hermes_nemisis

Release 0.2.1.dev6+gd81fafe

The HERMES Team

Apr 10, 2024

CONTENTS

This is the documentation for the hermes_nemisis Python package for processing and analyzing data from the Noise Eliminating Magnetometer Instrument in a Small Integrated System (NEMISIS) on the Lunar Gateway.

CHAPTER

ONE

ACKNOWLEDGING THIS PACKAGE

If you use this package in your scientific work, we would appreciate citing it in your publications. Proper citations and acknowledgement is key to a healthy scientific community and enables scientific reproducibility.

1.1 Citing in Publications

Please add the following line within your methods, conclusion or acknowledgements sections:

> *This research used version X.Y.Z (software citation) of the Hermes Instrument open source software package (paper citation).*

The software citation should be the specific Zenodo DOI for the version used in your work. A paper citation does not yet exist.

CHAPTER

TWO

RELEASE HISTORY

2.1 Full Changelog

CHAPTER

THREE

CALIBRATION AND MEASUREMENT ALGORITHM DOCUMENT (CMAD)

3.1 Scope

This section provides a brief description of the specific aspects of instrument calibration covered by this plan

3.2 Related Documentations

3.2.1 Applicable Documents

This section identifies (in tabular format) any other project/mission documentation with higher- level guiding requirements or that provide more detail or context. See example below:

Title	Document Number	Publication Date
Heliophysics Division Science Data Management Policy	HPD-SDMP	14 Feb 2022
HERMES Project Level Requirement Appendix (PLRA)	HERMES-SYS-REQ-0027	13 Oct 2021
HERMES Project Data Management Plan (PDMP)	HERMES-MGMT-PLAN-0015	7 Oct 2021

3.3 Overview and Background Information

This section briefly summarizes the instrument and its objectives to provide its role and importance within the context of the SMD portfolio.

3.3.1 Science Objectives

This section describes the science objective(s) of the mission.

3.3.2 Noise Eliminating Magnetometer Instrument in a Small Integrated System (NEMISIS) Instrument Description

3.3.3 Measurement Concept

This subsection summarizes the instrument parameters and associated requirements that must be fulfilled to attain mission success. The types of measurements or observations made as well as how the instrument executes those measurements are described. A table like the example below may be included.

Parameter	Minimum Success Criteria	Comprehensive Success Criteria	Design Goals
Wavelength λ	6 or more emissions to specify the chromosphere, TR, and corona, plus the He II 30.4 nm emission	0.1-105 nm	0.1-105 nm
Time Cadence	60 sec	< 20 sec	10 sec

3.3.4 Instrument Subsystem Descriptions

<Subsystem name>

This section (and any necessary subsections) provides details on the susbsystems or components responsible for directly obtaining the measurements or observations pertinent to the instrument. Details on the layout and design of the subsystem, examples of expected measurements, and interactions with any other subsystems should be provided.

<Instrument name> Heritage

Subsystem Heritage

This section summarizes any heritage from past missions for the instrument and its subsystems or components (e.g., detectors, cameras, signal processing electronics).

Algorithm and Calibration Heritage

This section identifies any heritage from past missions for the algorithms used to process/convert detector signals into the measurable quantities needed to meet the science requirements.

3.4 Noise Eliminating Magnetometer Instrument in a Small Integrated System (NEMISIS) Calibration Plan

3.4.1 Overall Calibration Scheme

This section summarizes the calibration philosophy and identifies any heritage tied to the calibration schemes of related missions and instruments.

3.4.2 Pre-flight Calibration Plans

This section details how unit level (e.g., individual detectors) and system level (e.g., instrument subsystem) are tested and calibrated to verify that they will meet the expected performance parameters prior to placement (i.e., launch) into the relevant operational environment.

<Subsystem name> Pre-flight Calibrations

This subsection describes the specific testing and examination methods used to characterize the build and performance of each subsystem or component (e.g., diffraction gratings, CCDs)

3.4.3 Instrument Description

This subsection describes the primary scientific objectives of the instrument, its hardware, physical configuration, etc. This subsection lists the major elements of the instrument and provides a schematic of the conceptual design. Known issues due to external factors that could impact any long-term comparison or analysis (e.g., optical distortion due to gradual radiation degradation) should be captured.

