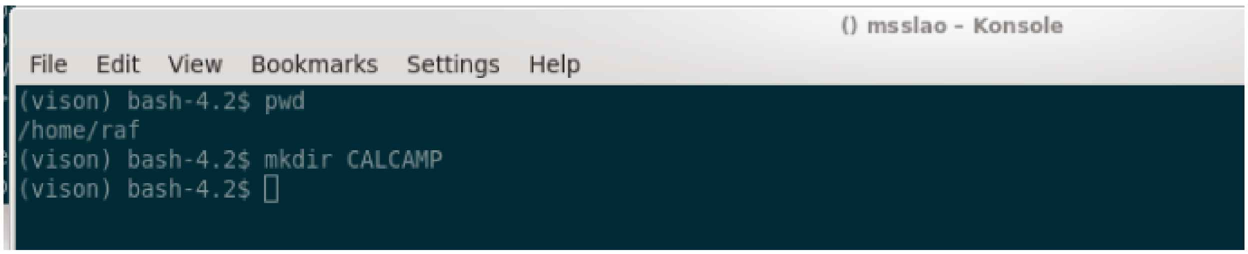
Before we can start using the pipeline, it is most convenient, and strongly suggested, to set up a consistent work environment, creating a number of folders and symbolic links in a workspace directory, so we can produce results that are useful, traceable, and easy to recognize by other members of the team (for inspection, or collaborative work). This is a guide to set up such a workflow. We'll go step-by-step.

1. Starting point: assumptions

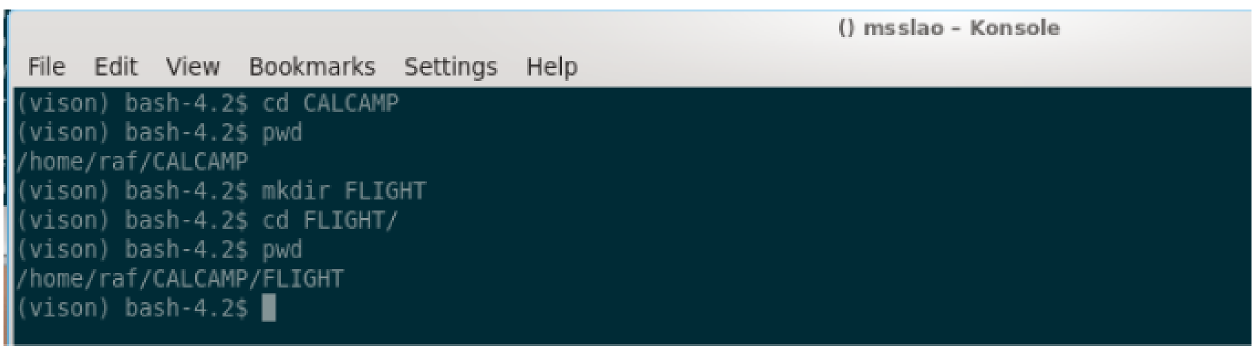
We'll assume that we are already ssh-connected to one of mssla? and activated the "vison" conda environment, following the indications of the first part of this tutorial: Accessing "vison", the pipeline.

2. Creating a working CALCAMP directory tree in our $HOME folder.

We'll first create a work folder in your $home folder, where we'll do all the check/analysis tasks related to the CALCAMP.



We move into CALCAMP and create a FLIGHT subfolder:



Note: the "pwd" command shows where we are, the current directory.

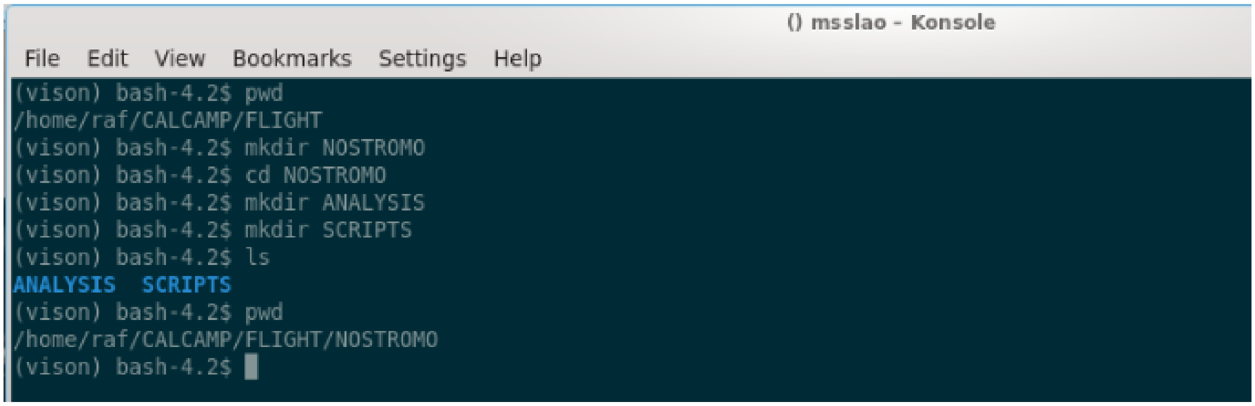
Now, for the sake of exercise, we'll assume we're going to calibrate a block dubbed "NOSTROMO". **We'll create the following folder structure below "FLIGHT"**:

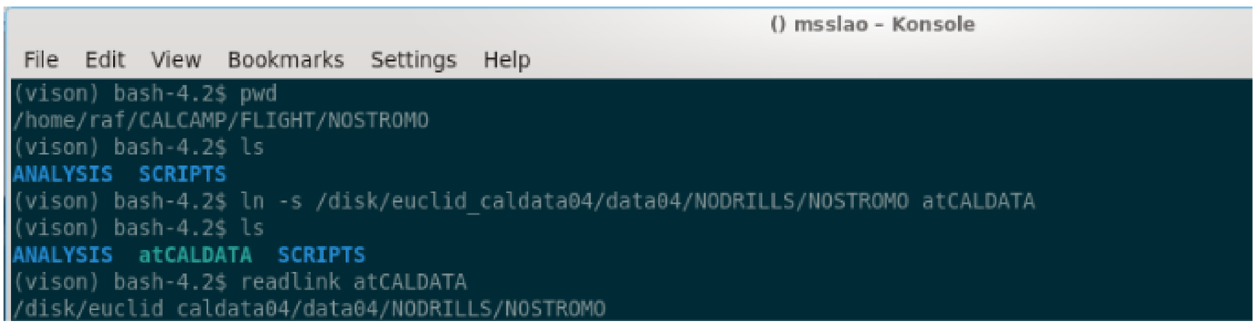
Note: the "ls" command lists the contents of the folder we're in, and "pwd" shows where we are.

Now we will create symbolic links to the folders where the data are, so the paths we give to the pipeline are conveniently short. So, while at the NOSTROMO folder, we first create a link to the path in euclid_caldata0X where the data are:

Note: of course, it is an assumption for the exercise that the data for the fictitious block NOSTROMO is in euclid_caldata04/data04… in practice you'll have to ask test_scheduler where the data is to be copied in euclid_caldata0X for the block of concern.

Now, if we're also writing scripts for the test_operators, and/or doing data acquisition monitoring using eyegore, it'll also be convenient to create a symbolic link to the folder in MSSL3M/EEDisk5 for this block. That's where the data is being copied by the laptop running ELVIS in ISO8 [lab room].





NOTE: In order to have permanently mounted access points to the EEDisk5 drive of MSSL3M you'll have to ask help from IT… and you're not guaranteed to get it, they're very reluctant about this. So, it's likely that only if you connect to MSSLLX, where those access points have already been created (and are at least accessible for user raf), instead of MSSLA[O/N/P], that you'll be able to link to / access that drive in the convenient way we describe in this tutorial.

For the time being we'll assume you also have EEDisk5 permanently mounted for you in /mssl3m/D5/, as it's done in MSSLLX (for user raf, at least).

So, we create a symbolic link to the NOSTROMO folder in MSSL3M/EEDisk5 at your NOSTROMO folder in CALCAMP.

So far we haven't really done anything but creating directories and some symbolic links. Now let's move to start copying some template input files and python scripts we'll need to actually do something.

3. Copying the script-writer session-builder scripts from the templates folder

The template scripts that are used by the pipeline to run, are all saved in: /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP

To do a local copy of the scripts-writer inputs file we'll do the following:

Now we rename the file to a name that's meaningful for the calibration campaign of NOSTROMO, in particular:

Where we have assumed that this BLOCK will be calibrated in April '19, on week 17 in the year, in particular.

```
(vison) bash-4.2$ cd SCRIPTS/
(vison) bash-4.2$ pwd
/home/raf/CALCAMP/FLIGHT/NOSTROMO/SCRIPTS
(vison) bash-4.2$ ls
(vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/SCRIPTS/scripts_inputs_BLOCK_MMM19_WN.json .
(vison) bash-4.2$
```

```
(vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/SCRIPTS/scripts_inputs_BLOCK_MMM19_WN.json .
(vison) bash-4.2$ mv scripts_inputs_BLOCK_MMM19_WN.json scripts_inputs_NOSTROMO_APR19_W17.json
(vison) bash-4.2$ ls
scripts_inputs_NOSTROMO_APR19_W17.json
(vison) bash-4.2$
```

Then you'll have to go to the specific tutorial on writing scripts for the campaign to follow on that.

Once you've written the scripts you'll also want to create a number of "sessions", with a specific set of tests in a specific order within them. This is most safely done using another pipeline script, vis_mksession.py. In order to run this script you'll also need an inputs file, which we'll copy from the templates folder:

```
(vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/SCRIPTS/sessions_builder_BLOCK_MMM19_YWNN.json .
(vison) bash-4.2$ mv sessions_builder_BLOCK_MMM19_YWNN.json sessions_builder_NOSTROMO_APR19_YW17.json
(vison) bash-4.2$ ls
scripts_inputs_NOSTROMO_APR19_W17.json  sessions_builder_NOSTROMO_APR19_YW17.json
(vison) bash-4.2$
```

4. Copying the check/analysis script from the templates folder

To copy the template analysis script to the local working folder, and renaming it with a meaningful name we'll do:

It is **very important** to rename the copied file to something that actually has meaning, so that when this file is moved to the results folder, for reference, it's easy to find, and we know what it is. The analysis script holds most of the relevant information about the configuration of the pipeline when the analysis was performed, which is useful for tracing back what has been done. The rest of the information about what the pipeline did will be stored in the outputs of the analysis itself.

To learn about how to edit this script and actually use the pipeline to check / analyze data, follow to the corresponding tutorial.

3.1.3 Workflow Description

In this section we describe the work-flow of operations for those doing the following tasks during the calibration campaign:

- Writing acquisition scripts (using vis_mkscripts.py and vis_mksession.py).
- Doing the data-acquisition monitoring in real-time (using eyegore).
- Doing the analysis (using vison_run).

These activities should be assigned to different roles/persons, and formally the are. But because in practice, they are usually done by the same person, and regardless of this, they are so closely related, we'll present them together and interleaved, as they are in practice.

WRITING SCRIPTS / SESSIONS

Step-by-step description of what to do regarding the writing of test scripts and building sessions.

Step	Action	Comment
1	Get hardware serials from Campaign director.	We need these to write the scripts, so that ELVIS writes the meta-data of the images and EXP-LOG correctly.
2	Using the CHAMBER profile that corresponds to the chamber, write scripts.	Here we're using the best focus from the last time we measured it in this chamber. So, it may be slightly off. Still, we only run tests without point-source in the first sessions, D00, D11 and D12 (except the FOCUS00 test that measures the focus), and so this is not a problem.
3	We create the sessions D00, D11, D12	Using the written scripts, and the schedule provided by Cal. Camp. Scheduler, we'll build the sessions D00, D11, D12 using vis_mksession.py. These sessions do not include point-source tests, except the FOCUS00, in some cases, the PSFLUX00 test, for which the focus setting is not critical, and the PERSIST00 test in which we deliberately de-focus the source by a large margin.
4	Inspect at least some of the scripts	At least verify the serials of the hardware are correct in the scripts.
5	We copy the sessions to MSSL3M/D5	We copy the scripts and sequence files from these sessions to the corresponding block folder in MSSL3M/D5, and let the test operators know via email (with test camp. Director on copy).
6	Update focus in pipeline based on results from PSF01_NNN (_800)	As soon as the results from the focus test in session D11 are out, we can update the focus position for this chamber. It's the pipeline custodian who does it, but we can give him the new best estimate given the results from test PSF01_NNN.
7	Update exposure times, if needed	If the results from the campaign so far require it (say from any test FLATFLUX00 or PSFLUX00), also update the exposure times at this point. We do not expect this to be necessary once the chamber has been commissioned.
8	Write scripts for sessions D21 and D22	We write the scripts again, with the new focus position update.
9	Make sessions D21 and D22	Using vis_mksession.py we create sessions D21 and D22 using the latest scripts.
10	Check new scripts	Check serials are right in at least one of the new scripts, and verify the changes in focus (and exposure time, if there have been), have been effective in the tests where this applies.
11	Copy session D12 and D22 to MSSL3M/D5	We copy the sessions D21 and D22 to MSSL3M/5 and let the test operators know via email with test director on copy.

Details on how to write scripts and sessions, using the pipeline scripts **vis_mkscripts.py** and **vis_mksession.py**, are provided elsewhere.

Real Time Data Inspections using Eyegore

As soon as we start acquiring, we should be running "**eyegore**" on the data that should be "flowing" to MSSL3M from the laptop running ELVIS in ISO8.

WARNING: Currently, many of the eyegore capabilities quoted below can only be performed by **user "raf"** from "his" machine MSSLLX. The reason for this are mostly down to limitations of permissions granted to users for (convenient) access to servers (to MSSL3M and MSSLUS) or files (with passwords needed to send warning sms, for example).

These are the main capabilities of eyegore, that justify its use in parallel with the acquisition at all times:

- Eyegore gives a **synoptic view** of the HK that can allow for quick identification of major issues with the hardware (voltages / currents / temperatures out-of-limits). This of course requires visual attention to the eyegore windows from time to time (**at least once every hour**), while in office hours.
- We can track progress with the **EXP-LOG display** of the data acquisition. We can also use it to send images to DS9 with a mouse double-click, for quick inspection of incoming data.

- We can activate the **warning system** of eyegore so we're warned of major HK OOL conditions (e.g. detectors being dangerously cold) via email and sms (though this latter function can only be used by user raf, by now).
- Eyegore can also be used for **automatic and on-the-fly back-up of the data to the euclid_caldata** drive of choice for this block. And this is not only a safety/precaution measure, we also need the data to be in a euclid_caldata folder in order to run the pipeline for checks/analysis from any of the designated linux machines in the network for this tasks.
- Eyegore can also be used to rely the data to the "**msslus**" server as we acquire it, so that external data inspectors can have access to it asap. These "external eyes" have proved to be very useful in identifying issues with the hardware in the past, and so it's deemed very important for the calibration campaign that we keep serving them the data through msslus as we acquire it.

Details on how to do the data acquisition monitoring using **eyegore** are provided elsewhere.

Checking Data Quality / Analysis

The data-checks and data-analysis are related tasks, but they are not the same thing. Let's clarify the differences, before proceeding to explain how and when to do both.

The pipeline has assigned to each test we perform in the calibration campaign a "Task". These Tasks are Python objects with some methods that effect some processing (not in-place), measurements, and in general analysis, on the data of a test (or type of tests). The, these "Tasks" are subdivided in "sub-tasks", and the pipeline, executes these sub-tasks in sequence when we call that Task to be run on a data-set. Then, the "data checks" are just a part of the sub-tasks that compose the Tasks. This is effected by a sub-task which invariably is named "check_data" in the Task classes, and is activated by setting check=True in the todo_flags dictionary of the Task. For doing the "data checks" we need to at least have been done before all the previous sub-tasks in the Task (Task initialisation, and in some cases also point-source locking / targets finding). The "data checks" are a very limited set of measurements performed on the raw data that allow to judge whether the data acquired is complete and of sufficient quality to perform the analysis which the data is intended for.

Without further ado, let's proceed to enumerate the steps we have to follow to do the data-checking and analysis during the calibration campaign, in parallell with the data acquisition.

Action	Comment
folder trees and links to data folders	Create the folder structure, with symbolic links as needed, described in section 2.
Rename and Fill the vison configuration script	Copy the vison configuration python script, rename it with the right block name and date, and edit it according to the expected distribution of tests in sessions, hardware serials, chamber ID, block ID, etc.
Check todo_flags	Check values of the todo_flags dictionaries in the vison configuration Python script.
Session **D00**: Run Pipeline on **MOT_WARM** test in **wait mode**	Run pipeline on test MOT_WARM (as soon as test is finished) to check data and give the go-ahead signal to pump down or not, according to results.
Session **D11**: run in **wait mode**.	Run pipeline in wait-mode along with acquisition of D11 session. Use results from FOCUS00, and FLATFLUX00/PSFLUX00 to update focus and/or saturation times in pipeline, as needed.
Fill up Test Record with results from session D11	Done by data analysts, using the results from the pipeline on session D11.
Alert pipeline custodian of need to change to pipeline / ogse configuration	Ask pipeline custodian to update ogse values (focus, fluxes), so that the script writers can write updated scripts for next sessions (in particular point-source tests).
Update day-folders and Obsid limits in config file	Update day-folder value for session D11 and OBSID limits for each test within the session in the vison configuration script.
Session D11: run in aim mode	After session D11 is finished, run the pipeline again on the session to complete analysis while the data from session D12 is being acquired. Some test results are more urgent than others, and this must be taken into account. Prioritize as needed.
Session D12: run in wait mode	For all tests in this session the pipeline will be asked to at least do the data-checks, and for some other tests we'll also ask full-analysis, so that those results can be put in the test report (excel) asap. These choices are efected by changes to the todo_flags in the vison Python configuration script.
Update day-folders and Obsid limits in config file	Update day-folder value for session D12 and OBSID limits for each test within the session in the vison configuration script.
Session D12: run in aim mode	After session D12 is finished, run the pipeline again on the session to complete analysis while the data from session D21 is being acquired. Some test results are more urgent than others, and this must be taken into account. Prioritize as needed.
Session D21: run in wait mode	While data from session D21 is being acquired, run pipeline in wait mode, in parallel, to produce check-data results.
Update day-folders and Obsid limits in config file	Update day-folder value for session D21 and OBSID limits for each test within the session in the vison configuration script.
Session D21: aim mode.	After session D21 is finished, run the pipeline again on the session to complete analysis while the data from session D22 is being acquired. Some test results are more urgent than others, and this must be taken into account. Prioritize as needed.
Session D22: run in wait mode	While data from session D21 is being acquired, run pipeline in wait mode, in parallel, to produce check-data results.
Update day-folders and Obsid limits in config file	Update day-folder value for session D22 and OBSID limits for each test within the session in the vison configuration script.
Session D22: run in aim mode.	After session D22 is finished, run the pipeline again on the session to complete analysis. Some test results are more urgent than others, and this must be taken into account. Prioritize as needed.

IMPORTANT: If there are test failures, or we just decide to change the order of some tests after the run has started (hopefully not a common situation) we'll need to reschedule tests, and this may require modifications to the vison configuration script so that the pipeline knows in which session are the tests, and where is the data.

Detailed instructions on how to run the pipeline using vison_run are provided elsewhere.

How to Write Test Scripts and Create Sessions

Writing Test Scripts

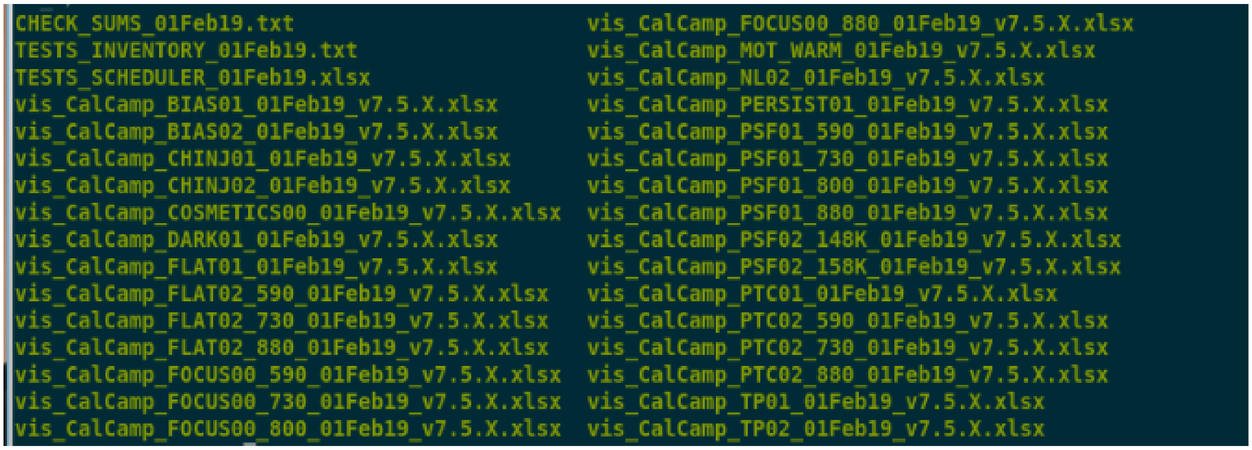
To write scripts we use a pipeline script, vis_mkscripts.py. This is called like this from a terminal:

```
bash$ vis_mkscripts.py -j scripts_inputs_BLOCK_MMM19_WN.json
```

The .json file has information about the hardware serials, ELVIS version to be used, Chamber and tests to be written. Here is an example of input json file to write scripts with the pipeline, with comments. The comments, even if they are behind "#" signs make the pipeline fail:

```
  {
    "equipment": {
        "sn_roe": "FMNN", # substitute NN with the right serial.
        "sn_rpsu": "FMNN", # substitute NN with the right serial.
        "operator": "unk", # do not modify, it's just not possible to know who will␣
↪execute the script in advance.
        "sn_ccd1": "14XXX-XX-XX", # substitute NN with the right serial.
        "sn_ccd2": "14XXX-XX-XX", # substitute NN with the right serial.
        "sn_ccd3": "14XXX-XX-XX"  # substitute NN with the right serial
    },
    "elvis": "7.5.X", # elvis version
    "CHAMBER": "CHAMBER_ID", # This is used by the pipeline to compute the right␣
↪exposure times, for example.
    "camptype": "Full",      # Type of campaign. Always "Full" for FM Calibration␣
↪campaign.
    "outpath": "Scripts_BLOCK_MMM19_WN", # output path, example: Scripts_BORN_MAR19_W1
    "toWrite": {   # 1 means write the script(s) and 0 do not write.
                Some test labels produce more than 1 script. For example, if there are
                several wavelengths.
        "BIAS01": 1,
        "BIAS02": 1,
        "DARK01": 1,
        "CHINJ01": 1,
        "CHINJ02": 1,
        "TP01": 1,
        "TP02": 1,
        "FLATFLUX00": 0,
        "FLAT01": 1,
        "FLAT02": 1,
        "PTC01": 1,
        "PTC02WAVE": 1,
        "PTC02TEMP": 0,
        "PTC02RD": 0,
        "NL01": 0,
        "NL02": 1,
        "NL02RD": 0,
        "PSFLUX00": 0,
        "PSF01": 1,
        "PSF02": 0,
        "FOCUS00": 1,
        "PERSIST01": 1,
        "MOT_WARM": 1,
        "COSMETICS00": 1
    }
}
```

Here's an example of the contents of the folder where the pipeline outputs all the scripts:



Contents of the folder where the scripts are written.

One-by-one, these files are:

- **CHECK_SUMS_01Feb19.txt**: this is a plain-text file with the name of the scripts and their checksums. I usually reuse this file, the listing of script names, to build the "sequence" files used by ELVIS to execute the tests in sequence. I just copy the names from this file to the sequence files, manually.
- **TESTS_INVENTORY_01Feb19.txt: plain-text file with a summary of the tests. Contents:**
  - **Header:**
    - Date and time when scripts have been written.
    - Name of the corresponding CHECK_SUMS file.
    - Pipeline version.
  - **For each test:**
    - Test Name: Nr. of columns, expected duration, frames in each column.
  - Total Nr. of Frames.
  - Total duration in minutes.
- **TESTS_SCHEDULER_01Feb19.xlsx**: An excel with all the tests, their number of columns, frames and durations. These test entries (rows in the excel) can be easily copied across to the "scheduler" spreadsheet for the Cal. Camp. of each block.
- The test scripts: **vis_CalCamp_TESTNAME_DDMmmYY_v7.X.X.xlsx**.

Making Sessions

As we have seen, the pipeline writes all scripts to the same folder. And there is no "sequence" file that can be used by ELVIS to ingest tests. Thus, dividing the tests in session folders, each with a sequence file (a list of script file names, one per line) that can be ingested by ELVIS, would have to be done by hand. But now it is possible to do it in a less prone-to-error fashion using the pipeline, through a script named vis_mksession.py. Here we describe how.

To create a number of sessions from a directory with test scripts, we'd command:

```
(vison) bash$ vis_mksession.py -j sessions_builder_BLOCK_MMM19_YWNN.json
```

The .json file has information about where to find the scripts (written with vis_mkscripts.py), where to create the session folders, and the tests that are within each session, and in which order. The tests are identified by their name on the script files, and they will be added to each session in the specified order. If there are more than 1 script for the same test in the inputs folder, the application will halt with a message alerting of the issue. The same would happen if it doesn't find at least one script for that test. Example of input json file to create the sessions with the pipeline, using vis_mksession.py:

```
{
    "inpath": "Scripts_BLOCK_MMM19_YWNN",
    "outpath": "SEQUENCES_BLOCK_MMM19_YWNN",
    "sessions": {
        "D00": ["MOT_WARM"],
        "D11": ["COSMETICS00",
                "FOCUS00_800",
                "FLATFLUX00_800",
                "PSFLUX00_800",
                "PTC02_730"],
        "D12": ["BIAS02",
                "BIAS01",
                "CHINJ01",
                "CHINJ02",
                "TP01",
                "TP02",
                "PTC01",
                "NL02",
                "FLAT01",
                "DARK01",
                "PTC02_590",
                "PTC02_880",
                "PERSIST01"],
        "D21": ["BIAS02",
                "PSF01_590",
                "FLAT02_590"],
        "D22": ["BIAS02",
                "PSF01_730",
                "FLAT02_730",
                "PSF01_800",
                "FLAT02_880",
                "PSF01_880"]
    }
}
```

WARNING: We may need to write scripts and make sessions more than once during a run, if there's an update to the OGSE parameters (focus / fluxes), or there's a change in the schedule for any reason.

3.1.4 Data Analysis Cookbook

Introduction

In this part we describe how to run the ground calibration pipeline, vison, to **analyse test data**. The pipeline has multiple functions, which can be accessed either as a Python package, or more usually for the end-user (aka you), via several command scripts. Here we will restrict ourselves to using the pipeline to do:

- **Analyse data on-the-fly (as we acquire it), and optionally restricting ourselves to just checking the integrity and quality of**

  - **Nota bene**: when we say analysis / check on-the-fly, it isn't strictly on-the-fly, as the pipeline has to wait for all the data from a given test to be acquired to actually start inspecting / analysing the test.
  - **Nota bene 2**: there are tools in the pipeline to inspect data / HK as it is acquired, via the "eyegore" tool, but that's the subject of another dedicated tutorial.

*(In depth) Analysis of the data post-acquisition.

Editing the vison_config_*.py file

The most delicate and complicated part of running the pipeline is editing the configuration script, a Python script itself, that's used to configure the pipeline, to tell it what to do, and with which data.

To execute the pipeline we'll use the command "**vison_run**", with a number of options, like for example:

```
(vison) bash$ vison_run vison_run -y vison_config_BLOCK_MMM19.py -R DNN -l -t _DNN -m␣
↪6
```

In the preceding command, we have highlighted the reference to the Python configuration file, **vison_config_BLOCK_MMM19.py** of concern here. Now we will go through a template for this file, and comment on what inputs we must provide in order to run the pipeline with it.

The script is about 400 lines long (the exact total length depends on how many tests are done within the run, as you will soon understand), but you only have to modify some of them.

To copy the template analysis script to the local working folder, we will do:

```
(vison) bash $ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/
↪ANALYSIS/vison_config_BLOCK_MMM19.py .
```

It is **very important** to rename the copied file to something that actually has meaning, so that when this file is moved to the results folder, for reference, it is easy to find, and we know what it is (years from now). So, you will replace "BLOCK" with the block-name (e.g. "BORN"), and "MMM19" with the month-year date.

In Fig. 1 we see a commented version of the first part (first ~70 lines) of the configuration script. There you have pointed out where you would put the name of the BLOCK you're about to analyse (e.g. BORN), and your name, for the record.

Then we will skip to the last part of the script, which we show in Fig. 2. Here we will edit a number of entries:

- **dataroot**: where the parent data folder is, relative to where the configuration file is.
- **datapath**: you can leave this with the default values.
- **cdppath**: a folder where you'll put some calibration data-products needed by the pipeline (namely the cosmetics masks, by now).
- **resroot** : the parent folder where the results from the pipeline will be written. The session folders will hang from this path. Usually this will be a symbolic link named "results" to a folder in euclid_caldata0X/data0X/results (exact location depends on block).
- **BLOCKID**: The block nickname.
- **CHAMBER**: What chamber profile we're using. Depends on what chamber we're testing in, obviously. This is a critical parameter, because depending on this we'll use the right exposure times, or not.
- **diffvalues**: in this Python dictionary we provide the pipeline with information about the hardware under testing. This is, the serials of the ROE, RPSU and CCDs. If the serials do not match those provided at the time of writing the acuisition scripts, the pipeline will report about it, but will still analyse the data.
- **inCDPs**: if you have cosmetics masks computed for the detectors, here's where you tell the pipeline where to find them.

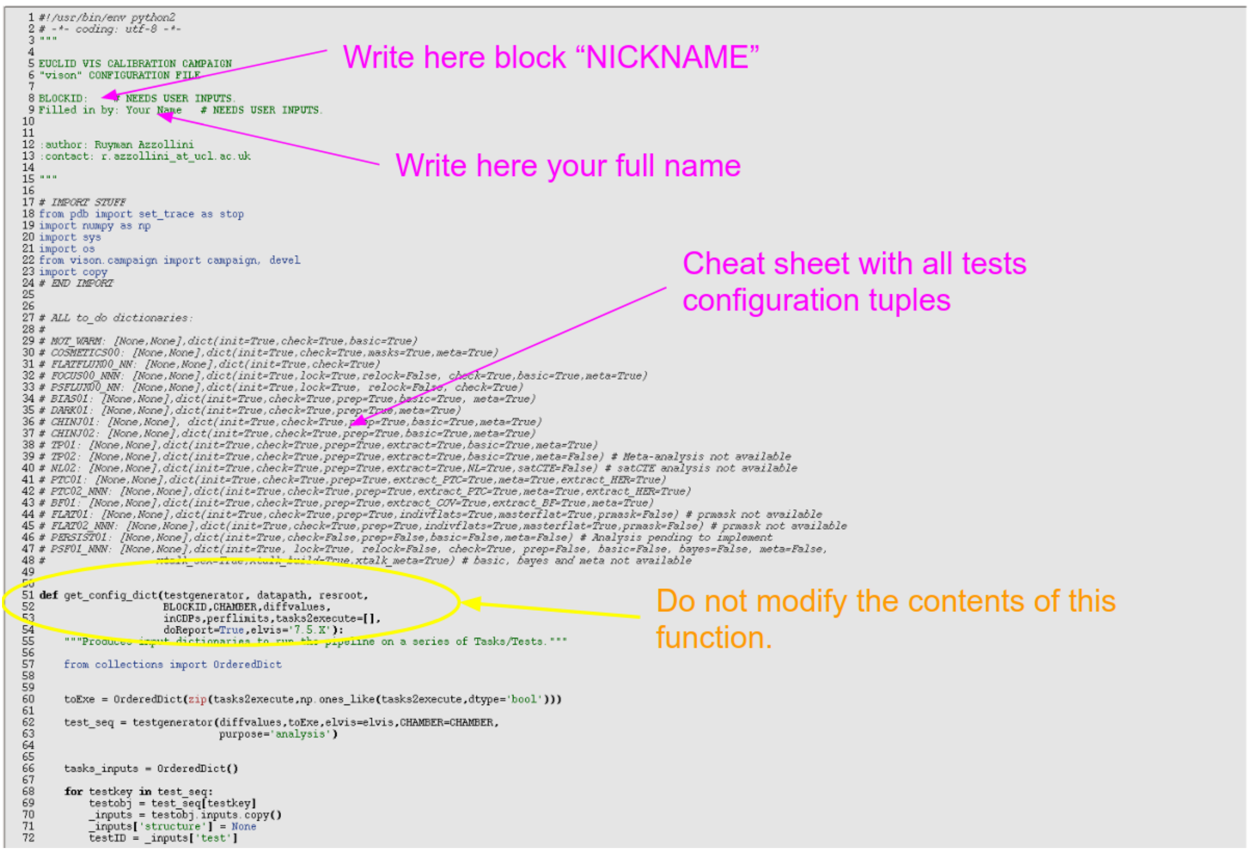


Fig. 3.1: Fig. 1: First part of the configuration file (edited first). Here you have to introduce the BLOCKID "nickname", as corresponding (e.g. "BORN"), and your name, for the record (the configuration file will be saved as part of the record of what has been done). In the commented-out area you have a cheat-sheet with the todo_flags for each test.

Then, there is the dictionary “tasks2execute_dict”, where you tell what “tasks” are to be executed within each session. Each “task” corresponds to a “test”. But, as you will see, **the task names in this dictionary do not exactly correspond to the test names**, but they are shorter and/or slightly different. This is because some of the tests are particular “instances” (using this Python term loosely here) of more general tests which can have different wavelengths or other characteristics. In this dictionary we provide “handles” to those more generic test types. The correspondence between test names and the generic handles is given in the following table.

