Space Physics WebServices Client Documentation

Release 1.2.7

Alexis Jeandet

May 03, 2024

CONTENTS

CHAPTER

ONE

INSTALLATION

1.1 Stable release

To install Space Physics WebServices Client, run this command in your terminal:

```
$ python -m pip install speasy
# or
$ python -m pip install --user speasy
```

This is the preferred method to install Space Physics WebServices Client, as it will always install the most recent stable release.

If you don't have pip installed, this Python installation guide can guide you through the process.

1.2 From sources

The sources for Space Physics WebServices Client can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone git://github.com/SciQLop/speasy
```

Or download the tarball:

```
$ curl  -OL https://github.com/SciQLop/speasy/tarball/main
```

Once you have a copy of the source, you can install it with:

```
$ python -m pip install .
```

CHAPTER

TWO

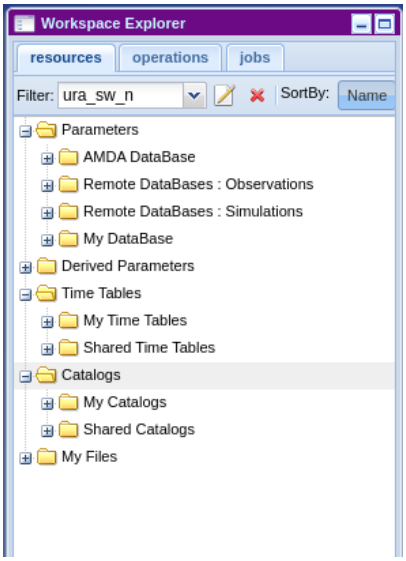
WEB SERVICES

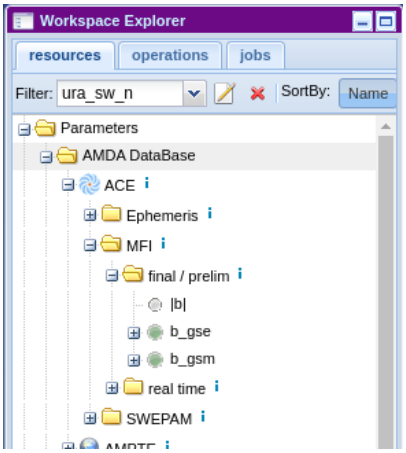
2.1 Automated Multi-Dataset Analysis (AMDA)

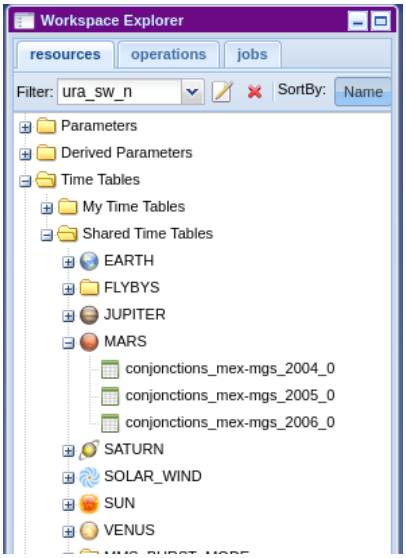
AMDA is one of the main data providers handled by speasy. All products are either available using directly the AMDA module or using `speasy.get_data()`. `speasy.get_data()` usage should be preferred over AMDA module methods since it is more flexible and it's interface is guaranteed to be more stable. The following documentation will focus on AMDA module specific usage.

2.1.1 Basics: Getting data from AMDA

AMDA distributes several public or private products such as Parameters, Datasets, Timetables and Catalogs. Speasy makes them accessible thanks to this module with `get_data()` or their dedicated methods such as `get_parameter()`, `get_user_parameter()`,... Note that you can browse the list of all available products from AMDA Workspace:







This module provides two kinds of operations, **list** or **get** and so user methods are prefixed with one of them.

- **get** methods retrieve the given product from AMDA server, they takes at least the product identifier and time range for time series
- **list** methods list available products of a given type on AMDA, they return a list of indexes that can be passed to a **get** method

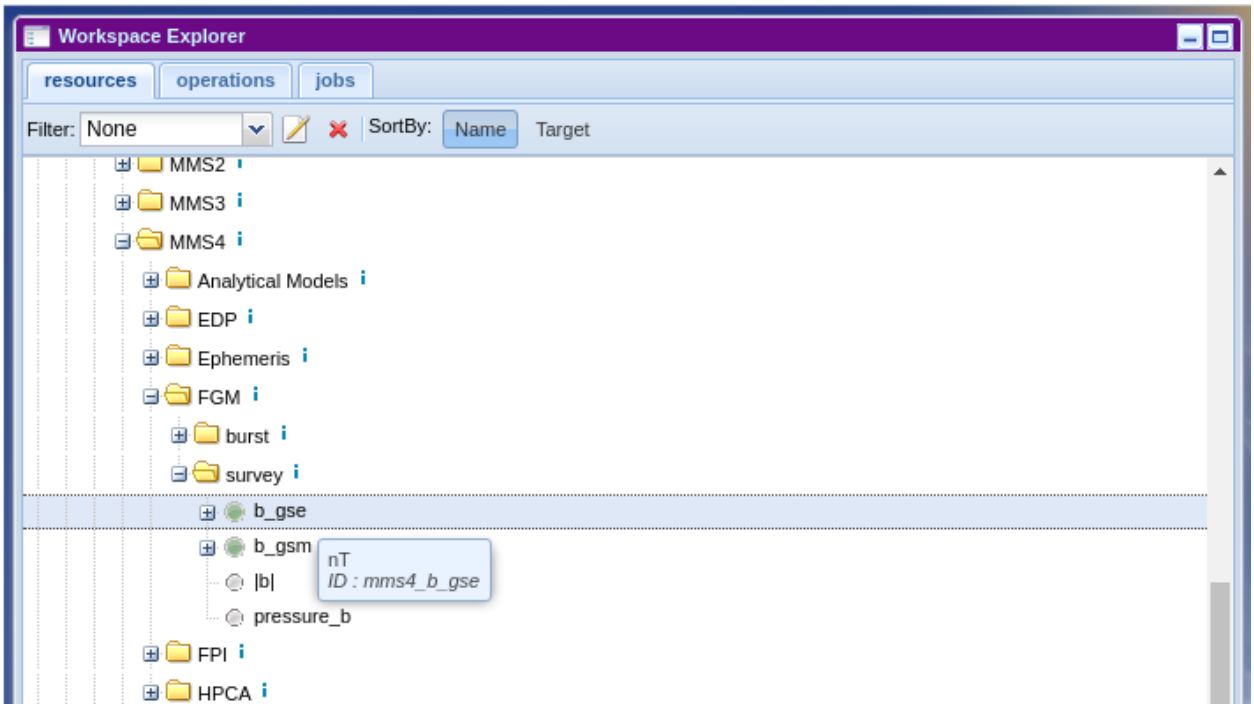
Parameters

Let's start with a simple example, we want to download the first parameter available on AMDA:

```
>>> from speasy import amda
>>> first_param_index=amda.list_parameters()[0]
>>> print(first_param_index)
<ParameterIndex: He flux>
>>> first_param=amda.get_parameter(first_param_index, "2018-01-01", "2018-01-02T01")
>>> first_param.columns
['He flux[0]', 'He flux[1]']
>>> len(first_param.time)
288
```

Usually you already know which product you want to download, two scenarios are available:

1. You are an AMDA web interface user, so you want some specific product from AMDA Workspace. You need first to get your product id, you will find the id from the tooltip while hovering any product (Dataset, Parameter, Timetable or Catalog):



Then simply:

```
>>> from speasy import amda
>>> mms4_fgm_btot=amda.get_parameter('mms4_b_tot', "2018-01-01", "2018-01-01T01")
>>> mms4_fgm_btot.columns
['|b|']
>>> len(mms4_fgm_btot.time)
57600
```

2. Second scenario, your are not much familiar with AMDA, then you can simply browse speasy dynamic inventory. In the following example, we alias AMDA data tree as amdatree, note that Python completion works and you will be able to discover AMDA products directly from your Python terminal or notebook:

```
>>> import speasy as spz
>>> from speasy import amda
>>> amdatree = spz.inventories.tree.amda
>>> mms4_fgm_btot=amda.get_parameter(amdatree.Parameters.MMS.MMS4.FGM.mms4_fgm_srvy.mms4_
↪b_tot, "2018-01-01", "2018-01-01T01")
>>> mms4_fgm_btot.columns
['|b|']
>>> len(mms4_fgm_btot.time)
57600
```

See `get_parameter()` or `get_data()` for more details.

Catalogs and TimeTables

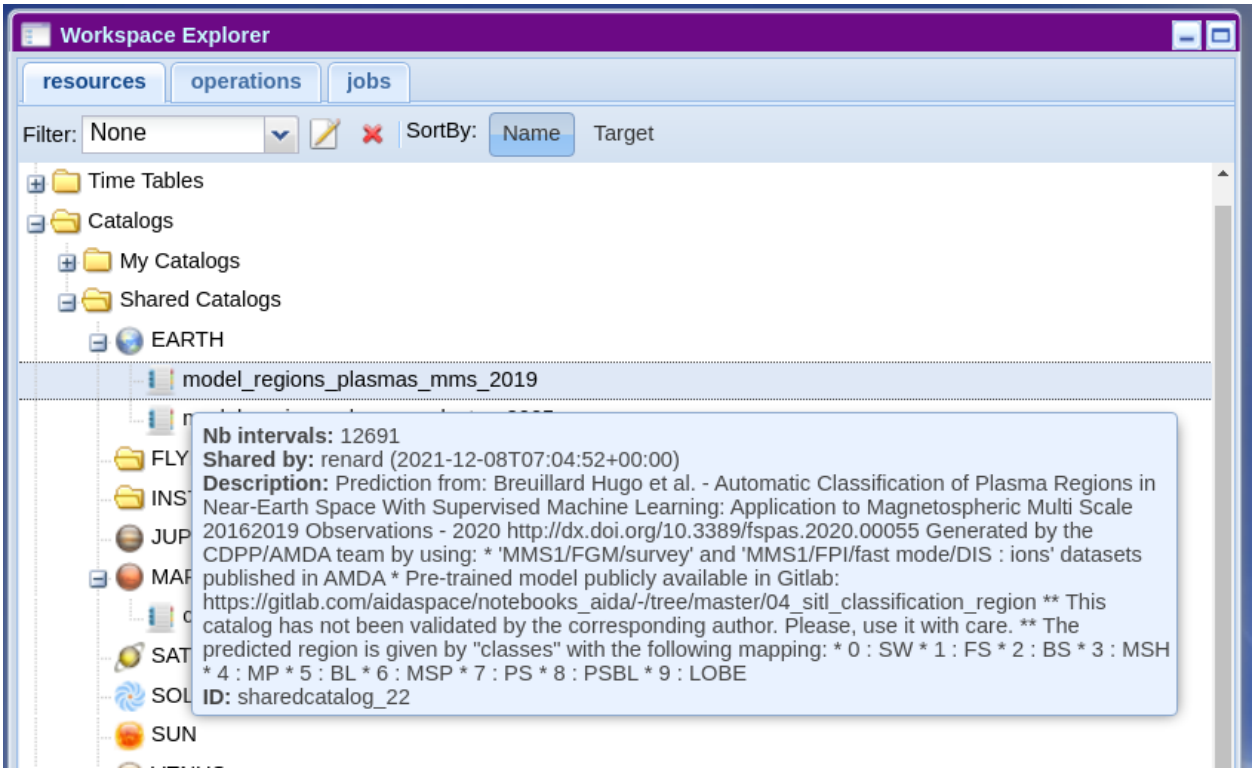
Downloading Catalogs and TimeTables from AMDA is similar to Parameters. For example let's assume you want to download the first available catalog:

```
>>> from speasy import amda
>>> first_catalog_index=amda.list_catalogs()[0]
>>> print(first_catalog_index)
<CatalogIndex: model_regions_plasmas_mms_2019>
>>> first_catalog=amda.get_catalog(first_catalog_index)
>>> first_catalog
<Catalog: model_regions_plasmas_mms_2019>
>>> len(first_catalog)
12691
>>> print(first_catalog[1])
<Event: 2019-01-01T00:24:04+00:00 -> 2019-01-01T00:24:04+00:00 | {'classes': '1'}>
```

Exactly the same with a TimeTable:

```
>>> from speasy import amda
>>> first_timetable_index=amda.list_timetables()[0]
>>> print(first_timetable_index)
<TimetableIndex: FTE_c1>
>>> first_timetable=amda.get_timetable(first_timetable_index)
>>> first_timetable
<TimeTable: FTE_c1>
>>> len(first_timetable)
782
>>> print(first_timetable[1])
<DateTimeRange: 2001-02-02T17:29:29+00:00 -> 2001-02-02T17:29:30+00:00>
```

As with Parameters you can also use the ID found on AMDA web user interface:



Then simply:

```
>>> from speasy import amda
>>> catalog_mms_2019=amda.get_catalog("sharedcatalog_22")
>>> catalog_mms_2019
<Catalog: model_regions_plasmas_mms_2019>
>>> len(catalog_mms_2019)
12691
>>> print(catalog_mms_2019[1])
<Event: 2019-01-01T00:24:04+00:00 -> 2019-01-01T00:24:04+00:00 | {'classes': '1'}>
```

And also alternatively you can use the dynamic inventory:

```
>>> from speasy import amda
>>> import speasy as spz
>>> amdatree = spz.inventories.tree.amda
>>> catalog_mms_2019=amda.get_catalog(amdatree.Catalogs.SharedCatalogs.EARTH.model_
↪regions_plasmas_mms_2019)
>>> catalog_mms_2019
<Catalog: model_regions_plasmas_mms_2019>
>>> len(catalog_mms_2019)
12691
>>> print(catalog_mms_2019[1])
<Event: 2019-01-01T00:24:04+00:00 -> 2019-01-01T00:24:04+00:00 | {'classes': '1'}>
```

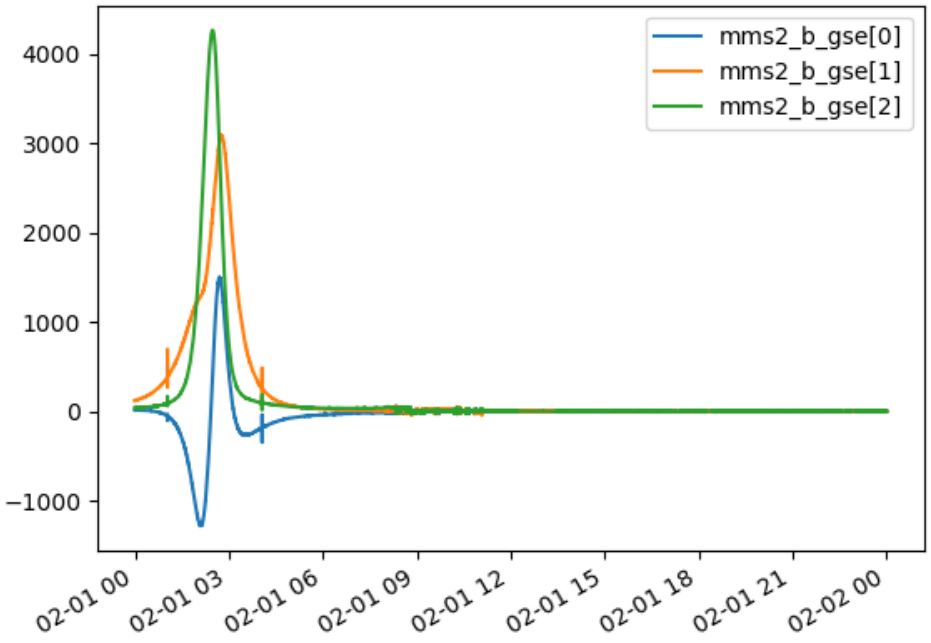
2.1.2 Some examples using AMDA products

My first plot from AMDA

In this example we will use AMDA module to retrieve and plot MMS2 FGM data, feel free to change the code and experiment!

```
>>> import matplotlib.pyplot as plt
>>> import speasy as spz
>>> amdatree = spz.inventories.tree.amda
>>> mms2_b_gse = spz.amda.get_parameter(amdatree.Parameters.MMS.MMS2.FGM.mms2_fgm_srvy.
↪mms2_b_gse, "2019-01-01", "2019-01-02")
>>> # Check that mms2_b_gse isn't empty
>>> len(mms2_b_gse)
816750
>>> # Then you can use the SpeasyVariable plot method for quick plots
>>> mms2_b_gse.plot()
>>> plt.show()
```

Then you should get something like this:



Note:

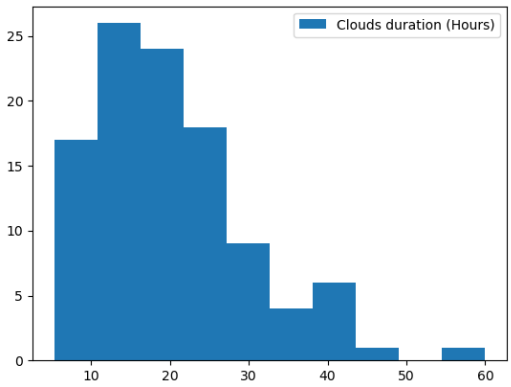
- Depending on your matplotlib backend and if you are using Jupyter Notebooks or a simple python terminal you may need to adapt this example.
- Speasy is not a plotting package, to produce publication ready figures, use something like matplotlib or seaborn directly.

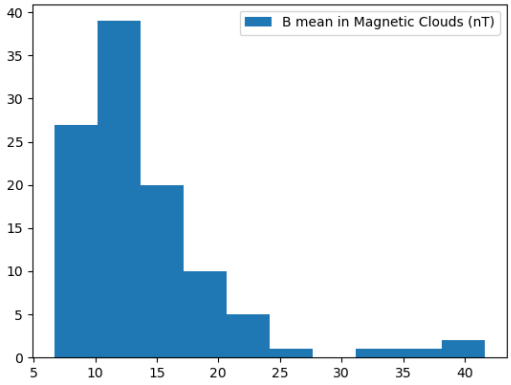
Using timetables to download data

In this example we will use AMDA to first retrieve a public timetable containing time intervals where Magnetic Clouds were detected with Wind spacecraft. Then download the magnetic field magnitude measured with MFI instrument for each interval where a Magnetic cloud was found. Once we have magnetic field measurements inside each cloud, we will as an example plot the average distribution.

```
>>> import matplotlib.pyplot as plt
>>> import numpy as np
>>> import speasy as spz
>>> amda_timetables = spz.inventories.tree.amda.TimeTables
>>> amda_parameters = spz.inventories.tree.amda.Parameters
>>> Magnetic_Clouds = spz.amda.get_timetable(amda_timetables.SharedTimeTables.SOLAR_WIND.
↪Magnetic_Clouds)
>>> print(Magnetic_Clouds.meta['description'])
Magnetic Clouds from WIND/MFI 1995-2007 -- Estimated start and end times from a magnetic␣
↪field model [Lepping et al., 1990] which assumes that the field within the magnetic␣
↪cloud is force free, i.e., so that the electrical current and the magnetic field are␣
↪parallel and proportional in strength everywhere within its volume -- see http://
↪lepmfi.gsfc.nasa.gov/mfi/mag_cloud_pub1.html ;
        Historic: From old AMDA;
        Creation Date :  2013-11-22T13:52:50;
>>> # Check that the timetable has at least some events (as expected)
>>> len(Magnetic_Clouds)
106
>>> # Then we can plot their duration distribution
>>> def duration(event):
...     return (event.stop_time.timestamp() - event.start_time.timestamp())/3600
...
>>> clouds_duration = [duration(cloud) for cloud in Magnetic_Clouds]
>>> plt.hist(clouds_duration, label="Clouds duration (Hours)")
>>> plt.legend()
>>> plt.show()
>>> # Now let's get MFI data for each cloud
>>> b_mfi_coulds = spz.get_data(amda_parameters.Wind.MFI.wnd_mfi_kp.wnd_bmag, Magnetic_
↪Clouds)
>>> # compute mean of B for each cloud and ignore NaNs
>>> b_mean_mfi_clouds = [ np.nanmean(cloud.values) for cloud in b_mfi_coulds ]
>>> plt.hist(b_mean_mfi_clouds, label="B mean in Magnetic Clouds (nT)")
>>> plt.legend()
>>> plt.show()
```

Then you should get something like these plots:





Note:

- Depending on your matplotlib backend and if you are using Jupyter Notebooks or a simple python terminal you may need to adapt this example.
- Speasy is not a plotting package, to produce publication ready figures, use something like matplotlib or seaborn directly.

2.1.3 Advanced: AMDA module configuration options

AMDA user login

Most AMDA features are available without login except user created product from web user interface. You can configure speasy to store your AMDA login, from your favourite python terminal:

```
>>> from speasy import config
>>> config.amda.username.set('my_username')
>>> config.amda.password.set('my_password')
>>> # check that your login/password are correctly set
>>> config.amda.username.get(), config.amda.password.get()
('my_username', 'my_password')
```

Then if you correctly typed your login you should be able to list and get user products:

```
>>> from speasy import amda
>>> # list user products
>>> amda.list_user_parameters()
[<ParameterIndex: test_param>]
>>> amda.list_user_catalogs()
[<CatalogIndex: MyCatalog>]
>>> amda.list_user_timetables()
[<TimetableIndex: test_alexis>, <TimetableIndex: test_alexis2>, <TimetableIndex: tt3>]
>>> # get my first user catalog
>>> amda.get_user_catalog(amda.list_user_catalogs()[0])
<Catalog: MyCatalog>
```

AMDA cache retention

While parameter download cache is not configurable and relies on product version to decide if local data is up to date compared to remote data. Requests like catalogs or time-tables download have a different dedicated cache based on duration, by default they will be cached for 15 minutes. As a consequence if a time-table has changed on AMDA servers it might take up to the configured duration to see it. This cache has been designed with interactive usage of speasy in mind where we want to minimize penalty of running multiple times the same command/line in a short amount of time.

To change this cache duration value:

```
>>> from speasy import config
>>> # set cache duration to 900 seconds
>>> config.amda.user_cache_retention.set(900)
>>> config.amda.user_cache_retention.get()
900
```

2.2 Satellite Situation Center (SSCWeb)

SSCWeb provides trajectories for our solar system space objects such as planets, moons and spacecrafts in different coordinate systems. It's integration into speasy makes easy to get any available object trajectory on any time range.

2.2.1 Basics: Getting data from SSCWeb module

First you need to ensure that the trajectory you want to get is available with this module. The easiest solution is use speasy dynamic inventory so you will always get an up to date inventory:

```
>>> import speasy as spz
>>> # Let's only print the first 10 trajectories
>>> print(list(spz.inventories.flat_inventories.ssc.parameters.keys())[:10])
['ace', 'active', 'aec', 'aed', 'aee', 'aerocube6a', 'aerocube6b', 'aim', 'akebono',
↪'alouette1']
```

Note that you can also use your python terminal completion and browse *spz.inventories.data_tree.ssc.Trajectories* to find your trajectory. Once you have found your trajectory, you may also want to chose in which coordinates system your data will be downloaded. The following coordinates systems are available: **geo**, **gm**, **gse**, **gsm**, **sm**, **geitod**, **geij2000**. By default **gse** is used. Now you can get your trajectory:

```
>>> import speasy as spz
>>> # Let's assume you wanted to get MMS1 trajectory
>>> mms1_traj = spz.ssc.get_data(spz.inventories.data_tree.ssc.Trajectories.mms1, "2018-
↪01-01", "2018-02-01", 'gsm')
>>> mms1_traj.columns
['X', 'Y', 'Z']
>>> mms1_traj.values
array([[57765.77891127, 39928.64689416, 36127.69757491],
       [57636.78726753, 39912.67690181, 36075.18117495],
       [57507.67093183, 39896.65117739, 36022.43945697],
       ...,
       [74135.04374424,   741.72325874, 27240.73393024],
       [74007.246673  ,   795.05699457, 27220.37053627],
       [73879.18392451,   848.35181084, 27199.87604795]])
```

2.3 Coordinated Data Analysis Web (CDAWeb)

The Coordinated Data Analysis Web (CDAWeb) contains selected public non-solar heliophysics data from current and past heliophysics missions and projects. Many datasets from current missions are updated regularly (even daily), including reprocessing older time periods, and SPDF only preserves the latest version.

2.3.1 Basics: Getting data from CDA module

The easiest solution is to use your python terminal completion and browse *spz.inventories.data_tree.cda* to find your product. Once you have found your product, then simply ask CDA module to get data for the provided time range:

```
>>> import speasy as spz
>>> # Let's assume you wanted to get Solar Orbiter 'Magnetic field vector in RTN␣
↪coordinates'
>>> solo_mag_rtn = spz.cda.get_data(spz.inventories.tree.cda.Solar_Orbiter.SOLO.MAG.SOLO_
↪L2_MAG_RTN_NORMAL_1_MINUTE.B_RTN, "2021-01-01", "2021-01-02")
>>> solo_mag_rtn.columns
['B_r', 'B_t', 'B_n']
>>> solo_mag_rtn.values.shape
(1438, 3)
```

2.4 Cluster Science Archive (CSA)

The Cluster Science Archive (CSA) provides access to all science and support data of the on-going Cluster (2000-) and Double Star (2004-2008) missions. It's integration into speasy makes easy to get any public data from the **CSA** handling both webservice **API** and **ISTP CDF** files read.

2.4.1 Basics: Getting data from CSA module

The easiest solution is to use your python terminal completion and browse *spz.inventories.data_tree.csa* to find your product. Once you have found your product, then simply ask CSA module to get data for the provided time range:

```
>>> import speasy as spz
>>> # Let's assume you wanted to get 'Cluster C3, Magnetic Field Vector, spin resolution␣
↪in GSE'
>>> c3_fgm_spin = spz.csa.get_data(spz.inventories.data_tree.csa.Cluster.Cluster_3.FGM3.
↪C3_CP_FGM_SPIN.B_vec_xyz_gse__C3_CP_FGM_SPIN, "2018-01-01", "2018-01-01T01")
>>> c3_fgm_spin.columns
['Bx', 'By', 'Bz']
>>> c3_fgm_spin.values
array([[  4.60300016,  13.44400024, -16.83200073],
       [  4.68400002,  12.85200024, -16.70800018],
       [  2.8599999 ,  12.79399967, -17.36199951],
       ...,
       [ 20.58600044,  -4.40700006, -29.24699974],
       [ 20.74099922,  -0.26800001, -29.07799911],
       [ 20.3560009 ,   1.05200005, -27.90399933]])
```

2.5 Direct archive access

The Direct Archive Access module in Speasy enables users to access any local or remote data archive that stores data in ISTP compliant CDF files. This module does not interact with any web service. Instead, it provides flexibility for users to configure and populate the necessary configuration files to expose the desired products.

Using this module, Speasy can seamlessly retrieve data from the specified data archive, leveraging predictable file names and paths within the archive. By adhering to the ISTP standards, Speasy ensures compatibility and smooth data access.

This module supports both regularly split files (one file per day for example) and randomly split files such as burst data.

To add your favourite products into Speasy, you need to add or edit an yaml file either located in Speasy lookup path, default user lookup path can be retrieved with `spz.webservices.generic_archive.user_inventory_dir()`. You need to add an entry per dataset with the following information:

- For a regularly split dataset, you can configure it using the following YAML structure:

```
tha_efi:
  inventory_path: cdpp/THEMIS/THA/L2
  master_cdf: http://cdpp.irap.omp.eu/themisdata/tha/l2/efi/0000/tha_l2_efi_00000000_v01.
↪cdf
  split_frequency: daily
  split_rule: regular
  url_pattern: http://cdpp.irap.omp.eu/themisdata/tha/l2/efi/{Y}/tha_l2_efi_{Y}{M:02d}{D:
↪02d}_v\d+.cdf
  use_file_list: true
```

Here's an explanation of the parameters::

- **tha_efi**: The name you want to assign to your dataset.
- **inventory_path:**: The desired inventory path for your dataset. In this example, it can be found in `spz.inventories.data_tree.archive.cdpp.THEMIS.THA.L2.tha_efi`.

- **master_cdf:**: The URL or path to download a master CDF or any sample CDF for this dataset. Speasy requires it to complete the inventory with the dataset's data variables. It is recommended to use master CDFs as they contain sufficient information while being smaller in size.
- **split_frequency:**: The frequency at which your dataset is split. For example, if you have one file per day, month, or year. Allowed values are *daily*, *monthly*, *yearly*.
- **url_pattern:**: The URL pattern to access each file. When requesting data within a specific interval, Speasy utilizes the *split_frequency* to determine the number of files to download and replaces the date/time information accordingly. It uses python *{}* format syntax, and available date/time placeholders are year (**Y**), month (**M**) and day (**D**) are available. You can also utilize Python regular expressions if you are unable to predict certain parts of the file name, such as the file version, but you have set *use_file_list* to true.
- **use_file_list:**: If set to true, Speasy lists the files in the specified directory after generating the URL based on the *url_pattern*. It then selects the last matching file.

- For a randomly split dataset, you can configure it using the following YAML structure:

```yaml
mms2_fpi_brst_l2_des_moms:
    url_pattern: 'https://cdaweb.gsfc.nasa.gov/pub/data/mms/mms2/fpi/brst/l2/des-moms/{Y}
↪/{M:02d}/mms2_fpi_brst_l2_des-moms_{Y}{M:02d}\d+_v\d+.\d+.\d+.cdf'
    use_file_list: true
    master_cdf: "https://cdaweb.gsfc.nasa.gov/pub/software/cdawlib/0MASTERS/mms2_fpi_
↪brst_l2_des-moms_00000000_v01.cdf"
    inventory_path: 'cda/MMS/MMS2/FPI/BURST/MOMS'
    split_rule: "random"
    split_frequency: "monthly"
    fname_regex: 'mms2_fpi_brst_l2_des-moms_(?P<start>\d+)_v(?P<version>[\d\.]+)\.cdf'
```

Here's an explanation of the parameters::

- **mms2_fpi_brst_l2_des_moms**: The name you want to assign to your dataset.
- **inventory_path:**: The desired inventory path for your dataset. In this example, it can be found in `spz.inventories.data_tree.archive.cda.MMS.MMS2.FPI.BURST.MOMS.mms2_fpi_brst_l2_des_moms`.
- **master_cdf:**: The URL or path to download a master CDF or any sample CDF for this dataset. Speasy requires it to complete the inventory with the dataset's data variables. It is recommended to use master CDFs as they contain sufficient information while being smaller in size.
- **split_frequency:**: The frequency at which folders are is split for this dataset. For example, if you have one folder per day, month, or year. Allowed values are *daily*, *monthly*, *yearly*.
- **url_pattern:**: The URL pattern to access files covering the current time range. When requesting data within a specific interval, Speasy utilizes the *split_frequency* to determine the number of folders to scan and replaces the date/time information accordingly. It uses python *{}* format syntax, and available date/time placeholders are year (**Y**), month (**M**) and day (**D**) are available. With randomly split datasets, it is important to ensure that the URL pattern includes the fixed and deterministic parts and rely on the **fname_regex** field to match files that cover the requested time range.
- **use_file_list:**: If set to true, Speasy lists the files in the specified directory after generating the URL based on the *url_pattern*. It then selects the last matching file.
- **fname_regex: This regular expression is used to extract information such as the start date, stop date, and file version from the file names. It follows Python's regular expression syntax and captures specific groups. The expected or supported groups are:**
  - *start*: Start date, it should be parsable by `dateutil.parser.parse` (mandatory)
  - *stop*: Stop date, same as start date (optional)

  - *version*: Dataset version (optional)

Speasy provides access to the following Web Services:

- *Automated Multi-Dataset Analysis (AMDA)*
- *Satellite Situation Center Web (SSCWeb)*
- *Coordinated Data Analysis Web (CDAWeb)*
- *Cluster Science Archive (CSA)*
- *Direct archive access*

While you can download any data with `speasy.get_data()`, each web service have specificities and might expose extra features through their dedicated modules.

Note: At any time, you can disable a web service by adding it to the `disabled_services` list in your configuration file, see *Disabling data providers*.

CHAPTER

THREE

SPEASY CONFIGURATION

Speasy is configured using the config module or setting environment variables or editing an ini file. The default location can be found by running:

```
>>> import speasy as spz
>>> print(spz.config.SPEASY_CONFIG_FILE)
```

Speasy current configuration can be displayed by running:

```
>>> import speasy as spz
>>> spz.config.show()
```

3.1 Core section

3.1.1 Disabling data providers

Sometimes you may want to disable some data providers either to speed up Speasy import or because you don't need them. This can be done by adding the provider name to the *disabled_providers* list in the configuration file.

For example, to disable AMDA and CDAWeb, add the following to the configuration file:

```
[core]
disabled_providers = amda,cdaweb
```

Or from Python:

```
>>> import speasy as spz
>>> spz.config.core.disabled_providers.set('amda,cdaweb')
```

3.2 Cache section

You can configure the cache location and maximum size by editing the *cache* section of the configuration file.

```
[CACHE]
path = /path/to/cache
size = 1e9
```

Or from Python:

```
>>> import speasy as spz
>>> spz.config.cache.path.set('/path/to/cache')
>>> spz.config.cache.size.set(1e9)
```

CHAPTER

FOUR

SPEASY EXAMPLES GALLERY

Browse example folder on MyBinder:

Browse example folder on Google Colab:

The following section was generated from `docs/examples/AMDA.ipynb`

4.1 AMDA first steps

4.1.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

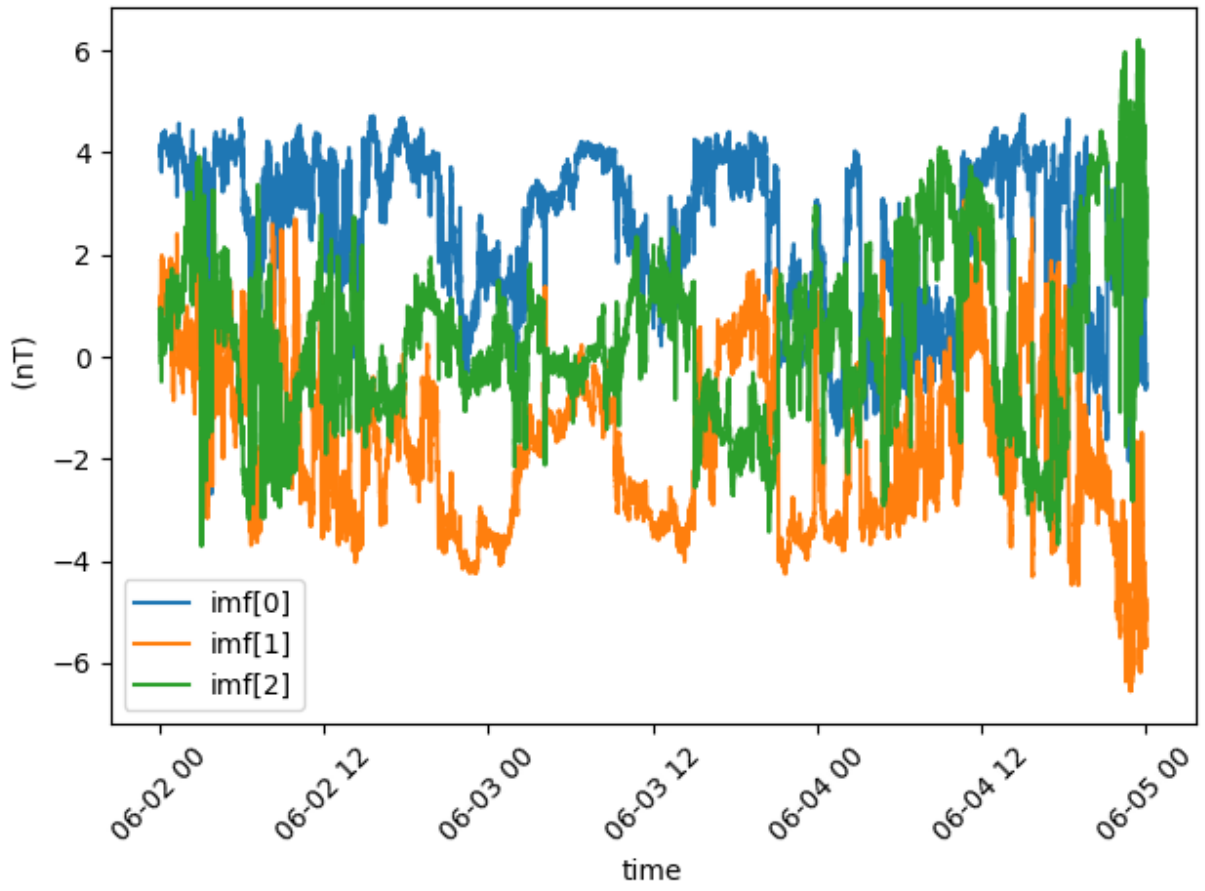
[ ]:
```
try:
    from google.colab import output
    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.1.2 For all users:

[1]:
```
import speasy as spz
%matplotlib widget
amda_tree = spz.inventories.tree.amda
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from datetime import datetime
```

4.1.3 A simple example with ACE IMF data

[2]:
```
plt.figure()
ace_mag = spz.get_data(amda_tree.Parameters.ACE.MFI.ace_imf_all.imf, datetime(2016,6,2),␣
↪datetime(2016,6,5))
ace_mag.plot()
plt.tight_layout()
plt.show()
```



The following section was generated from `docs/examples/CDAWeb.ipynb`

4.2 CDAWeb first steps

4.2.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output
    output.enable_custom_widget_manager()
```

(continues on next page)

(continued from previous page)

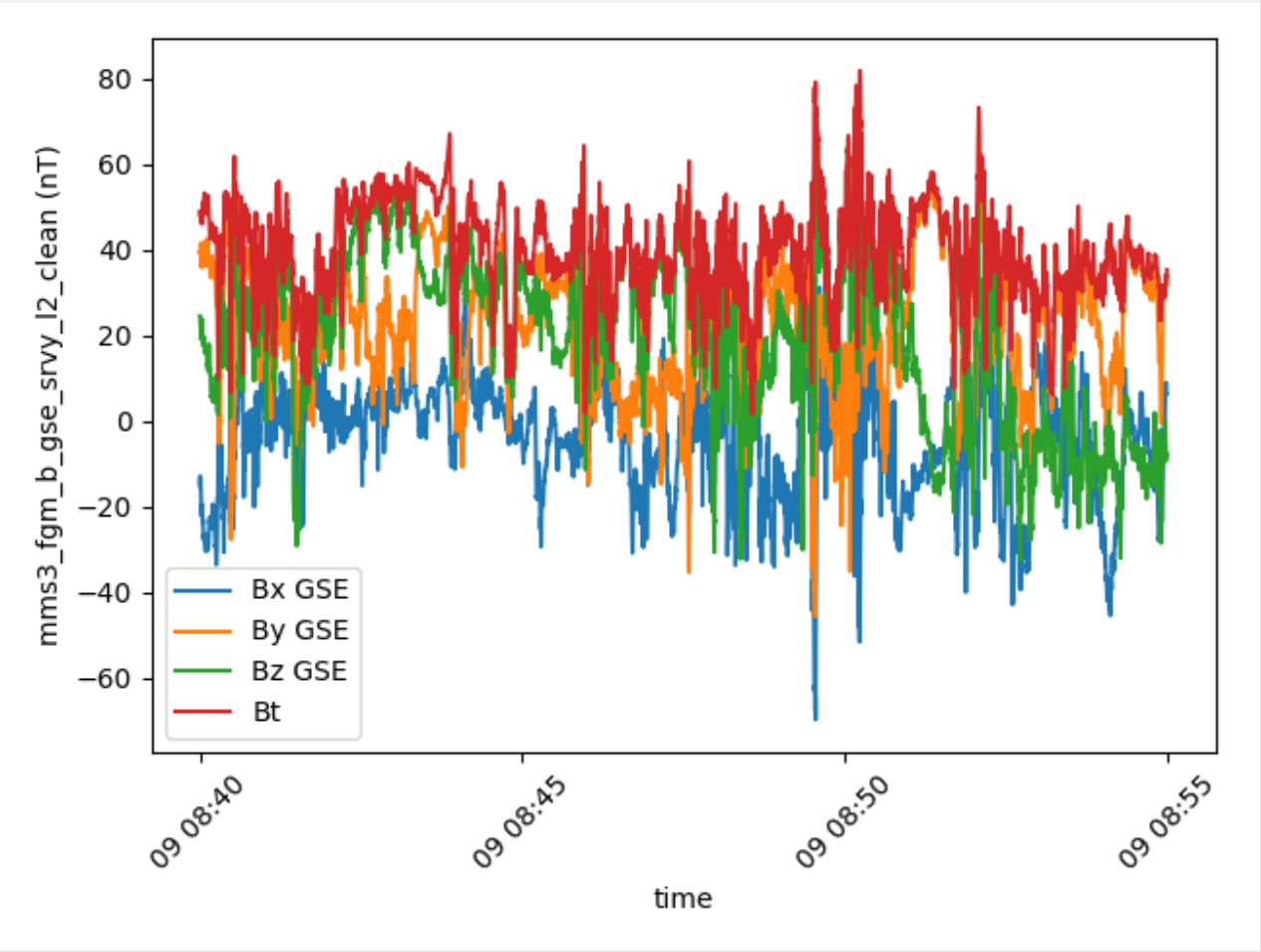
```
except:
    print("Not running inside Google Collab")
```

4.2.2 For all users:

[1]:
```
import speasy as spz
%matplotlib widget
cda_tree = spz.inventories.tree.cda
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from datetime import datetime
```

4.2.3 A simple example with MMS FGM data

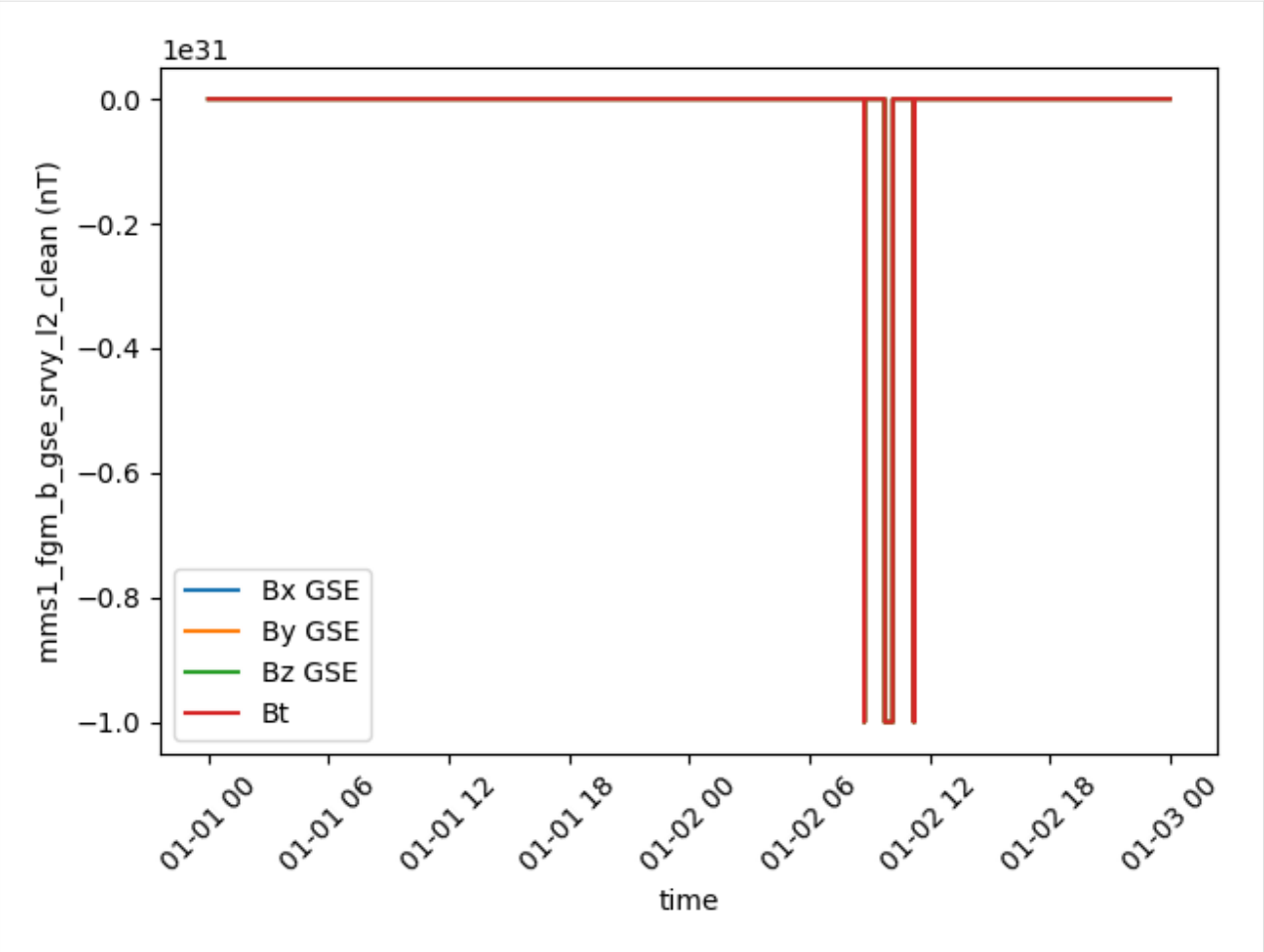
[5]:
```
fig = plt.figure()
mms3_fgm_b_gse_srvy = spz.get_data(cda_tree.MMS.MMS3.FGM.MMS3_FGM_SRVY_L2.mms3_fgm_b_gse_
↪srvy_l2_clean, "2015-09-09T08:40",
                                   "2015-09-09T08:55")
mms3_fgm_b_gse_srvy.plot()
plt.tight_layout()
plt.show()
```



4.2.4 Replacing fill values by NaN and filtering components

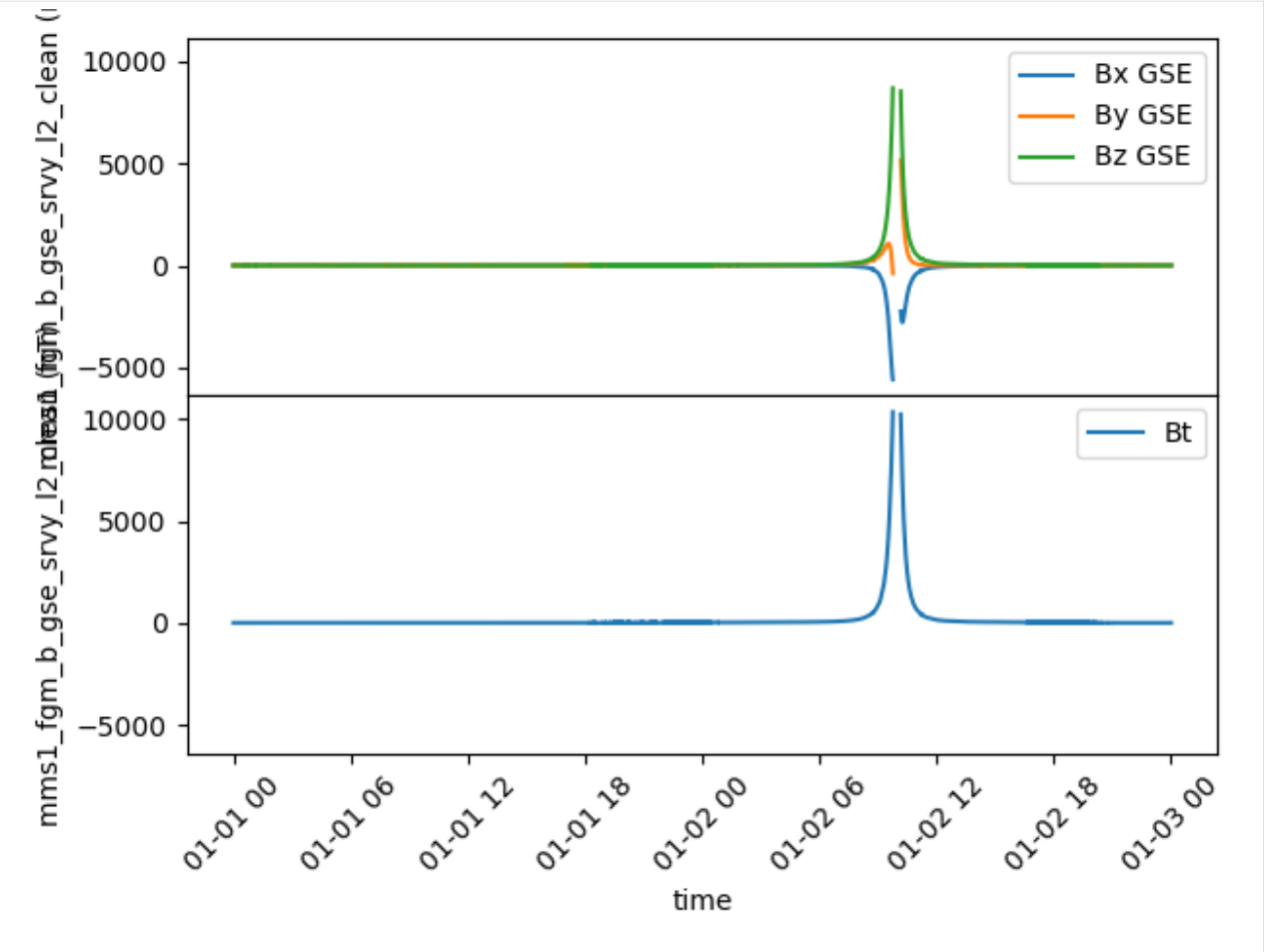
Let's look at the data as we get it on an interval where there are some fill values

[6]:
```
fig = plt.figure()
mms1_fgm_b_gse_srvy = spz.get_data(cda_tree.MMS.MMS1.FGM.MMS1_FGM_SRVY_L2.mms1_fgm_b_gse_
↪srvy_l2_clean, "2019-01-01",
                                   "2019-01-03")
mms1_fgm_b_gse_srvy.plot()
plt.tight_layout()
plt.show()
```



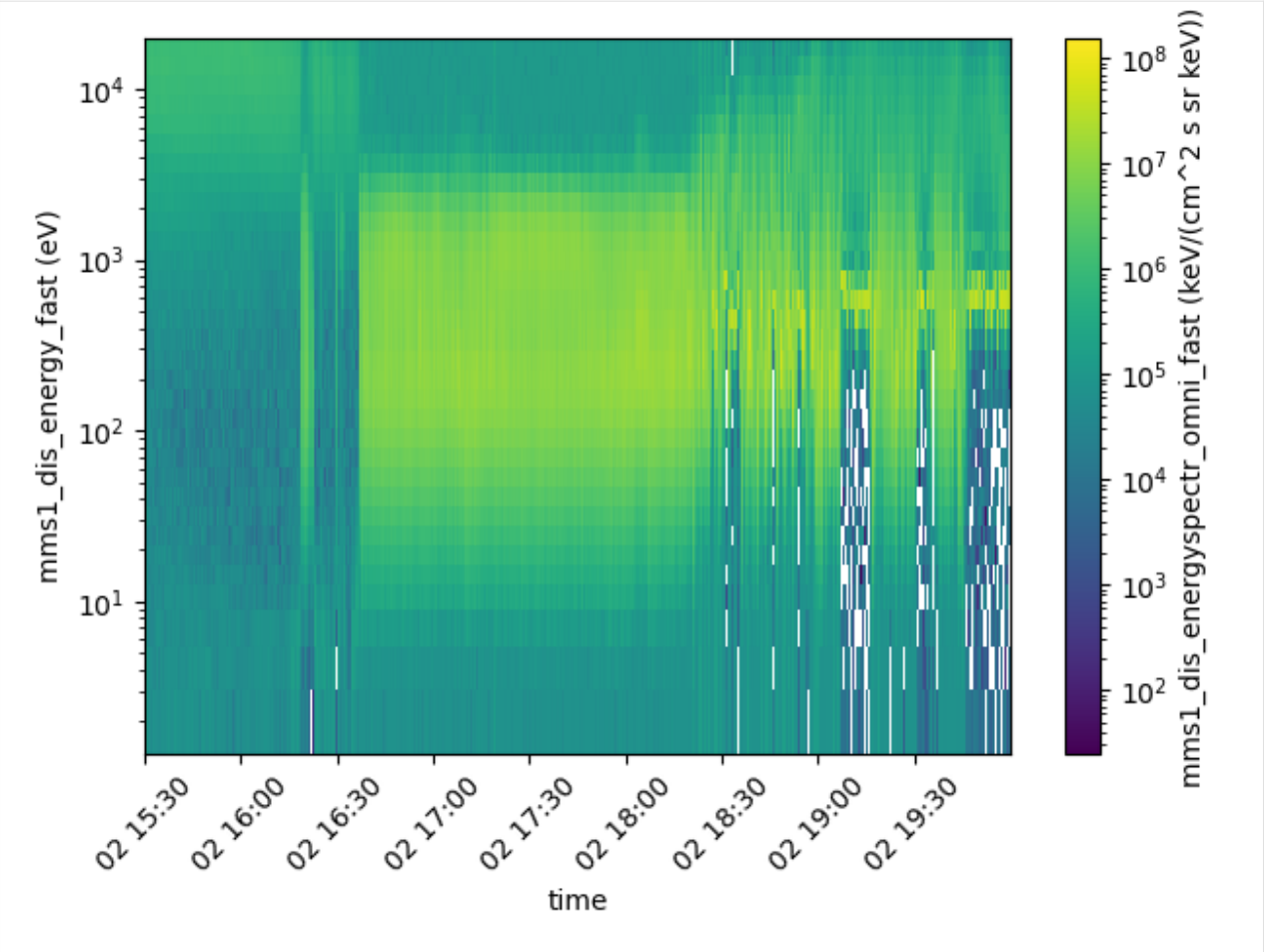
Now let's replace fill values by NaNs

[7]:
```
fig = plt.figure()
gs = fig.add_gridspec(2, hspace=0)
ax = gs.subplots(sharex=True, sharey=True)
mms1_fgm_b_gse_srvy = spz.get_data(cda_tree.MMS.MMS1.FGM.MMS1_FGM_SRVY_L2.mms1_fgm_b_gse_
↪srvy_l2_clean, "2019-01-01",
                                   "2019-01-03")
mms1_fgm_b_gse_srvy["Bx GSE", "By GSE", "Bz GSE"].replace_fillval_by_nan().plot(ax=ax[0])
mms1_fgm_b_gse_srvy["Bt"].replace_fillval_by_nan().plot(ax=ax[1])
plt.tight_layout()
plt.show()
```



4.2.5 Another example with an MMS FPI spectrogram

[9]:
```
mms1_dis_energyspectr_omni_fast = spz.get_data(
    cda_tree.MMS.MMS1.DIS.MMS1_FPI_FAST_L2_DIS_MOMS.mms1_dis_energyspectr_omni_fast,
↪"2019-01-02T15:30",
    "2019-01-02T20")