3.4.4 In-flight Tracking of Short-Term Changes

This section identifies any potential factors in the operational environment (e.g., radiation, temperature fluctuations, exposure-related degradation) that could eventually result in off- nominal changes in the instrument's measurements. The methods used to identify and track these changes are also described.

3.4.5 Long-term Absolute Calibration Tracking (Re-Calibration)

This section identifies any periodic re-calibration to absolute standards to be used over the course of the mission.

3.4.6 Validation

This describes the use of any other measurements (via complementary instruments) or models to validate the instrument's measurements

3.5 Noise Eliminating Magnetometer Instrument in a Small Integrated System (NEMISIS) Measurement Algorithm Description

3.5.1 Theoretical basis

This section provides context and background information for the quantity or phenomenon being detected. The algorithms and techniques used are described, with pertinent equations and references included. Logical groupings (i.e., separate subsections) may be used for clarity of the concepts introduced.

3.5.2 Conversion of Instrument Signals to <Measurable units>

Measurement Equations

This subsection (one for each subsystem) describes the equations used to derive measurable quantities from raw instrument signals.

3.5.3 Signal Estimates and Error Analyses for Subsystems

<Subsystem name> Signal Estimates and Error Analysis

This subsection (one for each subsystem) provides details on the expected signal values for the instrument subsystem. This can be expressed graphically. A table summarizing the "acceptable" values—that is, the minimum values that would meet mission requirements and be deemed still usable to meet the mission's science objectives—as well as the estimated uncertainty in the measured values and the error budget allowable for each parameter may be included. The equations for determining the uncertainties should be included.

3.5.4 Preflight Calibration Algorithms

This section describes the process for calibrating the instrument prior to shipment and/or installation. It may refer back to measurement equations detailed earlier in the document and identify the specific variables being solved for in order to determine proper calibration.

3.5.5 Appendix A. List of Variable Definitions

CHAPTER

FOUR

USER'S GUIDE

Welcome to our User guide. For more details checkout the *API Reference*.

4.1 A Brief Tour

Insert a tour here.

4.2 Data

4.2.1 Overview

Data Description

Level	Product	Description
1	Vector Magnetic Field from 3 sensors (M0, M1, M2) in spacecraft coordinates	CCSDS, each packet is 4-sec long at 10 Hz rate, 3-axis magnetic field components for all three sensors in coordinate system native to the HERMES Payload
1	Sensor Temperatures	Temperatures at M0, M1, M2
2	Vector Magnetic Field from 3 sensors (M0, M1, M2) in GSE coordinates	3d vector magnetic fields (Bx, By, Bz) in nT for each magnetometer (M0, M1, M2) using final calibrations for offsets and gains
2	Sensor Temperatures	Final calibrated temperatures
3	Magnetic field at Gateway	Background-subtracted and processed 3d vector magnetic field (Bx, By, Bz) in nT in a common coordinate system (e.g. GSE). Derived by combining individual sensor data.
QL	**Vector Magnetic Field from 3 sensors (M0, M1, M2)** in GSE coordinates (Unvalidated)	Despiked values for 3d vector magnetic fields (Bx, By, Bz) in nT for each magnetometer (M0, M1, M2) in their local coordinate system, plus temperatures in Celsius for each sensor. (and time-corrected and time-checked)

4.2.2 Getting Data

4.2.3 Reading Data

4.2.4 Calibrating Data

Data products below level 2 generally require calibration to be transformed into scientificically useable units. This section describes how to calibrate data files from lower to higher levels.

4.3 Customization and Global Configuration

4.3.1 The `configrc` file

This package uses a `configrc` configuration file to customize certain properties. You can control a number of key features such as where your data will download to. This configuration file in a platform specific directory, which you can see the path for by running:

```
>>> import hermes_nemisis
>>> hermes_nemisis.print_config()
```

To maintain your own customizations place a copy of the default file into the *first* path printed above. Do not edit the default file directly as every time you install or update, this file will be overwritten.

See below for the example config file.