Table 3.1: Correspondence between specific test names and generic test handles.

Action	Comment
“TEST/TASK” NAME”	**GENERIC TEST HANDLE**
MOT_WARM	MOT_WARM
BIAS01	BIAS01
BIAS02	BIAS02
DARK01	DARK01
COSMETICS00	COSMETICS00
FOCUS00_NNN [NNN=wavelength in nm]	FOCUS00
PSFLUX00_NNN [NNN=wavelength in nm]	PSFLUX00
FLATFUX00_NNN [NNN=wavelength in nm]	FLATFLUX00
CHNJ01	CHINJ01
CHINJ02	CHINJ02
TP11	TP11
TP21	TP21
NL02	NL02
PTC01	PTC01
PTC02_NNN [NNN=wavelength in nm]	PTC02WAVE
FLAT01	FLAT01
FLAT02_NNN [NNN=wavelength in nm]	FLAT02
BF01	BF01
BF01_NNN [NNN=wavelength in nm]	BF01WAVE
PSF01_NNN [NNN=wavelength in nm]	PSF01
PERSIST01	PERSIST01

Then we proceed to the third part to edit, which is actually in the ~middle of the script, and is shown in Fig. 3. What we have to edit is the first part of the function “add_RUN_specifics” (around line 113 in the script).

Because of its verbosity, this part of the configuration file is probably where it’s more likely that we’ll make errors that will lead to not getting results, or at least not those we hope for out of the pipeline. So tread with even more care here than in the previous sections.

In this section you “only” have to edit the contents of the dictionary “test_specifics”. In it, there are all the sessions which will compose this run. For each of these sessions (named DNN, like D11), there is a list of lists, each corresponding to a specific test.

Here, the tests are named by their specific name, not the generic ones. So, for example, including the wavelength, where it applies.

For each test, there’s an entry with the following structure:

```
['TEST_NAME', [Obsid_start, Obsid_end], dict(init=True,check=True,basic=True)]
```

Let’s break up these entries:

- **TEST_NAME** is just that, the test name (e.g. BIAS01).
- The second item is a list, which contains the first and last obsid for the test data. When we’re running the pipeline in “listening/wait” mode we won’t know this in advance, and this list will have the value: [None, None]

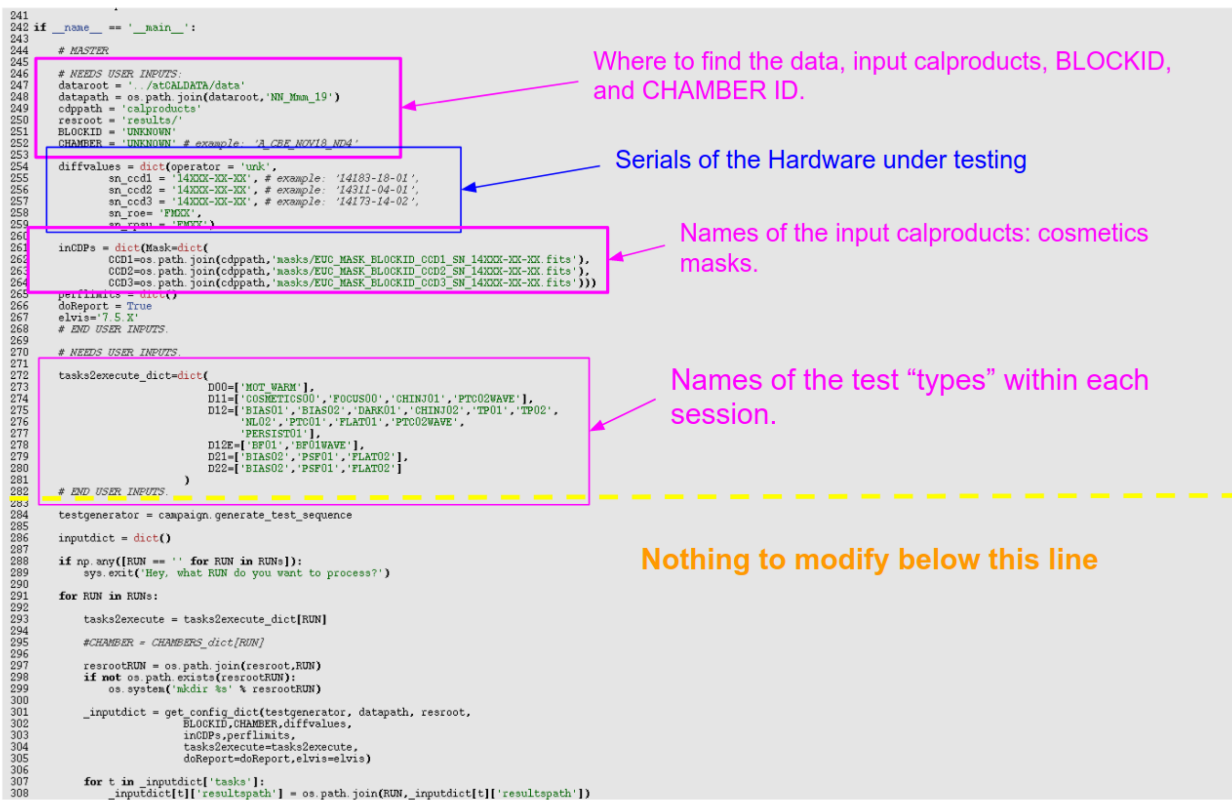


Fig. 3.2: Fig. 2.: Third/last part of the configuration file (edited second).

(None is a valid variable in Python which has a "null" value).

- The third item is a dictionary, called the "todo_flags". This tells the pipeline which sub-tasks within the task to execute. And as you may expect, "True" means you want the sub-task executed, and "False", that you don't. This allows for fine control of what we want to do for each test, and is most convenient as we may just want to do part of the analysis at a given time, or we may not want to repeat all the analysis for a test, but just part of it, when rerunning the pipeline.

  You'll find templates of how the specific inputs for each test look like at the top of the script (commented lines).

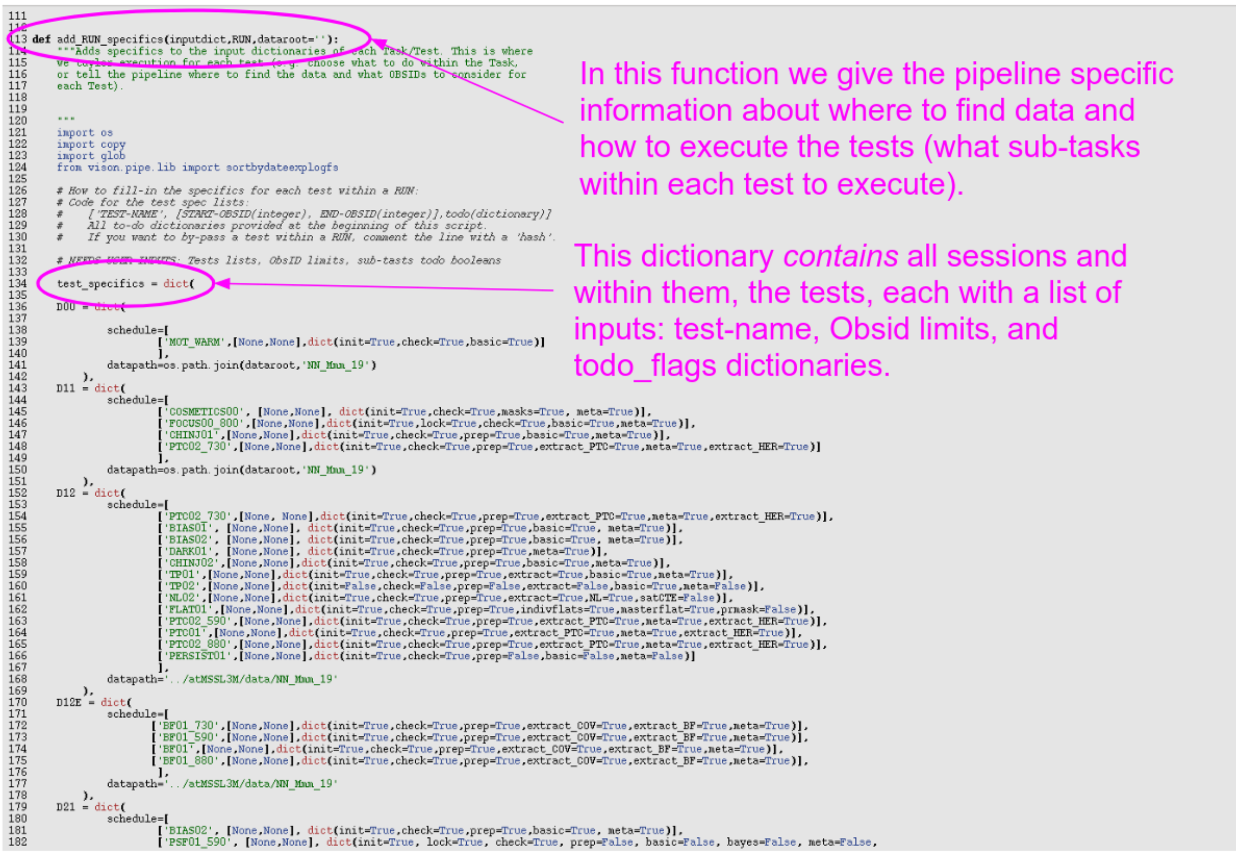


Fig. 3.3: Fig. 3. Second part of the configuration file, to be edited last.

Executing the pipeline

As we said above, we can run the pipeline in several "modes". Here we describe briefly what are these modes, and provide a high-level description of the inputs in each case, leaving the details for the recipes section below.

- **"Wait" mode**
  - In this mode, the pipeline is launched as soon as the data acquisition of a session starts, and the pipeline will process the data of each test as soon as it is completed.
  - This is the quickest way to obtain check/data-quality assessments of the tests.
  - **Basically we have to give the pipeline (via the configuration file and keyword inputs to the command):**
    - a location where to find the data (the path to the "data" folder)

    * A location where to store results.
    * Where to find calibration data products, if needed.
    * A BLOCKID and a CHAMBER identifier.
    * Serials for the hardware,
    * Names of the input calibration data products (e.g. cosmetics masks), if needed.
    * A test list.
    * Optionally, test-specific todo_flags (via the entries in the test_specifics dictionary within add_RUN_specifics function) if we're not just "checking" on all tests using option -k.

- **"Aim" Mode**
  - In this mode we tell the pipeline where to find the data, with specific starting and ending obsids, and, in general, we'll aim at doing more extensive analysis of the data. But the real difference with the previous mode is that the pipeline will not be waiting for the data, but rather assume it's already there, and it is told exactly where to find it.
  - This is the mode we'll use to produce the end-results of the calibration.
  - **We have to give the following inputs in this mode:**
    * a location where to find the data, including the day-folder, via the test_specifics dictionary within add_RUN_specifics function.
    * A location where to store results ("resroot").
    * Where to find calibration data products, if needed.
    * A BLOCKID and a CHAMBER identifier.
    * Serials for the hardware,
    * Names of the input calibration data products (e.g. cosmetics masks), if needed.
    * A test list, and a filled-in "test_specifics" dictionary in add_RUN_specifics.

Advanced: using "nohup"

In general, and in particular the windows users, we'll run the pipeline in a remote machine, via ssh, or Putty. But, if we close the window from which we run the "vison_run" command, the pipeline will halt. To prevent that from happening, we can use "nohup", that will let the process(es) running even if the window is terminated (because, for example, the connection is terminated). This is simple to do:

```
(vison) bash$ nohup vison_run -y .... > nohup_DNN
```

We have just preceded the pipeline executing command with the command "nohup", and appended piping the on-screen outputs to a text file called nohup_DNN (NN can be the session number, for example), so we that screen output is not lost.

Pipeline Feedback and Results

When we run the pipeline we will want to know a) that the pipeline is actually running, and b) what progress has been made.

The on-screen output of the pipeline, at start, is quite limited. It will just state something along the lines of:

```
(vison) bash$ vison_run -y ...
Running: TESTNAME
```

Where TESTNAME will be the name of one of the tests you've asked the pipeline to run upon. Most of the times, if the inputs are wrong, the pipeline will end abruptly on start, with a list of Python errors written to the screen, and/or the pipeline log. But sometimes, these errors on the inputs won't be noticed until much later (for example, if they're specific to a test, and not general to the overall pipeline session).

The most informative document will be the pipeline log, which will be created everytime we run the pipeline with the "-l" option. The name of the pipe-log is of the type:

Calib_FM20190323_193647_D22.log

Where the first 2 numbers are the data and time when we started the pipeline execution, and the _D22 comes from the tag option at running the pipeline ("-t _D22").

Let's look at the contents of the pipe-log file, highlighting the most important parts:

```
2019-03-23 19:36:47,504 - INFO - _
Starting FM Calib. Pipeline
Pipeline ID: FM20190323_193647_D22
BLOCK ID: BORN
Chamber: B_CBE_MAR19
vison version: 0.8+51.g5301278
Tasks: ['PSF01_730', 'PSF01_800', 'PSF01_880']

2019-03-23 19:36:47,506 - INFO -   Results will be saved in: results_atCALDATA/

2019-03-23 19:36:47,507 - INFO -   Running Task: PSF01_730
Inputs:
perflimits = {}
elvis = 7.5.X
BLOCKID = BORN
OBSID_lims = [17968, 18020]
offsetxy = [0.0, 0.0]
CHAMBER = B_CBE_MAR19
preprocessing = {'offsetkwargs': {'ignore_pover': True, 'trimscan': [25, 5],
'method': 'row', 'extension': -1, 'scan': 'pre'}}
frames = [20, 15, 10, 4, 4]
[...]
2019-03-23 19:36:47,729 - INFO - Executing xtalk_sex: vison.point.PSF0X

2019-03-23 19:46:27,062 - INFO - 9.7 minutes in running Sub-task: xtalk_sex

2019-03-23 19:46:27,133 - INFO - Executing xtalk_build: vison.point.PSF0X

2019-03-23 19:50:06,835 - INFO - 3.6 minutes in running Sub-task: xtalk_build

2019-03-23 19:50:06,916 - INFO - Executing xtalk_meta: vison.point.PSF0X

2019-03-23 19:50:10,906 - INFO - 0.1 minutes in running Sub-task: xtalk_meta

2019-03-23 19:51:18,023 - INFO - Finished PSF0X

2019-03-23 19:51:18,024 - INFO - 14.5 minutes in running Task: PSF01_730

2019-03-23 19:51:18,024 - INFO - Task PSF01_730 exited with Errors: False

2019-03-23 19:51:18,026 - INFO - _
```

As you can see, there's a first part where the general parameters of the pipeline execution are listed (block ID, chamber ID, tasks to be executed), and then comes, for each test, a listing of its inputs, and then reports of progress within the test/task.

Every time the pipeline finishes processing one test/task, it prints to the log a report of progress with a few lines on each executed task. At the end there will be a similar report with all the tasks executed:

```
######################################################################
Pipeline ID: FM20190323_193647_D22
BLOCK ID: BORN
Chamber: B_CBE_MAR19
vison version: 0.8+51.g5301278
Tasks: ['PSF01_730', 'PSF01_800', 'PSF01_880']
_
PSF01_730
Executed in 14.5 minutes
Raised Flags = ['FLUENCE_OOL', 'FOCUS_OOL', 'RON_OOL', 'HK_OOL', 'FLUX_OOL',
'POORQUALDATA']
_
PSF01_800
Executed in 14.8 minutes
Raised Flags = ['FLUENCE_OOL', 'FOCUS_OOL', 'RON_OOL', 'HK_OOL', 'FLUX_OOL',
'POORQUALDATA']
_
PSF01_880
Executed in 15.1 minutes
Raised Flags = ['FLUENCE_OOL', 'FOCUS_OOL', 'RON_OOL', 'HK_OOL', 'FLUX_OOL',
'POORQUALDATA']
_
_
_
_
######################################################################
```

If the test fails for any reason, then the pipeline will say "Executed with Error(s)" for that specific test in these reports.

The best way to consult this pipe-log is using "nedit", or just doing, from the command line:

```
(vison) bash $ cat Calib_FM20190323_193647_D22.log
```

Quick Reference on running the pipeline

Asking for Help

```
(vison) bash $ vison_run -h

Usage: vison_run [options]

Options:
    -h, --help            show this help message and exit
    -y PYCONFIG, --pyconfig=PYCONFIG
                        Python configuration file to run the pipeline.
    -R RUN, --run=RUN     Run to process - TESTS.
    -j JSON, --json=JSON  Json configuration file to run the pipeline.
    -d DAYFOLDER, --day=DAYFOLDER
                    Day-folder. Only needed in 'wait' mode.
    -v ELVIS, --elvis=ELVIS
                    ELVIS vrsion. Only needed in 'wait' mode.
```

```
    -W, --wait            Run in 'data-waiting/listening' mode.
    -k, --check           Check consistency and basic quality of data only.
    -l, --log             Start an Execution Log.
    -r, --drill           Do a drill execution.
    -g, --debug           Run in 'debug' mode.
    -T, --test            Run in 'test' mode (just ingest inputs and initialize
                        pipeline and output directories)
    -O STARTOBSID, --ObsID=STARTOBSID
                        Only use data from given ObsID and onwards. Only used
                        in 'wait' mode.
    -m MULTITHREAD, --multithread=MULTITHREAD
                        Use multithreading? Number of threads / cores to use.
                        Default=1 [single thread]. Number of threads must be <
                        number of available cores. Only some tasks are
                        parallelized.
    -t TAG, --tag=TAG     Tag to be added to log-file name for ease of
                        identification. Optional.
    -i, --interactive     Switch matplotlib to TkAgg backend for interactive
                        plotting (e.g. debugging)
```

Options to execute the pipeline using "vison_run" console command / script.

Listening/Wait Mode

To run the pipeline in "listening mode" (-W) doing only test "checks" (-k):

```
(vison) bash $ vison_run -y vison_config_BLOCK_MMM19.py -R RUN -d path_to_dayfolder -
↪v ELVIS_VERSION -W -k -l [-O STARTOBSID] -t TAG -m NTHREADS


-y vison_config_BLOCK_MMM19.py
    Configuration file. It's a python script itself.
-R RUN
    Which "RUN" to consider in the configuration file. This will restrict the list of␣
↪tests we're expecting data from in this session.
-d path_to_dayfolder
    Day-folder where the data is being stored for the session.
-v ELVIS_VERSION
    ELVIS version we're using (7.5.X is should work for FM).So write "-v 7.5.X"
-W
    Executing in "listening" mode. The pipeline knows what tests to expect, and their␣
↪structure. Only when it sees that the data for a test has been acquired, starts the␣
↪processing of that data-set. It waits a maximum of 4 hours since launching␣
↪application or the last test was completed. Then asks whether you want to abort␣
↪pipeline listening, as the data acquisition may have been interrupted or aborted.
-k
    Only executes the "check" subtask of each test. In tests with point source, it␣
↪will also do "source locking" before doing any checks, to be able to find the␣
↪sources. If you omit this option, then the todo_flags in "add_RUN_specifics"␣
↪function within the python configuration script.
-l
    Log pipeline execution results. This is to write a .log file with outputs from␣
↪the pipeline used to monitor progress with the pipeline execution. Each test has␣
↪its own test report.
-O STARTOBSID
    Use this option if you want to start from a particular OBSID. This is useful when␣
↪we resubmit a session after a failure. If a test has 15 frames and the pipeline␣
↪finds, say, 19, 4 from a previous aborted submission, and 15 from the second␣
↪submission, it won't identify the test as completed and won't analyse the data.␣
↪This option can be used to correct this, telling the pipeline to start the progress␣
↪tracking from the OBSID that corresponds to the 5th frame of that test, in this␣
↪example.
```

```
-t tag
    Use if we want to add a text tag to be added to the pipeline .log file name, for␣
↪ease of identification. Suggestion: use the RUN name: D00, D11, D12, etc.
-m NTHREADS
    Use multithreading. NTHREADS is an integer < number of CPU cores in the machine␣
↪we're using to process the data. The multithreading is only used in selected␣
↪subtasks. If you're only "checking", this is indeed not used.
```

NOTE: You can also run the pipeline in "Wait" mode and use the todo_flags for each test assigned in "add_RUN_specifics", if you just omit the "-k" option.

Aim Mode

To run the pipeline in "aim mode":

```
bash $ vison_run -y vison_config_BLOCK_MMM19.py -R RUN -l -m NTHREADS -t TAG
-y vison_config_BLOCK_MMM19.py
    Configuration file. It's a python script itself.
-R RUN
    Which "RUN" to consider in the configuration file. This will restrict the list of␣
↪tests we're expecting data from in this session.
-l
    Log pipeline execution results. This is to write a .log file with outputs from␣
↪the pipeline used to monitor progress with the pipeline execution. Each test has␣
↪its own test report.
-m NTHREADS
    Use multithreading. NTHREADS is an integer < number of CPU cores in the machine␣
↪we're using to process the data. The multithreading is only used in selected␣
↪subtasks.
-t tag
    Use if we want to add a text tag to be added to the pipeline .log file name, for␣
↪ease of identification. Suggestion: use the RUN name: D00, D11, D12, etc.
```

Estimated Processing Times

Here we provide some estimates of how long does it take the pipeline to process tests. These are based on running the pipeline in the MSSLA[N/O/P] machines, using multithreading (with 6 cores). But notice that only some sub-tasks in some tests can be run in "multi-thread" mode.

Table 3.2: Aprox. test processing times

Action	Comment		
TEST	ONLY CHECKS	FULL ANALYSIS	Comments
•	minutes	minutes	
MOT_WARM	3	4	
COSMETICS00	3	6	
FOCUS00	7	10	
PSFLUX00	7	N/A	
FLATFLUX00	3	N/A	
BIAS01/02	3	11	
DARK01	3	6	
CHINJ01	13	50	
CHINJ02	11	37	
TP01	23	92	
TP02	6	11	
NL02	10	32	
PTC01	9	26	
PTC02WAVE	4	10	Fewer frames than PTC01
FLAT01	14	97	
FLAT02	10	63	Fewer frames than FLAT01
BF01	9	86	
BF01WAVE	3	28	Fewer frames than BF01
PSF01	35	57	Task Analysis Incomplete!
PERSIST01	3	7	

3.1.5 Real-Time Data Acquisition Monitoring: "Eyegore"

Vison's data acquisition monitoring tool

Eyegore is part of the calibration pipeline, vison. It can be used to monitor data acquisition, as it happens: does automatic HK plots, checks HK against HK limits, and displays the EXPLOG as it grows. Also, it can be used to copy the data to a backup drive, and to a server "behind" the firewall so that the data can be accessed by selected people for independent and external data acquisition monitoring (e.g. by Ralf Kohley at ESAC/Madrid).

Asking for help

```
> eyegore -h

Usage: eyegore [options]

Options:
    -h, --help            show this help message and exit
    -p PATH, --path=PATH  day-path to be monitored.
    -B BROADCAST, --broadcast=BROADCAST
                        Synchronize data to gateway folder at msslus
    -E ELVIS, --elvis=ELVIS
                        ELVIS version.
```

```
    -b, --blind           Run without image displays.
    -L, --lite            Run a lighter version of the program (no image
                        displays and no ExpLog).
    -r ALTPATH, --rsync=ALTPATH
                    rsync to an alternative local path.
    -g, --log             keep a log
    -W, --Warnings        Raise warnings (via email and/or phone) if critical HK
                    is OOL.
```

Use Case 1

We just want to monitor the data acquisition, doing a backup, but without broadcasting to outside world. We also don't want to send warnings via email/sms if HK temps are OOL.

```
> eyegore -p ../atMSSL3M/data/DD_Mmm_YY -E 7.5.X -g -r atCALDATA/data
```

Notes:

- atMSSL3M is a symbolic link to the right folder for this run/BLOCK in MSSL3M/EEDisk5.
- Note the path (-p) has the date-folder included, but the remote (-r ALTpath) does not.
- -g does writes a log which can be helpful to find out what happened overnight.

Use Case 2

We just want to monitor the data acquisition, doing a backup, and also broadcasting to outside world via msslus server. We also don't want to send warnings via email/sms if HK temps are OOL.

- Note 1: currently, only user "raf" can do the broadcasting in this way.
- Note 2: the folder for the BLOCK in msslus where the data is to be copied across has to be created in advance to running eyegore in this way.

```
> eyegore -p ../atMSSL3M/data/DD_Mmm_YY -E 7.5.X -g -r ../atCALDATA/data -B /data2/
↪gaia/usdownloads/EuclidCaldata/Quarantine/BLOCKNAME/data
```

Use Case 3

Like Use Case 3, but we also want to issue warnings via email/sms.

Note: Sending sms can only be done by user "raf", by now.

```
> eyegore -p ../atMSSL3M/data/DD_Mmm_YY -E 7.5.X -W -r ../atCALDATA/data -B /data2/
↪gaia/usdownloads/EuclidCaldata/Quarantine/BLOCKNAME/data
```

3.1.6 Quickds9: quick-loading many images into ds9

When inspecting test data, it is very handy to use the SAO DS9 program, which allows to display the image with different options for scaling and palette, do area statistics in regions of different shapes, extract profile cuts, and many other useful things. In the pipeline what we have added is a script to load a number of images, which you can use to, for example, load all the images in a given test, to inspect them visually. The script is called "quickds9.py".

The script can only load images of one CCD at a time, which you have to specify, and only works with consecutive OBSIDs. It may be used to load images output by ELVIS, and also images produced with other programs, as long as the image names conform to some basic rules (and we provide a template for the name), but here we will only give a recipe to load calibration campaign images written by ELVIS.

Use case: We want to load all images between OBSIDs 1000 and 1020, from a day folder 21_Feb_80, selecting CCD 2. Here's how to load all those in DS9 images using the script:

```
(vison) bash-4.2$ quickds9.py -p ../atCALDATA/data/21_Feb_80 -r 1 -c 2 1000 1020
-p ../atCALDATA/data/21_Feb_80
    Where the images are.
-r 1
    ROE selected (always 1 for Cal. Camp data).
-c 2
    CCD selected.
1000 1020
    First and last OBSID to load.
```

WARNING: You may encounter problems when using the program if there are several instances of DS9 running (for example, launched by eyegore, or yourself). You may tell the program what instance of DS9 to use, using the option "-d" and then the ds9 instance ID. This you can get from the DS9 window itself: In the above menu, click on File> XPA > Information. Then, a window will pop-up with a message like this:

```
XPA_VERSION:    2.1.17
XPA_CLASS:  DS9
XPA_NAME:   ds9
XPA_METHOD: 802847c3:37111
```

Then to select this specific instance, you would have to call quickds9.py using the highlighted number, like this:

```
(vison) bash-4.2$ quickds9.py -p ../atCALDATA/data/21_Feb_80 -r 1 -c 2 -d ␣
↪802847c3:37111 1000 1020
```

If you want more help, you can just type:

```
(vison) bash-4.2$ quickds9.py -h
Usage: quickds9.py [options] arg1 [arg2]
arg1: starting OBSID
[arg2]: optional, end OBSID. If not provided, arg2==arg1.

Options:
    -h, --help            show this help message and exit
    -p PATH, --path=PATH  path where to look for FITS files. Search is not
                    recursive.
    -r ROE, --roe=ROE     ROE to select
    -c CCD, --ccd=CCD     CCD to select
    -t TEMP, --template=TEMP
                    Image Name Template
    -d DS9TARGET, --DS9=DS9TARGET
                    Specify DS9 target (pyds9)?
```

3.1.7 PRACTICAL CASE: "NOSTROMO"

In this section we will try to guide you through some practical cases covering the writing of acquisition scripts and performing data analysis on sample data, using the pipeline.

We'll use a fake block, called "NOSTROMO" (you know where this is going…), as a working example. The data we will process is part of the data that was acquired with block "BORN/FM1" in March 2019.

ENVIRONMENT SETUP

First we will connect to MSSLAP (using ssh or Putty), and activate the pipeline. We also check that we're using the pipeline installed in euclid_cadata06/

```
[raf@msslap ~]$ bash
bash-4.2$ source /disk/euclid_caldata06/data06/SOFTWARE_LITE/bashrc_lite
bash-4.2$ source activate vison
(vison) bash-4.2$ which vison
/disk/euclid_caldata06/data06/SOFTWARE_LITE/anaconda2/envs/vison/bin/vison
```

Now we create a folder structure to write the scripts and and process the data. I trace my steps from my $HOME folder.

```
(vison) bash-4.2$ pwd
/home/raf
(vison) bash-4.2$ mkdir CALCAMP
(vison) bash-4.2$ cd CALCAMP
(vison) bash-4.2$ pwd
/home/raf/CALCAMP
(vison) bash-4.2$ mkdir TRAIN
(vison) bash-4.2$ cd TRAIN/
(vison) bash-4.2$ mkdir NOSTROMO
(vison) bash-4.2$ cd NOSTROMO
(vison) bash-4.2$ mkdir ANALYSIS
(vison) bash-4.2$ mkdir SCRIPTS
(vison) bash-4.2$ ls
    ANALYSIS  SCRIPTS
```

Finally, we also create a link to the "block" folder in caldata so that the paths to data and results are conveniently short, when doing analysis. In the following box, we show how to do this, and including some commands to check we've done it correctly.

```
(vison) bash-4.2$ pwd
/home/raf/CALCAMP/TRAIN/NOSTROMO
(vison) bash-4.2$ ls
ANALYSIS  SCRIPTS
(vison) bash-4.2$ ln -s /disk/euclid_caldata06/data06/TRAIN/NOSTROMO atCALDATA
(vison) bash-4.2$ ls
    ANALYSIS  atCALDATA  SCRIPTS
(vison) bash-4.2$ readlink atCALDATA
    /disk/euclid_caldata06/data06/TRAIN/NOSTROMO
(vison) bash-4.2$ ls atCALDATA/
    calproducts data  solutions
```

Writing Acquisition Scripts for NOSTROMO

We are going to create the scripts first. First we copy the template script-writer and session-writer input json files, and rename them to something that makes sense (for this example).

```
(vison) bash-4.2$ cd SCRIPTS
(vison) bash-4.2$ pwd
```

```
    /home/raf/CALCAMP/TRAIN/NOSTROMO/SCRIPTS
(vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/
↪SCRIPTS/scripts_inputs_BLOCK_MMM19_WN.json .
(vison) bash-4.2$ mv scripts_inputs_BLOCK_MMM19_WN.json scripts_inputs_NOSTROMO_APR19_
↪W18.json
(vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/
↪SCRIPTS/sessions_builder_BLOCK_MMM19_WN.json .
(vison) bash-4.2$ mv sessions_builder_BLOCK_MMM19_WN.json sessions_builder_NOSTROMO_
↪APR19_W18.json
(vison) bash-4.2$ ls
    scripts_inputs_NOSTROMO_APR19_W18.json  sessions_builder_NOSTROMO_APR19_W18.json
```

First we edit the json to write scripts with the following inputs, but (VERY IMPORTANT) following the formatting of the template json:

```
ROE: FM15
RPSU: FM15
CCD1: 14888-88-81
CCD2: 14888-88-82
CCD3: 14888-88-83
Elvis: 7.5.X
CHAMBER: B_NOSTROMO
Outpath: Scripts_NOSTROMO_APR19_W18

Tests to be done:
FOCUS00, BIAS01, BIAS02, CHINJ01, CHINJ02, TP01
```

Then we execute the script-writer with the modified json file as input:

```
(vison) bash-4.2$ vis_mkscripts.py -j scripts_inputs_NOSTROMO_APR19_W18.json

WRITING SCRIPTS...
CHINJ02...
BIAS01...
TP01...
BIAS02...
CHINJ01...
FOCUS00_590...
FOCUS00_730...
FOCUS00_800...
FOCUS00_880...
```

These are the contents of the newly created folder, Scripts_BLOCK_APR19_w18:

```
(vison) bash-4.2$ ls -1 Scripts_NOSTROMO_APR19_W18/
    CHECK_SUMS_01Apr19.txt
    TESTS_INVENTORY_01Apr19.txt
    TESTS_SCHEDULER_01Apr19.xlsx
    vis_CalCamp_BIAS01_01Apr19_v7.5.X.xlsx
    vis_CalCamp_BIAS02_01Apr19_v7.5.X.xlsx
    vis_CalCamp_CHINJ01_01Apr19_v7.5.X.xlsx
    vis_CalCamp_CHINJ02_01Apr19_v7.5.X.xlsx
    vis_CalCamp_FOCUS00_590_01Apr19_v7.5.X.xlsx
    vis_CalCamp_FOCUS00_730_01Apr19_v7.5.X.xlsx
    vis_CalCamp_FOCUS00_800_01Apr19_v7.5.X.xlsx
    vis_CalCamp_FOCUS00_880_01Apr19_v7.5.X.xlsx
    vis_CalCamp_TP01_01Apr19_v7.5.X.xlsx
```

And the test inventory (TESTS_INVENTORY_01Apr19.txt) should look like:

```
(vison) bash-4.2$ cat Scripts_NOSTROMO_APR19_W18/TESTS_INVENTORY_01Apr19.txt
Scripts written on 01Apr19 12:56:31
checksumf: CHECK_SUMS_01Apr19.txt
vison version: 0.8+69.gf135883

CHINJ02: 26 [42.31 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1]
BIAS01: 1 [15.58 min] cols: [10]
TP01: 50 [120.31 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1]
BIAS02: 1 [15.58 min] cols: [10]
CHINJ01: 36 [58.58 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,␣
↪1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
FOCUS00_590: 9 [15.11 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1]
FOCUS00_730: 9 [14.25 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1]
FOCUS00_800: 9 [14.22 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1]
FOCUS00_880: 9 [14.39 min] cols: [1, 1, 1, 1, 1, 1, 1, 1, 1]

168 Frames Total

310.33 Minutes Total
```

Now we edit the json file to create the sequences (again following the format of the template):

```
inpath: Scripts_NOSTROMO_APR19_W18
outpath: SEQUENCES_NOSTROMO_APR19_W18
Sessions:

    D00: FOCUS00_800
    D11: BIAS01, BIAS02, CHINJ01, CHINJ02, TP01
```

Then we execute the sessions builder with the modified file as input:

```
(vison) bash-4.2$ vis_mksession.py -j sessions_builder_NOSTROMO_APR19_W18.json
session: D11
    vis_CalCamp_BIAS01_01Apr19_v7.5.X.xlsx
    vis_CalCamp_BIAS02_01Apr19_v7.5.X.xlsx
    vis_CalCamp_CHINJ01_01Apr19_v7.5.X.xlsx
    vis_CalCamp_CHINJ02_01Apr19_v7.5.X.xlsx
    vis_CalCamp_TP01_01Apr19_v7.5.X.xlsx
session: D00
    vis_CalCamp_FOCUS00_800_01Apr19_v7.5.X.xlsx
```

And we check the sessions folder has the following contents:

```
(vison) bash-4.2$ find SEQUENCES_NOSTROMO_APR19_W18/
SEQUENCES_NOSTROMO_APR19_W18/
SEQUENCES_NOSTROMO_APR19_W18/D11
SEQUENCES_NOSTROMO_APR19_W18/D11/vis_CalCamp_BIAS01_01Apr19_v7.5.X.xlsx
SEQUENCES_NOSTROMO_APR19_W18/D11/vis_CalCamp_CHINJ02_01Apr19_v7.5.X.xlsx
SEQUENCES_NOSTROMO_APR19_W18/D11/vis_CalCamp_CHINJ01_01Apr19_v7.5.X.xlsx
SEQUENCES_NOSTROMO_APR19_W18/D11/vis_CalCamp_BIAS02_01Apr19_v7.5.X.xlsx
SEQUENCES_NOSTROMO_APR19_W18/D11/TEST_SEQUENCE_D11.txt
SEQUENCES_NOSTROMO_APR19_W18/D11/vis_CalCamp_TP01_01Apr19_v7.5.X.xlsx
SEQUENCES_NOSTROMO_APR19_W18/D00
```

```
SEQUENCES_NOSTROMO_APR19_W18/D00/TEST_SEQUENCE_D00.txt
SEQUENCES_NOSTROMO_APR19_W18/D00/vis_CalCamp_FOCUS00_800_01Apr19_v7.5.X.xlsx
```

Then, we would copy the session folders to the appropiated folder in MSSL3M, and let the test-operator (and campaign director) where to find the scripts for the sessions.