plt.figure()
mms1_dis_energyspectr_omni_fast.plot["matplotlib"].colormap(cmap='viridis')
plt.tight_layout()
plt.show()
```



The following section was generated from `docs/examples/SSCWeb.ipynb`

4.3 SSCWeb first steps

4.3.1 Only for Google Colab users:

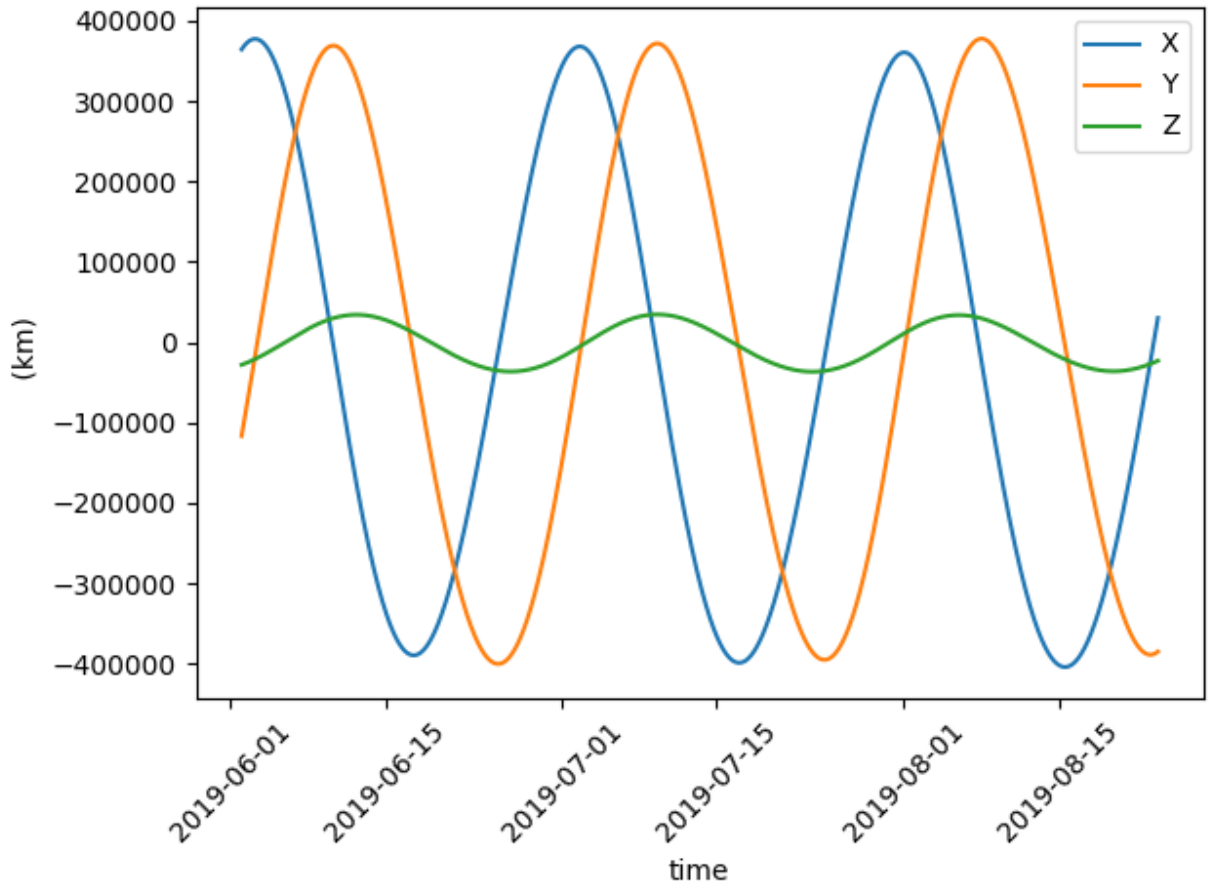
[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output

    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.3.2 For all users:

[1]:
```
import speasy as spz
ssc_tree = spz.inventories.tree.ssc
%matplotlib widget
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from datetime import datetime
from astropy import units
import numpy as np
```

[2]:
```
plt.figure()
moon_orbit = spz.get_data(ssc_tree.Trajectories.moon,
                          datetime(2019,6,2), datetime(2019,8,24), coordinate_system='gse
↪')
moon_orbit.plot()
plt.tight_layout()
plt.show()
```

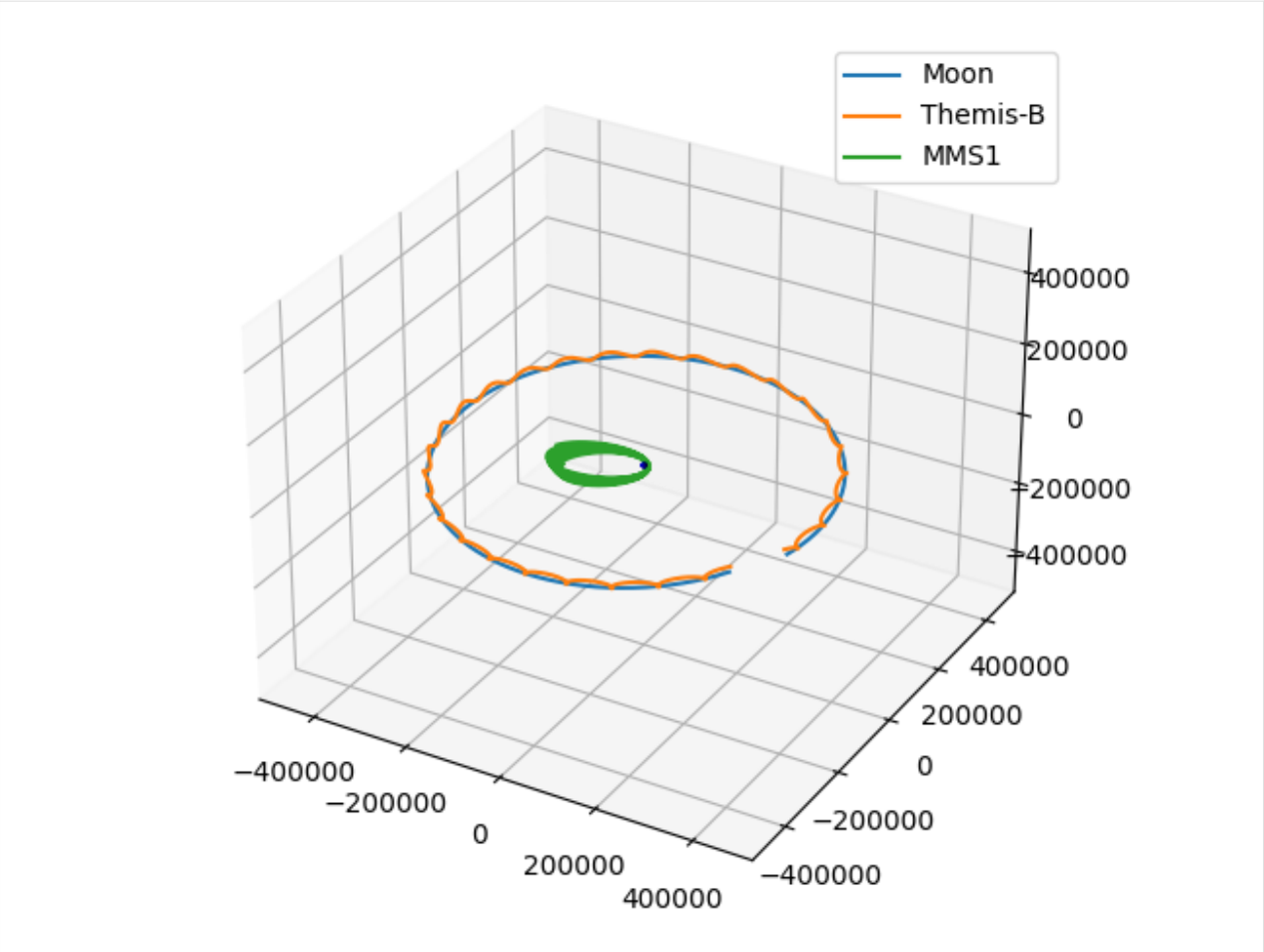


[3]:
```
def plot_traj(var, ax, label):
    ax.plot(var.values[:,0],var.values[:,1],var.values[:,2], label=label)
```

(continues on next page)

(continued from previous page)