4.3.2 Dynamic settings

You can also dynamically change the default settings in a Python script or interactively from the python shell. All of the settings are stored in a Python ConfigParser instance called `sunpy.config`, which is global to the package. Settings can be modified directly, for example:

```
import hermes_nemisis
hermes_nemisis.config.set('downloads', 'download_dir', '/home/user/Downloads')
```

A sample configrc file

```
;
; Configuration
;
; This is the default configuration file

;;;;;;;;;;;;;;;;;;;
; General Options ;
;;;;;;;;;;;;;;;;;;;
[general]

; Time Format to be used for displaying time in output (e.g. graphs)
; The default time format is based on ISO8601 (replacing the T with space)
; note that the extra '%'s are escape characters
```

(continues on next page)

(continued from previous page)

```
time_format = %Y-%m-%d %H:%M:%S

;;;;;;;;;;;;;
; Downloads ;
;;;;;;;;;;;;;
[downloads]

; Location to save download data to. Path should be specified relative to the
; SunPy working directory.
; Default value: data/
download_dir = data
```

4.4 Logging system

4.4.1 Overview

The logging system is an adapted version of `AstropyLogger`. Its purpose is to provide users the ability to decide which log and warning messages to show, to capture them, and to send them to a file.

All messages use this logging facility which is based on the Python `logging` module rather than print statements.

For more information on this system see the documentation in hermes-core.

which will save the messages to a local file called `myfile.log`.

CHAPTER

FIVE

DEVELOPER'S GUIDE

This article describes the guidelines to be followed by developers working on this repository. If you are planning on contributing to this repository please read the following carefully. This guide borrows heavily from the one developed by the SunPy Project. It is consistent with the standards recommended by the Python in Heliophysics (PyHC).

The guidelines are

5.1 Developer Environment

This Python package is used in the pipeline processing of scientific data from HERMES. Special consideration is therefore required to ensure that development is compatible with the pipeline environment. It is also important to ensure that this package is compatible with a user's systems such as a mac and windows.

See the parent package for the documentation.

5.2 Coding Standards

The purpose of the page is to describe the standards that are expected of all the code in this repository. All developers should read and abide by the following standards. Code which does not follow these standards closely will generally not be accepted.

The projects coding standards are documented in HERMES-core.

The following standards are specific to this repository.

5.2.1 HERMES Instrument Standards

insert instrument specific standards here.

5.3 Testing Guidelines

This section describes the testing framework and format standards for tests. Here we have heavily adapted the Astropy version, and **it is worth reading that link.**

The testing framework used by sunpy is the pytest framework, accessed through the `pytest` command.

Note: The `pytest` project was formerly called `py.test`, and you may see the two spellings used interchangeably.

5.3.1 Writing tests

`pytest` has the following test discovery rules:

```
* ``test_*.py`` or ``*_test.py`` files
* ``Test`` prefixed classes (without an ``__init__`` method)
* ``test_`` prefixed functions and methods
```

We use the first one for our test files, `test_*.py` and we suggest that developers follow this.

A rule of thumb for unit testing is to have at least one unit test per public function.

Where to put tests

Each package should include a suite of unit tests, covering as many of the public methods/functions as possible. These tests should be included inside each package, e.g:

```
sunpy/map/tests/
```

"tests" directories should contain an `__init__.py` file so that the tests can be imported.

doctests

Code examples in the documentation will also be run as tests and this helps to validate that the documentation is accurate and up to date. We use the same system as Astropy, so for information on writing doctests see the astropy documentation.

You do not have to do anything extra in order to run any documentation tests. Within our `setup.cfg` file we have set default options for `pytest`, such that you only need to run:

```
$ pytest <rst to test>
```

to run any documentation test.

Bugs Testing

In addition to writing unit tests new functionality, it is also a good practice to write a unit test each time a bug is found, and submit the unit test along with the fix for the problem. This way we can ensure that the bug does not re-emerge at a later time.

5.4 Documentation Rules

5.4.1 Overview

All code must be documented and we follow the style conventions described here:

- numpydoc

Referring to other code

To link to methods, classes, or modules in your repo you have to use backticks, for example:

```
`hermes_nemisis.io.read_file`
```

generates a link like this: `hermes_nemisis.io.read_file`.