UPDATES to the Scripts

Imagine now that we've executed session D00, which has test FOCUS00_800, and after analysing results, we see a need to update the focus position. Then the campaign director would have to ask the pipeline custodian to modify the focus for this chamber profile (B_NOSTROMO), and we would have to write scripts again for session D11, and build a new session folder D11. We will not cover this case in this exercise though, but we aware that would be a common case when writing (and re-writing) scripts.

Using vison to analyse acquired data

Now we're going to run the pipeline in "Wait" mode on the data of session D00, using the pipeline.

First we move to the ANALYSIS subfolder we created above:

```
(vison) bash-4.2$ ls
ANALYSIS  atCALDATA  SCRIPTS
(vison) bash-4.2$ cd ANALYSIS/
(vison) bash-4.2$ ls
(vison) bash-4.2$
```

Then we copy the template analysis input script from the templates and rename it:

:: (vison) bash-4.2$ cp /disk/euclid_caldata06/data06/SOFTWARE_LITE/TEMPLATES_CALCAMP/ANALYSIS/vison_config_BLOC
 . (vison) bash-4.2$ mv vison_config_BLOCK_MMM19.py vison_config_NOSTROMO_APR19.py

In the "header" of the file vison_config_NOSTROMO_ARP19.py we put the BLOCKID value and the "Filled in by" with the values that apply:

Note: I'm using "Spyder" to edit the configuration script, but you can use nedit, or emacs, as well, if you find it more convenient (almost any text editor would work... as long as it can edit and save ascii without adding formatting code, as WordTM does).

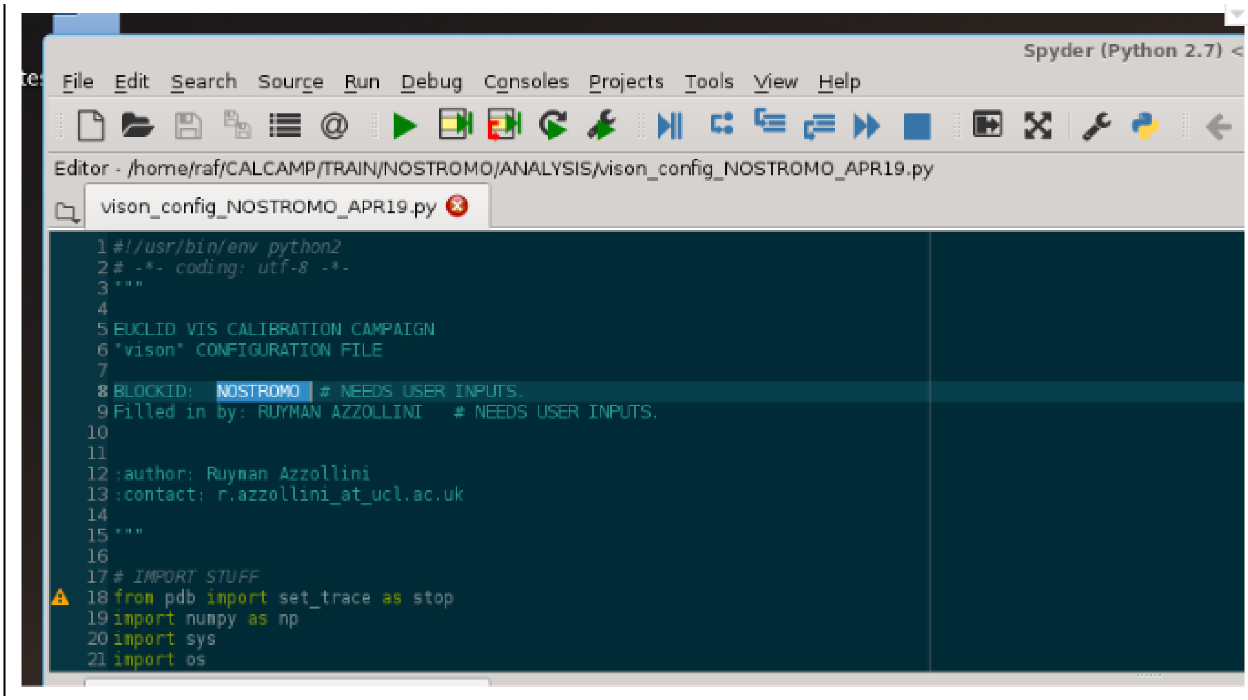
Now we got to the bottom part of the script, where we introduce general inputs about the data and hardware under test. Notice that we only consider two sessions, D00 and D11, following what we assumed when writing the scripts. Also, look at the line numbers to an idea of where in the script you have to do the edits.

Now, you'll have noticed that the results folder, "results_atCALDATA", which is suppossed to be in ANALYSIS (where the vison_config*py file is), hasn't been created. We'll fix that now.

That folder is, in fact, a symbolic link, to a results folder in caldata (we'll see exactly how). But as several of you are going to follow this exercise, you'll have to create a dedicated folder in euclid_caldata, so that you don't collide with each other when following this exercise.

So, I'll create a results folder in euclid_caldata, adding my initials to it, so it's "unique".

```
(vison) bash-4.2$ pwd
/home/raf/CALCAMP/TRAIN/NOSTROMO/ANALYSIS
(vison) bash-4.2$ ls
    vison_config_NOSTROMO_APR19.py
(vison) bash-4.2$ ls ../atCALDATA/
```




```
    data  solutions
(vison) bash-4.2$ mkdir ../atCALDATA/results_raf
```

By the way, remember the ../atCALDATA folder is in fact a symbolic link:

```
(vison) bash-4.2$ readlink ../atCALDATA
    /disk/euclid_caldata06/data06/TRAIN/NOSTROMO
```

Now we create a symbolic link to the results_xxx folder (substitute xxx with your 3 letters linux user id), in the ANALYSIS folder (I'll use results_raf in this example, as I'm user raf):

```
(vison) bash-4.2$ ln -s ../atCALDATA/results_raf results_atCALDATA
(vison) bash-4.2$ ls
    results_atCALDATA  vison_config_NOSTROMO_APR19.py
(vison) bash-4.2$ readlink results_atCALDATA
    ../atCALDATA/results_raf
```

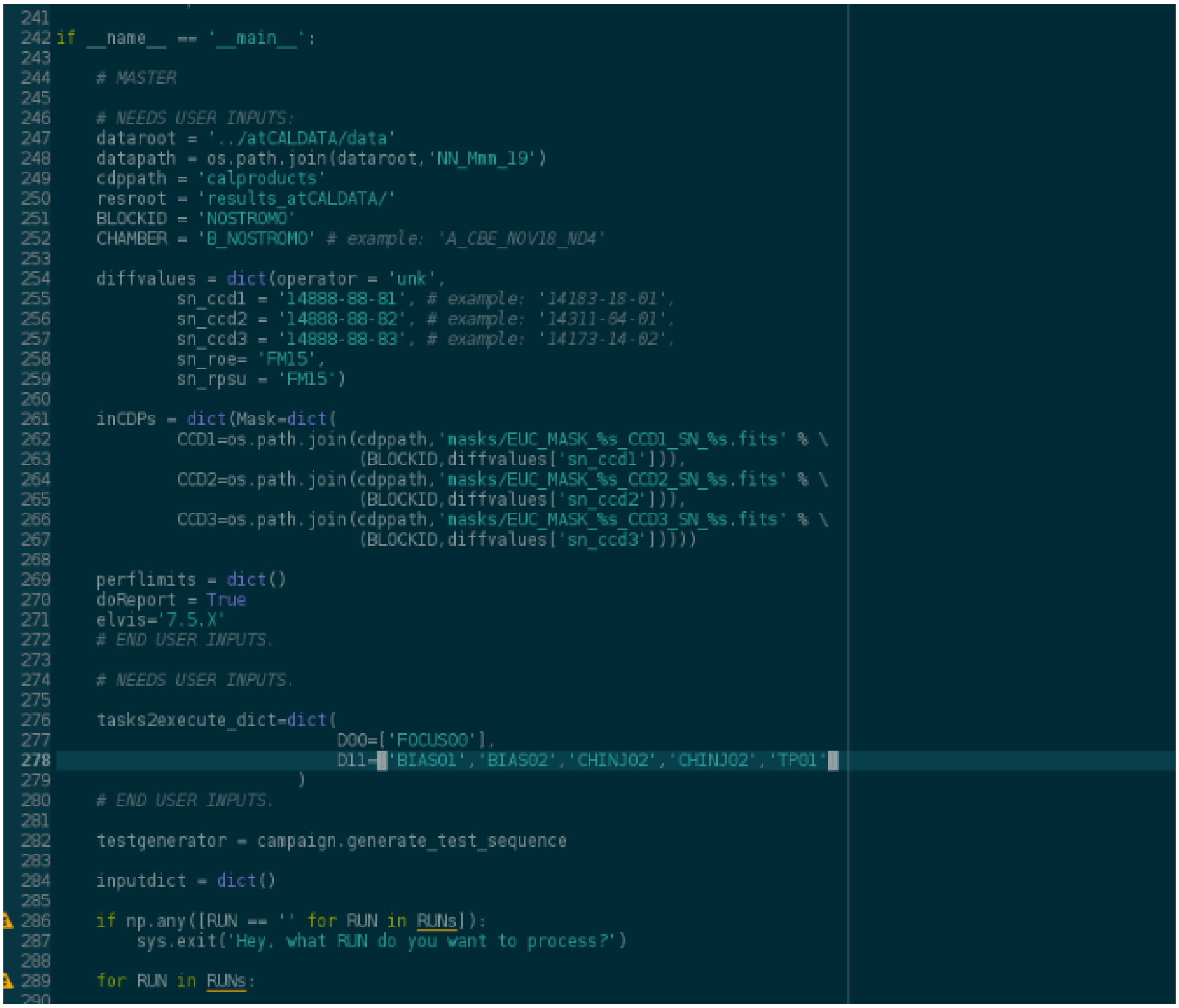
Then we'll check that the test_specifics dictionary in add_RUN_specifics function, for session D00, is correctly configured, and ready to do the FULL analysis:

Session D00: running the Pipeline in "wait" mode

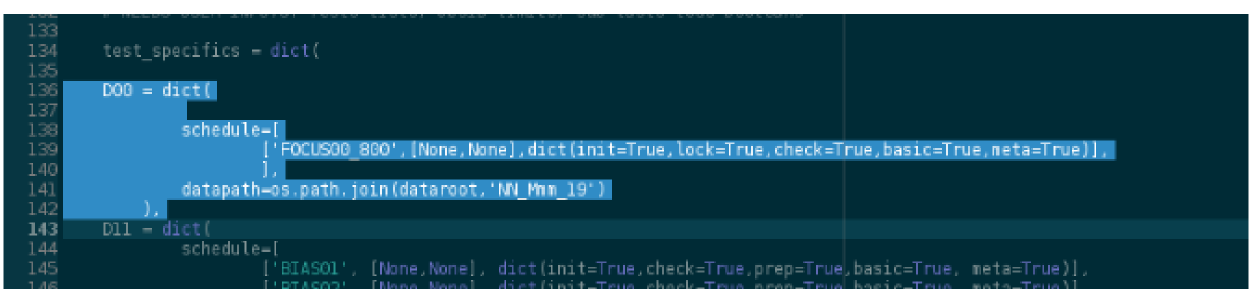
Now we can and will run the pipeline in "wait" mode, using the configuration file we've just edited as input. We're assuming that the data is being copied to caldata from MSSL3M by eyegore, in the background (either we're running that code as well, or somebody else is doing it).

```
(vison) bash-4.2$ vison_run -y vison_config_NOSTROMO_APR19.py -R D00 -d ../atCALDATA/
↪data/12_Mar_19/ -v 7.5.X -W -l -m 6 -t _D00
```

Reminders: * We're telling the pipeline on session D00 (-R D00). * We're telling the pipeline where to look for the data in the command line (-d ../atCALDATA/data/12_Mar_19/). * We're running in "Wait" mode (-W). * We're using



```
if __name__ == '__main__':

    # MASTER

    # NEEDS USER INPUTS:
    dataroot = '../atCALDATA/data'
    datapath = os.path.join(dataroot,'NN_Mmm_19')
    cdppath = 'calproducts'
    resroot = 'results_atCALDATA/'
    BLOCKID = 'NOSTROMO'
    CHAMBER = 'B_NOSTROMO' # example: 'A_CBE_NOV18_ND4'

    diffvalues = dict(operator = 'unk',
            sn_ccd1 = '14888-88-81', # example: '14183-18-01',
            sn_ccd2 = '14888-88-82', # example: '14311-04-01',
            sn_ccd3 = '14888-88-83', # example: '14173-14-02',
            sn_roe= 'FM15',
            sn_rpsu = 'FM15')

    inCDPs = dict(Mask=dict(
            CCD1=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD1_SN_%s.fits' % \
                                (BLOCKID,diffvalues['sn_ccd1'])),
            CCD2=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD2_SN_%s.fits' % \
                                (BLOCKID,diffvalues['sn_ccd2'])),
            CCD3=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD3_SN_%s.fits' % \
                                (BLOCKID,diffvalues['sn_ccd3']))))

    perflimits = dict()
    doReport = True
    elvis='7.5.X'
    # END USER INPUTS.

    # NEEDS USER INPUTS.

    tasks2execute_dict=dict(
                        D00=['FOCUS00'],
                        D11=['BIAS01','BIAS02','CHINJ02','CHINJ02','TP01']
                    )
    # END USER INPUTS.

    testgenerator = campaign.generate_test_sequence

    inputdict = dict()

    if np.any([RUN == '' for RUN in RUNs]):
        sys.exit('Hey, what RUN do you want to process?')

    for RUN in RUNs:
```



```
    test_specifics = dict(

    D00 = dict(

            schedule=[
                ['FOCUS00_800',[None,None],dict(init=True,lock=True,check=True,basic=True,meta=True)],
                ],
            datapath=os.path.join(dataroot,'NN_Mmm_19')
        ),
    D11 = dict(
            schedule=[
                ['BIAS01', [None,None], dict(init=True,check=True,prep=True,basic=True, meta=True)],
```

multithreading (to 6 cores, -m 6).

The output pipeline log will be named something like: Calib_FM20190401_141840_D00.log.

When the pipeline has finished running, it will provide this screen output:

```
  "Unnamed" / no ext. header / 4238x4172 / 32 bits (floats)
(M+D) Background: 2682.43    RMS: 4.38461    / Threshold: 100
  Objects: detected 46       / sextracted 46

> All done (in 2.2 s: 1916.9 lines/s , 21.1 detections/s)
rm: cannot remove ‘sex.param’: No such file or directory
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
Algorithm terminated at max iterations without convergence.
8.9 minutes in running Task: FOCUS00_800
Task FOCUS00_800 exited with Errors: False
```

Then, the pipeline will start waiting for the data from tests FOCUS00_590, _730, and _880, but those will never arrive (remember that we entered “FOCUS00” in the declaration of tasks to be executed).

Just to be sure that the pipeline is just waiting for data that we know won’t never come, we just display the contents of the pipeline log, and loot at the end:

```
(vison) bash-4.2$ cat Calib_FM20190401_141840_D00.log
[...a lot of outputs....]
_
_
_
######################################################################
Pipeline ID: FM20190401_141840_D00
BLOCK ID: NOSTROMO
Chamber: B_NOSTROMO
vison version: 0.8+69.gf135883
Tasks: ['FOCUS00_590', 'FOCUS00_730', 'FOCUS00_800', 'FOCUS00_880']
_
FOCUS00_800
Executed in 8.9 minutes
OBSIDs range = [17210,17218]
Raised Flags = ['FLUENCE_OOL', 'FOCUS_OOL', 'BGD_OOL', 'HK_OOL', 'FLUX_OOL',
'MISSDATA', 'POORQUALDATA']
_
_
_
_
######################################################################

2019-04-01 14:27:37,934 - INFO - Pipeline sleeping for 120 seconds...
```

So, we see the pipeline is “sleeping”, waiting for the data, and what we can do to terminate this early is just “Ctrl+c”.

About the summary of results in pipeline (the part within the long lines of hashes #), we see that: * Task FOCUS00_800 was executed in 8.9 minutes. * The OBSIDs range where the data was found is [17210, 17218]. * The following flags were raised: [’FLUENCE_OOL’, ‘FOCUS_OOL’, ‘BGD_OOL’, ‘HK_OOL’, ‘FLUX_OOL’, ‘MISSDATA’, ‘POORQUALDATA’]

And in results_atCALDATA/D00 you should see the following contents:

```
(vison) bash-4.2$ ls -1 results_atCALDATA/D00/FOCUS00_800/
    figs
    FOCUS00_800_DataDict.pick
    FOCUS00_800_report.aux
    FOCUS00_800_report.dvi
    FOCUS00_800_report.log
    FOCUS00_800_report.out
    FOCUS00_800_report.pdf
    FOCUS00_800_Report.pick
    FOCUS00_800_report.soc
    FOCUS00_800_report.tex
    FOCUS00_800_report.toc
    products
```

Check the contents of the pdf report. The best focus position should be 47.21 mm, and the CBE_FWHM, 2.34 pixels. This is what the FWHM(x) plot should look like:

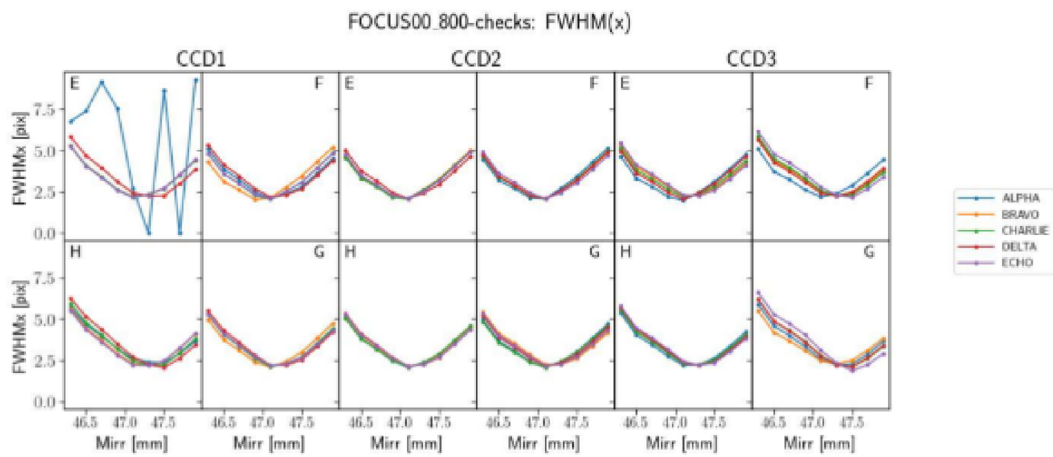


Figure 6.5. FOCUS00_800: FWHM(x) vs. Mirror Position.

The pipeline complained by raising the "MISSDATA" flag. Does it mean that the data-set is incomplete for this test? No, in this case it's because the acquired data has the wrong ccd, roe and rpsu serials, and also exposure times. Remember the input data is actually from another block, BORN (this is just a copy of real data), and in our analysis script we're using fake serials. Also the exposure times are wrong, but that's a known (non-)issue with this data-set in particular.

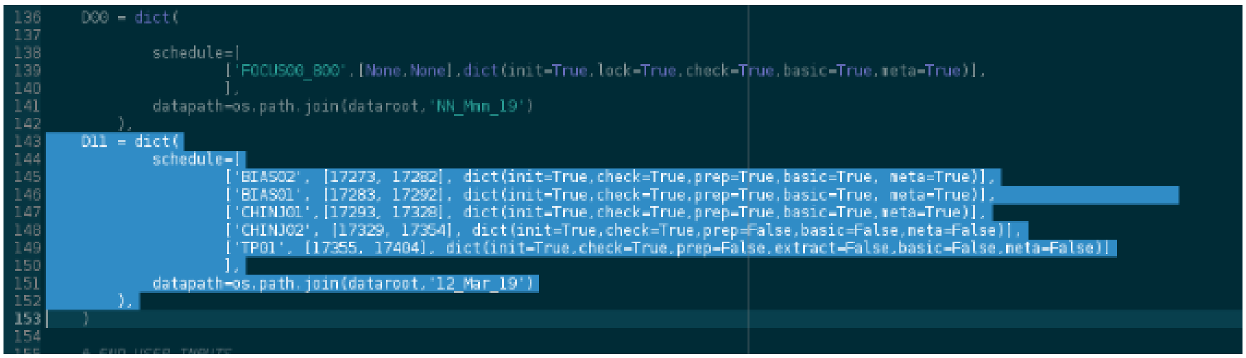
Then, the CCD?_IG1_T/B HK parameters do not match the inputs, but that again was known, and is due to ELVIS using a default HK calibration for the block.

Note: As pipeline "runners", we'd probably update, in the google-sheets test report for the run of NOSTROMO, tab "TESTS_RECORD", the values of OBSID_lims, test-script name, session name and day-folder for the test FOCUS00_800.

Session D11: running the Pipeline in "aim" mode

Now let's run the pipeline in "aim" mode for session D11. This is just an example for didactic purposes; when acquiring the data of session D11 we'd actually run the pipeline in "wait" mode in parallel with the acquisition. But as we already did that in the previous section of the tutorial, we'll now use the "aim" mode instead. The aim mode is the preferred mode we'll use whenever the data has already been acquired.

In the configuration file we just have to edit the test_specifics dict, session D11, in add_RUN_specifics. Use these todo_flags (careful with what is set True, and what's False, and try to predict from them what the pipeline will do given those):



```
    D00 = dict(

            schedule=[
                    ['FOCUS00_800',[None,None],dict(init=True,lock=True,check=True,basic=True,meta=True)],
                    ],
            datapath=os.path.join(dataroot,'NN_Mnn_19')
        ),
    D11 = dict(
            schedule=[
                    ['BIAS02', [17273, 17282], dict(init=True,check=True,prep=True,basic=True, meta=True)],
                    ['BIAS01', [17283, 17292], dict(init=True,check=True,prep=True,basic=True, meta=True)],
                    ['CHINJ01',[17293, 17328], dict(init=True,check=True,prep=True,basic=True,meta=True)],
                    ['CHINJ02', [17329, 17354], dict(init=True,check=True,prep=False,basic=False,meta=False)],
                    ['TP01', [17355, 17404], dict(init=True,check=True,prep=False,extract=False,basic=False,meta=False)]
                    ],
            datapath=os.path.join(dataroot,'12_Mar_19')
        ),
    )
```

As you can see, in some of the test we require to do the image processing (prep=True), and in general, this implies using cosmetics masks, specific to the CCDs at hand. In order for the pipeline to find them, we'll have to link them in the ANALYSIS folder (we have the masks already generated from a previous run on the data-set BORN, for these CCDs).

Let's do that extra link:

```
(vison) bash-4.2$ ln -s ../atCALDATA/calproducts
(vison) bash-4.2$ find calproducts/
    calproducts/
    calproducts/masks
    calproducts/masks/EUC_MASK_NOSTROMO_CCD2_SN_14888-88-82.fits
    calproducts/masks/EUC_MASK_NOSTROMO_CCD3_SN_14888-88-83.fits
    calproducts/masks/EUC_MASK_NOSTROMO_CCD1_SN_14888-88-81.fits
```

NOTE: these masks are the output (renamed to NOSTROMO and fake serials for the tutorial) from COSMETICS00 test. You just have to copy them to a suitable folder (calproducts/masks/) in the block-folder in euclid_caldata, from the "products" subfolder of the COSMETICS00 test-folder, and then link the calproducts folder, as we've done above.

Then we'll just have to run the pipeline, which is done with this (shorter) command:

```
(vison) bash-4.2$ vison_run -y vison_config_NOSTROMO_APR19.py -R D11 -l -m 6 -t _D11
```

This time it will take quite longer to run (~50 minutes). Here's the tasks execution report at the end of the Pipeline log for this analysis session:

:: Pipeline ID: FM20190401_170628_D11 BLOCK ID: NOSTROMO Chamber: B_NOSTROMO vison version: 0.8+69.gf135883 Tasks: ['BIAS01', 'BIAS02', 'CHINJ02', 'TP01'] _ BIAS01 Executed in 9.8 minutes OBSIDs range = [17283,17292] Raised Flags = ['RON_OOL', 'HK_OOL', 'MISSDATA', 'POORQUALDATA'] _ BIAS02 Executed in 9.4 minutes OBSIDs range = [17273,17282] Raised Flags = ['RON_OOL', 'HK_OOL', 'MISSDATA', 'POORQUALDATA'] _ CHINJ02 Executed in 8.8 minutes OBSIDs range = [17329,17354] Raised Flags = ['FLUENCE_OOL', 'RON_OOL', 'FLUENCEGRAD_OOL', 'MISSDATA', 'POORQUALDATA'] _ TP01 Executed in 19.5 minutes OBSIDs range = [17355,17404] Raised Flags =

['FLUENCE_OOL', 'RON_OOL', 'FLUENCEGRAD_OOL', 'MISSDATA', 'POORQUALDATA'] _ _ _ _ ##

There should be a pdf report for each test in the D11 results sub-folder:

::

(vison) bash-4.2$ find results_atCALDATA/ -type f -name *.pdf results_atCALDATA/D11/TP01/TP01_report.pdf results_atCALDATA/D11/CHINJ02/CHINJ02_report.pdf results_atCALDATA/D11/BIAS02/BIAS02_report.pdf results_atCALDATA/D11/BIAS01/BIAS01_report.pdf results_atCALDATA/D00/FOCUS00_800/FOCUS00_800_report.pd

And the total "weight" of the D11 results sub-folder is ~3.3 GB:

```
(vison) bash-4.2$ du --si results_atCALDATA/D11
    512     results_atCALDATA/D11/TP01/products
    512     results_atCALDATA/D11/TP01/ccdpickles
    5.0M    results_atCALDATA/D11/TP01/figs
    5.7M    results_atCALDATA/D11/TP01
    512     results_atCALDATA/D11/CHINJ02/products
    512     results_atCALDATA/D11/CHINJ02/ccdpickles
    4.7M    results_atCALDATA/D11/CHINJ02/figs
    5.3M    results_atCALDATA/D11/CHINJ02
    7.0M    results_atCALDATA/D11/BIAS02/profiles
    7.3M    results_atCALDATA/D11/BIAS02/figs
    980M    results_atCALDATA/D11/BIAS02/ccdpickles
    462M    results_atCALDATA/D11/BIAS02/products
    1.5G    results_atCALDATA/D11/BIAS02
    8.0M    results_atCALDATA/D11/BIAS01/figs
    619M    results_atCALDATA/D11/BIAS01/products
    1.2G    results_atCALDATA/D11/BIAS01/ccdpickles
    7.6M    results_atCALDATA/D11/BIAS01/profiles
    1.9G    results_atCALDATA/D11/BIAS01
    3.3G    results_atCALDATA/D11
```

Check that the reports make sense, given your experience as (Cal-camp.) data analyst. Just be aware that we're using data with the "wrong" meta-data information (namely hardware serials), and that makes the pipeline to report the data as not being conformant with expectations (ast it should).

Cleaning Up after Ourselves

After doing these analysis exercises, we've generated a fair amount of accessory files we can/should get rid off so we don't waste storage resources in euclid_caldata06. This can be done with another pipeline tool, vis_clear_space.py.

The pipeline stores modified versions of the images (for example, with the offset subtracted) in a sub-folder (of each test) called "ccdpickles". Once we've passed that stage, and done other subtasks, these images are no longer needed by the pipeline, and we can erase them. Also, when doing flat-fields, the individual flat-fields for each image can be erased once we've produced the master flat-fields. In the session D00 we just did "checks", and so there was no production of prepared images (and so the whole D00 sub-folder is only 6.5MB). In D11 we didn't flat-fields, but we did "prep" for some tests, and so there are some hefty "ccdpickles" folders in there. Let's clear them up. The program will warn us of what we're going to erase, and then ask for confirmation.

```
(vison) bash-4.2$ vis_clear_space.py -p results_atCALDATA/D11/ -k ccdpickles

    Directories that will be WIPED OUT clear:

    results_atCALDATA/D11/TP01/ccdpickles
    results_atCALDATA/D11/CHINJ02/ccdpickles
    results_atCALDATA/D11/BIAS02/ccdpickles
```

```
    results_atCALDATA/D11/BIAS01/ccdpickles


    Are you happy with the selection? yes=y/Y y

    Preparing to CLEAR OUT:

    results_atCALDATA/D11/TP01/ccdpickles, 0 files, 0.0e+00 bytes
    results_atCALDATA/D11/CHINJ02/ccdpickles, 0 files, 0.0e+00 bytes
    results_atCALDATA/D11/BIAS02/ccdpickles, 30 files, 2.7e+09 bytes
    results_atCALDATA/D11/BIAS01/ccdpickles, 30 files, 2.7e+09 bytes

    TOTAL: 60 files, 5.3e+09 bytes

    To be executed: "find results_atCALDATA/D11/ -type d -name 'ccdpickles' -exec sh -
↪c 'rm -r "$0"/*' {} \;"

    Still want to proceed? yes=y/Y y
        rm: cannot remove 'results_atCALDATA/D11/TP01/ccdpickles/*': No such file or␣
↪directory
        rm: cannot remove 'results_atCALDATA/D11/CHINJ02/ccdpickles/*': No such file␣
↪or directory
```

After we've done this, the D11 sub-folder is significantly lighter. There's still some "bulk" in it, because of the master bias images (which are now multi-extension, and stored in two formats, FITS, and as python pickles):

```
(vison) bash-4.2$ du --si results_atCALDATA/D11/
512     results_atCALDATA/D11/TP01/products
512     results_atCALDATA/D11/TP01/ccdpickles
5.0M    results_atCALDATA/D11/TP01/figs
5.7M    results_atCALDATA/D11/TP01
512     results_atCALDATA/D11/CHINJ02/products
512     results_atCALDATA/D11/CHINJ02/ccdpickles
4.7M    results_atCALDATA/D11/CHINJ02/figs
5.3M    results_atCALDATA/D11/CHINJ02
7.0M    results_atCALDATA/D11/BIAS02/profiles
7.3M    results_atCALDATA/D11/BIAS02/figs
512     results_atCALDATA/D11/BIAS02/ccdpickles
462M    results_atCALDATA/D11/BIAS02/products
477M    results_atCALDATA/D11/BIAS02
8.0M    results_atCALDATA/D11/BIAS01/figs
619M    results_atCALDATA/D11/BIAS01/products
512     results_atCALDATA/D11/BIAS01/ccdpickles
7.6M    results_atCALDATA/D11/BIAS01/profiles
636M    results_atCALDATA/D11/BIAS01
1.2G    results_atCALDATA/D11/
```

CHAPTER

FOUR

GUIDE THROUGH THE CODE

In this section we provide an overal description of the capabilities and organisation of the code.

4.1 Code Guide

Here we provide a succint description of the code, what it does and how it is organised, to help the user / developer understand how things work, to better make use of it / repurpose it.