```
def plot_earth(ax):
    u = np.linspace(0, 2 * np.pi, 100)
    v = np.linspace(0, np.pi, 100)
    x = 6371 * np.outer(np.cos(u), np.sin(v))
    y = 6371 * np.outer(np.sin(u), np.sin(v))
    z = 6371 * np.outer(np.ones(np.size(u)), np.cos(v))
    ax.plot_surface(x, y, z, color='b')

themisb_orbit = spz.get_data(ssc_tree.Trajectories.themisb,
                         datetime(2019,6,2), datetime(2019,6,30), coordinate_system='gse
↪')

mms1_orbit = spz.get_data(ssc_tree.Trajectories.mms1,
                         datetime(2019,6,2), datetime(2019,6,30), coordinate_system='gse
↪')

moon_orbit = spz.get_data(ssc_tree.Trajectories.moon,
                         datetime(2019,6,2), datetime(2019,6,30), coordinate_system='gse
↪')

fig = plt.figure()
ax = fig.add_subplot(projection='3d')
plot_traj(moon_orbit, ax, 'Moon')
plot_traj(themisb_orbit, ax, 'Themis-B')
plot_traj(mms1_orbit, ax, 'MMS1')
plot_earth(ax)
ax.set_xlim(-50e4,50e4)
ax.set_ylim(-50e4,50e4)
ax.set_zlim(-50e4,50e4)
ax.legend()
plt.tight_layout()
plt.show()
```



The following section was generated from `docs/examples/Caches.ipynb`

4.4 Speasy caches levels analysis

4.4.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output
    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.4.2 For all users:

[ ]:
```
import speasy as spz
%matplotlib widget
amda_tree = spz.inventories.tree.amda
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from datetime import datetime
```

[1]:
```
import speasy as spz

amda_tree = spz.inventories.tree.amda
%matplotlib widget
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from datetime import datetime, timedelta
import time
import numpy as np
```

First ensure that speasy is setup to use SciQLop cache

[2]:
```
spz.config.proxy.url.set('http://sciqlop.lpp.polytechnique.fr/cache-dev')
spz.config.proxy.enabled.set(True)
```

[3]:
```
start_time = datetime(2016, 6, 2)
stop_time = datetime(2016, 6, 8)
reference_data = spz.get_data(amda_tree.Parameters.ACE.MFI.ace_imf_all.imf, start_time,␣
↪stop_time, progress=False)
print(f"Data shape: {reference_data.values.shape}")
print(f"Data size in Bytes: {reference_data.nbytes}")
```

```
Data shape: (32400, 3)
Data size in Bytes: 1039238
```

[4]:
```
def times(f, *args, n=10, **kwargs):
    def time_once():
        start = time.perf_counter_ns()
        f(*args, **kwargs, progress=False)
        stop = time.perf_counter_ns()
        return (stop - start) / 1e6

    return [time_once() for _ in range(n)]

def best_99_percent(times):
    return sorted(times)[:int(len(times)*.99)]

def best_90_percent(times):
    return sorted(times)[:int(len(times)*.9)]
```

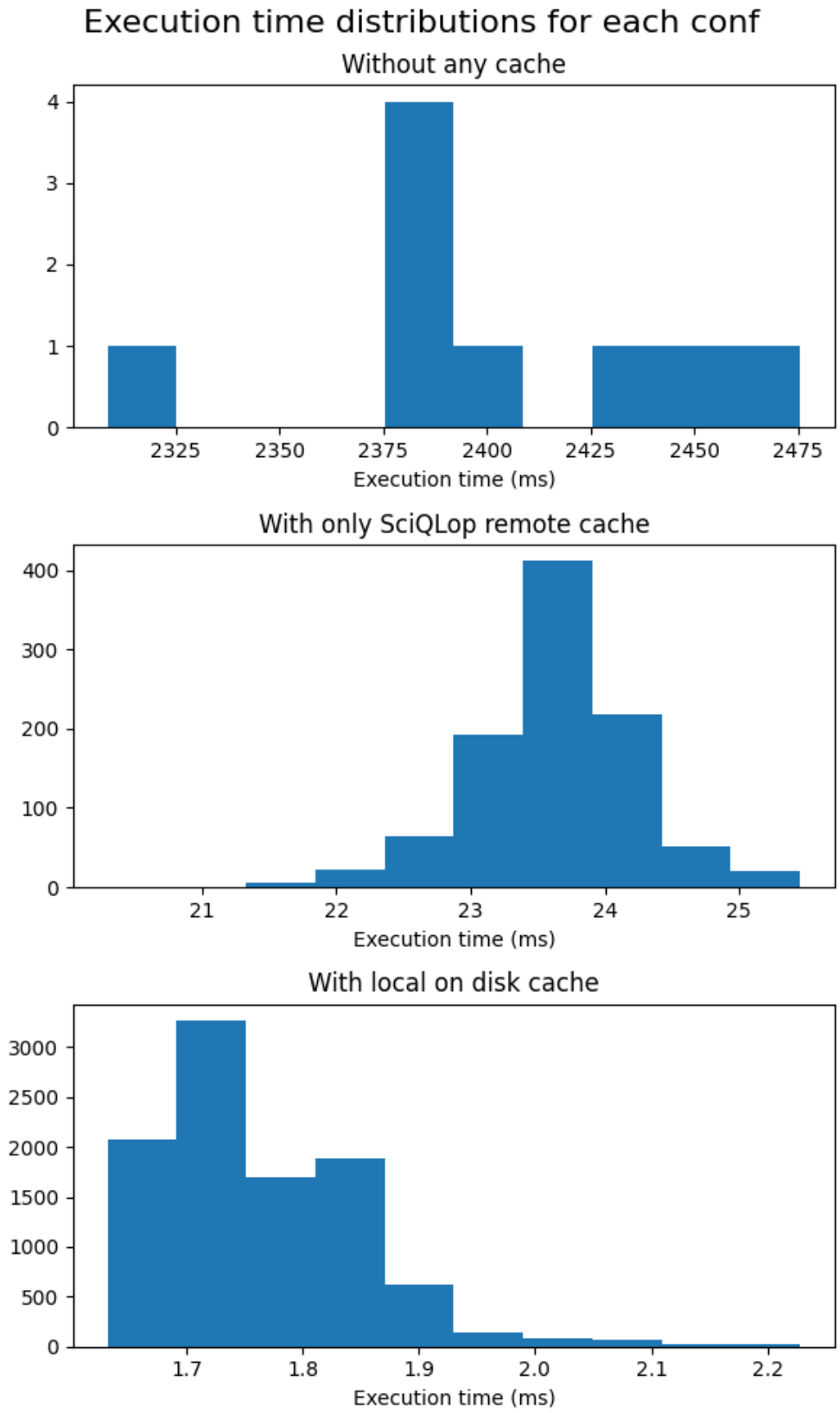
4.4.3 Cache level comparison

Then request data several times with all 3 configurations:

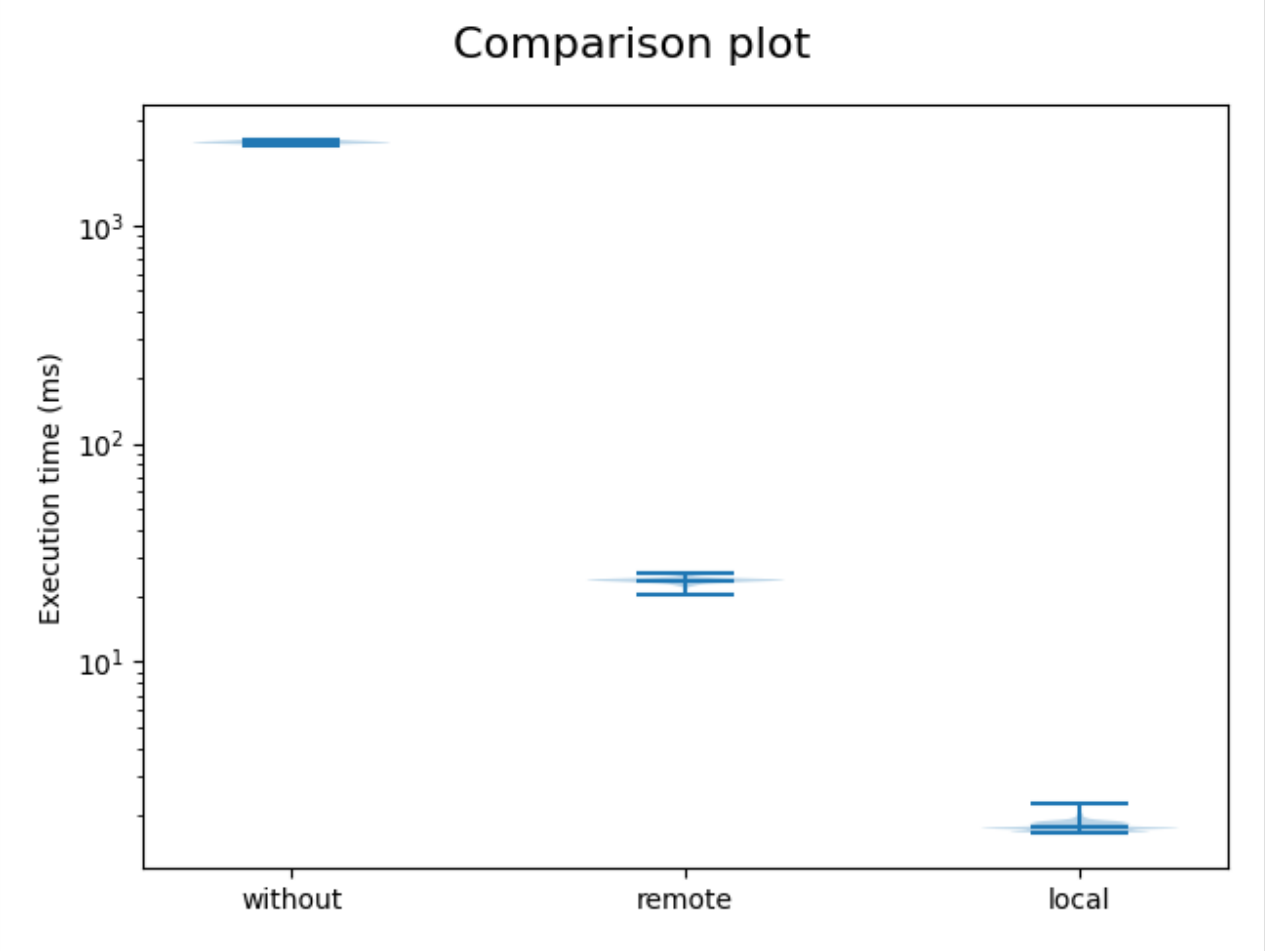
- **without any cache**, each time speasy will download data from AMDA
- **with remote cache only**, each time speasy download data from our remote cahe hosted here
- **with local cahe**, each time after the first request speasy will load data from your disk

[5]:
```
durations_without_any_cache = times(spz.get_data, amda_tree.Parameters.ACE.MFI.ace_imf_
↪all.imf, start_time, stop_time,
                                    disable_cache=True, disable_proxy=True, n=10);
durations_with_remote_cache = times(spz.get_data, amda_tree.Parameters.ACE.MFI.ace_imf_
↪all.imf, start_time, stop_time,
                                    disable_cache=True, n=1000);
durations_with_local_cache = times(spz.get_data, amda_tree.Parameters.ACE.MFI.ace_imf_
↪all.imf, start_time, stop_time,
                                   n=10000);
```

[6]:
```
fig, axs = plt.subplots(3, 1, figsize=(6, 10))
for i, data, title in ((0, best_99_percent(durations_without_any_cache), 'Without any␣
↪cache'),
                       (1, best_99_percent(durations_with_remote_cache), 'With only␣
↪SciQLop remote cache'),
                       (2, best_99_percent(durations_with_local_cache), 'With local on␣
↪disk cache')):
    axs[i].hist(data)
    axs[i].set_xlabel('Execution time (ms)')
    axs[i].set_title(title)