Links can also be generated to external packages via intersphinx:

```
`numpy.mean`
```

will return this link: `numpy.mean`. This works for Python, Numpy and Astropy (full list is in `docs/conf.py`).

With Sphinx, if you use `:func:` or `:meth:`, it will add closing brackets to the link. If you get the wrong pre-qualifier, it will break the link, so we suggest that you double check if what you are linking is a method or a function.

```
:class:`numpy.mean()`
:meth:`numpy.mean()`
:func:`numpy.mean()`
```

will return two broken links ("class" and "meth") but "func" will work.

Project-specific Rules

- For **all** RST files, we enforce a one sentence per line rule and ignore the line length.

5.4.2 Sphinx

All of the documentation (like this page) is built by Sphinx, which is a tool especially well-suited for documenting Python projects. Sphinx works by parsing files written using a a Mediawiki-like syntax called reStructuredText. It can also parse markdown files. In addition to parsing static files of reStructuredText, Sphinx can also be told to parse code comments. In fact, in addition to what you are reading right now, the Python documentation was also created using Sphinx.

Usage and Building the documentation

All of the documentation is contained in the "docs" folder and code documentation strings. Sphinx builds documentation iteratively, only adding things that have changed. For more information on how to use Sphinx, consult the Sphinx documentation.

HTML

To build the html documentation locally use the following command, in the docs directory run:

```
$ make html
```

This will generate HTML documentation in the "docs/_build/html" directory. You can open the "index.html" file to browse the final product.

If you'd like to rebuild the documentation from scratch. This is normally not necessary since Sphinx will detect and only build the required changes. But if you are running into strange errors you may want to try this. The following command will wipe all generated files.

$ make clean

Sphinx can also build documentation as a PDF but this requires latex to be installed.

5.5 Workflow for Maintainers

This page is for maintainers who can merge our own or other peoples' changes into the upstream repository.

Seeing as how you're a maintainer, you should be completely on top of the basic git workflow in Developer's Guide and Astropy's git workflow.

5.5.1 Integrating changes via the web interface (recommended)

Whenever possible, merge pull requests automatically via the pull request manager on GitHub. Merging should only be done manually if there is a really good reason to do this!

Make sure that pull requests do not contain a messy history with merges, etc. If this is the case, then follow the manual instructions, and make sure the fork is rebased to tidy the history before committing.

To check out a particular pull request to test out locally:

```
$ git checkout pr/999
Branch pr/999 set up to track remote branch pr/999 from upstream.
Switched to a new branch 'pr/999'
```

When to remove or combine/squash commits

In all cases, be mindful of maintaining a welcoming environment and be helpful with advice, especially for new contributors. It is expected that a maintainer would offer to help a contributor who is a novice git user do any squashing that that maintainer asks for, or do the squash themselves by directly pushing to the PR branch.

Pull requests **must** be rebased and at least partially squashed (but not necessarily squashed to a single commit) if large (approximately >10KB) non-source code files (e.g. images, data files, etc.) are added and then removed or modified in the PR commit history (The squashing should remove all but the last addition of the file to not use extra space in the repository).

Combining/squashing commits is **encouraged** when the number of commits is excessive for the changes made. The definition of "excessive" is subjective, but in general one should attempt to have individual commits be units of change, and not include reversions. As a concrete example, for a change affecting < 50 lines of source code and including a changelog entry, more than a two commits would be excessive. For a larger pull request adding significant functionality, however, more commits may well be appropriate.

As another guideline, squashing should remove extraneous information but should not be used to remove useful information for how a PR was developed. For example, 4 commits that are testing changes and have a commit message of just "debug" should be squashed. But a series of commit messages that are "Implemented feature X", "added test for feature X", "fixed bugs revealed by tests for feature X" are useful information and should not be squashed away without reason.

When squashing, extra care should be taken to keep authorship credit to all individuals who provided substantial contribution to the given PR, e.g. only squash commits made by the same author.