4.1.1 Capabilities of the Code

First, it is convenient to know what the code here provided can do:

1. Write **excel scripts for the acquisition of data** using ELVIS. Each test has an associated class, and this class has a method which generates the description of the test that is then converted to excel. The test description can be modified at runtime by means of test object parameters. These test descriptions are a key element of the pipeline. Monitoring and analysis capabilities depend on them (i.e. knowing what *should* happen).
2. **Monitor the data acquisition** of the calibration campaign in real time. This is done through the sub-package "eyegore". It can monitor the data flow, values in the HK data stream (cheking value against limits), display images and the exposure log, and issue warnings via sms and e-mail.
3. **Analyse the data** from the tests. This is the core functionality of the pipeline.
4. Produce calibration / analysis results in a range of formats: **Json, FITS, excel**, etc.
5. Produce **test reports** for each test in pdf (via LaTeX).
6. Collate and produce summary reports from a set of tests across all calibrated blocks (**metatests**).
7. Analyse data from the whole VIS FPA (**fpatests**).
8. **Simulate data** (very basic capabilities).

4.1.2 Code Architecture

First, it is convenient to introduce some nomenclature regarding the executiong of the pipeline that you may see used in the naming of classes and functions therein:

- A **Task** is basically a test (e.g. BIAS02), with a description of how the test data should be acquired, methods to analyse the data once acquired, and others to plot, produce reports, check compliances, etc.
- The execution of a Task is broken down into **subtasks**, which are methods of the the Task classes.

- A **pipeline** is a sequential execution of a number of a tasks. In the pipeline it takes shape as a class (**Pipe**) that has to be instantiated with inputs to perform analysis on a number of tests, using the data stored somewhere.

A diagram with the classes for the parent classes Task and Pipe (which inherits from a generic pipeline class, GenPipe) is shown in the next figure.

All tests in the campaign have an associated class, which inherits from **Task**. For example, tests BIAS01 and BIAS02 are created as "instances" of the class BIAS0X, and test DARK01 from the class DARK01. Both inherit from a common class, DarkTask, which in turn inherits from Task.

These Task subclasses have some methods which are common to all tests and are worth explaining what they do:

- **build_scriptdict()**: builds the a dictionary with the structure of the data acquisition of the test. This is, assigns values to each of the keywords in the *excel script* used to acquire the data, and for each column (exposure) in the script, according to the internal structure of the test inherent to the class (usually with some free parameters accessible to input parameters).
- **filterexposures()**: This method sub-selects the exposures in the (ELVIS generated) EXPLOG that correspond to the test, taking into account the values in the *Test* column, and usually a user-specified range of OBSIDs to consider. It also validates the acquisition parameters (collected in the EXPLOG) against the expected structure of the test.
- **check_data()** (in parent class DarkTask): This abstract method performs validation of the HK and the image values (e.g. RON, offsets, fluences) against expected values, according to the test design. The polymorphism of this method, which has to catter to the structure of very different tests, is managed through the call to sub-methods which are test-specific (and thus subclass-specific).
- **prep_data()**: Prepares images for further analysis. For example, converting the FITS files to CCD objects with analysis methods, subtracting offsets, dividing by flat-fields, etc. Depending on the test/subclass, it will perform different corrections.
- **basic_analysis()**: This is a generic method that performs the basic steps of analysis. For example, in the case of PTC analysis, this method may just extract means and variances from the pairs of images in the sequnce acquired.
- **meta_analysis()**: This is another generic method, that usually performs the important / final part of the analysis, building on the preparation of images and the *basic_analysis()* performed before. In the example of the PTC analysis, this method actually builds the PTC curves and extracts the gain and other parameters from its analysis.

4.1.3 Code Flow

It is perhaps easier to describe what the pipeline does, and how it is organised, following what it does when we use to perform different tasks.

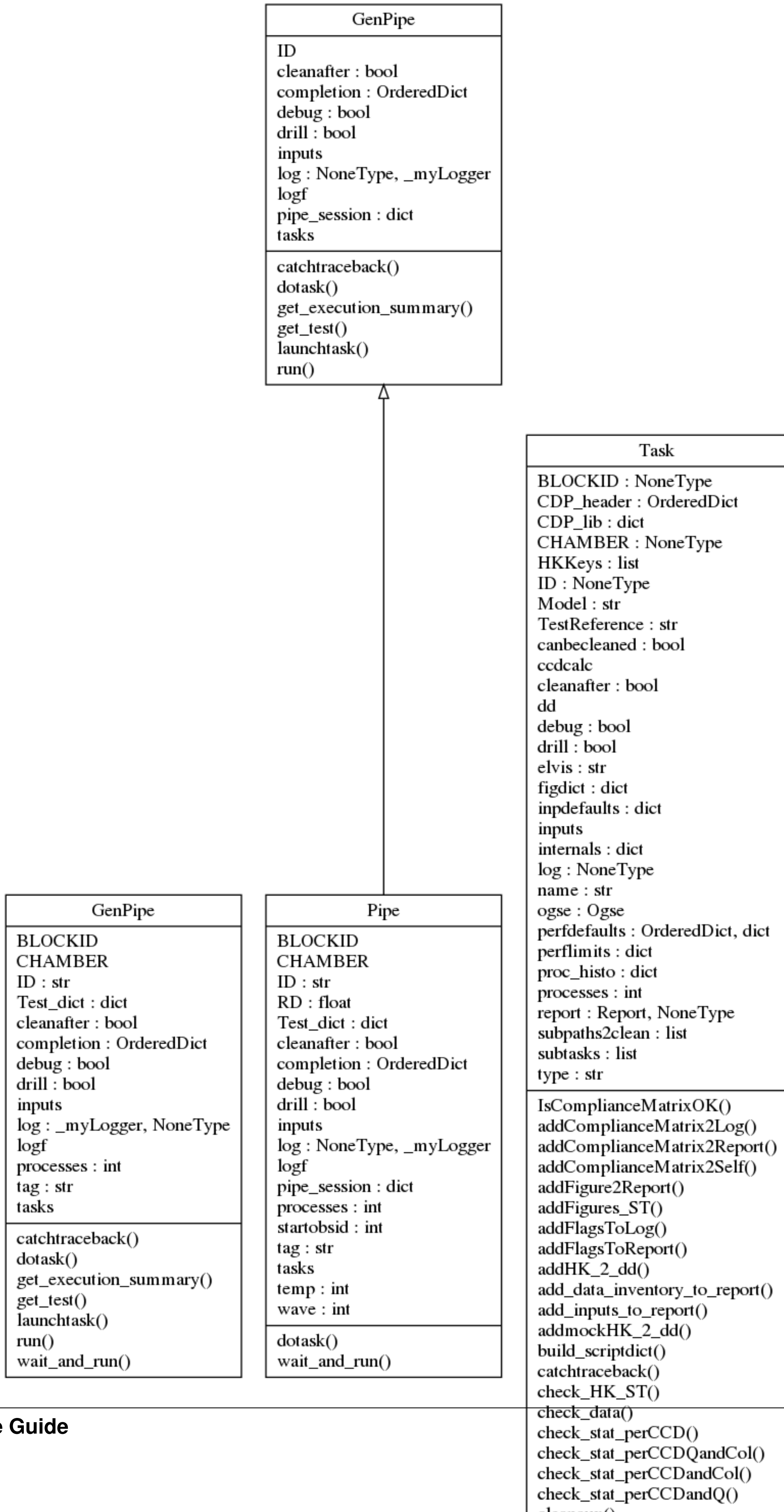
Data-set Analysis

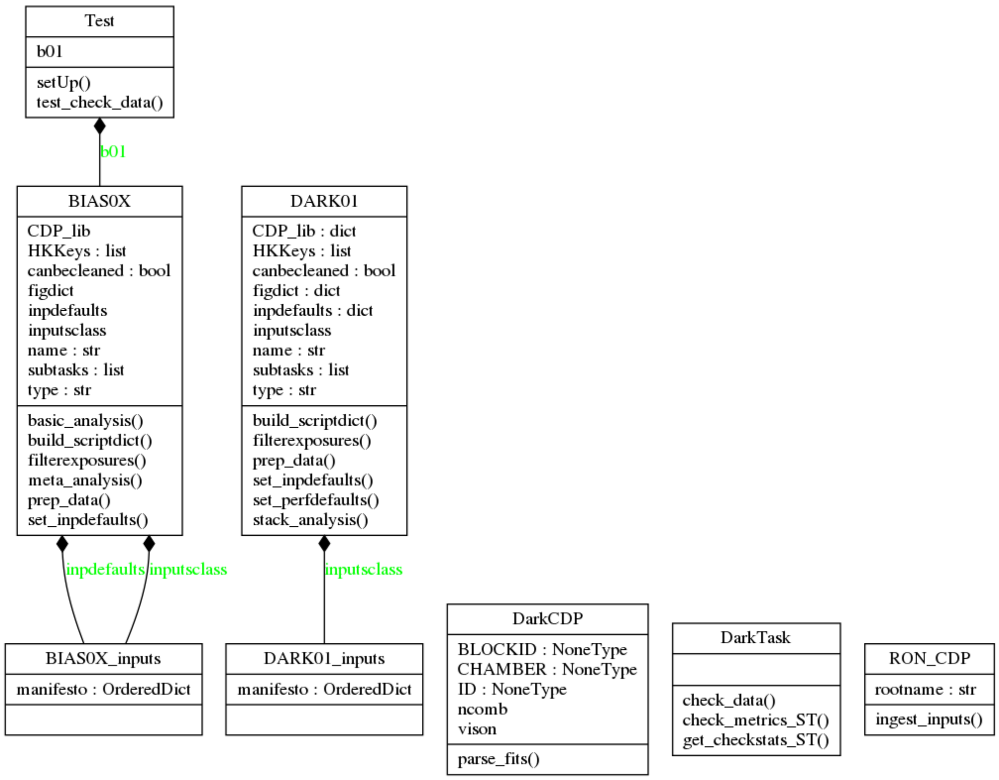
Let's first try to analyse a data-set.

We will call the script **vison_run** which instantiates a Pipeline object, loads it with the tests that we are going to process, the inputs to the tasks for those tests, and then runs the pipeline object. Let's go step by step with an example.

```
~$ vison_run -y [vison_config.py] -R [SESSION] -l -t [TAG]
```

Here vison_config.py stands for a python script with inputs (more on that soon), SESSION is a name to select the acquisition session within the the configuration file we want to select for analysis (there usually are several sessions within a configuration script, as there are data acquisition sessions in a multi-day campaign), and TAG is just a character string to label the directory with results, and the text log file, for ease of identification.


GenPipe
ID
cleanafter : bool
completion : OrderedDict
debug : bool
drill : bool
inputs
log : NoneType, _myLogger
logf
pipe_session : dict
tasks
catchtraceback()
dotask()
get_execution_summary()
get_test()
launchtask()
run()
Task
BLOCKID : NoneType
CDP_header : OrderedDict
CDP_lib : dict
CHAMBER : NoneType
HKKeys : list
ID : NoneType
Model : str
TestReference : str
canbecleaned : bool
ccdcalc
cleanafter : bool
dd
debug : bool
drill : bool
elvis : str
figdict : dict
inpdefaults : dict
inputs
internals : dict
log : NoneType
name : str
ogse : Ogse
perfdefaults : OrderedDict, dict
perflimits : dict
proc_histo : dict
processes : int
report : Report, NoneType
subpaths2clean : list
subtasks : list
type : str
IsComplianceMatrixOK()
addComplianceMatrix2Log()
addComplianceMatrix2Report()
addComplianceMatrix2Self()
addFigure2Report()
addFigures_ST()
addFlagsToLog()
addFlagsToReport()
addHK_2_dd()
add_data_inventory_to_report()
add_inputs_to_report()
addmockHK_2_dd()
build_scriptdict()
catchtraceback()
check_HK_ST()
check_data()
check_stat_perCCD()
check_stat_perCCDQandCol()
check_stat_perCCDandCol()
check_stat_perCCDandQ()
cleanaux()
GenPipe
BLOCKID
CHAMBER
ID : str
Test_dict : dict
cleanafter : bool
completion : OrderedDict
debug : bool
drill : bool
inputs
log : _myLogger, NoneType
logf
processes : int
tag : str
tasks
catchtraceback()
dotask()
get_execution_summary()
get_test()
launchtask()
run()
wait_and_run()
Pipe
BLOCKID
CHAMBER
ID : str
RD : float
Test_dict : dict
cleanafter : bool
completion : OrderedDict
debug : bool
drill : bool
inputs
log : NoneType, _myLogger
logf
pipe_session : dict
processes : int
startobsid : int
tag : str
tasks
temp : int
wave : int
dotask()
wait_and_run()



Before we go on, some basic notions regarding the organisation of the GCC:

- the campaign was sub-divided in campaigns for each block.
- within each block-campaign there were sessions/runs (-R comes from "run") in which several tests are executed one after another, autonomously by ELVIS.

For each block-campaign we created a vison_config script, with the name of the block in question and the date of the campaign (e.g. vison_config_EINSTEIN_JUL19.py). This script is important because it is required to run the pipeline, and because it serves as registry of the inputs used to run it.

Let's have a look at the contents of a vison_config.py script, in simplified form.

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
"""

EUCLID VIS CALIBRATION CAMPAIGN
"vison" CONFIGURATION FILE

BLOCKID: EINSTEIN   # NEEDS USER INPUTS.
Filled in by: RAF   # NEEDS USER INPUTS.

:author: Ruyman Azzollini
:contact: r.azzollini_at_ucl.ac.uk

"""

# IMPORT STUFF
from pdb import set_trace as stop
import numpy as np
```

```
import sys
import os
from vison.campaign import campaign, devel
import copy
# END IMPORT


# ALL to_do dictionaries:
#
# MOT_WARM: [None,None],dict(init=True,check=True,basic=True)
# COSMETICS00: [None,None],dict(init=True,check=True,masks=True,meta=True)
# [...]

def get_config_dict(testgenerator, datapath, resroot,
                    BLOCKID,CHAMBER,diffvalues,
                    inCDPs,perflimits,tasks2execute=[],
                    doReport=True,elvis='7.5.X'):
    """Produces input dictionaries to run the pipeline on a series of Tasks/Tests."""

    return inputdict



def add_RUN_specifics(inputdict,RUN,dataroot=''):
    """Adds specifics to the input dictionaries of each Task/Test. This is where
    we taylor execution for each test (e.g. choose what to do within the Task,
    or tell the pipeline where to find the data and what OBSIDs to consider for
    each Test).

    """
    import os
    import copy
    import glob
    from vison.pipe.lib import sortbydateexplogfs

    # How to fill-in the specifics for each test within a RUN:
    # Code for the test spec lists:
    #    ['TEST-NAME', [START-OBSID(integer), END-OBSID(integer)],todo(dictionary)]
    #    All to-do dictionaries provided at the beginning of this script.
    #    If you want to by-pass a test within a RUN, comment the line with a 'hash'.

    # NEEDS USER INPUTS: Tests lists, ObsID limits, sub-tasts todo booleans

    test_specifics = dict(

    D00 = dict(

            schedule=[
                    ['MOT_WARM',[30332, 30339],dict(init=True,check=True,basic=True)]
                    ],
            datapath=os.path.join(dataroot,'25_Jul_19')
        )
    )

    # END USER INPUTS

    # [...]
```

```
    return inputdict

if __name__ == '__main__':

    # MASTER

    # NEEDS USER INPUTS:
    dataroot = '../atCALDATA/data'
    datapath = os.path.join(dataroot,'NN_Mmm_19')
    cdppath = 'calproducts'
    resroot = 'results_atCALDATA/'
    BLOCKID = 'EINSTEIN'
    CHAMBER = 'B_EINSTEIN' # example: 'A_MAX'

    diffvalues = dict(operator = 'unk',
            sn_ccd1 = '14471-19-01', # example: '14183-18-01',
            sn_ccd2 = '15081-15-02', # example: '14311-04-01',
            sn_ccd3 = '14471-10-02', # example: '14173-14-02',
            sn_roe= 'FM14', # example: FM01
            sn_rpsu = 'FM14') # example: FM02

    inCDPs = dict(Mask=dict(
            CCD1=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD1_SN_%s.fits' % \
                              (BLOCKID,diffvalues['sn_ccd1'])),
            CCD2=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD2_SN_%s.fits' % \
                              (BLOCKID,diffvalues['sn_ccd2'])),
            CCD3=os.path.join(cdppath,'masks/EUC_MASK_%s_CCD3_SN_%s.fits' % \
                              (BLOCKID,diffvalues['sn_ccd3']))),
        Gain=dict(
            nm730=os.path.join(cdppath, 'gain/nm730/PTC02_730_GAIN_TB.pick')
            ),
        FF=dict(
            nm730=dict(
                CCD1=os.path.join(cdppath,'flats/nm730/EUC_FF_730nm_col001_ROE1_CCD1.
↪fits'),
                CCD2=os.path.join(cdppath,'flats/nm730/EUC_FF_730nm_col001_ROE1_CCD2.
↪fits'),
                CCD3=os.path.join(cdppath,'flats/nm730/EUC_FF_730nm_col001_ROE1_CCD3.
↪fits')))
        )

    perflimits = dict()
    doReport = True
    elvis='7.5.X'
    # END USER INPUTS.

    # NEEDS USER INPUTS.

    tasks2execute_dict=dict(
                            D00=['MOT_WARM'],
                            D11=['COSMETICS00', 'FOCUS00', 'PSFLUX00', 'FLATFLUX00',
↪'FLAT_STB', 'PTC02WAVE'],
                            D12=['BIAS02','BIAS01','CHINJ01','CHINJ02','TP11','TP21',
                                 'FLATFLUX00', 'PTC01','NL02','FLAT01','DARK01',
↪'PTC02WAVE',
                                 'PERSIST01','PSFLUX00'],
                            D12E=['BF01','BF01WAVE'],
```

```
                            D21=['BIAS02','PSF01','PTC02WAVE'],
                            D21E=['BF01WAVE'],
                            D22=['BIAS02','PSF01','FLAT02'],
                            DTEST=['BF01WAVE']#'BF01WAVE']
                        )
    # END USER INPUTS.

    testgenerator = campaign.generate_test_sequence

    inputdict = dict()

    if np.any([RUN == '' for RUN in RUNs]):
        sys.exit('Hey, what RUN do you want to process?')

    for RUN in RUNs:

        tasks2execute = tasks2execute_dict[RUN]

            # [...]


            inputdict.update(_inputdict)
```

When vison_run is executed, this literally *executes* the vison_config script which generates a dictionary called **inputdict**. This dictionary has all the information necessary for the pipeline to run:

- in which facility (*chamber*) was the data acquired (relevant to know exposure times, for example).
- what tasks (tests) are to be executed/analysed.
- what values to assign to the free parameters of the tasks.
- what hardware are we testing.
- where to find the input data, and where to put the output results.
- where to find input calibration data products (e.g. cosmetics masks).

When vison_config.py is executed, it starts from:

```
if __name__ == '__main__'
```

From there, it calls the function *get_config_dict* to create a standard version of the inputs dictionary, setting up the tasks and their standard inputs. Then it calls *add_RUN_specifics* to add data locations, OBSID ranges, and apply a selector of sub-tasks to execute for each task.

Going back to *vison_run*, once the inputs in the configuration file are ingested, the next important thing is to create an instance of the pipeline class:

```
pipe = Pipe(inputdict, dolog=dolog, drill=drill,
                debug=debug, startobsid=startobsid,
                processes=multithread, tag=tag,
                cleanafter=cleanafter)
```

This takes as input the input dictionary (inputdict) and other keywords to control the execution of the pipeline.

Then, the pipeline is executed, either in wait-for-data mode (in parallel with acquisition), or directly (assuming all data has already been acquired previously):

```
if wait:
    pipe.wait_and_run(dayfolder, elvis=elvis)
else:
    pipe.run()
```

In the **run** method, the pipe object cyles over the list of tasks to be executed, and launches each task with its specific inputs:

:: for taskname in tasknames:

> taskinputs = self.inputs[taskname] taskinputs['resultspath'] = os.path.join(resultsroot, taskinputs['resultspath'])
>
> **if explogf is not None:** taskinputs['explogf'] = explogf
>
> **if elvis is not None:** taskinputs['elvis'] = elvis
>
> self.inputs[taskname] = taskinputs
>
> self.launchtask(taskname)
>
> [...]

The **launchtask** method is also a method of Pipe. Here, some further inputs parsing, and task execution logging are handled. But the actual instantiation of the Task object happens in the **dotask** method within **launchtask**.

```
[...]
taskreport = self.dotask(taskname, taskinputs,
                drill=self.drill, debug=self.debug,
                cleanafter=self.cleanafter)
[...]
```

In **Pipe.dotask(...)** is where the Task object gets instantiated and executed:

```
[...]

test = self.get_test(strip_taskname, inputs=inputs, log=self.log,
        drill=drill, debug=debug, cleanafter=cleanafter)

[...]

Errors = test()

[...]
```

The rest in that method is handling exceptions, and logging errors and test analysis execution durations.

Each Task has a **__call__**() method that is what is called when we do Errors = test(), and obviously returns any errors raised during the execution of the task.

The **__call__**() method is common to all Task subclasses, because it is inherited from Task class itself. The polymorphism of the tasks in their execution (e.g. different subtask methods), is handled through subclass methods, which are called via internal dictionaries that link methods to subtask names.

4.1.4 Data Models

See *Data Model*

4.1.5 Reusing / adapting the code

CHAPTER

FIVE

PIPELINE CORE

Pipeline master classes.

5.1 Pipeline

5.1.1 genmaster.py

Created on Wed Sep 25 17:07:21 2019

@author: raf

class `vison.pipe.genmaster.`**`GenPipe`**(*inputdict*, *dolog=True*, *drill=False*, *debug=False*, *startobsid=0*, *processes=1*, *tag=' '*, *cleanafter=False*)

Abstract Master Class of FM-analysis, any level of assembly.

`catchtraceback`()

`dotask`(*taskname*, *inputs*, *drill=False*, *debug=False*, *cleanafter=False*)

Generic test master function.

`get_execution_summary`(*exectime=None*)

`get_test`(*taskname*, *inputs={}*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`launchtask`(*taskname*)

`run`(*explogf=None*, *elvis=None*)

`wait_and_run`(*dayfolder*, *elvis='7.5.X'*)

class `vison.pipe.genmaster.`**`GenPipe`**(*inputdict*, *dolog=True*, *drill=False*, *debug=False*, *startobsid=0*, *processes=1*, *tag=' '*, *cleanafter=False*)

Abstract Master Class of FM-analysis, any level of assembly.

`catchtraceback`()

`dotask`(*taskname*, *inputs*, *drill=False*, *debug=False*, *cleanafter=False*)

Generic test master function.

`get_execution_summary`(*exectime=None*)

`get_test`(*taskname*, *inputs={}*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`launchtask`(*taskname*)

`run`(*explogf=None*, *elvis=None*)

`wait_and_run`(*dayfolder*, *elvis='7.5.X'*)

5.1.2 master.py

This is the main script that will orchestrate the analysis of Euclid-VIS FM Ground Calibration Campaign.

The functions of this module are:

- Take inputs as to what data is to be analyzed, and what analysis scripts are to be run on it.
- Set the variables necessary to process this batch of FM calib. data.
- Start a log of actions to keep track of what is being done.
- Provide inputs to scripts, execute the analysis scripts and report location of analysis results.

Some Guidelines for Development:

- Input data is "sacred": read-only.
- Each execution of Master must have associated a unique ANALYSIS-ID.
- All the Analysis must be divided in TASKS. TASKS can have SUB-TASKS.
- All data for each TASK must be under a single day-folder.
- All results from the execution of FMmaster must be under a single directory with subdirectories for each TASK run.
- **A subfolder of this root directory will contain the logging information:** inputs, outputs, analysis results locations.

Created on Wed Jul 27 12:16:40 2016

author Ruyman Azzollini

class `vison.pipe.master.`**`GenPipe`**(*inputdict*, *dolog=True*, *drill=False*, *debug=False*, *cleanafter=False*)

Master Pipeline Class.

`catchtraceback`()

`dotask`(*taskname*, *inputs*, *drill=False*, *debug=False*, *cleanafter=False*)

Generic test master function.

`get_execution_summary`(*exectime=None*)

`get_test`(*taskname*, *inputs={}*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`launchtask`(*taskname*)

`run`(*explogf=None*, *elvis=None*)

class `vison.pipe.master.`**`Pipe`**(*inputdict*, *dolog=True*, *drill=False*, *debug=False*, *startobsid=0*, *processes=1*, *tag=''*, *cleanafter=False*)

Master Class of FM-analysis at block-level of assembly.

class **`BF01`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)

Builds PTC0X script structure dictionary.

#:param exptimes: list of ints [ms], exposure times. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 800 nm. :param diffvalues: dict, opt, differential values.

`correct_BFE_G15`()

`extract_BF`()

Performs basic analysis of images:
- extracts BF matrix for each COV matrix

extract_COV()
Performs basic analysis of images:
- extracts COVARIANCE matrix for each fluence

extract_PTCs()

f_correct_BFE_G15(*ccdobjname*, *fixA=False*)
Applies BFE solutions from G+15 to images, to later test effectivity through PTC.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()
Analyzes the BF results across fluences.

set_inpdefaults(***kwargs*)

class Pipe.**BIAS0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()
BIAS0X: Basic analysis of data.

METACODE

```
f. e. ObsID:
   f.e.CCD:

       load ccdobj of ObsID, CCD

       with ccdobj, f.e.Q:
           produce a 2D poly model of bias, save coefficients
           produce average profile along rows
           produce average profile along cols
           # save 2D model and profiles in a pick file for each OBSID-CCD
           measure and save RON after subtracting large scale structure

plot RON vs. time f. each CCD and Q
plot average profiles f. each CCD and Q (color coded by time)
```

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds BIAS0X script structure dictionary.

###:param N: integer, number of frames to acquire. :param diffvalues: dict, opt, differential values. :param elvis: char, ELVIS version.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()
METACODE

```
f. each CCD:
   stack all ObsIDs to produce Master Bias
   f. e. Q:
       measure average profile along rows
       measure average profile along cols
plot average profiles of Master Bias(s) f. each CCD,Q
(produce table(s) with summary of results, include in report)
save Master Bias(s) (3 images) to FITS CDPs
```

```
show Master Bias(s) (3 images) in report
save name of MasterBias(s) CDPs to DataDict, report
```

prep_data()

BIAS0X: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction cosmetics masking

class Pipe.**CHINJ00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds CHINJ00 script structure dictionary.

Parameters diffvalues – dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

set_inpdefaults(***kwargs*)

class Pipe.**CHINJ01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds CHINJ01 script structure dictionary.

#:param IDL: float, [V], value of IDL (Inject. Drain Low). #:param IDH: float, [V], Injection Drain High. #:param IG2: float, [V], Injection Gate 2. #:param IG1s: list of 2 floats, [V], [min,max] values of IG1. #:param id_delays: list of 2 floats, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()

Plot and model charge injection vs. IG1 Find injection threshold: Min IG1 Find notch injection amount.

set_inpdefaults(***kwargs*)

class Pipe.**CHINJ02**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds CHINJ02 script structure dictionary.

#:param IDLs: list of 2 ints, [V], [min,max] values of IDL (Inject. Drain Low). #:param IDH: int, [V], Injection Drain High. #:param id_delays: list of 2 ints, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()

Finds the Injection Threshold for each CCD half.

METACODE

```
f.e.CCD:
    f.e.Q:
        load injection vs. IDL cuve
        find&save injection threshold on curve

report injection threshold as a table
```

set_inpdefaults(***kwargs*)

class `Pipe.`**`COSMETICS00`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds COSMETICS00 script structure dictionary.

Parameters
- **`diffvalues`** – dict, opt, differential values.
- **`elvis`** – char, ELVIS version.

`check_data`()

`check_metrics_ST`(***kwargs*)

`do_masks`()

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`get_checkstats_ST`(***kwargs*)

`meta`()

class `Pipe.`**`DARK01`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds DARK01 script structure dictionary.

Parameters **`diffvalues`** – dict, opt, differential values.

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`prep_data`()
DARK01: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [BIAS SUBTRACTION] cosmetics masking

`stack_analysis`()
METACODE

```
f. each CCD:
    f. e. Q:
        stack all ObsIDs to produce Master Dark
        produce mask of hot pixels / columns
        count hot pixels / columns
        measure average profile along rows
        measure average profile along cols

plot average profiles of Master Bias f. each CCD,Q
show Master Dark (images), include in report
report stats of defects, include in report
save name of MasterDark to DataDict, report
save name of Defects in Darkness Mask to DD, report
```

class `Pipe.`**`FLAT0X`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds FLAT0X script structure dictionary.

Parameters **`diffvalues`** – dict, opt, differential values.

`do_indiv_flats`()
METACODE

```
Preparation of data for further analysis and
produce flat-field for each OBSID.

f.e. ObsID:
    f.e.CCD:

        load ccdobj

        f.e.Q:

            model 2D fluence distro in image area
            produce average profile along rows
            produce average profile along cols

        save 2D model and profiles in a pick file for each OBSID-CCD
        divide by 2D model to produce indiv-flat
        save indiv-Flat to FITS(?), update add filename

plot average profiles f. each CCD and Q (color coded by time)
```

do_master_flat()

METACODE

```
Produces Master Flat-Field

f.e.CCD:
    f.e.Q:
        stack individual flat-fields by chosen estimator
save Master FF to FITS
measure PRNU and
report PRNU figures
```

do_prdef_mask()

METACODE

```
Produces mask of defects in Photo-Response
Could use master FF, or a stack of a subset of images (in order
to produce mask, needed by other tasks, quicker).

f.e.CCD:
    f.e.Q:
        produce mask of PR defects
        save mask of PR defects
        count dead pixels / columns

report PR-defects stats
```

filterexposures(*structure*, *explog*, *OBSID_lims*)

prepare_images()

FLAT0X: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [bias structure subtraction, if available] cosmetics masking

set_inpdefaults(***kwargs*)

class Pipe.**FOCUS00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()

This is just an assignation of values measured in check_data.

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds FOCUS00 script structure dictionary.

#:param wavelength: int, [nm], wavelength. #:param exptime: int, [ms], exposure time. :param diffvalues: dict, opt, differential values.

filterexposures (*structure*, *explog*, *OBSID_lims*)

lock_on_stars ()

meta_analysis ()

prep_data ()

class Pipe.**MOT_FF** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

extract_HER ()

class Pipe.**MOT_WARM** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis ()
EXPOSURES: BIAS, RAMP, CHINJ, FLAT, POINT_w x waves_PNT

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds MOT_WARM script structure dictionary.

Parameters
- **diffvalues** – dict, opt, differential values.
- **elvis** – char, ELVIS version.

check_data ()

check_metrics_ST (***kwargs*)

filterexposures (*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST (***kwargs*)

lock_on_stars ()

class Pipe.**NL01** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds NL01 script structure dictionary.

#:param expts: list of ints [ms], exposure times. #:param exptinter: int, ms, exposure time of interleaved source-stability exposures. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 0 (Neutral Density Filter) :param diffvalues: dict, opt, differential values.

do_satCTE ()
METACODE

```
select ObsIDs with fluence(exptime) >~ 0.5 FWC

f.e. ObsID:
    CCD:
        Q:
            measure CTE from amount of charge in over-scan relative to␣
↪fluence
```

```
f.e. CCD:
    Q:
        get curve of CTE vs. fluence
        measure FWC from curve in ADU

report FWCs in electrons [via gain in inputs] f.e. CCD, Q (table)
```

extract_stats()

Performs basic analysis: extracts statistics from image regions to later build NLC.

METACODE

```
create segmentation map given grid parameters

f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            f.e. "img-segment": (done elsewhere)
                measure central value
                measure variance
```

filterexposures(*structure*, *explog*, *OBSID_lims*)

Loads a list of Exposure Logs and selects exposures from test NL01.

The filtering takes into account an expected structure for the acquisition script.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

prep_data()

Takes Raw Data and prepares it for further analysis.

METACODE

```
f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            mask-out bad pixels
            mask-out detector cosmetics
            subtract offset
            opt: [sub bias frame]
```

produce_NLCs()

METACODE

```
Obtains Best-Fit Non-Linearity Curve

f.e. CCD:
    f.e. Q:

        [opt] apply correction for source variability (interspersed␣
↪exposure
          with constant exptime)
        Build NL Curve (NLC) - use stats and exptimes
        fit poly. shape to NL curve

plot NL curves for each CCD, Q
report max. values of NL (table)
```

recalibrate_exptimes (*exptimes*)
Corrects exposure times given independent calibration of the shutter.

class Pipe.**NL02** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

build_scriptdict (*diffvalues={}, elvis='7.5.X'*)
Builds NL02 script structure dictionary.

do_satCTE ()
METACODE

```
select ObsIDs with fluence(exptime) >~ 0.5 FWC

f.e. ObsID:
    CCD:
        Q:
            measure CTE from amount of charge in over-scan relative to␣
↪fluence

f.e. CCD:
    Q:
        get curve of CTE vs. fluence
        measure FWC from curve in ADU

report FWCs in electrons [via gain in inputs] f.e. CCD, Q (table)
```

prep_data ()
Takes Raw Data and prepares it for further analysis.

METACODE

```
f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            mask-out bad pixels
            mask-out detector cosmetics
            subtract offset
            opt: [sub bias frame]
```

produce_NLCs ()
METACODE

```
Obtains Best-Fit Non-Linearity Curve

f.e. CCD:
    f.e. Q:

        [opt] apply correction for source variability (interspersed␣
↪exposure
          with constant exptime)
        Build NL Curve (NLC) - use stats and exptimes
        fit poly. shape to NL curve

plot NL curves for each CCD, Q
report max. values of NL (table)
```

class Pipe.**PERSIST01** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

basic_analysis()
Basic analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        measure stats in pix satur MASK across OBSIDs
         (pre-satur, satur, post-satur)
```

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds PERSISTENCE01 script structure dictionary.