fig.suptitle('Execution time distributions for each conf', fontsize=16)
plt.tight_layout()
plt.show()
```


Execution time distributions for each conf
Without any cache
4
3
2
1
0
2325
2350
2375
2400
2425
2450
2475
Execution time (ms)
With only SciQLop remote cache
400
300
200
100
0
21
22
23
24
25
Execution time (ms)
With local on disk cache
3000
2500
2000
1500
1000
500
0
1.7
1.8
1.9
2.0
2.1
2.2
Execution time (ms)

[7]:
```
fig, ax = plt.subplots()
ax.violinplot([best_99_percent(durations_without_any_cache), best_99_percent(durations_
↪with_remote_cache), best_99_percent(durations_with_local_cache), ], showmeans=False,
              showmedians=True)
ax.set_xticks([1, 2, 3], labels=['without', 'remote', 'local'])
ax.set_ylabel('Execution time (ms)')
plt.semilogy()
fig.suptitle('Comparison plot', fontsize=16)
plt.tight_layout()
plt.show()
```



4.4.4 Scaling

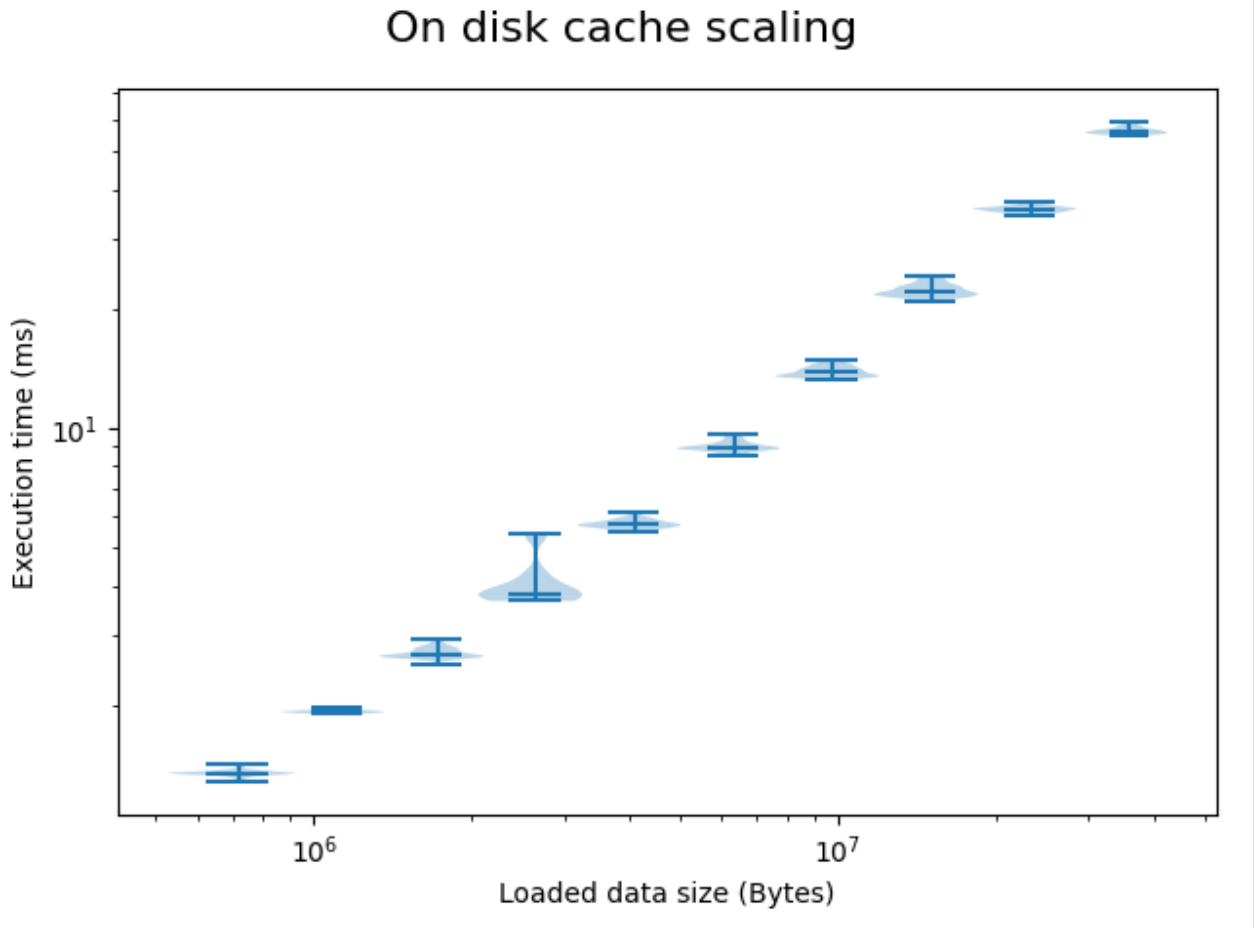
On disk cache scaling

[8]:
```
start_time = datetime(2016, 6, 2)


def scaling_point(delta):
    stop_time = start_time + timedelta(hours=delta)
    data = spz.get_data(amda_tree.Parameters.ACE.MFI.ace_imf_all.imf, start_time, stop_
```

(continues on next page)

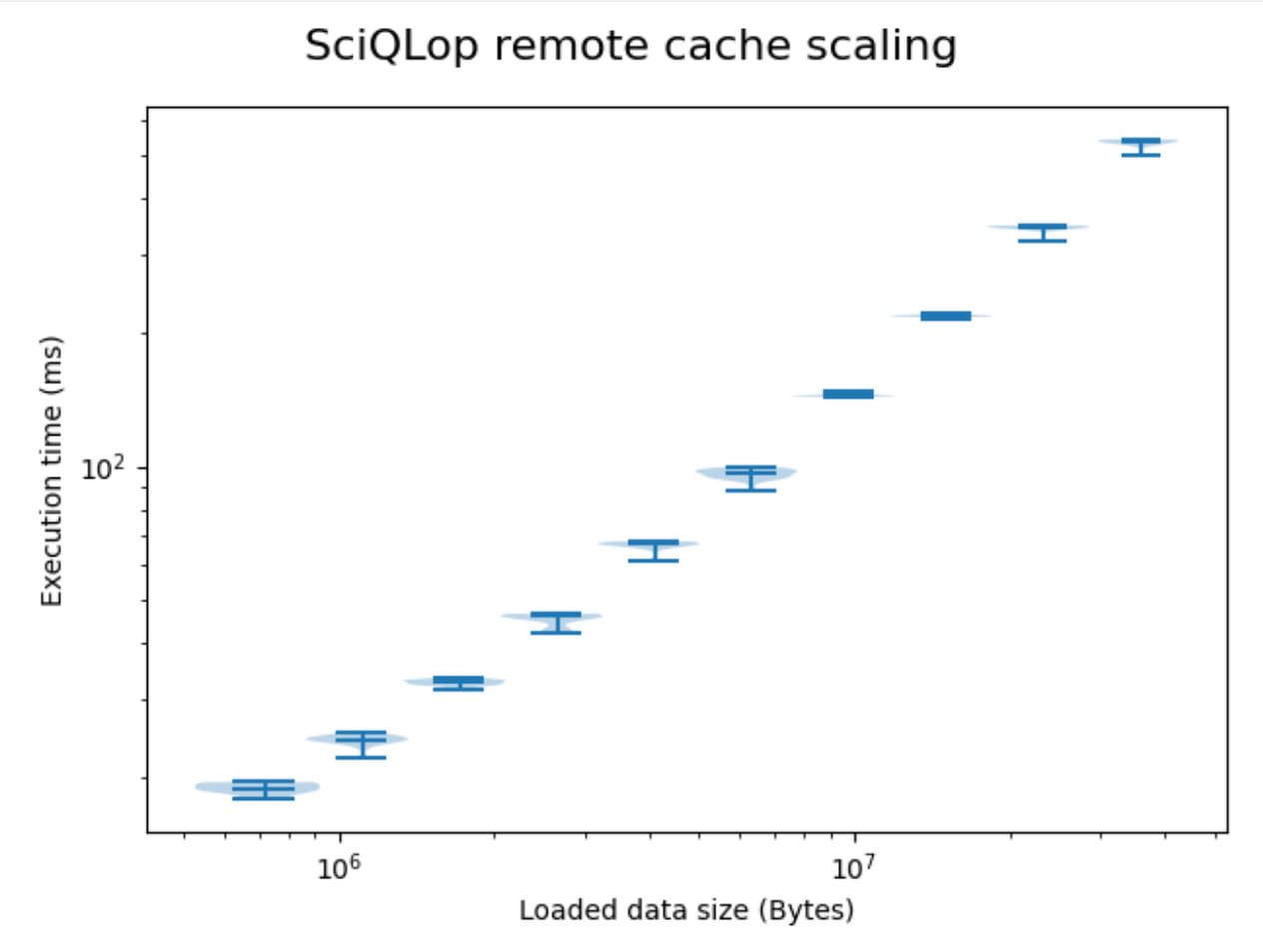
(continued from previous page)