When to rebase

Pull requests **must** be rebased (but not necessarily squashed to a single commit) if:

- There are commit messages include offensive language or violate the code of conduct (in this case the rebase must also edit the commit messages)

Pull requests **may** be rebased (either manually or with the rebase and merge button) if:

- There are conflicts with main
- There are merge commits from upstream/main in the PR commit history (merge commits from PRs to the user's fork are fine)

Asking contributors who are new to the project or inexperienced with using git is **discouraged**, as is maintainers rebasing these PRs before merge time, as this requires resetting of local git checkouts.

A few commits

If there are only a few commits, consider rebasing to upstream:

```
# Fetch upstream changes
$ git fetch upstream-rw

# Rebase
$ git rebase upstream-rw/main
```

A long series of commits

If there are a longer series of related commits, consider a merge instead:

```
$ git fetch upstream-rw
$ git merge --no-ff upstream-rw/main
```

Note the `--no-ff` above. This forces git to make a merge commit, rather than doing a fast-forward, so that these set of commits branch off trunk then rejoin the main history with a merge, rather than appearing to have been made directly on top of trunk.

Check the history

Now, in either case, you should check that the history is sensible and you have the right commits:

```
$ git log --oneline --graph
$ git log -p upstream-rw/main..
```

The first line above just shows the history in a compact way, with a text representation of the history graph. The second line shows the log of commits excluding those that can be reached from trunk (`upstream-rw/main`), and including those that can be reached from current HEAD (implied with the `..` at the end). So, it shows the commits unique to this branch compared to trunk. The `-p` option shows the diff for these commits in patch form.

Push to open pull request

Now you need to push the changes you have made to the code to the open pull request:

```
$ git push git@github.com:<username>/hermes_nemisis.git HEAD:<name of branch>
```

You might have to add `--force` if you rebased instead of adding new commits.

5.5.2 IOssue Milestones and Labels

Current milestone guidelines:

- Only confirmed issues or pull requests that are release critical or for some other reason should be addressed before a release, should have a milestone. When in doubt about which milestone to use for an issue, do not use a milestone and ask other the maintainers.

Current labelling guidelines:

- Issues that require fixing in main, but that also are confirmed to apply to supported stable version lines should be marked with a "Affects Release" label.
- All open issues should have a "Priority <level>", "Effort <level>" and "Package <level>", if you are unsure at what level, pick higher ones just to be safe. If an issue is more of a question or discussion, you can omit these labels.
- If an issue looks to be straightforward, you should add the "Good first issue" and "Hacktoberfest" label.
- For other labels, you should add them if they fit, like if an issue affects the net submodule, add the "net" label or if it is a feature request etc.

5.5.3 Updating and Maintaining the Changelog

The changelog will be read by users, so this description should be aimed at users instead of describing internal changes which are only relevant to the developers.

The current changelog is kept in the file "CHANGELOG.rst" at the root of the repository. You do not need to update this file as we use towncrier to update our changelog. This is built and embedded into our documentation.

Towncrier will automatically reflow your text, so it will work best if you stick to a single paragraph, but multiple sentences and links are OK and encouraged. You can install towncrier and then run `towncrier --draft` if you want to get a preview of how your change will look in the final release notes. This tool was built by the SunPy community and they provide a great guide on how to use it.

Instructions on how to write a changelog..

5.5.4 Releases

We have a step by step checklist on the Wiki on how to make a release.

5.6 Global Settings

We make use of a settings file (`<hermes_nemisis>rc`). This file contains a number of global settings such as where files should be downloaded by default or the default format for displaying times. When developing new functionality check this file and make use of the default values if appropriate or, if needed, define a new value. More information can be found in *Customization and Global Configuration*.

CHAPTER

SIX

API REFERENCE

6.1 hermes_nemisis Package

6.1.1 Functions

read_file(data_filename)	Read a file.

read_file

hermes_nemisis.**read_file**(*data_filename*)

Read a file.

Parameters
data_filename (*str*) – A file to read.

Returns
data (*str*)

Examples

6.2 hermes_nemisis.calibration.calibration Module

A module for all things calibration.