Parameters

- **exptSATUR** – int, saturation exposure time.
- **exptLATEN** – int, latency exposure time.
- **diffvalues** – dict, opt, differential values.

check_data()
PERSIST01: Checks quality of ingested data.

METACODE

```
check common HK values are within safe / nominal margins
check voltages in HK match commanded voltages, within margins

f.e.ObsID:
    f.e.CCD:
        f.e.Q.:
            measure offsets in pre-, over-
            measure std in pre-, over-
            measure fluence in apertures around Point Sources

assess std in pre- (~RON) is within allocated margins
assess offsets in pre-, and over- are equal, within allocated  margins
assess fluence is ~expected within apertures (PS) for each frame (pre-
↪satur, satur, post-satur)


plot point source fluence vs. OBSID, all sources
[plot std vs. time]

issue any warnings to log
issue update to report
```

check_metrics_ST(***kwargs*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

get_satur_masks()
Basic analysis of data.

METACODE

```
f.e.CCD:
    use SATURATED frame to generate pixel saturation MASKs
```

meta_analysis()
Meta-analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        estimate delta-charge_0 and decay tau from time-series

report:
    persistence level (delta-charge_0) and time constant
```

prep_data()
PERSIST01: Preparation of data for further analysis. Calls task.prepare_images().
Applies: offset subtraction cosmetics masking

set_inpdefaults(***kwargs*)

class Pipe.**PTC0X**(*inputs, log=None, drill=False, debug=False, cleanafter=False*)

build_scriptdict(*diffvalues={}, elvis='7.5.X'*)
Builds PTC0X script structure dictionary.

#:param exptimes: list of ints [ms], exposure times. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 800 nm. :param diffvalues: dict, opt, differential values.

extract_HER()
Hard Edge Response Analysis

extract_PTC()
Performs basic analysis of images:
- builds PTC curves: both on non-binned and binned images

METACODE

```
create list of OBSID pairs

create segmentation map given grid parameters

f.e. OBSID pair:
    CCD:
        Q:
            subtract CCD images
            f.e. segment:
                measure central value
                measure variance
```

f_extract_PTC(*ccdobjcol, medcol, varcol*)

filterexposures(*structure, explog, OBSID_lims*)

meta_analysis()
Analyzes the variance and fluence: gain, and gain(fluence)

METACODE

```
f.e. CCD:
    Q:
        (using stats across segments:)
        fit PTC to quadratic model
        solve for gain
        solve for alpha (pixel-correls, Guyonnet+15)
        solve for blooming limit (ADU)
```

```
            convert bloom limit to electrons, using gain

plot PTC curves with best-fit f.e. CCD, Q
report on gain estimates f. e. CCD, Q (table)
report on blooming limits (table)
```

set_inpdefaults (***kwargs*)

class Pipe.**STRAY00** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds STRAY00 script structure dictionary. :param diffvalues: dict, opt, differential values.

filterexposures (*structure*, *explog*, *OBSID_lims*)

set_inpdefaults (***kwargs*)

class Pipe.**TP00** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)

check_data ()
TP01: Checks quality of ingested data.

METACODE

```
check common HK values are within safe / nominal margins
check voltages in HK match commanded voltages, within margins

f.e.ObsID:
    f.e.CCD:
        f.e.Q.:
            measure offsets in pre-, over-
            measure std in pre-, over-
            measure mean in img-

assess std in pre- (~RON) is within allocated margins
assess offsets in pre-, and over- are equal, within allocated margins
assess offsets are within allocated margins
assess injection level is within expected margins

plot histogram of injected levels for each Q
[plot std vs. time]

issue any warnings to log
issue update to report
```

filterexposures (*structure*, *explog*, *OBSID_lims*)

set_inpdefaults (***kwargs*)

class Pipe.**TP01** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis ()
Basic analysis of data.

METACODE

```
f. e. ObsID [there are different TOI_TP and TP-patterns]:
    f.e.CCD:
        f.e.Q:
            load "map of relative pumping"
            find_dipoles:
                x, y, rel-amplitude, orientation

produce & report:
    map location of dipoles
    PDF of dipole amplitudes (for N and S)
    Counts of dipoles (and N vs. S)
```

build_scriptdict (*diffvalues={}, elvis='7.5.X'*)

extract ()

Obtain maps of dipoles.

METACODE

```
f.e. id_delay (there are 2):
    f.e. CCD:
        f.e. Q:
            produce reference non-pumped injection map

f. e. ObsID:
    f.e. CCD:

        load ccdobj
        f.e.Q.:
            divide ccdobj.Q by injection map

        save dipole map and store reference
```

filterexposures (*structure, explog, OBSID_lims*)

meta_analysis ()

Meta-analysis of data:

Try to identify tau and pixel-phase location for each trap. Need to associate dipoles across TOI_TPs and TP-patterns

METACODE

```
across TOI_TP, patterns:

    build catalog of traps: x,y, tp-mode, tau, Pc
    tau, Pc = f({A,TOI})

Report on :
    Histogram of Taus
    Histogram of Pc (capture probability)
    Histogram of I-phases (larger phases should have more traps,
                    statistically) -> check

    Total Count of Traps
```

set_inpdefaults (***kwargs*)

class Pipe.**TP02** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

basic_analysis ()
Basic analysis of data.

METACODE

```
f. e. ObsID [there are different TOI_TP and TP-patterns]:
    f.e.CCD:
        f.e.Q:
            load raw 1D map of relative pumping (from extract_data)
            identify dipoles:
                x, rel-amplitude, orientation (E or W)

produce & report:
    map location of dipoles
    PDF of dipole amplitudes (for E and W)
    Counts of dipoles (and E vs. W)
```

build_scriptdict (*diffvalues={}, elvis='7.5.X'*)

extract ()
Obtain Maps of Serial Dipoles.

filterexposures (*structure, explog, OBSID_lims*)

meta_analysis ()
Meta-analysis of data:

Try to identify tau and pixel-phase location for each trap. Need to associate dipoles across TOI_TPs and TP-patterns

METACODE

```
across TOI_TP, patterns:
    build catalog of traps: x,y,R-phase, amp(dwell)
    from Amp(dwell) -> tau, Pc

Report on :
   Histogram of Taus
   Histogram of Pc (capture probability)
   Histogram of R-phases

   Total Count of Traps
```

set_inpdefaults (***kwargs*)

class Pipe.**TP11** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

class Pipe.**TP21** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

Pipe.**dotask** (*taskname, inputs, drill=False, debug=False, cleanafter=False*)
Generic test master function.

Pipe.**wait_and_run** (*dayfolder, elvis='7.5.X'*)

class vison.pipe.master.**Pipe** (*inputdict, dolog=True, drill=False, debug=False, startobsid=0, processes=1, tag='', cleanafter=False*)
Master Class of FM-analysis at block-level of assembly.

class BF01 (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

build_scriptdict (*diffvalues={}, elvis='7.5.X'*)
Builds PTC0X script structure dictionary.

#:param exptimes: list of ints [ms], exposure times. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 800 nm. :param diffvalues: dict, opt, differential values.

correct_BFE_G15()

extract_BF()

Performs basic analysis of images:

- extracts BF matrix for each COV matrix

extract_COV()

Performs basic analysis of images:

- extracts COVARIANCE matrix for each fluence

extract_PTCs()

f_correct_BFE_G15(*ccdobjname*, *fixA=False*)

Applies BFE solutions from G+15 to images, to later test effectivity through PTC.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()

Analyzes the BF results across fluences.

set_inpdefaults(***kwargs*)

class Pipe.**BIAS0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()

BIAS0X: Basic analysis of data.

METACODE

```
f. e. ObsID:
   f.e.CCD:

       load ccdobj of ObsID, CCD

       with ccdobj, f.e.Q:
           produce a 2D poly model of bias, save coefficients
           produce average profile along rows
           produce average profile along cols
           # save 2D model and profiles in a pick file for each OBSID-CCD
           measure and save RON after subtracting large scale structure

plot RON vs. time f. each CCD and Q
plot average profiles f. each CCD and Q (color coded by time)
```

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds BIAS0X script structure dictionary.

###:param N: integer, number of frames to acquire. :param diffvalues: dict, opt, differential values. :param elvis: char, ELVIS version.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()

METACODE

```
f. each CCD:
   stack all ObsIDs to produce Master Bias
```

```
   f. e. Q:
       measure average profile along rows
       measure average profile along cols
plot average profiles of Master Bias(s) f. each CCD,Q
(produce table(s) with summary of results, include in report)
save Master Bias(s) (3 images) to FITS CDPs
show Master Bias(s) (3 images) in report
save name of MasterBias(s) CDPs to DataDict, report
```

prep_data()
BIAS0X: Preparation of data for further analysis. Calls task.prepare_images().
Applies: offset subtraction cosmetics masking

class Pipe.**CHINJ00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds CHINJ00 script structure dictionary.
Parameters diffvalues – dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

set_inpdefaults(***kwargs*)

class Pipe.**CHINJ01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds CHINJ01 script structure dictionary.

#:param IDL: float, [V], value of IDL (Inject. Drain Low). #:param IDH: float, [V], Injection Drain High. #:param IG2: float, [V], Injection Gate 2. #:param IG1s: list of 2 floats, [V], [min,max] values of IG1. #:param id_delays: list of 2 floats, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()
Plot and model charge injection vs. IG1 Find injection threshold: Min IG1 Find notch injection amount.

set_inpdefaults(***kwargs*)

class Pipe.**CHINJ02**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds CHINJ02 script structure dictionary.

#:param IDLs: list of 2 ints, [V], [min,max] values of IDL (Inject. Drain Low). #:param IDH: int, [V], Injection Drain High. #:param id_delays: list of 2 ints, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()
Finds the Injection Threshold for each CCD half.

METACODE

```
f.e.CCD:
    f.e.Q:
```

```
        load injection vs. IDL cuve
        find&save injection threshold on curve

report injection threshold as a table
```

set_inpdefaults(***kwargs*)

class Pipe.**COSMETICS00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds COSMETICS00 script structure dictionary.

Parameters
- **diffvalues** – dict, opt, differential values.
- **elvis** – char, ELVIS version.

check_data()

check_metrics_ST(***kwargs*)

do_masks()

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

meta()

class Pipe.**DARK01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds DARK01 script structure dictionary.

Parameters **diffvalues** – dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

prep_data()
DARK01: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [BIAS SUBTRACTION] cosmetics masking

stack_analysis()
METACODE

```
f. each CCD:
    f. e. Q:
        stack all ObsIDs to produce Master Dark
        produce mask of hot pixels / columns
        count hot pixels / columns
        measure average profile along rows
        measure average profile along cols

plot average profiles of Master Bias f. each CCD,Q
show Master Dark (images), include in report
report stats of defects, include in report
save name of MasterDark to DataDict, report
save name of Defects in Darkness Mask to DD, report
```

class Pipe.**FLAT0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds FLAT0X script structure dictionary.
Parameters **diffvalues** – dict, opt, differential values.

do_indiv_flats ()
METACODE

```
Preparation of data for further analysis and
produce flat-field for each OBSID.

f.e. ObsID:
    f.e.CCD:

        load ccdobj

        f.e.Q:

            model 2D fluence distro in image area
            produce average profile along rows
            produce average profile along cols

        save 2D model and profiles in a pick file for each OBSID-CCD
        divide by 2D model to produce indiv-flat
        save indiv-Flat to FITS(?), update add filename

plot average profiles f. each CCD and Q (color coded by time)
```

do_master_flat ()
METACODE

```
Produces Master Flat-Field

f.e.CCD:
    f.e.Q:
        stack individual flat-fields by chosen estimator
save Master FF to FITS
measure PRNU and
report PRNU figures
```

do_prdef_mask ()
METACODE

```
Produces mask of defects in Photo-Response
Could use master FF, or a stack of a subset of images (in order
to produce mask, needed by other tasks, quicker).

f.e.CCD:
    f.e.Q:
        produce mask of PR defects
        save mask of PR defects
        count dead pixels / columns

report PR-defects stats
```

filterexposures (*structure*, *explog*, *OBSID_lims*)

prepare_images ()
FLAT0X: Preparation of data for further analysis. Calls task.prepare_images().
Applies: offset subtraction [bias structure subtraction, if available] cosmetics masking

set_inpdefaults(***kwargs*)

class Pipe.**FOCUS00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()
This is just an assignation of values measured in check_data.

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds FOCUS00 script structure dictionary.

#:param wavelength: int, [nm], wavelength. #:param exptime: int, [ms], exposure time. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

lock_on_stars()

meta_analysis()

prep_data()

class Pipe.**MOT_FF**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

extract_HER()

class Pipe.**MOT_WARM**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()
EXPOSURES: BIAS, RAMP, CHINJ, FLAT, POINT_w x waves_PNT

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds MOT_WARM script structure dictionary.

Parameters

- **diffvalues** – dict, opt, differential values.
- **elvis** – char, ELVIS version.

check_data()

check_metrics_ST(***kwargs*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

lock_on_stars()

class Pipe.**NL01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds NL01 script structure dictionary.

#:param expts: list of ints [ms], exposure times. #:param exptinter: int, ms, exposure time of interleaved source-stability exposures. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 0 (Neutral Density Filter) :param diffvalues: dict, opt, differential values.

do_satCTE()
METACODE

```
select ObsIDs with fluence(exptime) >~ 0.5 FWC

f.e. ObsID:
    CCD:
        Q:
            measure CTE from amount of charge in over-scan relative to␣
↪fluence

f.e. CCD:
    Q:
        get curve of CTE vs. fluence
        measure FWC from curve in ADU

report FWCs in electrons [via gain in inputs] f.e. CCD, Q (table)
```

extract_stats()

Performs basic analysis: extracts statistics from image regions to later build NLC.

METACODE

```
create segmentation map given grid parameters

f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            f.e. "img-segment": (done elsewhere)
                measure central value
                measure variance
```

filterexposures(*structure*, *explog*, *OBSID_lims*)

Loads a list of Exposure Logs and selects exposures from test NL01.

The filtering takes into account an expected structure for the acquisition script.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

prep_data()

Takes Raw Data and prepares it for further analysis.

METACODE

```
f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            mask-out bad pixels
            mask-out detector cosmetics
            subtract offset
            opt: [sub bias frame]
```

produce_NLCs()

METACODE

```
Obtains Best-Fit Non-Linearity Curve

f.e. CCD:
    f.e. Q:

        [opt] apply correction for source variability (interspersed␣
↪exposure
```

```
          with constant exptime)
        Build NL Curve (NLC) - use stats and exptimes
        fit poly. shape to NL curve

plot NL curves for each CCD, Q
report max. values of NL (table)
```

recalibrate_exptimes (*exptimes*)
Corrects exposure times given independent calibration of the shutter.

class Pipe.**NL02** (*inputs, log=None, drill=False, debug=False, cleanafter=False*)

build_scriptdict (*diffvalues={}, elvis='7.5.X'*)
Builds NL02 script structure dictionary.

do_satCTE ()
METACODE

```
select ObsIDs with fluence(exptime) >~ 0.5 FWC

f.e. ObsID:
    CCD:
        Q:
            measure CTE from amount of charge in over-scan relative to␣
↪fluence

f.e. CCD:
    Q:
        get curve of CTE vs. fluence
        measure FWC from curve in ADU

report FWCs in electrons [via gain in inputs] f.e. CCD, Q (table)
```

prep_data ()
Takes Raw Data and prepares it for further analysis.

METACODE

```
f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            mask-out bad pixels
            mask-out detector cosmetics
            subtract offset
            opt: [sub bias frame]
```

produce_NLCs ()
METACODE

```
Obtains Best-Fit Non-Linearity Curve

f.e. CCD:
    f.e. Q:

        [opt] apply correction for source variability (interspersed␣
↪exposure
          with constant exptime)
        Build NL Curve (NLC) - use stats and exptimes
```

```
        fit poly. shape to NL curve

plot NL curves for each CCD, Q
report max. values of NL (table)
```

class Pipe.**PERSIST01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()

Basic analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        measure stats in pix satur MASK across OBSIDs
         (pre-satur, satur, post-satur)
```

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds PERSISTENCE01 script structure dictionary.

Parameters

- **exptSATUR** – int, saturation exposure time.
- **exptLATEN** – int, latency exposure time.
- **diffvalues** – dict, opt, differential values.

check_data()

PERSIST01: Checks quality of ingested data.

METACODE

```
check common HK values are within safe / nominal margins
check voltages in HK match commanded voltages, within margins

f.e.ObsID:
    f.e.CCD:
        f.e.Q.:
            measure offsets in pre-, over-
            measure std in pre-, over-
            measure fluence in apertures around Point Sources

assess std in pre- (~RON) is within allocated margins
assess offsets in pre-, and over- are equal, within allocated  margins
assess fluence is ~expected within apertures (PS) for each frame (pre-
↪satur, satur, post-satur)


plot point source fluence vs. OBSID, all sources
[plot std vs. time]

issue any warnings to log
issue update to report
```

check_metrics_ST(***kwargs*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

get_satur_masks()

Basic analysis of data.

METACODE

```
f.e.CCD:
    use SATURATED frame to generate pixel saturation MASKs
```

meta_analysis()

Meta-analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        estimate delta-charge_0 and decay tau from time-series

report:
    persistence level (delta-charge_0) and time constant
```

prep_data()

PERSIST01: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction cosmetics masking

set_inpdefaults(***kwargs*)

class Pipe.**PTC0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds PTC0X script structure dictionary.

#:param exptimes: list of ints [ms], exposure times. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 800 nm. :param diffvalues: dict, opt, differential values.

extract_HER()

Hard Edge Response Analysis

extract_PTC()

Performs basic analysis of images:

- builds PTC curves: both on non-binned and binned images

METACODE

```
create list of OBSID pairs

create segmentation map given grid parameters

f.e. OBSID pair:
    CCD:
        Q:
            subtract CCD images
            f.e. segment:
                measure central value
                measure variance
```

f_extract_PTC(*ccdobjcol*, *medcol*, *varcol*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

meta_analysis()

Analyzes the variance and fluence: gain, and gain(fluence)

METACODE

```
f.e. CCD:
    Q:
        (using stats across segments:)
        fit PTC to quadratic model
        solve for gain
        solve for alpha (pixel-correls, Guyonnet+15)
        solve for blooming limit (ADU)
            convert bloom limit to electrons, using gain

plot PTC curves with best-fit f.e. CCD, Q
report on gain estimates f. e. CCD, Q (table)
report on blooming limits (table)
```

set_inpdefaults(***kwargs*)

class Pipe.**STRAY00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds STRAY00 script structure dictionary. :param diffvalues: dict, opt, differential values.

filterexposures(*structure*, *explog*, *OBSID_lims*)

set_inpdefaults(***kwargs*)

class Pipe.**TP00**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

check_data()
TP01: Checks quality of ingested data.

METACODE

```
check common HK values are within safe / nominal margins
check voltages in HK match commanded voltages, within margins

f.e.ObsID:
    f.e.CCD:
        f.e.Q.:
            measure offsets in pre-, over-
            measure std in pre-, over-
            measure mean in img-

assess std in pre- (~RON) is within allocated margins
assess offsets in pre-, and over- are equal, within allocated margins
assess offsets are within allocated margins
assess injection level is within expected margins

plot histogram of injected levels for each Q
[plot std vs. time]

issue any warnings to log
issue update to report
```

filterexposures(*structure*, *explog*, *OBSID_lims*)

set_inpdefaults(***kwargs*)

class `Pipe.TP01` (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis ()

Basic analysis of data.

METACODE

```
f. e. ObsID [there are different TOI_TP and TP-patterns]:
    f.e.CCD:
        f.e.Q:
            load "map of relative pumping"
            find_dipoles:
                x, y, rel-amplitude, orientation

produce & report:
    map location of dipoles
    PDF of dipole amplitudes (for N and S)
    Counts of dipoles (and N vs. S)
```

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)

extract ()

Obtain maps of dipoles.

METACODE

```
f.e. id_delay (there are 2):
    f.e. CCD:
        f.e. Q:
            produce reference non-pumped injection map

f. e. ObsID:
    f.e. CCD:

        load ccdobj
        f.e.Q.:
            divide ccdobj.Q by injection map

        save dipole map and store reference
```

filterexposures (*structure*, *explog*, *OBSID_lims*)

meta_analysis ()

Meta-analysis of data:

Try to identify tau and pixel-phase location for each trap. Need to associate dipoles across TOI_TPs and TP-patterns

METACODE

```
across TOI_TP, patterns:

    build catalog of traps: x,y, tp-mode, tau, Pc
    tau, Pc = f({A,TOI})

Report on :
    Histogram of Taus
    Histogram of Pc (capture probability)
    Histogram of I-phases (larger phases should have more traps,
                      statistically) -> check
```

```
    Total Count of Traps
```

set_inpdefaults (***kwargs*)

class Pipe.**TP02** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis ()

Basic analysis of data.

METACODE

```
f. e. ObsID [there are different TOI_TP and TP-patterns]:
    f.e.CCD:
        f.e.Q:
            load raw 1D map of relative pumping (from extract_data)
            identify dipoles:
                x, rel-amplitude, orientation (E or W)

produce & report:
    map location of dipoles
    PDF of dipole amplitudes (for E and W)
    Counts of dipoles (and E vs. W)
```

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)

extract ()

Obtain Maps of Serial Dipoles.

filterexposures (*structure*, *explog*, *OBSID_lims*)

meta_analysis ()

Meta-analysis of data:

Try to identify tau and pixel-phase location for each trap. Need to associate dipoles across TOI_TPs and TP-patterns

METACODE

```
across TOI_TP, patterns:
    build catalog of traps: x,y,R-phase, amp(dwell)
    from Amp(dwell) -> tau, Pc

Report on :
   Histogram of Taus
   Histogram of Pc (capture probability)
   Histogram of R-phases