6.2.1 Functions

process_file(data_filename)	This is the entry point for the pipeline processing.
parse_l0_sci_packets(data_filename)	Parse a level 0 nemisis binary file containing CCSDS packets.
l0_sci_data_to_cdf(data, original_filename)	Write level 0 nemisis science data to a level 1 cdf file.
calibrate_file(data_filename)	Given an input data file, raise it to the next level (e.g. level 0 to level 1, level 1 to quicklook) it and return a new file.
get_calibration_file(data_filename[, time])	Given a time, return the appropriate calibration file.
read_calibration_file(calib_filename)	Given a calibration, return the calibration structure.

process_file

hermes_nemisis.calibration.calibration.**process_file**(*data_filename: Path*) → list

This is the entry point for the pipeline processing. It runs all of the various processing steps required.

Parameters
data_filename (*str*) – Fully specificied filename of an input file

Returns
output_filenames (*list*) – Fully specificied filenames for the output files.

parse_l0_sci_packets

hermes_nemisis.calibration.calibration.**parse_l0_sci_packets**(*data_filename: Path*) → dict

Parse a level 0 nemisis binary file containing CCSDS packets.

Parameters
data_filename (*str*) – Fully specificied filename

Returns
result (*dict*) – A dictionary of arrays which includes the ccsds header fields

Examples

```
>>> import hermes_nemisis.calibration as calib
>>> data_filename = "hermes_MAG_l0_2022339-000000_v0.bin"
>>> data = calib.parse_nemisis_sci_packets(data_filename)
```

l0_sci_data_to_cdf

hermes_nemisis.calibration.calibration.**l0_sci_data_to_cdf**(*data: dict*, *original_filename: Path*) → Path

Write level 0 nemisis science data to a level 1 cdf file.

Parameters

- **data** (*dict*) – A dictionary of arrays which includes the ccsds header fields
- **original_filename** (*Path*) – The Path to the originating file.

Returns
output_filename (*Path*) – Fully specificied filename of cdf file

Examples

```
>>> from pathlib import Path
>>> from hermes_core.util.util import parse_science_filename
>>> import hermes_nemisis.calibration as calib
>>> data_filename = Path("hermes_MAG_l0_2022339-000000_v0.bin")
>>> metadata = parse_science_filename(data_filename)
>>> data_packets = calib.parse_l0_sci_packets(data_filename)
>>> cdf_filename = calib.l0_sci_data_to_cdf(data_packets, data_filename)
```

calibrate_file

hermes_nemisis.calibration.calibration.**calibrate_file**(*data_filename: Path*) → Path

Given an input data file, raise it to the next level (e.g. level 0 to level 1, level 1 to quicklook) it and return a new file.

Parameters
data_filename (*Path*) – Fully specificied filename of the input data file.

Returns
output_filename (*Path*) – Fully specificied filename of the output file.

Examples

```
>>> from hermes_nemisis.calibration import calibrate_file
>>> level1_file = calibrate_file('hermes_MAG_l0_2022239-000000_v0.bin')
```

get_calibration_file

hermes_nemisis.calibration.calibration.**get_calibration_file**(*data_filename: Path*, *time=None*) → Path

Given a time, return the appropriate calibration file.

Parameters

- **data_filename** (*str*) – Fully specificied filename of the non-calibrated file (data level < 2)
- **time** (*Time*) –

Returns
calib_filename (*str*) – Fully specificied filename for the appropriate calibration file.

Examples

read_calibration_file

hermes_nemisis.calibration.calibration.**read_calibration_file**(*calib_filename: Path*)

Given a calibration, return the calibration structure.

Parameters
calib_filename (*str*) – Fully specificied filename of the non-calibrated file (data level < 2)

Returns
output_filename (*str*) – Fully specificied filename of the appropriate calibration file.

Examples

6.3 hermes_nemisis.io.file_tools Module

This module provides a generic file reader.

6.3.1 Functions

read_file(data_filename)	Read a file.

read_file

hermes_nemisis.io.file_tools.**read_file**(*data_filename*)

Read a file.

Parameters
data_filename (*str*) – A file to read.

Returns
data (*str*)

Examples

PYTHON MODULE INDEX

h

INDEX