   Total Count of Traps
```

set_inpdefaults (***kwargs*)

class Pipe.**TP11** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

class Pipe.**TP21** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

Pipe.**dotask** (*taskname*, *inputs*, *drill=False*, *debug=False*, *cleanafter=False*)

Generic test master function.

Pipe.**wait_and_run** (*dayfolder*, *elvis='7.5.X'*)

5.1.3 task.py

Generic Task (Test) Class.

Created on Tue Nov 14 14:20:04 2017

author Ruyman Azzollini

class `vison.pipe.task.Task`(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`IsComplianceMatrixOK`(*complidict*)

`addComplianceMatrix2Log`(*complidict*, *label=''*)

`addComplianceMatrix2Report`(*complidict*, *label=''*, *caption=''*)

`addFigure2Report`(*figkey*)

`addFigures_ST`(*dobuilddata=True*, ***kwargs*)

`addFlagsToLog`()

`addFlagsToReport`()

`addHKPlotsMatrix`()
Adds to self.report a table-figure with HK [self.HKKeys] during test.

`addHK_2_dd`()

`add_data_inventory_to_report`(*tDict*)

`add_inputs_to_report`()

`add_labels_to_explog`(*explog*, *structure*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)

`catchtraceback`()

`check_HK`(*HKKeys*, *reference='command'*, *limits='P'*, *tag=''*, *doReport=False*, *doLog=True*)

`check_HK_ST`()

`check_data`(***kwargs*)
Generic check_data method

`check_metrics_T`()

`check_stat_perCCD`(*arr, CCDlims, CCDs=['CCD1', 'CCD2', 'CCD3']*)

`check_stat_perCCDQandCol`(*arr, lims, CCDs=['CCD1', 'CCD2', 'CCD3']*)

`check_stat_perCCDandCol`(*arr, lims, CCDs=['CCD1', 'CCD2', 'CCD3']*)

`check_stat_perCCDandQ`(*arr, CCDQlims, CCDs=['CCD1', 'CCD2', 'CCD3']*)

`cleanaux`()

`create_mockexplog`(*OBSID0=1000*)

`doPlot`(*figkey*, ***kwargs*)

`filterexposures`(*structure*, *explog*, *OBSID_lims*, *colorblind=False*, *wavedkeys=[]*, *surrogate=''*)
Loads a list of Exposure Logs and selects exposures from test 'test'.

The filtering takes into account an expected structure for the acquisition script.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

get_checkstats_T()
“

prepare_images(*doExtract=True*, *doBadPixels=False*, *doMask=False*, *doOffset=False*, *doBias=False*, *doFF=False*)

recover_progress(*DataDictFile*, *reportobjFile*)

save_progress(*DataDictFile*, *reportobjFile*)

skipMissingPlot(*key*, *ref*)

class vison.pipe.task.**Task**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

IsComplianceMatrixOK(*complidict*)

addComplianceMatrix2Log(*complidict*, *label=’‘*)

addComplianceMatrix2Report(*complidict*, *label=’‘*, *caption=’‘*)

addFigure2Report(*figkey*)

addFigures_ST(*dobuilddata=True*, ***kwargs*)

addFlagsToLog()

addFlagsToReport()

addHKPlotsMatrix()
Adds to self.report a table-figure with HK [self.HKKeys] during test.

addHK_2_dd()

add_data_inventory_to_report(*tDict*)

add_inputs_to_report()

add_labels_to_explog(*explog*, *structure*)

build_scriptdict(*diffvalues={}*, *elvis=‘7.5.X’*)

catchtraceback()

check_HK(*HKKeys*, *reference=’command’*, *limits=’P’*, *tag=’‘*, *doReport=False*, *doLog=True*)

check_HK_ST()

check_data(***kwargs*)
Generic check_data method

check_metrics_T()

check_stat_perCCD(*arr, CCDlims, CCDs=[’CCD1’, ‘CCD2’, ‘CCD3’]*)

check_stat_perCCDQandCol(*arr, lims, CCDs=[’CCD1’, ‘CCD2’, ‘CCD3’]*)

check_stat_perCCDandCol(*arr, lims, CCDs=[’CCD1’, ‘CCD2’, ‘CCD3’]*)

check_stat_perCCDandQ(*arr, CCDQlims, CCDs=[’CCD1’, ‘CCD2’, ‘CCD3’]*)

cleanaux()

create_mockexplog(*OBSID0=1000*)

doPlot(*figkey*, ***kwargs*)

filterexposures (*structure*, *explog*, *OBSID_lims*, *colorblind=False*, *wavedkeys=[]*, *surrogate=''*)
Loads a list of Exposure Logs and selects exposures from test 'test'.

The filtering takes into account an expected structure for the acquisition script.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

get_checkstats_T ()
"

prepare_images (*doExtract=True*, *doBadPixels=False*, *doMask=False*, *doOffset=False*, *doBias=False*, *doFF=False*)

recover_progress (*DataDictFile*, *reportobjFile*)

save_progress (*DataDictFile*, *reportobjFile*)

skipMissingPlot (*key*, *ref*)

CHAPTER

SIX

DATA MODEL

Modules with classes to hold data model for inputs and outputs: exposure log, HK files, FITS files, etc.

6.1 Data Model

6.1.1 ccd.py

Data model for Euclid-VIS CCDs (ground testing at MSSL).

VIS CCDs have 4 quadrants (E,F,G & H). Each quadrant has 2048 (hor.) x 2066 (ver.) active pixels. To these we have to add 51 columns of prescan and 20 columns of overscan (29 in flight/FM). There are also 20 lines of parallel overscan. These pre and over scan pixels are virtual.

The main class in this module, CCD, allows to load, store, manipulate and save images generated by the instrument ROEs and saved in FITS format by the acquisition software, ELVIS.

History

Created on Fri Nov 13 17:42:36 2015

author Ruyman Azzollini

class `vison.datamodel.ccd.`**`CCD`**(*infits=None*, *extensions=None*, *getallextensions=False*, *withpover=True*, *overscan=20*)

Class of CCD273 objects.

Input are Euclid Images as acquired by ELVIS software (Euclid LabView Imaging Software).

The class has been extended to handle multi-extension images. This is useful to also "host" calibration data-products, such as Flat-Fields.

A note on Coordinates Systems:

- **'CCD': referenced to the first pixel readout from channel H. All 4 quadrants** are in a single array, their detection nodes in the 4 "corners" of the rectangle. Same system as images are displayed on DS9. In clock-wise sense, quadrants are H (bottom-left), E (top-left), F (top-right), and G (bottom-right).
- Physical: same as 'CCD' but takes into account virtual and non-active pixels. It is the closest to measuring real "distances" on the silicon.
- 'Quadrant-canonical': Quadrant coordinates system in which the first pixel is the first pixel read out (closest pixel to the readout node), and the last is the last readout. In this system, the serial pre-scan comes before the image area, and this before the serial overscan. Parallel overscan comes after image area in the parallel direction. In this system, coordinates of pixels across quadrants, for a single readout, correspond to the same point in time. Useful when doing cross-talk analysis, for example.

- 'Quadrant-relative': quadrant coordinates system with the same relative orientation as in the 'CCD' system, but referenced to the 'lower-left' pixel of the given quadrant in such system. In this system, the readout node is in a different corner for each quadrant: lower-left for H, top-left for E, top-right for F and bottom-right for G.

add_extension (*data*, *header=None*, *label=None*, *headerdict=None*)
Appends an extension to self (extensions are in a list).

add_to_hist (*action*, *extension=-1*, *vison=u'0.9+374.gd7313d7'*, *params={}*)
Adds information to historial of operations applied on object.

cooconvert (*x*, *y*, *insys*, *outsys*, *Q='U'*)
Coordinates conversion between different systems.

del_extension (*ixextension*)
Deletes an extension from self, by index.

divide_by_flatfield (*FF*, *extension=-1*)
Divides CCD image by a Flat-field.

do_Vscan_Mask (*VSTART*, *VEND*)
Returns a vertical scan mask.

dummyrebin (*arr*, *new_shape*, *stat='median'*)

extract_region (*ccdobj*, *Q*, *area='img'*, *vstart=0*, *vend=2086*, *Full=False*, *canonical=True*, *extension=-1*)

flip_tocanonical (*array*, *Quad*)
Reorients an array to canonical orientation, according to quadrant. Assuming the array is in "relative" orientation

get_1Dprofile (*ccdobj*, *Q*, *orient='hor'*, *area='img'*, *stacker='mean'*, *vstart=0*, *vend=2086*, *extension=-1*)

get_Q (*x*, *y*, *w*, *h*)

get_cutout (*corners*, *Quadrant*, *canonical=False*, *extension=-1*)
Returns a cutout from the CCD image, either in canonical or non-canonical orientation.

Parameters

- **corners** (`list (of int)`) – [x0,x1,y0,y1]
- **Quadrant** (`char`) – Quadrant, one of 'E', 'F', 'G', 'H'
- **canonical** (`bool`) – Canonical [True] = with readout-node at pixel index (0,0) regardless of quadrant. This is the orientation which corresponds to the data-readin order (useful for cross-talk measurements, for example). Non-Canonical [False] = with readout-node at corner matching placement of quadrant on the CCD. This is the orientation that would match the representation of the image on DS9.
- **extension** (`int`) – extension number. Default = -1 (last)

get_mask (*mask*)
Loads a mask into the extensions.

get_quad (*Quadrant*, *canonical=False*, *extension=-1*)
Returns a quadrant in canonical or non-canonical orientation.

Parameters

- **Quadrant** (`char`) – Quadrant, one of 'E', 'F', 'G', 'H'
- **canonical** –

Canonical [True] = with readout-node at pixel index (0,0) regardless of quadrant. This is the orientation which corresponds to the data-reading order (useful for cross-talk measurements, for example). Non-Canonical [False] = with readout-node at corner matching placement of quadrant on the CCD. This is the orientation that would match the representation of the image on DS9.

Parameters **extension** (*int*) – extension number. Default = -1 (last)

get_region2Dmodel(*ccdobj*, *Q*, *area='img'*, *kind='spline'*, *splinemethod='cubic'*, *pdegree=2*, *doFilter=False*, *doBin=True*, *filtsize=1*, *binsize=1*, *filtertype='mean'*, *recoveredges=False*, *vstart=0*, *vend=2086*, *canonical=True*, *extension=-1*)

Parameters

- **ccdobj** (*object*) – ccd object
- **Q** (*char*) – Quadrant
- **kind** (*char*) – type of interpolation
- **doFilter** (*bool*) – boolean to control whether to apply image filtering or not
- **filtsize** (*int*) – size of the filter to be applied (on a side)
- **doBin** (*bool*) – apply binning?
- **binsize** (*int*) – size of the binning window (on a side)
- **filtertype** (*char*) – type of the filter, if applied
- **recoveredges** (*bool*) – try to recover edges of array after binning / filtering?
- **vstart** (*int*) – windowing parameter, start line (from 0) to consider in image area
- **vend** (*int*) – windowing parameter, end line (+1) to consider in image area
- **canonical** (*bool*) – [quadrant] orientation of the resulting model
- **extension** (*int*) – image extension to be used from ccdobj

get_stats(*Quadrant*, *sector='img'*, *statkeys=None*, *trimscan=None*, *ignore_pover=True*, *extension=-1*, *VSTART=0*, *VEND=2086*, *clip=None*)

get_tile_coos(*Quadrant*, *wpx*, *hpx*, *noedges=False*)

Returns a dictionary with a tiling [coordinates of corners of tiles] of quadrant Q, with tiles of size wpx[width] x hpx[height].

CAUTION: Returned coordinates are Q-relative.

Parameters

- **Quadrant** – str, Quadrant, one of ['E','F','G','H']
- **wpx** – int, width [along NAXIS1] of tiles, in pixels.
- **hpx** – int, height [along NAXIS2] of tiles, in pixels.

Returns tiles_dict = dict(wpx='Width of tiles, integer', hpx='Height of tiles, integer', llpix='Lower left corner of tiles, list of tuples', ccpix= 'Central pixel of tiles, list of tuples', Nsamps='Number of tiles, integer')

get_tiles(*Quadrant*, *tile_coos*, *extension=-1*)

Returns cutouts from Quadrant using the coordinates in tile_coos.

Parameters

- **Quadrant** (*str*) – Quadrant where to take the cutouts from.

- **tile_coos** (*dict ()*) – A dictionary with tiles coordinates, as output by get_tile_coos.
- **extension** (*int*) – extension to consider, by index. Last is -1.

Returns A list with the cutouts.

get_tiles_stats (*Quad*, *tile_coos*, *statkey*, *extension=-1*)
Returns statistics on a list of tiles.

Parameters

- **Quad** (*str*) – Quadrant where to take the cutouts from.
- **tile_coos** (*dict ()*) – A dictionary with tiles coordinates, as output by get_tile_coos.
- **statkey** (*str*) – stat to retrieve (one of mean, median, std)
- **extension** (*int*) – Extension index, last is -1

Returns A 1D numpy array with the stat values for the tiles.

getsectioncollims (*Q*)
Returns limits of [HORIZONTAL] sections: prescan, image and overscan

getsectionrowlims (*Q*)
Returns limits of [VERTICAL] sections: image [and vertical overscan]

loadfromFITS (*fitsfile, extensions=[-1], getallextensions=False*)
Loads contents of self from a FITS file.

or_mask (*mask*)
Adds (OR) a mask to self.extensions[*].data.mask

set_extension (*data*, *header=None*, *label=None*, *headerdict=None*, *extension=-1*)
Sets extension 'extension' in self.

set_quad (*inQdata*, *Quadrant*, *canonical=False*, *extension=-1*)
Sets the contents of a quadrant in an extension.

sim_window (*ccdobj*, *vstart*, *vend*, *extension=-1*)

simadd_flatilum (*ccdobj*, *levels=None*, *extension=-1*)

simadd_points (*ccdobj*, *flux*, *fwhm*, *CCDID='CCD1'*, *dx=0*, *dy=0*, *extension=-1*)

simadd_poisson (*ccdobj*, *extension=-1*)

simadd_ron (*ccdobj*, *extension=-1*)

sub_bias (*superbias*, *extension=-1*)
Subtracts a superbias from CCD image.

sub_offset (*Quad, method='row', scan='pre', trimscan=[3, 2], ignore_pover=True, extension=-1*)
Subtracts the offset from a quadrant.

writeto (*fitsf*, *clobber=False*, *unsigned16bit=False*)
Writes self to a FITS file.

class vison.datamodel.ccd.**CCDPile** (*infitsList=None*, *ccdobjList=None*, *extension=-1*, *withpover=True*)
Class to hold and operate (e.g. stack) on a bunch of CCD images. Each image (a single extension picked from each) becomes an extension in the pile.

stack (*method='median'*, *dostd=False*)
Stacking images with an stat.

class `vison.datamodel.ccd.`**`Extension`**(*data*, *header=None*, *label=None*, *headerdict=None*)
Extension Class. The images, as FITS files, have extensions, each with an extension, and optionally, a label, and data.

`vison.datamodel.ccd.`**`cooconv_arrays_decorate`**(*func*)
Decorator for conversion of coordinates on arrays.

`vison.datamodel.ccd.`**`test_create_from_scratch`**()

`vison.datamodel.ccd.`**`test_load_ELVIS_fits`**()

6.1.2 ccd_aux.py

Module auxiliary to ccd.py

History

Created on Mon Feb 19 13:14:02 2018

author raf

class `vison.datamodel.ccd_aux.`**`Model2D`**(*img*, *corners=None*)
Class for 2D models of images and images sections.

`bin_img`(*boxsize*, *stat='median'*)
Bins down image in self.

`filter_img`(*filtsize=15*, *filtertype='median'*, *Tests=False*)
Returns filtered version of self.img.

`fit2Dpol_xyz`(*xx*, *yy*, *zz*, *degree=1*)

`get_model_poly2D`(*sampling=1*, *pdegree=5*, *useBin=False*)

`get_model_splines`(*sampling=1*, *splinemethod='cubic'*, *useBin=False*, *recoveredges=False*, *pdegree=5*)

class `vison.datamodel.ccd_aux.`**`Profile1D`**(*x*, *y*)
Class for 1D profiles of images and images sections.

`vison.datamodel.ccd_aux.`**`extract_region`**(*ccdobj*, *Q*, *area='img'*, *vstart=0*, *vend=2086*, *Full=False*, *canonical=True*, *extension=-1*)

`vison.datamodel.ccd_aux.`**`get_1Dprofile`**(*ccdobj*, *Q*, *orient='hor'*, *area='img'*, *stacker='mean'*, *vstart=0*, *vend=2086*, *extension=-1*)

`vison.datamodel.ccd_aux.`**`get_region2Dmodel`**(*ccdobj*, *Q*, *area='img'*, *kind='spline'*, *splinemethod='cubic'*, *pdegree=2*, *doFilter=False*, *doBin=True*, *filtsize=1*, *binsize=1*, *filtertype='mean'*, *recoveredges=False*, *vstart=0*, *vend=2086*, *canonical=True*, *extension=-1*)

Parameters

- **`ccdobj`** (*`object`*) – ccd object
- **`Q`** (*`char`*) – Quadrant
- **`kind`** (*`char`*) – type of interpolation
- **`doFilter`** (*`bool`*) – boolean to control whether to apply image filtering or not
- **`filtsize`** (*`int`*) – size of the filter to be applied (on a side)

- **doBin** (*bool*) – apply binning?
- **binsize** (*int*) – size of the binning window (on a side)
- **filtertype** (*char*) – type of the filter, if applied
- **recoveredges** (*bool*) – try to recover edges of array after binning / filtering?
- **vstart** (*int*) – windowing parameter, start line (from 0) to consider in image area
- **vend** (*int*) – windowing parameter, end line (+1) to consider in image area
- **canonical** (*bool*) – [quadrant] orientation of the resulting model
- **extension** (*int*) – image extension to be used from ccdobj

vison.datamodel.ccd_aux.**rebin**(*arr*, *new_shape*, *stat='mean'*)
“Rebin 2D array arr to shape new_shape by averaging.

6.1.3 cdp.py

Classes to store Calibration Data Products.

History

Created on Tue Feb 27 10:58:42 2018

author Ruyman Azzollini

class vison.datamodel.cdp.**CCD_CDP**(**args*, ***kwargs*)
CCD Calibration Data Product

ingest_inputs(*data*, *meta=None*, *header=None*)

savehardcopy(*filef=''*)

class vison.datamodel.cdp.**CDP**(**args*, ***kwargs*)
Parent CDP Class.

loadfrompickle(*pickf=''*)

savehardcopy(*filef=''*)

savetopickle(*pickf=''*)

class vison.datamodel.cdp.**FitsTables_CDP**(**args*, ***kwargs*)
Fits Table CDP.

fill_Header()

fill_Meta()

fill_Table(*sheet*)

fill_allTables()

ingest_inputs(*data*, *meta=None*, *header=None*, *figs=None*)

init_HDUList()

savehardcopy(*filef=''*)

class vison.datamodel.cdp.**Json_CDP**(**args*, ***kwargs*)
Generic Json Object CDP.

ingest_inputs(*data*, *meta=None*, *header=None*)

loadhardcopy(*filef=''*)

savehardcopy (*filef=''*)

class vison.datamodel.cdp.**LE1_CDP** (**args*, ***kwargs*)
LE1 FPA Image CDP. One extension per Quadrant.

ingest_inputs (*data*, *header=None*, *inextension=-1*, *fillval=0*)

savehardcopy (*filef=''*, *clobber=True*, *uint16=False*)

class vison.datamodel.cdp.**Tables_CDP** (**args*, ***kwargs*)
Table CDP. Can export to excel.

fill_Header (*title=''*)

fill_Meta ()

fill_Sheet (*sheet*)

fill_allDataSheets ()

get_textable (*sheet*, *caption=''*, *fitwidth=False*, *tiny=False*, ***kwargs*)

ingest_inputs (*data*, *meta=None*, *header=None*, *figs=None*)

init_workbook ()

savehardcopy (*filef=''*)

vison.datamodel.cdp.**loadCDPfromPickle** (*pickf*)
Function to load a CDP from a pickle file.

vison.datamodel.cdp.**wraptextable** (*tex*, *ncols=1*, *caption=''*, *fitwidth=False*, *tiny=False*, *longtable=False*)
Auxiliary function to Tables_CDP class

6.1.4 ccdsim.py

Methods to simulate data. Used by ccd.CCD class.

History

Created on Wed Apr 4 11:13:30 2018

author Ruyman Azzollini

vison.datamodel.ccdsim.**sim_window** (*ccdobj*, *vstart*, *vend*, *extension=-1*)

vison.datamodel.ccdsim.**simadd_flatilum** (*ccdobj*, *levels=None*, *extension=-1*)

vison.datamodel.ccdsim.**simadd_points** (*ccdobj*, *flux*, *fwhm*, *CCDID='CCD1'*, *dx=0*, *dy=0*, *extension=-1*)

vison.datamodel.ccdsim.**simadd_poisson** (*ccdobj*, *extension=-1*)

vison.datamodel.ccdsim.**simadd_ron** (*ccdobj*, *extension=-1*)

6.1.5 compliance.py

Some functions to produce COMPLIANCE MATRICES.

History

Created on Mon Apr 9 17:32:03 2018

author raf

vison.datamodel.compliance.**convert_compl_to_nesteditemlist**(*complidict*)

vison.datamodel.compliance.**gen_compliance_tex**(*indict*, *escape=True*, *caption=''*)

vison.datamodel.compliance.**removescalars_from_dict**(*indict*)

6.1.6 core.py

DataDict Class : holds data and results across sub-tasks of a "task" (Test).
This is the **CORE data-structure** used to do analysis and report results.

History

Created on Thu Sep 21 16:47:09 2017

author Ruyman Azzollini

class vison.datamodel.core.**DataDict**(*meta=None*)

A Task object has associated a DataDict object where the input data for the Task/Test, from the EXPLOG and HK files, and also results obtained through the Task.methods() are stored.

So, DataDict is a data structure that usually grows as the Task execution progress.

A DataDict is basically a dictionary of arrays, but with some specific properties:

- All the arrays have a common dimension, equal to the number of frames / OBSIDs in the test.
- **Other dimensions of the arrays may vary, depending on contents.**
  - For example, the OBSID column only has this common dimension.
  - a column holding PSF FWHM ofspots, may have this common dimension, plus a dimension for the 3 CCDs, another for the 4 Quadrants in each CCD, and another for the 5 spots in each Quadrant.
- The DataDict is composed of Column arrays.
- The DataDict object has methods to save / reload from hard copies.

addColumn(*array*, *name*, *indices*, *ix=-1*)
Adds a Column to self.

col_has_index(*colname*, *indexname*)
Verifies whether column *colname* has an index called *indexname*

dropColumn(*colname*)
Removes column *colname* from self.

flattentoTable()

Flattens the multidimensional contents of self to a 2D table.

Returns an astropy.table.Table object.

initColumn(*name*, *indices*, *dtype='float32'*, *valini=0.0*)
Initialises a Column in self.

loadExpLog(*explog*)
Loads the contents of an EXPLOG.

name_indices()
Returns the names of the indices in self.

saveToFile(*outfile*, *format='ascii.commented_header'*)

Saves self to a hardcopy.

uses the .write method of astropy.tables.Table.

class vison.datamodel.core.**FpaDataDict**(*meta=None*)

loadExpLog(*explog*)

vison.datamodel.core.**useCases**()
#TODO:

create a DataDict object from an exposure log. # add a column indexed by ObsID, CCD and Quad # drop a column # create a column from an operation on several columns with different dimensions # save to a text / excel file # save to a pickle file

class vison.datamodel.core.**vColumn**(*array*, *name*, *indices*)
|Class for Column objects. | A column has contents (an array) and an Index/vMultiIndex object associated.

name_indices()

class vison.datamodel.core.**vIndex**(*name*, *vals=None*, *N=0*)
Class for indexes of a Column.

class vison.datamodel.core.**vMultiIndex**(*IndexList=None*)

Class for indices of a DataDict, which is made of Columns.

A MultiIndex is made up of individual Index objects.

append(**args*)
Adds indices to self.

find(*indexname*)
finds the index an index name in self.

get_len(*indexname*)
Returns the length of index indexname in self.

get_names()
Returns the names of all indices in self.

get_shape()
Gets the dimensions of the indices in self.

get_vals(*indexname*)
Returns the values of the index *indexname* in self.

pop(**args*)
Removes indices

update_names()

update_shape()

6.1.7 elvis.py

ELVIS variables dictionaries.

History

Created on Fri Sep 22 12:04:09 2017

author Ruyman Azzollini

6.1.8 EXPLOGtools.py

class `vison.datamodel.EXPLOGtools.`**`ExpLogClass`**(*elvis='7.5.X'*)

`addRow`(*row*)

`iniExplog`()

`summary`()

`writeto`(*outfile*)

`vison.datamodel.EXPLOGtools.`**`iniExplog`**(*elvis*)

`vison.datamodel.EXPLOGtools.`**`loadExpLog`**(*expfile, elvis='7.5.X', safe=False*)
Loads an Exposure Log from file.

`vison.datamodel.EXPLOGtools.`**`mergeExpLogs`**(*explogList, addpedigree=False, verbose=False*)
Merges explog objects in a list.

`vison.datamodel.EXPLOGtools.`**`test`**()
This Tests needs UPDATE (for data access and probably data format)

6.1.9 fpa_dm.py

FPA Data Model(s).

Created on Thu Aug 1 17:05:12 2019

@author: raf

class `vison.datamodel.fpa_dm.`**`FPA_LE1`**(*infits=None*)

`add_extension`(*data, header, label=None, headerdict=None*)

`apply_function_to_ccds`(*ccdfunction, **kwargs*)

`del_extension`(*ixextension*)

`get_CCDID_from_BLCCD`(*BLOCK, CCD*)
BLOCK: block nickname (e.g. 'CURIE') CCDk: 'CCD1', 'CCD2' or 'CCD3'

`get_ccdobj`(*CCDID*)
Returns a CCD Object given a CCDID.

`get_extid`(*CCDID, Q*)
CCDID: e.g. 'C_11' Q: 'E', 'F', 'G' or 'H'

`initialise_as_blank`(*fillval=None*)

`loadfromFITS`(*infits*)

savetoFITS (*outfits*, *clobber=True*, *unsigned16bit=False*)

set_ccdobj (*ccdobj*, *CCDID*, *inextension=-1*)

set_extension (*iext*, *data*, *header*, *label=None*, *headerdict=None*)

simul (*simputs=None*, *zerofirst=False*)

vison.datamodel.fpa_dm.**test1** ()

6.1.10 generator.py

Script to generate simulated data for pipeline testing purposes.

Created on Tue Aug 29 11:08:56 2017

author Ruyman Azzollini

vison.datamodel.generator.**IMG_bias_gen** (*ccdobj*, *ELdict*, *ogse=None*)

vison.datamodel.generator.**IMG_chinj_gen** (*ccdobj*, *ELdict*, *ogse=None*)

vison.datamodel.generator.**IMG_chinj_gen_v2** (*ccdobj*, *ELdict*, *ogse=None*)

vison.datamodel.generator.**IMG_flat_gen** (*ccdobj*, *ELdict*, *ogse=None*)

vison.datamodel.generator.**IMG_point_gen** (*ccdobj*, *ELdict*, *ogse=None*)

vison.datamodel.generator.**generate_Explog** (*scrdict*, *defaults*, *elvis='7.5.X'*, *explog=None*, *OBSID0=1000*, *date=datetime.datetime(1980, 2, 21, 7, 0)*, *CHAMBER=None*)

Generates a fake ExposureLog from a test structure dictionary.

DEVELOPMENT NOTES:

To be generated: (EASY) *ObsID, *File_name, *CCD, *ROE=ROE1, *DATE, *BUNIT=ADU, SPW_clk=0?,EGSE_ver=elvis,

Temporal: SerRdDel

To be provided in defaults: (EASY) Lab_ver,Con_file,CnvStart, Flsh-Rdout_e_time,C.Inj-Rdout_e_time, FPGA_ver,Chmb_pre,R1CCD[1,2,3]T[T,B]

To be read/parsed/processed from struct: (DIFFICULT)

SerRDel?,SumWell?, IniSweep?,+etc.

vison.datamodel.generator.**generate_FITS** (*ELdict*, *funct*, *filename=''*, *elvis='7.5.X'*, *ogse=None*)

vison.datamodel.generator.**generate_FITS_fromExpLog** (*explog*, *datapath*, *elvis='7.5.X'*, *CHAMBER=None*)

vison.datamodel.generator.**generate_HK** (*explog*, *vals*, *datapath=''*, *elvis='7.5.X'*)

vison.datamodel.generator.**merge_HKfiles** (*HKfilefs*, *masterHKf*)

6.1.11 HKtools.py

House-Keeping inspection and handling tools.

History

Created on Thu Mar 10 12:11:58 2016

author Ruyman Azzollini

vison.datamodel.HKtools.**check_HK_abs** (*HKKeys*, *dd*, *limits='S'*, *elvis='7.5.X'*)
Returns report on HK parameters, in DataDict (dd), compared to absolute limits.

HK Keys which have "relative" limits, always return False.

Parameters

- **HKKeys** – list of HK parameters, as named in HK files (without **HK_** suffix)
- **dd** – DataDict object
- **limits** – type of limits to use, either "P" (Performance) or "S" (Safe)
- **elvis** – ELVIS version to find correspondence between HK key and Exposure Log input (commanded voltage).

Returns report dictionary with pairs of HK-key : Bool. True = All values for given key are within limits. False = At least one value for given key is outside limits.

vison.datamodel.HKtools.**check_HK_vs_command** (*HKKeys*, *dd*, *limits='P'*, *elvis='7.5.X'*)
Returns report on HK parameters, in DataDict (dd), comparing inputs (commanded) vs. output (HK data).

HK Keys which do not correspond to commanded voltages always return 'True'.

Parameters

- **HKKeys** – list of HK parameters, as named in HK files (without **HK_** suffix)
- **dd** – DataDict object
- **limits** – type of limits to use, either "P" (Performance) or "S" (Safe)
- **elvis** – ELVIS version to find correspondence between HK key and Exposure Log input (commanded voltage).

Returns report dictionary with pairs of HK-key : Bool. True = All values are within limits, referred to commanded value. False = At least one value is outside limits, referred to commanded value.

vison.datamodel.HKtools.**doHKSinglePlot** (*dtobjs*, *HK*, *HKkey*, *ylabel='V'*, *HKlims=[]*, *filename=''*, *fontsize=10*)
Plots the values of a HK parameter as a function of time.

Parameters

- **dtobjs** – datetime objects time axis.
- **HK** – HK values (array)
- **HKkey** –
- **ylabel** –
- **HKlims** –
- **filename** – file-name to store plot [empty string not to save].

Returns None!!

vison.datamodel.HKtools.**filtervalues** (*values*, *key*)

vison.datamodel.HKtools.**iniHK_QFM** (*elvis='7.5.X'*, *length=0*)

vison.datamodel.HKtools.**loadHK_QFM** (*filename*, *elvis='7.5.X'*, *validate=False*, *safe=False*)
Loads a HK file, or list of HK files.

Structure: astropy table. First column is a timestamp, and there may be a variable number of rows (readings).

Parameters

- **filename** – path to the file to be loaded, including the file itself, or list of paths to HK files.
- **elvis** – "ELVIS" version

Returns astropy table with pairs parameter:[values]

vison.datamodel.HKtools.**loadHK_QFMsingle**(*filename*, *elvis='7.5.X'*, *validate=False*, *safe=False*)

Loads a HK file

Structure: tab separated columns, one per Keyword. First column is a timestamp, and there may be a variable number of rows (readings).

Parameters

- **filename** – path to the file to be loaded, including the file itself
- **elvis** – "ELVIS" version

Returns astropy table with pairs parameter:[values]

vison.datamodel.HKtools.**loadHK_preQM**(*filename*, *elvis='5.7.07'*)

Loads a HK file

It only assumes a structure given by a HK keyword followed by a number of of tab-separated values (number not specified). Note that the length of the values arrays is variable (depends on length of exposure and HK sampling rate).

Parameters **filename** – path to the file to be loaded, including the file itself

Returns dictionary with pairs parameter:[values]

vison.datamodel.HKtools.**mergeHK**(*HKList*)

vison.datamodel.HKtools.**parseDTstr**(*DTstr*)

vison.datamodel.HKtools.**parseHKfiles**(*HKlist*, *elvis='7.5.X'*)

Parameters

- **HKlist** – list of HK files (path+name).
- **elvis** – "ELVIS" version.

Returns [obsids],[dtobjs],[tdeltasec],[HK_keys], [data(nfiles,nstats,nHKparams)]

vison.datamodel.HKtools.**parseHKfname**(*HKfname*)

Parses name of a HK file to retrieve OBSID, date and time, and ROE number.

Parameters **HKfname** – name of HK file.

Returns obsid,dtobj=datetime.datetime(yy,MM,dd,hh,mm,ss),ROE

vison.datamodel.HKtools.**reportHK**(*HKs*, *key*, *reqstat='all'*)

Returns (mean, std, min, max) for each keyword in a list of HK dictionaries (output from loadHK).

Parameters

- **HK** – dictionary with HK data.
- **key** – HK key.

Reqstat what statistic to retrieve.

vison.datamodel.HKtools.**synthHK**(*HK*)

Synthetizes the values for each parameter in a HK dictionary into [mean,std,min,max].

Parameters **HK** – a dictionary as those output by loadHK.

Returns dictionary with pairs parameter:[mean,std,min,max]

6.1.12 inputs.py

Inputs Handling Classes and utilities.

Created on Thu Jan 11 10:34:43 2018

author Ruyman Azzollini

class `vison.datamodel.inputs.Inputs`(**args*, ***kwargs*)
Class to hold, transfer and 'document' Task Inputs.

6.1.13 QLAtools.py

Quick-Look-Analysis Tools.

History

Created on Wed Mar 16 11:31:58 2016

@author: Ruyman Azzollini

`vison.datamodel.QLAtools.dissectFITS`(*FITSfile*, *path=''*)

`vison.datamodel.QLAtools.getacrosscolscut`(*CCDobj*)

`vison.datamodel.QLAtools.getacrossrowscut`(*CCDobj*)

`vison.datamodel.QLAtools.getsectionstats`(*CCDobj*, *QUAD*, *section*, *xbuffer=(0, 0)*, *ybuffer=(0, 0)*)

`vison.datamodel.QLAtools.plotAcCOLcuts`(*dissection*, *filename=None*, *suptitle=''*)

`vison.datamodel.QLAtools.plotAcROWcuts`(*dissection*, *filename=None*, *suptitle=''*)

`vison.datamodel.QLAtools.plotQuads`(*CCDobj*, *filename=None*, *suptitle=''*)

`vison.datamodel.QLAtools.reportFITS`(*FITSfile*, *outpath=''*)

6.1.14 scriptic.py

Classes and functions to generate ELVIS commanding scripts automatically.

Created on Wed May 24 15:31:54 2017

author Ruyman Azzollini

class `vison.datamodel.scriptic.Script`(*defaults=None*, *structure=None*, *elvis='7.5.X'*)
Core Class that provides automatic test script generation and validation.

build_cargo()
Updates 'cargo' attribute. 'cargo': list of lists, each corresponding to a column in the script.

Each element in the inner lists is a register value. The first column corresponds to the column with key names.

Note: the number of frames is accumuled across columns, as ELVIS expects.

get_struct_from_cargo()

load(**args*, ***kwargs*)
alias method. Points to 'load_to_cargo'.

load_to_cargo(*scriptname*, *elvis='7.5.X'*)
Loads an script from an excel file.

Parameters

- **scriptname** – char, script to load
- **elvis** – char, ELVIS version of script to load

validate(*defaults*, *structure*, *elvis='7.5.X'*)
Not sure 'validation' will work like as implemented... TODO: validate self.validate

write(*scriptname*)
Writes self.cargo (script) to an excel file.

Parameters **scriptname** – char, name of file where to write script.

vison.datamodel.scriptic.**test0**()

vison.datamodel.scriptic.**update_structdict**(*sdict*, *commvalues*, *diffvalues*)
Updates an script structure with common values and differential values.

Parameters

- **sdict** – dict, dictionary with script structure. Takes precedence over commvalues.
- **commvalues** – dict, dictionary with common values to update sdict.
- **diffvalues** – dict, dictionaty with "differential" values to update "sdict". Takes precedence over sdict and commvalues.

CHAPTER

SEVEN

ANALYSIS (SHARED)

7.1 Analysis (Shared)

7.1.1 ellipse.py

Auxiliary module with functions to generate generalized ellipse masks.

author Ruyman Azzollini

class vison.analysis.ellipse.**TestEllipse**(*methodName='runTest'*)
Unit tests for the ellipse module.

vison.analysis.ellipse.**area_superellip**(*r*, *q*, *c=0*)
Returns area of superellipse, given the semi-major axis length

vison.analysis.ellipse.**dist_superellipse**(*n*, *center*, *q=1.0*, *pos_ang=0.0*, *c=0.0*)
Form an array in which the value of each element is equal to the semi-major axis of the superellipse of specified center, axial ratio, position angle, and c parameter which passes through that element. Useful for super-elliptical aperture photometry.

Inspired on dist_ellipse.pro from AstroLib (IDL).

Note: this program doesn't take into account the change in the order of axes from IDL to Python. That means, that in 'n' and in 'center', the order of the coordinates must be reversed with respect to the case for dist_ellipse.pro, in order to get expected results. Nonetheless, the polar angle means the counter-clock wise angle with respect to the 'y' axis.

Parameters

- **n** – shape of array (N1,N2), it can be an integer (squared shape NxN)
- **center** – center of superellipse radii: (c1,c2)
- **q** – axis ratio r2/r1
- **pos_ang** – position angle of isophotes, in degrees, CCW from axis 1
- **c** – boxyness (c>0) /diskyness (c<0)

vison.analysis.ellipse.**effective_radius**(*area*, *q=1.0*, *c=0.0*)
Returns semi-major axis length of superellipse, given the area

7.1.2 Guyonnet15.py

Library with functions that implement the algorithms described in Guyonnet+15. "Evidence for self-interaction of charge distribution in CCDs" Guyonnet, Astier, Antilogus, Regnault and Doherty 2015

Notes:

- I renamed "x" (pixel boundary index) to "b", to avoid confusion with cartesian "x".
- In paper, X belongsto [(0,1),(1,0),(0,-1),(-1,0)]. Here b is referred to as cardinal points "N","E","S","W". It is linked to matrix index ib, running between 0 and 3.

Created on Thu Sep 22 11:38:24 2016

author Ruyman Azzollini

`vison.analysis.Guyonnet15.`**`correct_estatic`**(*img*, *aijb*)

Corrects an image from pixel-boundaries deformation due to electrostatic forces. Subtracts delta-Q.

Parameters

- **`img`** – image, 2D array
- **`aijb`** – Aijb matrix, 3D array

Returns array, img - delta-Q

`vison.analysis.Guyonnet15.`**`degrade_estatic`**(*img*, *aijb*)

Degrades an image according to matrix of pixel-boundaries deformations. Follows on Eq. 11 of G15. Adds delta-Q.

Parameters

- **`img`** – image, 2D array
- **`aijb`** – Aijb matrix, 3D array

Returns array, img + delta-Q

`vison.analysis.Guyonnet15.`**`fpred_aijb`**(*p*, *i*, *j*, *ib*)

'The smoothing model assumes that a_{ij}^x coefficients are the product of a function of distance from the source charge to the considered boundary (r_{ij}) and that it also trivially depends on the angle between the source-boundary vector and the normal to the boundary (theta_{i,j}^x)'

Eq. 18

Parameters

- **`p`** – parameters of the radial function (list of 2)
- **`i`** – pixel coordinate i
- **`j`** – pixel coordinate j
- **`ib`** – boundary index [0, 1, 2, 3]

Returns f(rij)cos(theta_ij^x)

`vison.analysis.Guyonnet15.`**`frdist`**(*i*, *j*, *ib*)

Distance from the source charge to considered boundary "b"

Parameters

- **`i`** – pixel coordinate i
- **`j`** – pixel coordinate j
- **`ib`** – boundary index [0, 1, 2, 3]

Returns distance r(ijb)

`vison.analysis.Guyonnet15.`**`ftheta_bound`**(*i*, *j*, *ib*)

"[theta_i,j^X is] the angle between the source-boundary vector and the normal to the boundary".

Parameters

- **i** – pixel coordinate i
- **j** – pixel coordinate j
- **ib** – boundary index [0, 1, 2, 3]

Returns theta_i,j^x

vison.analysis.Guyonnet15.**fun_p**(*x*, **p*)
auxiliary function to 'solve_for_psmooth'

vison.analysis.Guyonnet15.**generate_GaussPSF**(*N*, *sigma*)
Create a circular symmetric Gaussian centered on the centre of a NxN matrix/image.

vison.analysis.Guyonnet15.**get_Rdisp**(*img*, *aijb*)
Retrieves map of relative displacements of pixel boundaries, for input img and Aijb matrix.

See G15 - Eq. 6

Parameters

- **img** – image, 2D array
- **aijb** – aijb matrix, 3D array NxNx4

Returns array, relative displacements all boundaries of pixels in img

vison.analysis.Guyonnet15.**get_cross_shape_rough**(*cross*, *pitch=12.0*)

vison.analysis.Guyonnet15.**get_deltaQ**(*img*, *aijb*)
Retrieves deltaQ map for input image and aijb matrix.

See G15 - Eq. 11

Parameters

- **img** – image, 2D array
- **aijb** – Aijb matrix, 3D array

Returns array, matrix with delta-Q for each pixel in img, given aijb

vison.analysis.Guyonnet15.**get_kernel**(*aijb*)
'kernel' is an array (2N-1)x(2N-1)x4. Each plane kernel[:,:,b] is a 2D array with the displacement coefficients aijb, in all directions around a pixel at (0,0).

Parameters **aijb** – array, matrix with displacements in 1st quadrant

Returns kernel matrix, (2N-1)x(2N-1)x4

vison.analysis.Guyonnet15.**plot_map**(*z*, *ii*, *jj*, *title=''*)

vison.analysis.Guyonnet15.**plot_maps_ftheta**(*f*, *ii*, *jj*, *suptitle=''*)

vison.analysis.Guyonnet15.**show_disps_CCD273**(*aijb*, *stretch=5.0*, *peak=28571.428571428572*, *N=25*, *sigma=1.6*, *title=''*, *figname=''*)

vison.analysis.Guyonnet15.**solve_for_A_linalg**(*covij*, *var=1.0*, *mu=1.0*, *doplot=False*, *psmooth=None*, *returnAll=False*, *verbose=False*)
Function to retrieve the A matrix of pixel boundaries displacements, given a matrix of pixel covariances, variance, and mu.

if var==1 and mu==1, it is understood that covij is the correlation matrix.

See section 6.1 of G15.

Parameters

- **covij** – array, squared matrix with pixel covariances.
- **var** – float, variance of the flat-field.
- **mu** – float, mean value of the flat-field.
- **doplot** – if True, plot the fit of the fpred(ijb) function
- **psmooth** – coefficients of the fpred(aijb) function (Eq. 18)
- **returnAll** – bool, controls return values
- **verbose** – bool, be verbose or not.

Returns if returnAll == True, return (aijb, psmooth), otherwise return aijb only

vison.analysis.Guyonnet15.**solve_for_psmooth**(*covij*, *var*, *mu*, *doplot=False*)

Solving (p0,p1) parameters in Eq. 18 using covariance matrix and measured covariance matrix.

Parameters

- **covij** – array, covariance matrix
- **var** – float, variance
- **mu** – float, expected value of pixel values ("mean" of flat-field)
- **doplot** – bool, if True, plot data and best fit model

Returns best-fit parameters, and errors: 2 tuples of 2 elements each

vison.analysis.Guyonnet15.**test0**()

vison.analysis.Guyonnet15.**test_getkernel**()

vison.analysis.Guyonnet15.**test_selfconsist**()

vison.analysis.Guyonnet15.**test_solve**()

CHAPTER

EIGHT

CHARGE INJECTION TOOLS

8.1 Charge Injection Tools

8.1.1 InjTask.py

Created on Wed Dec 6 15:56:00 2017

author Ruyman Azzollini

class `vison.inject.InjTask.`**`InjTask`**`(*args, **kwargs)`

`basic_analysis()`
Basic analysis of data.

METACODE

```
f. e. ObsID:
    f.e.CCD:
        f.e.Q:
            extract average 2D injection pattern (and save)
            produce average profile along/across lines
            measure charge-inj. non-uniformity
            measure charge spillover into non-injection
            measure stats of injection (mean, med, std, min/max, percentiles)

plot average inj. profiles along lines f. each CCD, Q and VOLTAGE
    save as a rationalized set of curves
plot average inj. profiles across lines f. each CCD, Q and VOLTAGE
    save as a rationalized set of  curves

Report injection stats as a table/tables
```

`check_data(**kwargs)`

`check_metrics_ST(**kwargs)`
TODO:

- offset levels (pre and over-scan), abs. and relative
- RON in pre and overscan
- mean fluence/signal in image area [script-column-dependent]
- med fluence/signal in image area [script-column-dependent]
- std in image area [script-column-dependent]

get_FluenceAndGradient_limits()

get_checkstats_ST(***kwargs*)

predict_expected_injlevels(*teststruct*)

prepare_images(*doExtract=True*, *doBadPixels=True*, *doMask=True*, *doOffset=True*, *doBias=True*, *doFF=False*)

InjTask: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [bias structure subtraction, if available] cosmetics masking

8.1.2 lib.py

NEEDSREVISION

Module to provide common tools for analysis of Charge Injection acquisitions.

Created on Thu Sep 14 15:32:10 2017

author Ruyman Azzollini

8.1.3 plot.py

Charge Injection Plotting Tools.

Created on Thu Sep 14 15:39:34 2017

author Ruyman Azzollini

CHAPTER

NINE

"FLAT" ACQ. ANALYSIS TOOLS

9.1 "Flat" Acq. Analysis Tools

9.1.1 FlatTask.py

Created on Mon Dec 4 16:00:10 2017

author Ruyman Azzollini

class `vison.flat.FlatTask.`**`FlatTask`**(**args*, ***kwargs*)

`check_data`()

`check_metrics_ST`(***kwargs*)

TODO:

- offset levels (pre and over-scan), abs. and relative
- RON in pre and overscan
- fluence in image area [script-column-dependent]
- variance in image area [script-column-dependent]

`get_checkstats_ST`(***kwargs*)

9.1.2 FlatFielding.py

Flat-fielding Utilities.

Created on Fri Apr 22 16:13:22 2016

@author: raf

class `vison.pipe.FlatFielding.`**`FlatField`**(*fitsfile='', data={}, meta={}*)

`parse_fits`()

`vison.pipe.FlatFielding.`**`fit2D`**(*xx, yy, zz, degree=1*)

`vison.pipe.FlatFielding.`**`get_ilum`**(*img, pdegree=5, filtsize=15, filtertype='median', Tests=False*)

`vison.pipe.FlatFielding.`**`get_ilum_splines`**(*img, filtsize=25, filtertype='median', Tests=False*) vison.pipe.FlatFielding.**produce_IndivFlats**(*infits*, *outfits*, *settings*, *runonTests*, *processes=6*)

vison.pipe.FlatFielding.**produce_MasterFlat**(*infits*, *outfits*, *mask=None*, *settings={}*)

Produces a Master Flat out of a number of flat-illumination exposures. Takes the outputs from produce_IndivFlats.

vison.pipe.FlatFielding.**produce_SingleFlatfield**(*infits*, *outfits*, *settings={}*, *runonTests=False*)

9.1.3 nl.py

NEEDSREVISION

Module with tools used in NL analysis.

Created on Mon Feb 5 15:51:00 2018

author Ruyman Azzollini

vison.flat.nl.**fNL**(*x*, **p*)

vison.flat.nl.**fNL_wExp**(*x*, **p*)

vison.flat.nl.**fitNL_pol**(*X*, *Y*, *W*, *Exptimes*, *minfitFl*, *maxfitFl*, *display=False*)

vison.flat.nl.**fitNL_taylored**(*X*, *Y*, *W*, *Exptimes*, *minfitFl*, *maxfitFl*, *display=False*, *addExp=False*)

vison.flat.nl.**getXYW_NL**(*fluencesNL*, *exptimes*, *nomG*, *pivotfrac=0.5*, *maxrelflu=None*, *method='spline'*)

vison.flat.nl.**getXYW_NL02**(*fluencesNL*, *exptimes*, *nomG*, *minrelflu=None*, *maxrelflu=None*)

vison.flat.nl.**get_exptime_atfracdynrange**(*flu1D*, *exp1D*, *frac=0.5*, *method='spline'*, *maxrelflu=None*, *debug=False*)

vison.flat.nl.**recalibrate_exptimes**(*exptimes*, *calibrationfile*)

vison.flat.nl.**test_wrap_fitNL**()

vison.flat.nl.**wrap_fitNL_SingleFilter**(*fluences*, *variances*, *exptimes*, *times=array([], dtype=float64)*, *TrackFlux=True*, *subBgd=True*)

vison.flat.nl.**wrap_fitNL_TwoFilters**(*fluences*, *variances*, *exptimes*, *wave*, *times=array([], dtype=float64)*, *TrackFlux=True*, *subBgd=True*, *debug=False*)

vison.flat.nl.**wrap_fitNL_TwoFilters_Alt**(*fluences*, *variances*, *exptimes*, *wave*, *times=array([], dtype=float64)*, *TrackFlux=True*, *debug=False*, *ObsIDs=None*, *NLdeg=4*, *offset=0.0*, *XX=None*, *YY=None*)

9.1.4 ptc.py

NEEDSREVISION

Module with tools used in PTC analysis.

Created on Thu Sep 14 16:29:36 2017

author Ruyman Azzollini

`vison.flat.ptc.`**`fitPTC`**(*means*, *var*, *debug=False*)
Fits Photon Transfer Curve to obtain gain.

`vison.flat.ptc.`**`foo_bloom_advanced`**(*means*, *var*, *_fit*, *debug=False*)
Finds blooming limit (where variance drops, if it does...).

`vison.flat.ptc.`**`foo_bloom_advanced_demoted`**(*means*, *var*, *_fit*, *debug=False*)
Finds blooming limit (where variance drops, if it does...).

CHAPTER

TEN

FPA TESTS

10.1 FPA Tests

These are tests made at FPA level, instead of block-level.

10.1.1 fpamaster.py

Pipelining for FPA analysis.

Created on Wed Oct 2 16:15:56 2019

author Ruyman Azzollini

class `vison.fpatests.fpamaster.`**`FpaPipe`**(*inputdict*, *dolog=True*, *drill=False*, *debug=False*, *startobsid=0*, *processes=1*, *tag=''*, *cleanafter=False*)
Master Class of FM-analysis at block-level of assembly.

10.1.2 fpatask

Generic FPA Test Class

Created on Tue Aug 20 11:25:55 2019

@author: raf

class `vison.fpatests.fpatask.`**`FpaTask`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`add_StandardQuadsTable`(*extractor*, *cdp=None*, *cdpdict=None*)

`catchtraceback`()

`check_data`()

`filterexposures`(*explog*, *OBSID_lims*)
Loads a list of Exposure Logs and selects exposures from test 'test'.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

`get_ImgDictfromLE1`(*LE1*, *doequalise=False*)

`ingest_data`()

`inputsclass`
alias of `Inputs`

iter_overCCDs(*data*, *assigner*, *RetDict=None*)

iterate_over_CCDs_inLE1(*LE1*, *_method*, ***kwargs*)

iterate_over_CCDs_parallel_inLE1(*LE1*, *method*, ***kwargs*)
DOES NOT WORK!!

load_LE1(*LE1fits*)

load_references()

CHAPTER

ELEVEN

IMAGE

11.1 Image Analysis

11.1.1 bits.py

NEEDSREVISION

Image bits analysis tools.

Created on Thu Sep 14 15:54:14 2017

author Ruyman Azzollini

`vison.image.bits.`**`get_histo_bits`**(*ccdobj*, *Q*, *vstart=0*, *vend=2086*)

11.1.2 calibration.py

Common use CDP functions / methods.

Created on Thu Nov 2 16:54:28 2017

author Ruyman Azzollini

`vison.image.calibration.`**`load_FITS_CDPs`**(*FDict*, *dataclass*, ***kwargs*)
Dummy function to load CDPs for all 3 CCDs. Input is of type dict(CCD1='',CCD2='',CCD3='')

11.1.3 cosmetics.py

Created on Wed Aug 1 11:55:12 2018

@author: Ruyman Azzollini

`vison.image.cosmetics.`**`get_Thresholding_DefectsMask`**(*maskdata*, *thresholds*)

`vison.image.cosmetics.`**`mask_badcolumns`**(*mask*, *colthreshold=200*)
Flags entire column of pixels if N>colthreshold pixels in column are bad.

`vison.image.cosmetics.`**`set_extrascans`**(*mask*, *val=0*)

11.1.4 covariance.py

Tools to retrieve covariance matrices for (differences of) Flat-Field images. Used in the context of Brighter-Fatter analysis, mainly.

Created on Wed Mar 7 11:54:54 2018

author Ruyman Azzollini

`vison.image.covariance.`**`f_get_corrmap`**(*sq1*, *sq2*, *N*, *submodel=False*, *estimator='median'*, *clipsigma=4.0*, *debug=False*)

`vison.image.covariance.`**`f_get_corrmap_tests`**(*sq1*, *sq2*, *N*, *submodel=False*, *estimator='median'*, *clipsigma=4.0*, *debug=False*)

`vison.image.covariance.`**`f_get_corrmap_v2`**(*sq1*, *sq2*, *N*, *submodel=False*, *estimator='median'*, *clipsigma=4.0*, *debug=False*)

`vison.image.covariance.`**`get_cov_maps`**(*ccdobjList*, *Npix=4*, *vstart=0*, *vend=2066*, *clipsigma=4.0*, *covfunc='ver1'*, *doBiasCorr=False*, *central='median'*, *doTest=False*, *debug=False*)

`vison.image.covariance.`**`get_sigmaclipcorr`**(*vardif*, *clipsigma*, *estimator*, *dims=None*)

11.1.5 ds9reg.py

DS9 Regions tool.

Created on Fri May 18 15:02:07 2018

author raf

`vison.image.ds9reg.`**`get_body_circles`**(*X*, *Y*, *R=None*, *radius=6.0*)

`vison.image.ds9reg.`**`get_body_ellipses`**(*X*, *Y*, *A=None*, *B=None*, *THETA=None*)

`vison.image.ds9reg.`**`save_spots_as_ds9regs`**(*data*, *regfilename=None*, *regfile=None*, *regtype='circle'*, *clobber=True*)

11.1.6 performance.py

Performance parameters of the ROE+CCDs. Compilation of CCD offsets, offset gradients, RONs... used for checks.

Created on Wed Nov 1 09:57:44 2017

author Ruyman Azzollini

`vison.image.performance.`**`get_offsets_lims`**(*offsets*, *offsets_margins*)

`vison.image.performance.`**`get_perf_rdout`**(*BLOCKID*)

11.1.7 pixbounce.py

Pixel Bounce Analysis methods.

Created on Fri Mar 9 09:50:16 2018

author Ruyman Azzollini

`vison.image.pixbounce.`**`get_pixbounce_from_overscan`**(*ccdobj*, *thresholds=None*)

Retrieves Hard Edge Respose for all Quadrants of a CCD. Uses the transition from image to overscan (along rows). Averages across rows. Input image should have high image-area fluence but not saturating. Rows can be filtered by average fluence in them via "thresholds" keyword. Do not use on images acquired with irradiated CCDs.

11.1.8 sextractor.py

Sextractor interface.

Created on Thu May 17 13:29:05 2018

author raf

class `vison.image.sextractor.`**`VSExtractor`**(*img=None*)

`load_catalog`(*catpath*)

`run_SEx`(*catroot*, *config=None*, *checks=None*, *cleanafter=False*)

`save_img_to_tmp`(*img*, *delete=True*, *close=False*)

CHAPTER

TWELVE

METATESTS

12.1 Metatests

These are "meta-tests" that produce summary test reports for the whole FPA/campaign, from the invidual tests reports of a given kind (e.g. "flats", or "chinj01") of the calibrated blocks.

12.1.1 metacal.py

Created on Fri Jul 19 16:06:40 2019

@author: raf

12.1.2 bf.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.bf.`**`MetaBF`**(**args*, ***kwargs*)

> **`dump_aggregated_results`**()
>
> **`init_fignames`**()
>
> **`parse_single_test`**(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.3 bias.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.bias.`**`MetaBias`**(***kwargs*)

> **`dump_aggregated_results`**()
>
> **`init_fignames`**()
>
> **`init_outcdpnames`**()
>
> **`parse_single_test`**(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.4 chinj01.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.chinj01.`**`MetaChinj01`**(***kwargs*)

`dump_aggregated_results`()

`get_injprof_fits_cdp`(*direction*, *inCDP_header=None*, *IG1=4.5*)

`get_injprof_xlsx_cdp`(*direction*, *inCDP_header=None*, *IG1=4.5*)

`init_fignames`()

`parse_single_test`(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.5 chinj02.py

Created on Thu Aug 22 14:05:36 2019

@author: raf

class `vison.metatests.chinj02.`**`MetaChinj02`**(***kwargs*)

`dump_aggregated_results`()

`init_fignames`()

`parse_single_test`(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.6 cosmetics.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.cosmetics.`**`MetaCosmetics`**(**args*, ***kwargs*)

`dump_aggregated_results`()

`init_fignames`()

`init_outcdpnames`()

`parse_single_test`(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.7 dark.py

Metatests: DARKS

Created on Mon Feb 10 16:07:00 2020

@author: raf

class `vison.metatests.dark.`**`MetaDark`**(***kwargs*)

dump_aggregated_results()

init_fignames()

init_outcdpnames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.8 flat.py

Created on Wed Aug 14 11:49:00 2019

@author: raf

class vison.metatests.flat.**MetaFlat**(***kwargs*)

dump_aggregated_results()

init_fignames()

init_outcdpnames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.9 mot_warm.py

Created on Thu Aug 29 16:28:00 2019

@author: raf

class vison.metatests.mot_warm.**MetaMOT**(***kwargs*)

dump_aggregated_results()

init_fignames()

init_outcdpnames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.10 nl.py

Created on Thu Aug 22 10:33:00 2019

@author: raf

class vison.metatests.nl.**MetaNL**(**args*, ***kwargs*)

dump_aggregated_results()

init_fignames()

init_outcdpnames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.11 persistence.py

Created on Mon Tue 24 16:34:00 2020

@author: raf

class vison.metatests.persistence.**MetaPersist**(**args*, ***kwargs*)

dump_aggregated_results()

init_fignames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

plot_XY(*XYdict*, ***kwargs*)

12.1.12 psf.py

Created on Fri Aug 16 14:19:00 2019

@author: raf

class vison.metatests.psf.**MetaPsf**(**args*, ***kwargs*)

dump_aggregated_results()

get_XTALKDICT_from_PT(*testname*)

init_fignames()

init_outcdpnames()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

plot_XtalkMAP(*XTALKs*, ***kwargs*)

verify_reqs(*XTALKs*)

for each victim channel, the maximum (dominant) coupling with any other channel shall be abs(c) <= 6x10-4.

for each victim channel, only one other channel, the "dominant", can have a coupling factor abs(c)> 7.6x10-5.

12.1.13 ptc.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class vison.metatests.ptc.**MetaPTC**(**args*, ***kwargs*)

dump_aggregated_results()

gen_GAIN_MXdict()

parse_single_test(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.14 tpx1.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.tpx1.`**`MetaTPX1`**(***kwargs*)

`dump_aggregated_results`()

`init_fignames`()

`parse_single_test`(*jrep*, *block*, *testname*, *inventoryitem*)

12.1.15 tpx2.py

Created on Mon Jul 22 17:01:36 2019

@author: raf

class `vison.metatests.tpx2.`**`MetaTPX2`**(***kwargs*)

`dump_aggregated_results`()

`init_fignames`()

`load_block_results`(*inventoryfile=None*)

`parse_single_test`(*jrep*, *block*, *testname*, *inventoryitem*)

CHAPTER

THIRTEEN

MONITORING ("EYEGORE")

Tools to monitor data acquisition on real time: plots of HK, auto-updating of visual display of Exposure Log with some interactive capabilities, and display of latest images.

13.1 Monitoring ("Eyegore")

Fig. 13.1: You must be Igor...

13.1.1 eyegore.py

13.1.2 eyeCCDs.py

Eyegore: CCDs display.

Created on Fri Oct 13 16:16:08 2017

author raf

class vison.eyegore.eyeCCDs.**ImageDisplay**(*parent*, *path*, *elvis='7.5.X'*, *tag=''*)

gen_render()

setup_fig()

13.1.3 eyeHK.py

Eyegore: House Keeping Monitoring.

Created on Fri Oct 13 14:11:41 2017

author raf

class vison.eyegore.eyeHK.**HKDisplay**(*root*, *path*, *interval*, *elvis='7.5.X'*, *dolite=False*, *tag=''*)

get_data()

search_HKfiles()

select_HKkeys()

class vison.eyegore.eyeHK.**HKFlags**(*root*, *parent*, *interval=5000*, *elvis='7.5.X'*, *tag=''*)

MuteFlag(*event*)

ResetFlag(*event*)

UnmuteFlag(*event*)

bind_buttons_to_methods(*ix*)

changeColor(*ix*, *color*)

isflagraised(*ix*)

lowerflag(*ix*)

raiseflag(*ix*)

class vison.eyegore.eyeHK.**SingleHKplot**(*root*)

vison.eyegore.eyeHK.**sort_HKfiles**(*HKfiles*)

vison.eyegore.eyeHK.**validate_within_HKlim**(*val*, *HKlim*)

violation: 0: None -1: below lower limit 1: above upper limit 2: different from limit, if limit is a single value

13.1.4 eyeObs.py

Eyegore: Exposure Log Monitoring.

Created on Fri Oct 13 16:22:36 2017

author raf

class vison.eyegore.eyeObs.**ExpLogDisplay**(*parent*, *path*, *interval*, *elvis='7.5.X'*, *ds9target='DS9:*'*, *tag=''*)

build_elementList()

get_data()

loadExplogs()

search_EXPLOGs()

sortBy(*tree*, *col*, *descending*)
sort tree contents when a column header is clicked

vison.eyegore.eyeObs.**changeNumeric**(*data*)
if the data to be sorted is numeric change to float

vison.eyegore.eyeObs.**isNumeric**(*s*)
test if a string s is numeric

13.1.5 eyeWarnings.py

CHAPTER

FOURTEEN

OGSE

OGSE stands for Optical Ground Support Equipment.

14.1 OGSE Tools

14.1.1 ogse.py

Model of the calibration OGSE

Created on Fri Sep 8 12:11:55 2017

author Ruyman Azzollini

`vison.ogse.ogse.`**`get_FW_ID`**(*wavelength*, *FW={'F1': 590, 'F2': 640, 'F3': 730, 'F4': 800, 'F5': 880, 'F6': 0}*)

returns FW key corresponding to input wavelength. :param wavelength: integer, wavelength.

CHAPTER

FIFTEEN

PLOTTING

General use plotting facilities.

15.1 Plotting

15.1.1 baseplotclasses.py

vison pipeline: Classes to do plots.

Created on Mon Nov 13 17:54:08 2017

author Ruyman Azzollini

class `vison.plot.baseplotclasses.`**`Beam1DHist`**`(`*data*, ***kwargs*`)`

class `vison.plot.baseplotclasses.`**`BeamPlot`**`(`*data*, ***kwargs*`)`

`populate_axes``()`

class `vison.plot.baseplotclasses.`**`ShellPlot`**`(`*data*, ***kwargs*`)`

`vison.plot.baseplotclasses.`**`testBeam2ImgShow`**`()`

class `vison.plot.baseplotclasses.`**`BasicPlot`**`(`***kwargs*`)`

class `vison.plot.baseplotclasses.`**`Beam1DHist`**`(`*data*, ***kwargs*`)`

class `vison.plot.baseplotclasses.`**`BeamImgShow`**`(`*data*, ***kwargs*`)`

class `vison.plot.baseplotclasses.`**`BeamPlot`**`(`*data*, ***kwargs*`)`

`populate_axes``()`

class `vison.plot.baseplotclasses.`**`BeamPlotYvX`**`(`*data*, ***kwargs*`)`

class `vison.plot.baseplotclasses.`**`CCD2DPlot`**`(`*data*, ***kwargs*`)`

class `vison.plot.baseplotclasses.`**`ImgShow`**`(`*data*, ***kwargs*`)`

`plt_trimmer``()`

`populate_axes``()`

15.1.2 figclasses.py

Created on Mon Apr 16 16:17:13 2018

author Ruyman Azzollini

class `vison.plot.figclasses.`**`BlueScreen`**

`build_data``(`**args*, ***kwargs*`)`

`configure``(`***kwargs*`)`

class `vison.plot.figclasses.`**`Fig_Dynamic`**`(`*figname=''*`)`

`plotclass`
alias of `ShellPlot`

class `vison.plot.figclasses.`**`Fig_Husk`**`(`*figname=''*`)`

15.1.3 plots_fpa.py

Classes to handle FPA-level plots

Created on Fri Jul 5 16:16:20 2019

@author: raf

class `vison.plot.plots_fpa.`**`FpaFindingChart`**`(`***kwargs*`)`

`populate_axes``()`

class `vison.plot.plots_fpa.`**`FpaHeatMap`**`(`*data*, ***kwargs*`)`

`populate_axes``()`

class `vison.plot.plots_fpa.`**`FpaPlot`**`(`*data*, ***kwargs*`)`

`populate_axes``()`

15.1.4 trends.py

Plotting classes shared across tasks/sub-tasks and derived from plots.baseclasses. They have in common that they show trends with time of some variables / stats.

Created on Fri Jan 26 16:18:43 2018

author raf

CHAPTER

SIXTEEN

POINT-SOURCE ANALYSIS

16.1 Point-Source Analysis

16.1.1 basis.py

author Ruyman Azzollini

Created on Thu Apr 20 18:56:40 2017

class `vison.point.basis.`**`SpotBase`**(*data*, *log=None*, *verbose=False*)

16.1.2 display.py

Display Library for Point-Source Analysis

Created on Fri Apr 21 14:02:57 2017

requires matplotlib

author Ruyman Azzollini

`vison.point.display.`**`show_spots_allCCDs`**(*spots_bag*, *title=''*, *filename=''*, *dobar=True*)

16.1.3 gauss.py

Gaussian Model of Point-like Sources

Simple class to do Gaussian Fitting to a spot.

requires NumPy, astropy

Created on Thu Apr 20 16:42:47 2017

author Ruyman Azzollini

class `vison.point.gauss.`**`Gaussmeter`**(*data*, *log=None*, *verbose=False*, ***kwargs*)

Provides methods to measure the shape of an object using a 2D Gaussian Model.

Parameters

- **data** (*np.ndarray*) – stamp to be analysed.
- **log** (*instance*) – logger
- **kwargs** (*dict*) – additional keyword arguments

Settings dictionary contains all parameter values needed.

fit_Gauss()

16.1.4 models.py

Models (Point-Like Sources)

Library module with models for processing of point-source imaging data.

requires NumPy

author Ruyman Azzollini

Created on Wed Apr 19 11:47:00 2017

vison.point.models.**fgauss2D**(*x*, *y*, *p*)

A gaussian fitting function where p[0] = amplitude p[1] = x0 p[2] = y0 p[3] = sigmax p[4] = sigmay p[5] = floor

16.1.5 photom.py

Aperture Photometry of point-like objects

Simple class to do aperture photometry on a stamp of a point-source.

requires NumPy

Created on Thu Apr 20 14:37:46 2017

author Ruyman Azzollini

class vison.point.photom.**Photometer**(*data*, *log=None*, *verbose=False*, ***kwargs*)

Provides methods to measure the shape of an object.

Parameters

- **data** (*np.ndarray*) – stamp to be analysed.
- **log** (*instance*) – logger
- **kwargs** (*dict*) – additional keyword arguments

Settings dictionary contains all parameter values needed.

doap_photom(*centre*, *rap*, *rin=-1.0*, *rout=-1.0*, *gain=3.5*, *doErrors=True*, *subbgd=False*)

get_centroid(*rap=None*, *full=False*)

TODO: add aperture masking

measure_bgd(*rin*, *rout*)

sub_bgd(*rin*, *rout*)

16.1.6 shape.py

Quadrupole Moments Shape Measurement

Simple class to measure quadrupole moments and ellipticity of an object.

requires NumPy, PyFITS

author Sami-Matias Niemi, Ruyman Azzollini

class `vison.point.shape.`**`Shapemeter`**(*data*, *log=None*, *verbose=False*, ***kwargs*)
Provides methods to measure the shape of an object.

Parameters

- **`data`** (*`np.ndarray`*) – stamp to be analysed.
- **`log`** (*`instance`*) – logger
- **`kwargs`** (*`dict`*) – additional keyword arguments

Settings dictionary contains all parameter values needed.

`circular2DGaussian`(*x*, *y*, *sigma*)
Create a circular symmetric Gaussian centered on x, y.

Parameters

- **`x`** (*`float`*) – x coordinate of the centre
- **`y`** (*`float`*) – y coordinate of the centre
- **`sigma`** (*`float`*) – standard deviation of the Gaussian, note that sigma_x = sigma_y = sigma

Returns circular Gaussian 2D profile and x and y mesh grid

Return type dict

`ellip2DGaussian`(*x*, *y*, *sigmax*, *sigmay*)
Create a two-dimensional Gaussian centered on x, y.

Parameters

- **`x`** (*`float`*) – x coordinate of the centre
- **`y`** (*`float`*) – y coordinate of the centre
- **`sigmax`** (*`float`*) – standard deviation of the Gaussian in x-direction
- **`sigmay`** (*`float`*) – standard deviation of the Gaussian in y-direction

Returns circular Gaussian 2D profile and x and y mesh grid

Return type dict

`measureRefinedEllipticity`()
Derive a refined iterated polarisability/ellipticity measurement for a given object.

By default polarisability/ellipticity is defined in terms of the Gaussian weighted quadrupole moments. If self.shsettings['weighted'] is False then no weighting scheme is used.

The number of iterations is defined in self.shsettings['iterations'].

Returns centroids [indexing stars from 1], ellipticity (including projected e1 and e2), and R2

Return type dict

quadrupoles (*image*)
Derive quadrupole moments and ellipticity from the input image.

Parameters **img** (*ndarray*) – input image data

Returns quadrupoles, centroid, and ellipticity (also the projected components e1, e2)

Return type dict

writeFITS (*data*, *output*)
Write out a FITS file using PyFITS.

Parameters

- **data** (*ndarray*) – data to write to a FITS file
- **output** (*string*) – name of the output file

Returns None

16.1.7 spot.py

Spot Stamp Class.

Created on Thu Apr 20 15:35:08 2017

author Ruyman Azzollini

class vison.point.spot.**Spot** (*data*, *log=None*, *verbose=False*, *lowerleft=(None,)*, ***kwargs*)
Provides methods to do point-source analysis on a stamp. Aimed at basic analysis:

•Photometry

•Quadrupole Moments

•Gaussian Fit

Parameters

- **data** (*np.ndarray*) – stamp to be analysed.
- **log** (*instance*) – logger
- **kwargs** (*dict*) – additional keyword arguments

Settings dictionary contains all parameter values needed.

get_photom ()
measurements:'apflu','eapflu','bgd','ebgd'

get_shape_Gauss ()

Returns res = dict(i0,ei0,x,ex,y,ey, sigma_x,esigma_x,sigmay,esigma_y, fwhm_x,efwhm_x, fwhm_y,efwhm_y, fluence,efluence)

get_shape_Moments ()

Returns res = dict(x,y,ellip,e1,e2,a,b)

get_shape_easy (*method='G'*, *debug=False*)

measure_basic (*rap=10*, *rin=15*, *rout=-1*, *gain=3.1*, *debug=False*)
TODO: # get basic statistics, measure and subtract background # update centroid # do aperture photometry # pack-up results and return

Parameters

- **rap** – source extraction aperture radius.
- **rin** – inner radius of background annulus.
- **rout** – outer radius of background annulus (-1 to set bound by image area).
- **gain** – image gain (e-/ADU).

16.1.8 lib.py

Library module with useful data and functions for processing of point-source imaging data.

Created on Wed Apr 5 10:21:05 2017

author Ruyman Azzollini (except where indicated)

`vison.point.lib.`**`extract_spot`**(*ccdobj*, *coo*, *Quad*, *log=None*, *stampw=25*) vison.point.lib.**gen_point_mask**(*Quad, width=75, sources='all', coodict={'CCD2': OrderedDict([('H', OrderedDict([('BRAVO', (1725.0, 1606.0)), ('CHARLIE', (1131.0, 1029.0)), ('ALPHA', (554.0, 1626.0)), ('ECHO', (1706.0, 435.0)), ('DELTA', (531.0, 446.0))])), ('E', OrderedDict([('BRAVO', (1716.0, 1700.0)), ('CHARLIE', (1126.0, 1124.0)), ('ALPHA', (542.0, 1725.0)), ('ECHO', (1695.0, 537.0)), ('DELTA', (521.0, 551.0))])), ('G', OrderedDict([('BRAVO', (1702.0, 1571.0)), ('CHARLIE', (1139.0, 1033.0)), ('ALPHA', (534.0, 1590.0)), ('ECHO', (1685.0, 394.0)), ('DELTA', (515.0, 415.0))])), ('F', OrderedDict([('BRAVO', (1745.0, 1668.0)), ('CHARLIE', (1141.0, 1144.0)), ('ALPHA', (578.0, 1686.0)), ('ECHO', (1723.0, 496.0)), ('DELTA', (553.0, 522.0))]))]), 'CCD3': {'H': {'BRAVO': (1689.4, 1668.8000000000002), 'ALPHA': (460.6, 1668.8000000000002), 'DELTA': (460.6, 417.20000000000005), 'ECHO': (1689.4, 417.20000000000005), 'CHARLIE': (1075.0, 1043.0)}, 'E': {'BRAVO': (1689.4, 3754.8), 'ALPHA': (460.6, 3754.8), 'DELTA': (460.6, 2503.2), 'ECHO': (1689.4, 2503.2), 'CHARLIE': (1075.0, 3129.0)}, 'G': {'BRAVO': (3808.4, 1668.8000000000002), 'ALPHA': (2579.6, 1668.8000000000002), 'DELTA': (2579.6, 417.20000000000005), 'ECHO': (3808.4, 417.20000000000005), 'CHARLIE': (3194.0, 1043.0)}, 'F': {'BRAVO': (3808.4, 3754.8), 'ALPHA': (2579.6, 3754.8), 'DELTA': (2579.6, 2503.2), 'ECHO': (3808.4, 2503.2), 'CHARLIE': (3194.0, 3129.0)}}, 'CCD1': {'H': {'ALPHA': (460.6, 1668.8000000000002), 'CHARLIE': (1075.0, 1043.0), 'DELTA': (460.6, 417.20000000000005), 'ECHO': (1689.4, 417.20000000000005), 'BRAVO': (1689.4, 1668.8000000000002)}, 'E': {'ALPHA': (460.6, 3754.8), 'CHARLIE': (1075.0, 3129.0), 'DELTA': (460.6, 2503.2), 'ECHO': (1689.4, 2503.2), 'BRAVO': (1689.4, 3754.8)}, 'G': {'ALPHA': (2579.6, 1668.8000000000002), 'CHARLIE': (3194.0, 1043.0), 'DELTA': (2579.6, 417.20000000000005), 'ECHO': (3808.4, 417.20000000000005), 'BRAVO': (3808.4, 1668.8000000000002)}, 'F': {'ALPHA': (2579.6, 3754.8), 'CHARLIE': (3194.0, 3129.0), 'DELTA': (2579.6, 2503.2), 'ECHO': (3808.4, 2503.2), 'BRAVO': (3808.4, 3754.8)}}, 'names': ['ALPHA', 'BRAVO', 'CHARLIE', 'DELTA', 'ECHO']}*)

CHAPTER

SEVENTEEN

SCRIPTS

These are pipeline scripts, not the Test Scripts (for those keep scrolling down).

17.1 Scripts

17.1.1 HKmonitor.py

TODO

- find HK files in a folder
- parse HK files
- plot HK parameters vs. time
- assemble all plots into a pdf file

Script to produce HK reports out of HK files in a folder. Aimed at quick inspection of data from Characterization and Calibration Campaigns of Euclid-VIS.

History

Created on Tue Mar 15 10:35:43 2016

author

Ruyman Azzollini (MSSL)

17.1.2 QLA.py

QLA script. Aimed at quick inspection of data from Characterization and Calibration Campaigns of Euclid-VIS.

TODO

- find FITS files in a folder
- load FITS file into a CCD object
- **obtain metrics on the data:**
  - image of 4 quadrants
  - across-rows and across-columns plots
  - statistics of image, pre and over scan regions: mea, med, std, p25, p75
- parse header

- do plots
- assemble all plots into a pdf file per image
- merge all pdfs into a single pdf file

History

Created on Wed Mar 16 11:33:21 2016

author

Ruyman Azzollini

17.1.3 quickds9.py

Wrap-up of SAO DS9 to quickly load a number of images, for inspection.

Core engine: pyds9 (http://hea-www.harvard.edu/RD/pyds9/)

History

Created on Thu Mar 17 13:18:10 2016

author

Ruyman Azzollini

17.1.4 vis_mkscripts.py

Automatically Generating Calibration Campaign Data Acquisition Scripts. Aimed at ELVIS.

History

Created on Fri Sep 08 12:03:00 2017

autor Ruyman Azzollini

`vison.scripts.vis_mkscripts.`**`f_write_script`**(*struct*, *filename*, *outpath*, *elvis*)

Function that writes the test structure (as a dictionary) to an .xlsx file.

Calls vison.datamodel.scriptic.Script class.

`vison.scripts.vis_mkscripts.`**`scwriter`**(*toWrite*, *test_generator*, *outpath*, *equipment*, *elvis='7.5.X'*, *CHAMBER=None*)

Parent function writing the test scripts.

- calls test_generator to create a sequence of test objects, instantiated with their inputs.
- the method build_scriptdict() of each test object is called to generate the dictionary-based structure of the test
- test acquisition durations are estimated
- the function f_write_script is called to write the script to an excel.
- a txt file is written with a summary of the tests written.
- a .xlsx file is written with the same summary.
- the checksums of the scripts are written to a .txt file.

Parameters

- **toWrite** (*dict ()*) – a dictionary with pairs of test_name: flat, where flag is either 1 (create script), or 0 (do not create).
- **test_generator** (*function*) – a function that instanstiates test objects for a given campaign type.
- **outpath** (*str*) – outputs directory. Created on the fly if not already-existent.
- **equipment** (*dict ()*) – dictionary with equipment serials.
- **elvis** (*str*) – elvis version to write scripts for.
- **CHAMBER** (*str*) – chamber where the tests will be executed (affects exposure times, for exapmle).

Returns None

`vison.scripts.vis_mkscripts.`**`write_summary_asexcel`**(*summarydict*, *excelf*, *meta*)
Writes a summary of the tests generated in .xlsx format. Uses vison.support.excel.ReportXL class and Pandas.

`vison.scripts.vis_mkscripts.`**`write_summary_astextfile`**(*summarylines*, *inventoryfile*, *meta*)
Writes a summary of the scripts generated, in .txt format.

17.1.5 vis_genDataSet.py

Development: Creating Calibration Campaign Fake Data-sets.

History

Created on Tue Sep 05 16:07:00 2017

autor

Ruyman Azzollini

`vison.scripts.vis_genDataSet.`**`datasetGenerator`**(*TestsSelector*, *doGenExplog*, *doGenHK*, *doGenFITS*, *outpath*, *elvis*, *CHAMBER*, *Nrows=0*)

`vison.scripts.vis_genDataSet.`**`genExpLog`**(*toGen*, *explogf*, *equipment*, *elvis='7.5.X'*, *CHAMBER=None*)

17.1.6 vison_run

This is the main script that executes the pipeline. Because it does not have a .py extension, it is not correctly parsed into Sphynx.

17.1.7 vis_explogs_merger.py

Merges [concatenates] a list of [ELVIS] exposure logs.

History

Created on Fri Feb 9 15:20:01 2018

author

Ruyman Azzollini

vison.scripts.vis_explogs_merger.**explog_merger**(*EList*, *output='EXP_LOG_merged.txt'*, *elvis='7.5.X'*)

17.1.8 vis_run_xtalk.py

Master Script to measure and report cross-talk levels among 12 ROE channels. Takes as input a data-set composed of 3x12 CCD images, corresponding to injecting a "ladder" of signal on each of the 12 channels, using the **ROE-TAB**.

History

Created on Thu Mar 22 16:17:39 2018

author

Ruyman Azzollini

vison.scripts.vis_run_xtalk.**run_xtalk**(*incat*, *inpath=''*, *respath=''*, *metafile=''*, *doCompute=False*)

17.1.9 vis_ROE_LinCalib.py

Non-Linearity Calibration of ROE (on bench).

History

Created on Thu Mar 15 15:32:11 2018

author Ruyman Azzollini

vison.scripts.vis_ROE_LinCalib.**find_adu_levels**(*qdata*, *Nlevels*, *debug=False*)

vison.scripts.vis_ROE_LinCalib.**run_ROE_LinCalib**(*inputsfile*, *incatfile*, *datapath=''*, *respath=''*, *doExtractFits=True*, *dopolyRT=False*, *debug=False*)

17.1.10 vis_ROETAB_LinCalib.py

Linearity Calibration of ROE-TAB.

History

Created on Tue Mar 27 14:42:00 2018 Modified on Fri Sep 14 10:53:00 2018

author Ruyman Azzollini

vison.scripts.vis_ROETAB_LinCalib.**filter_Voltage_uni**(*rV*, *filt_kernel*)

vison.scripts.vis_ROETAB_LinCalib.**find_discrete_voltages_inwaveform**(*rV*, *levels*, *filtered=None*, *debug=False*)

vison.scripts.vis_ROETAB_LinCalib.**load_WF**(*WFf*, *chkNsamp=None*, *chkSampInter=None*)

vison.scripts.vis_ROETAB_LinCalib.**plot_waveform**(*WF*, *disc_voltages=[]*, *figname=''*, *chan='Unknown'*) vison.scripts.vis_ROETAB_LinCalib.**run_ROETAB_LinCalib**(*inputsfile*, *incatfile*, *datapath=''*, *respath=''*, *doBayes=False*, *debug=False*)

17.1.11 vis_star_finder.py

Script to find point sources in VIS Ground Calibration Campaign. Used to 'prime' the position tables of point-source objects.

History

Created on Tue Jun 12 16:09:31 2018

author Ruyman Azzollini

vison.scripts.vis_star_finder.**write_ID_chart**(*filename*, *Quads*, *Starnames*)

17.1.12 vis_load_DD.py

Loading a DataDict object for inspection.

History

Created on Wed Aug 01 10:00:00 2018

autor Ruyman Azzollini

17.1.13 vis_cosmetics_masker.py

Script to create cosmetics masks in VIS Ground Calibration Campaign.

History

Created on Wed Aug 1 11:02:00 2018

author Ruyman Azzollini

vison.scripts.vis_cosmetics_masker.**do_Mask**(*inputs*, *masktype*, *subbgd=True*, *normbybgd=False*, *validrange=None*, *flagWholeColumns=False*)

vison.scripts.vis_cosmetics_masker.**pre_process**(*FITS_list*, *subOffset=False*, *validrange=None*)

vison.scripts.vis_cosmetics_masker.**read_OBSID_list**(*ff*)

vison.scripts.vis_cosmetics_masker.**run_maskmaker**(*inputs*)

17.1.14 vis_clear_space.py

Script to clean-up processing files produced by the pipeline. Useful to free-up space.

USE WITH CAUTION.

History

Created on Thu Dec 6 10:19:24 2018

author raf

`vison.scripts.vis_clear_space.`**`find_and_erase`**(*path*, *keyword*)

17.1.15 vis_reports_merger.py

Reports Merger - Tests

History

Created on Wed Dec 19 10:02:02 2018

author raf

`vison.scripts.vis_reports_merger.`**`run_merger`**(*infile*)

`vison.scripts.vis_reports_merger.`**`run_merger_plus`**(*infile*, *Issue=0.0*)

17.1.16 vis_mksession.py

Script to build test SEQUENCES easily and reliably.

History

Created on Thu Mar 14 10:30:33 2019

author raf

`vison.scripts.vis_mksession.`**`session_builder`**(*inputs*)

17.1.17 vis_dnl_extract.py

Exploring the histograms of counts in images. Do we have "missing codes" due to larger-than-expected DNL? DNL: Differential Non Linearity

This issue was brought to attention of VIS IDT by Ralf Kohley on Jan 10th 2019.

History

Created on Tue Feb 5 09:35:25 2019

author raf

`vison.scripts.vis_dnl_extract.`**`batch_extract_dnls`**(*explog*, *respath*, *interptype*, *multithread=1*)

`vison.scripts.vis_dnl_extract.`**`batch_extract_histos`**(*explog*, *respath*, *multithread=1*)

`vison.scripts.vis_dnl_extract.`**`explog_selector`**(*explog*, *testkeys*)

`vison.scripts.vis_dnl_extract.`**`extract_histograms`**(*infile_name*, *i*, *N*)

`vison.scripts.vis_dnl_extract.`**`get_average_dnls`**(*explog*, *respath*, *tag*)

`vison.scripts.vis_dnl_extract.`**`interpol_histo_cubic`**(*histo*)
NOT WORKING PROPERLY, DO NOT USE

`vison.scripts.vis_dnl_extract.`**`interpol_histo_linear`**(*histo*)

`vison.scripts.vis_dnl_extract.`**`save_to_fits`**(*array*, *fitsfile*)

CHAPTER

EIGHTEEN

SUPPORT CODE

18.1 Support Code

18.1.1 context.py

Common Values which are used by functions and classes throughout pipeline.

Created on Tue Jan 16 10:53:40 2018

author Ruyman Azzollini

18.1.2 ET.py

Module to issue WARNING / ALERT phone calls to designated phone numbers. Uses Twilio.

'... E.T. phone home...'

Created on Thu Sep 14 10:13:12 2017

author raf

class `vison.support.ET.`**`ET`**

Class to do phone calls.

`dial_numbers`(*url*)

Dials one or more phone numbers from a Twilio phone number.

Parameters **`url`** – char, URL with the TwiML code that Twilio uses as instructions on call. Basically, it provides a message to be voiced, as intended.

`send_sms`(*body*)

`vison.support.ET.`**`grab_numbers_and_codes`**()

Retrieves phone numbers and access codes necessary to make the phone calls.

18.1.3 excel.py

Excel Files Interfaces.

Created on Mon Mar 26 12:07:54 2018

author Ruyman Azzollini

`vison.support.excel.`**`test0`**()

Just a dummy test to show we can use openpyxl

18.1.4 files.py

IO related functions.

requires PyFITS

requires NumPy

author Sami-Matias Niemi

`vison.support.files.`**`cPickleDump`**(*data*, *output*, *protocol=2*)
Dumps data to a cPickled file.

Parameters

- **`data`** – a Python data container
- **`output`** – name of the output file

Returns None

`vison.support.files.`**`cPickleDumpDictionary`**(*dictionary*, *output*, *protocol=2*)
Dumps a dictionary of data to a cPickled file.

Parameters

- **`dictionary`** – a Python data container does not have to be a dictionary
- **`output`** – name of the output file

Returns None

`vison.support.files.`**`cPickleRead`**(*ffile*)
Loads data from a pickled file.

`vison.support.files.`**`convert_fig_to_eps`**(*figname*)
Converts a figure to .eps. Returns new file name.

18.1.5 flags.py

Functions and variables related to flags for vison.

Created on Wed Sep 20 17:05:00 2017

author Ruyman Azzollini

class `vison.support.flags.`**`Flags`**(*indict=None*)

18.1.6 latex.py

Just a collection of LaTeX-generating functions for use in report.py

History

Created on Mon Jan 30 2017

author Ruyman Azzollini

`vison.support.latex.`**`generate_header`**(*test*, *model*, *author*, *reference='7-XXX'*, *issue=0.0*, *doDraft=False*)

`vison.support.latex.`**`replace_in_template`**(*texf*, *values*)

18.1.7 logger.py

These functions can be used for logging information.

Warning: logger is not multiprocessing safe.

author Sami-Matias Niemi

version 0.3

class vison.support.logger.**SimpleLogger**(*filename*, *verbose=False*)
A simple class to create a log file or print the information on screen.

write(*text*)
Writes text either to file or screen.

vison.support.logger.**f_text_wrapper**(*msg*)

vison.support.logger.**setUpLogger**(*log_filename*, *loggername='logger'*)
Sets up a logger.

Param log_filename: name of the file to save the log.

Param loggername: name of the logger

Returns logger instance

18.1.8 report.py

LaTEx - PDF Reporting Utilities.

History

Created on Wed Jan 25 16:58:33 2017

author Ruyman Azzollini

class vison.support.report.**Chapter**(*Title=''*)

generate_Latex()

class vison.support.report.**Container**

add_to_Contents(*item*)

class vison.support.report.**Content**(*contenttype=''*)

class vison.support.report.**FigsTable**(*FigsList*, *Ncols*, *figswidth*, *caption=None*)
Class to generate table of figures

generate_Latex()
Generates LaTeX as list of strings

class vison.support.report.**Figure**(*figpath*, *textfraction=0.7*, *caption=None*, *label=None*)

generate_Latex()
Generates LaTeX as list of strings.

class vison.support.report.**Part**(*Title=''*)

generate_Latex()

class vison.support.report.**Section**(*keyword*, *Title=''*, *level=0*)

generate_Latex()

class vison.support.report.**Table**(*tableDict*, *formats=None*, *names=None*, *caption=None*, *col_align=None*, *longtable=False*)

PENDING:

- adjust width of table to texwidth:

esizebox{ extwidth}{!}{

... end{tabular}}

- include option to rotate table to show in landscape

generate_Latex()

Generates LaTeX as list of strings.

class vison.support.report.**Text**(*text*)

generate_Latex()

18.1.9 utils.py

General Purpose Utilities

Created on Tue Apr 10 15:18:07 2018

author Ruyman Azzollini

18.1.10 vistime.py

Accesory library: time related operations

Created on Tue Oct 10 15:08:28 2017

author Ruyman Azzollini

vison.support.vistime.**get_dtobj**(*DT*)

vison.support.vistime.**get_time_tag**()

18.1.11 vjson.py

json files handling utilities.

Created on Tue Mar 27 14:25:43 2018

author Ruyman Azzollini

vison.support.vjson.**dumps_to_json**(*pydict*)

vison.support.vjson.**load_jsonfile**(*jsonfile*, *useyaml=False*)

vison.support.vjson.**save_jsonfile**(*pydict*, *jsonfile*)

CHAPTER

NINETEEN

UNIT TESTING

19.1 Unit Testing

19.1.1 test_ccdpile.py

Unit-testing for CCDPile class.

Created on Mon May 7 09:47:07 2018

author Ruyman Azzollini

19.1.2 test_ccd.py

Unit-testing for CCD class.

Created on Mon May 7 09:47:07 2018

author Ruyman Azzollini

CHAPTER

TWENTY

TEST SCRIPTS

These are the scripts that hold the description, execution, data validation and analysis of the tests that make the campaign. They are served by the infrasctructure and tools provided by the pipeline.

20.1 Charge Injection Scripts

20.1.1 Charge Injection Scripts

CHINJ01

VIS Ground Calibration TEST: CHINJ01

Charge injection calibration (part 1) Injection vs. IG1-IG2

Created on Tue Aug 29 17:36:00 2017

author Ruyman Azzollini

class `vison.inject.CHINJ01.`**`CHINJ01`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds CHINJ01 script structure dictionary.

#:param IDL: float, [V], value of IDL (Inject. Drain Low). #:param IDH: float, [V], Injection Drain High. #:param IG2: float, [V], Injection Gate 2. #:param IG1s: list of 2 floats, [V], [min,max] values of IG1. #:param id_delays: list of 2 floats, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`meta_analysis`()
Plot and model charge injection vs. IG1 Find injection threshold: Min IG1 Find notch injection amount.

`set_inpdefaults`(***kwargs*)

CHINJ02

VIS Ground Calibration TEST: CHINJ02

Charge injection calibration (part 2) Injection vs. IDL (injection threshold)

Created on Tue Aug 29 17:36:00 2017

author Ruyman Azzollini

class `vison.inject.CHINJ02.`**`CHINJ02`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds CHINJ02 script structure dictionary.

#:param IDLs: list of 2 ints, [V], [min,max] values of IDL (Inject. Drain Low). #:param IDH: int, [V], Injection Drain High. #:param id_delays: list of 2 ints, [us], injection drain delays. #:param toi_chinj: int, [us], TOI-charge injection. :param diffvalues: dict, opt, differential values.

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`meta_analysis`()
Finds the Injection Threshold for each CCD half.

METACODE

```
f.e.CCD:
    f.e.Q:
        load injection vs. IDL cuve
        find&save injection threshold on curve

report injection threshold as a table
```

`set_inpdefaults`(***kwargs*)

20.2 Dark Scripts

20.2.1 "Dark Acquisitions" Scripts

BIAS01

TEST: BIAS0X

Bias-structure/RON analysis script

Created on Tue Aug 29 16:53:40 2017

author Ruyman Azzollini

class `vison.dark.BIAS0X.`**`BIAS0X`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`basic_analysis`()
BIAS0X: Basic analysis of data.

METACODE

```
f. e. ObsID:
   f.e.CCD:

       load ccdobj of ObsID, CCD

       with ccdobj, f.e.Q:
           produce a 2D poly model of bias, save coefficients
           produce average profile along rows
           produce average profile along cols
           # save 2D model and profiles in a pick file for each OBSID-CCD
           measure and save RON after subtracting large scale structure
```

```
plot RON vs. time f. each CCD and Q
plot average profiles f. each CCD and Q (color coded by time)
```

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds BIAS0X script structure dictionary.

###:param N: integer, number of frames to acquire. :param diffvalues: dict, opt, differential values. :param elvis: char, ELVIS version.

filterexposures (*structure*, *explog*, *OBSID_lims*)

meta_analysis ()
METACODE

```
f. each CCD:
   stack all ObsIDs to produce Master Bias
   f. e. Q:
       measure average profile along rows
       measure average profile along cols
plot average profiles of Master Bias(s) f. each CCD,Q
(produce table(s) with summary of results, include in report)
save Master Bias(s) (3 images) to FITS CDPs
show Master Bias(s) (3 images) in report
save name of MasterBias(s) CDPs to DataDict, report
```

prep_data ()
BIAS0X: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction cosmetics masking

class vison.dark.BIAS0X.**Test** (*methodName='runTest'*)
Unit tests for the BIAS0X class.

test_check_data ()

Returns None

DARK01

TEST: DARK01

"Dark Current" analysis script

Created on Tue Aug 29 17:21:00 2017

author Ruyman Azzollini

class vison.dark.DARK01.**DARK01** (*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds DARK01 script structure dictionary.

Parameters diffvalues – dict, opt, differential values.

filterexposures (*structure*, *explog*, *OBSID_lims*)

prep_data ()
DARK01: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [BIAS SUBTRACTION] cosmetics masking

stack_analysis()

METACODE

```
f. each CCD:
    f. e. Q:
        stack all ObsIDs to produce Master Dark
        produce mask of hot pixels / columns
        count hot pixels / columns
        measure average profile along rows
        measure average profile along cols

plot average profiles of Master Bias f. each CCD,Q
show Master Dark (images), include in report
report stats of defects, include in report
save name of MasterDark to DataDict, report
save name of Defects in Darkness Mask to DD, report
```

20.3 Flat-Illumination Scripts

20.3.1 Flat-Illumination Scripts

FLAT0X

VIS Ground Calibration TEST: FLAT0X

Flat-fields acquisition / analysis script

Created on Tue Aug 29 17:32:52 2017

author Ruyman Azzollini

class vison.flat.FLAT0X.**FLAT0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)

Builds FLAT0X script structure dictionary.

Parameters diffvalues – dict, opt, differential values.

do_indiv_flats()

METACODE

```
Preparation of data for further analysis and
produce flat-field for each OBSID.

f.e. ObsID:
    f.e.CCD:

        load ccdobj

        f.e.Q:

            model 2D fluence distro in image area
            produce average profile along rows
            produce average profile along cols

        save 2D model and profiles in a pick file for each OBSID-CCD
```

```
        divide by 2D model to produce indiv-flat
        save indiv-Flat to FITS(?), update add filename

plot average profiles f. each CCD and Q (color coded by time)
```

do_master_flat()

METACODE

```
Produces Master Flat-Field

f.e.CCD:
    f.e.Q:
        stack individual flat-fields by chosen estimator
save Master FF to FITS
measure PRNU and
report PRNU figures
```

do_prdef_mask()

METACODE

```
Produces mask of defects in Photo-Response
Could use master FF, or a stack of a subset of images (in order
to produce mask, needed by other tasks, quicker).

f.e.CCD:
    f.e.Q:
        produce mask of PR defects
        save mask of PR defects
        count dead pixels / columns

report PR-defects stats
```

filterexposures(*structure*, *explog*, *OBSID_lims*)

prepare_images()

FLAT0X: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction [bias structure subtraction, if available] cosmetics masking

set_inpdefaults(***kwargs*)

NL01

VIS Ground Calibration TEST: NL01

End-To-End Non-Linearity Curve

Tasks:

- Select exposures, get file names, get metadata (commandig, HK).
- Check exposure time pattern matches test design.
- Check quality of data (rough scaling of fluences with Exposure times).
- Subtract offset level.
- Divide by Flat-field.
- **Synoptic analysis:** fluence ratios vs. extime ratios >> non-linearity curve

- extract: Non-Linearity curve for each CCD and quadrant
- produce synoptic figures
- Save results.

Created on Mon Apr 3 17:38:00 2017

author raf

class `vison.flat.NL01.`**`NL01`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)

Builds NL01 script structure dictionary.

#:param expts: list of ints [ms], exposure times. #:param exptinter: int, ms, exposure time of interleaved source-stability exposures. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 0 (Neutral Density Filter) :param diffvalues: dict, opt, differential values.

`do_satCTE`()

METACODE

```
select ObsIDs with fluence(exptime) >~ 0.5 FWC

f.e. ObsID:
    CCD:
        Q:
            measure CTE from amount of charge in over-scan relative to fluence

f.e. CCD:
    Q:
        get curve of CTE vs. fluence
        measure FWC from curve in ADU

report FWCs in electrons [via gain in inputs] f.e. CCD, Q (table)
```

`extract_stats`()

Performs basic analysis: extracts statistics from image regions to later build NLC.

METACODE

```
create segmentation map given grid parameters

f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            f.e. "img-segment": (done elsewhere)
                measure central value
                measure variance
```

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

Loads a list of Exposure Logs and selects exposures from test NL01.

The filtering takes into account an expected structure for the acquisition script.

The datapath becomes another column in DataDict. This helps dealing with tests that run overnight and for which the input data is in several date-folders.

`prep_data`()

Takes Raw Data and prepares it for further analysis.

METACODE

```
f.e. ObsID:
    f.e.CCD:
        f.e.Q:
            mask-out bad pixels
            mask-out detector cosmetics
            subtract offset
            opt: [sub bias frame]
```

produce_NLCs()

METACODE

```
Obtains Best-Fit Non-Linearity Curve

f.e. CCD:
    f.e. Q:

        [opt] apply correction for source variability (interspersed exposure
          with constant exptime)
        Build NL Curve (NLC) - use stats and exptimes
        fit poly. shape to NL curve

plot NL curves for each CCD, Q
report max. values of NL (table)
```

recalibrate_exptimes(*exptimes*)

Corrects exposure times given independent calibration of the shutter.

PTC0X

VIS Ground Calibration TEST: PTC0X

Photon-Transfer-Curve Analysis PTC01 - nominal temperature and wavelength PTC02 - alternative temperatures / wavelengths / RD

Tasks:

- Select exposures, get file names, get metadata (commandig, HK).
- Check exposure time pattern matches test design.
- Check quality of data (rough scaling of fluences with Exposure times).
- Subtract pairs of exposures with equal fluence
- **Synoptic analysis:** variance vs. fluence variance(binned difference-frames) vs. fluence
- extract: RON, gain, gain(fluence)
- produce synoptic figures
- Save results.

Created on Mon Apr 3 17:00:24 2017

author raf

class vison.flat.PTC0X.**PTC0X**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

build_scriptdict (*diffvalues={}*, *elvis='7.5.X'*)
Builds PTC0X script structure dictionary.

#:param exptimes: list of ints [ms], exposure times. #:param frames: list of ints, number of frames for each exposure time. #:param wavelength: int, wavelength. Default: 800 nm. :param diffvalues: dict, opt, differential values.

extract_HER ()
Hard Edge Response Analysis

extract_PTC ()

Performs basic analysis of images:

- builds PTC curves: both on non-binned and binned images

METACODE

```
create list of OBSID pairs

create segmentation map given grid parameters

f.e. OBSID pair:
    CCD:
        Q:
            subtract CCD images
            f.e. segment:
                measure central value
                measure variance
```

f_extract_PTC (*ccdobjcol*, *medcol*, *varcol*)

filterexposures (*structure*, *explog*, *OBSID_lims*)

meta_analysis ()
Analyzes the variance and fluence: gain, and gain(fluence)

METACODE

```
f.e. CCD:
    Q:
        (using stats across segments:)
        fit PTC to quadratic model
        solve for gain
        solve for alpha (pixel-correls, Guyonnet+15)
        solve for blooming limit (ADU)
            convert bloom limit to electrons, using gain

plot PTC curves with best-fit f.e. CCD, Q
report on gain estimates f. e. CCD, Q (table)
report on blooming limits (table)
```

set_inpdefaults (***kwargs*)

20.4 Point-Source Scripts

20.4.1 Point-Source Scripts

FOCUS00

TEST: FOCUS00

Focus analysis script

Tasks:

- Select exposures, get file names, get metadata (commandig, HK).
- Check quality of data (integrated fluxes are roughly constant, matching expected level).
- Subtract offset level.
- Divide by Flat-field.
- **Crop stamps of the sources on each CCD/Quadrant.**
  - save snapshot figures of sources.
- **for each source (5 x Nquadrants):**
  - measure shape using Gaussian Fit
- Find position of mirror that minimizes PSF sizes
- **Produce synoptic figures:** source size and ellipticity across combined FOV (of 3 CCDs)
- Save results.

Created on Mon Apr 03 16:21:00 2017

author Ruyman Azzollini

class `vison.point.FOCUS00.`**`FOCUS00`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`basic_analysis`()
This is just an assignation of values measured in check_data.

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds FOCUS00 script structure dictionary.

#:param wavelength: int, [nm], wavelength. #:param exptime: int, [ms], exposure time. :param diffvalues: dict, opt, differential values.

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`lock_on_stars`()

`meta_analysis`()

`prep_data`()

PSF0X

TEST: PSF0X

PSF vs. Fluence, and Wavelength PSF01 - nominal temperature PSF02 - alternative temperatures

Tasks:

- Select exposures, get file names, get metadata (commandig, HK).
- Check exposure time pattern matches test design.
- Check quality of data (rough scaling of fluences with Exposure times).

- Subtract offset level.
- Divide by Flat-field.
- **Crop stamps of the sources on each CCD/Quadrant.**
  - save snapshot figures of sources.
- **for each source:**
  - measure shape using weighted moments
  - measure shape using Gaussian Fit
  - Bayesian Forward Modelling the optomechanic+detector PSF
- Produce synoptic figures.
- Save results.

Created on Thu Dec 29 15:01:07 2016

author Ruyman Azzollini

20.5 Trap-Pumping Scripts

20.5.1 Trap-Pumping Scripts

TP11

VIS Ground Calibration TEST: TP11

Trap-Pumping calibration (vertical)

Created on Thu May 23 10:42:00 2019

author Ruyman Azzollini

class `vison.pump.TP11.`**`TP11`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

TP21

VIS Ground Calibration TEST: TP02

Trap-Pumping calibration (serial)

Created on Tue Aug 29 17:38:00 2017

author Ruyman Azzollini

class `vison.pump.TP21.`**`TP21`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

PumpTask

Common Use Task for Trap-Pumping Analysis.

Created on Tue Jan 2 17:44:04 2018

author Ruyman Azzollini

tptools

Trap-pumping Analysis Tools.

Created on Fri Mar 16 14:38:51 2018

author Ruyman Azzollini

`vison.pump.tptools.`**`batch_fit_PcTau_stp`**(*Amplitudes*, *dwells*, *Nshuffles=5000*)

`vison.pump.tptools.`**`batch_fit_PcTau_vtp`**(*Amplitudes*, *tois*, *Nshuffles=5000*)

`vison.pump.tptools.`**`charact_injection`**(*ccdobj*)

`vison.pump.tptools.`**`fcomp_distamp_dipoles_1D`**(*merged*, *mcat*)

`vison.pump.tptools.`**`fcomp_distamp_dipoles_2D`**(*merged*, *mcat*)

`vison.pump.tptools.`**`find_dipoles_stpump`**(*ccdobj*, *threshold*, *Q*, *vstart=0*, *vend=2066*, *extension=-1*)

Using Jesper Skottfelt's algorithm, as described in Trap_Pumping_Analysis_GCALCAMP_17OCT17_Azzollini.pdf

'West' dipole: brighter pixel closer to serial register than dimmer pixel. 'East' dipole: brighter pixel farther to serial register than dimmer pixel.

`vison.pump.tptools.`**`find_dipoles_vtpump`**(*ccdobj*, *threshold*, *Q*, *vstart=0*, *vend=2066*, *extension=-1*)

Using Jesper Skottfelt's algorithm, as described in Trap_Pumping_Analysis_GCALCAMP_17OCT17_Azzollini.pdf

'South' dipole: brighter pixel closer to serial register than dimmer pixel. 'North' dipole: brighter pixel farther to serial register than dimmer pixel.

`vison.pump.tptools.`**`fit_PcTau_stp`**(*A*, *dwells*, *stoi*, *Nshuffles=5000*)

`vison.pump.tptools.`**`fit_PcTau_vtp`**(*A*, *tois*, *Nshuffles=5000*)

`vison.pump.tptools.`**`gen_raw_dpmap_stpump`**(*ccdobj*, *injprofiles*, *vstart=0*, *vend=2066*, *extension=-1*)

`vison.pump.tptools.`**`gen_raw_dpmap_vtpump`**(*ccdobj*, *Navgrows=-1*, *vstart=0*, *vend=2066*, *extension=-1*)

`vison.pump.tptools.`**`get_InjProfile`**(*ccdobj*, *Q*, *Navgrows=-1*, *vstart=0*, *vend=2066*, *extension=-1*)

`vison.pump.tptools.`**`get_injprofile_tpnorm`**(*ccdobj*, *vstart*, *vend*)

Produces a 2D Map of charge injection to be used in trap-pumping analysis, to obtain dipole maps.

`vison.pump.tptools.`**`merge_2dcats_generic`**(*catsdict*, *catkeys*, *parentkey*, *columns*, *opcolumns*, *fcomp*, *dropna=False*)

`vison.pump.tptools.`**`merge_vtp_dipole_cats_bypos`**(*catsdict*, *catkeys*, *parentkey*, *dropna=False*)

`vison.pump.tptools.`**`save_dipcat2D_as_ds9regs`**(*df*, *regfilename*, *clobber=True*)

`vison.pump.tptools.`**`wrap_gen_InjProfiles`**(*ccdobj*, *Navgrows=-1*, *vstart=0*, *vend=2066*, *extension=-1*)

20.6 Other Test Scripts

20.6.1 Other Scripts

COSMETICS00

TEST: COSMETICS00

Cosmetics masks for the detectors: defects in darkness and PR

Created on Wed Dec 05 17:59:00 2018

author Ruyman Azzollini

class `vison.other.COSMETICS00.`**`COSMETICS00`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds COSMETICS00 script structure dictionary.

Parameters

- **`diffvalues`** – dict, opt, differential values.
- **`elvis`** – char, ELVIS version.

`check_data`()

`check_metrics_ST`(***kwargs*)

`do_masks`()

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`get_checkstats_ST`(***kwargs*)

`meta`()

MOT_FF

VIS Ground Calibration TEST: MOT_FF

Brighter-Fatter Analysis Using data from test PTC01 (via BF01)

Hard Edge Response in serial / parallel Bit Correlations (ADC health)

Created on Tue Jul 31 18:04:00 2018

author raf

class `vison.other.MOT_FF.`**`MOT_FF`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`extract_HER`()

MOT_WARM

TEST: MOT_WARM

Readiness verification: Warm Test before Cooling Down.

Created on Mon Oct 22 17:11:00 2018

author Ruyman Azzollini

class `vison.other.MOT_WARM.`**`MOT_WARM`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()

EXPOSURES: BIAS, RAMP, CHINJ, FLAT, POINT_w x waves_PNT

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds MOT_WARM script structure dictionary.

Parameters

- **diffvalues** – dict, opt, differential values.
- **elvis** – char, ELVIS version.

check_data()

check_metrics_ST(***kwargs*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

lock_on_stars()

PERSIST01

VIS Ground Calibration TEST: PERSIST01

CCD Persistence test

Created on Tue Aug 29 17:39:00 2017

author Ruyman Azzollini

class vison.other.PERSIST01.**PERSIST01**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

basic_analysis()
Basic analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        measure stats in pix satur MASK across OBSIDs
         (pre-satur, satur, post-satur)
```

build_scriptdict(*diffvalues={}*, *elvis='7.5.X'*)
Builds PERSISTENCE01 script structure dictionary.

Parameters

- **exptSATUR** – int, saturation exposure time.
- **exptLATEN** – int, latency exposure time.
- **diffvalues** – dict, opt, differential values.

check_data()
PERSIST01: Checks quality of ingested data.

METACODE

```
check common HK values are within safe / nominal margins
check voltages in HK match commanded voltages, within margins

f.e.ObsID:
    f.e.CCD:
        f.e.Q.:
            measure offsets in pre-, over-
            measure std in pre-, over-
            measure fluence in apertures around Point Sources

assess std in pre- (~RON) is within allocated margins
assess offsets in pre-, and over- are equal, within allocated  margins
assess fluence is ~expected within apertures (PS) for each frame (pre-satur,
↪satur, post-satur)


plot point source fluence vs. OBSID, all sources
[plot std vs. time]

issue any warnings to log
issue update to report
```

check_metrics_ST(***kwargs*)

filterexposures(*structure*, *explog*, *OBSID_lims*)

get_checkstats_ST(***kwargs*)

get_satur_masks()

Basic analysis of data.

METACODE

```
f.e.CCD:
    use SATURATED frame to generate pixel saturation MASKs
```

meta_analysis()

Meta-analysis of data.

METACODE

```
f.e.CCD:
    f.e.Q:
        estimate delta-charge_0 and decay tau from time-series

report:
    persistence level (delta-charge_0) and time constant
```

prep_data()

PERSIST01: Preparation of data for further analysis. Calls task.prepare_images().

Applies: offset subtraction cosmetics masking

set_inpdefaults(***kwargs*)

STRAY00

VIS Ground Calibration TEST: STRAY00 - used to investigate STRAY-LIGHT sources in OGSE.

NOT intended for performance evaluation. COMMISSIONING.

Created on Thu Feb 08 14:07:00 2018

author Ruyman Azzollini

class `vison.other.STRAY00.`**`STRAY00`**(*inputs*, *log=None*, *drill=False*, *debug=False*, *cleanafter=False*)

`build_scriptdict`(*diffvalues={}*, *elvis='7.5.X'*)
Builds STRAY00 script structure dictionary. :param diffvalues: dict, opt, differential values.

`filterexposures`(*structure*, *explog*, *OBSID_lims*)

`set_inpdefaults`(***kwargs*)

CHAPTER

TWENTYONE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

V

INDEX

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W