```
↪time)
    capacity = data.nbytes
    t = best_90_percent(times(spz.get_data, amda_tree.Parameters.ACE.MFI.ace_imf_all.imf,
↪ start_time, stop_time, n=200))
    return capacity, t


deltas = np.logspace(np.log10(100), np.log10(5000), num=10)
values = [scaling_point(delta) for delta in deltas]


fig, ax = plt.subplots()
stats = [t for c, t in values]
capacities = np.array([c for c, t in values])
ax.violinplot(stats, positions=capacities, widths=np.gradient(capacities),␣
↪showmeans=False, showmedians=True)
ax.set_ylabel('Execution time (ms)')
ax.set_xlabel('Loaded data size (Bytes)')
fig.suptitle('On disk cache scaling', fontsize=16)
plt.tight_layout()
plt.loglog()
plt.show()
```



SciQLop remote cache scaling

[9]:
```
start_time = datetime(2016, 6, 2)


def scaling_point(delta):
    stop_time = start_time + timedelta(hours=delta)
    data = spz.get_data(amda_tree.Parameters.ACE.MFI.ace_imf_all.imf, start_time, stop_
↪time, disable_cache=True)
    capacity = data.nbytes
    t = best_90_percent(times(spz.get_data, amda_tree.Parameters.ACE.MFI.ace_imf_all.imf,
↪ start_time, stop_time, disable_cache=True,
                n=20))
    return capacity, t


deltas = np.logspace(np.log10(100), np.log10(5000), num=10)
values = [scaling_point(delta) for delta in deltas]

fig, ax = plt.subplots()
stats = [t for c, t in values]
capacities = np.array([c for c, t in values])
ax.violinplot(stats, positions=capacities, widths=np.gradient(capacities),␣
↪showmeans=False, showmedians=True)
ax.set_ylabel('Execution time (ms)')
ax.set_xlabel('Loaded data size (Bytes)')
fig.suptitle('SciQLop remote cache scaling', fontsize=16)
plt.tight_layout()
plt.loglog()
plt.show()
```



The following section was generated from `docs/examples/CompleteDemo.ipynb`

4.5 A more complete demo of Speasy

4.5.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output

    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

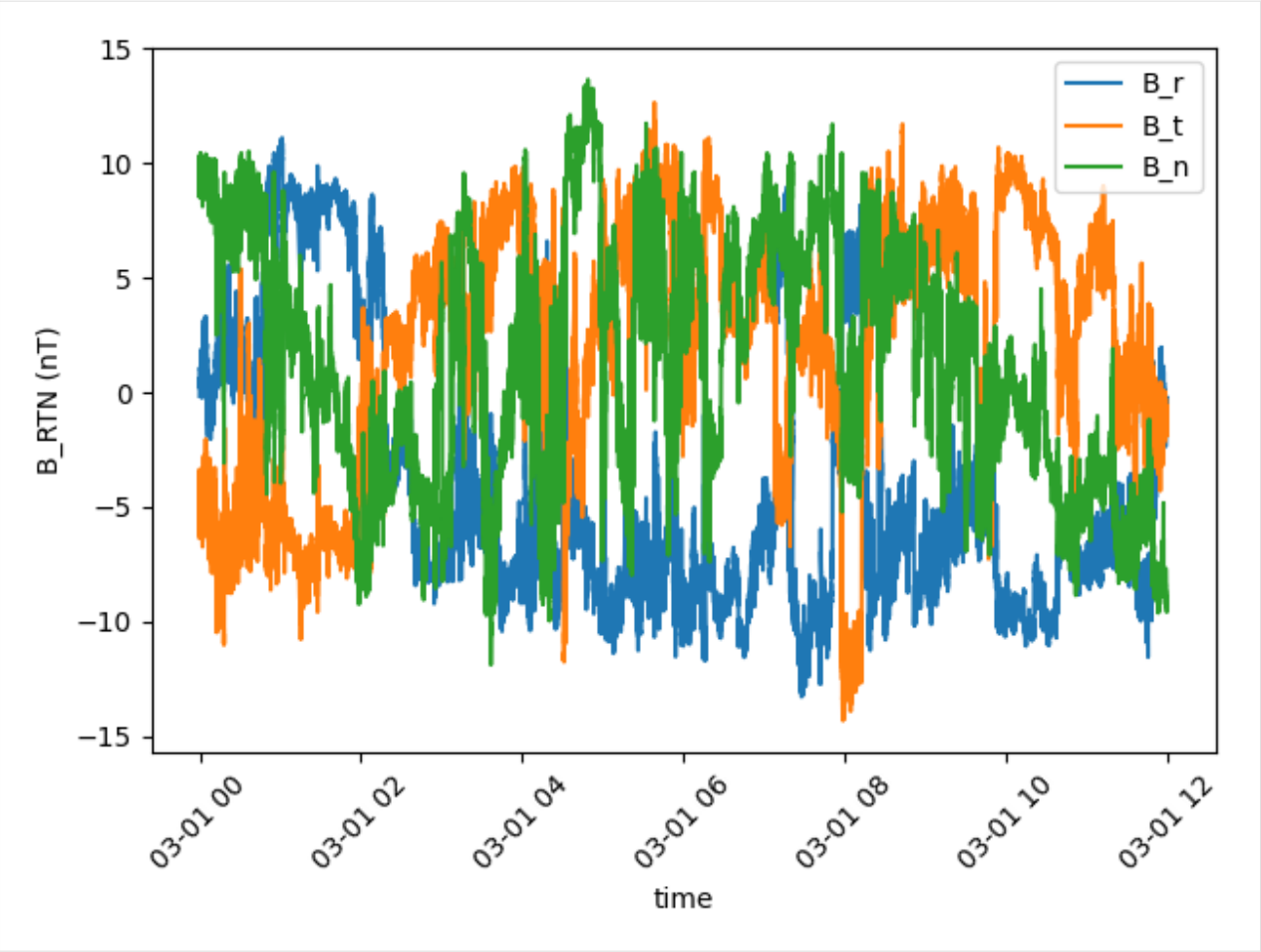
4.5.2 For all users:

[1]:
```
%matplotlib widget

import matplotlib.pyplot as plt
import speasy as spz
from speasy.products import SpeasyVariable
from typing import List
import numpy as np
from datetime import datetime

#plt.rcParams["figure.figsize"] = (20, 4)
```

4.5.3 Few products from CDAWeb

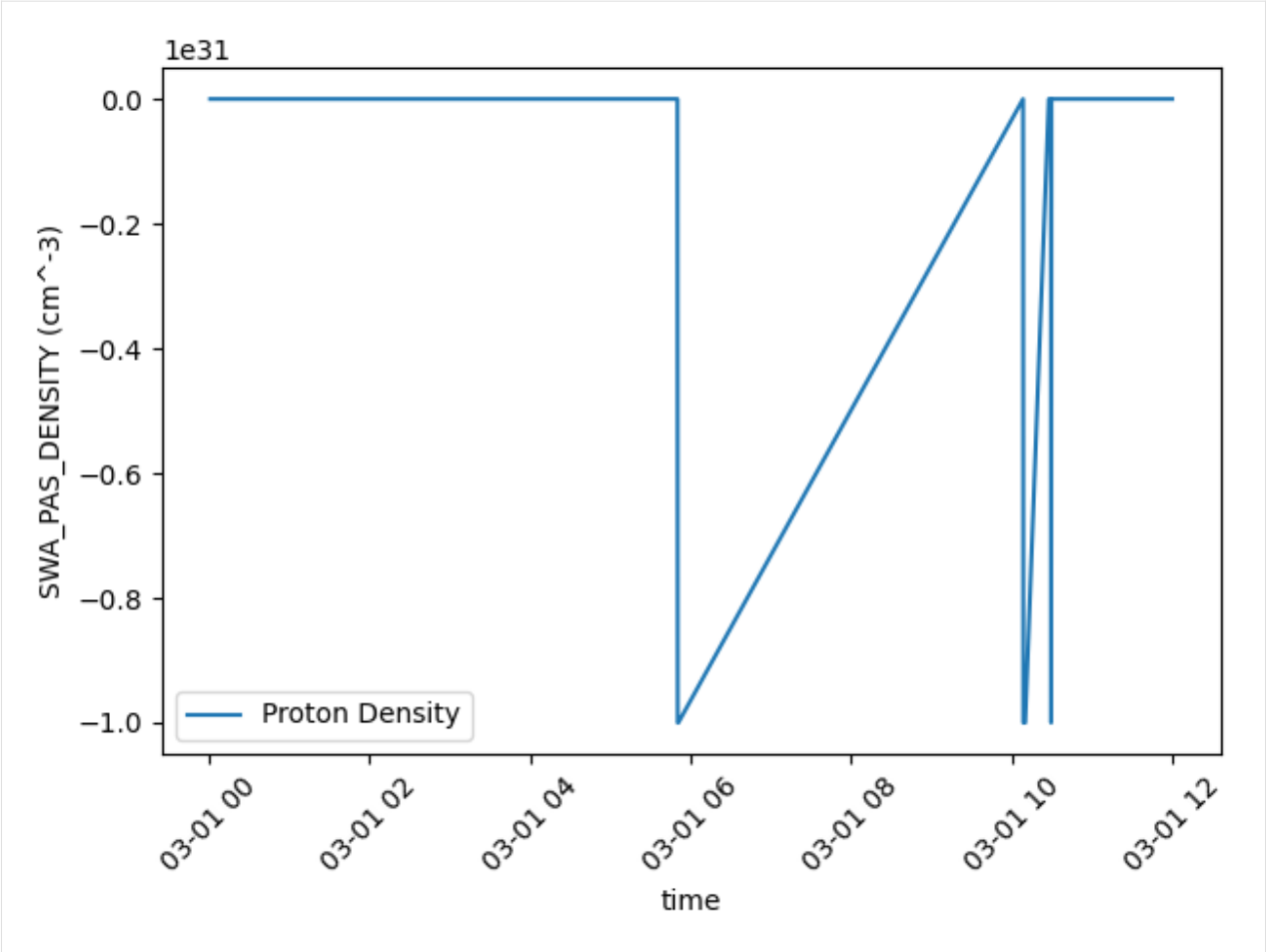
You can browse CDAWeb inventory using your python environement completion and get any 'variable'(CDA name) or 'parameter' (Speasy name) on any valid time interval. Then most SpeasyVariables can be plotted directly (using matplotlib backend).

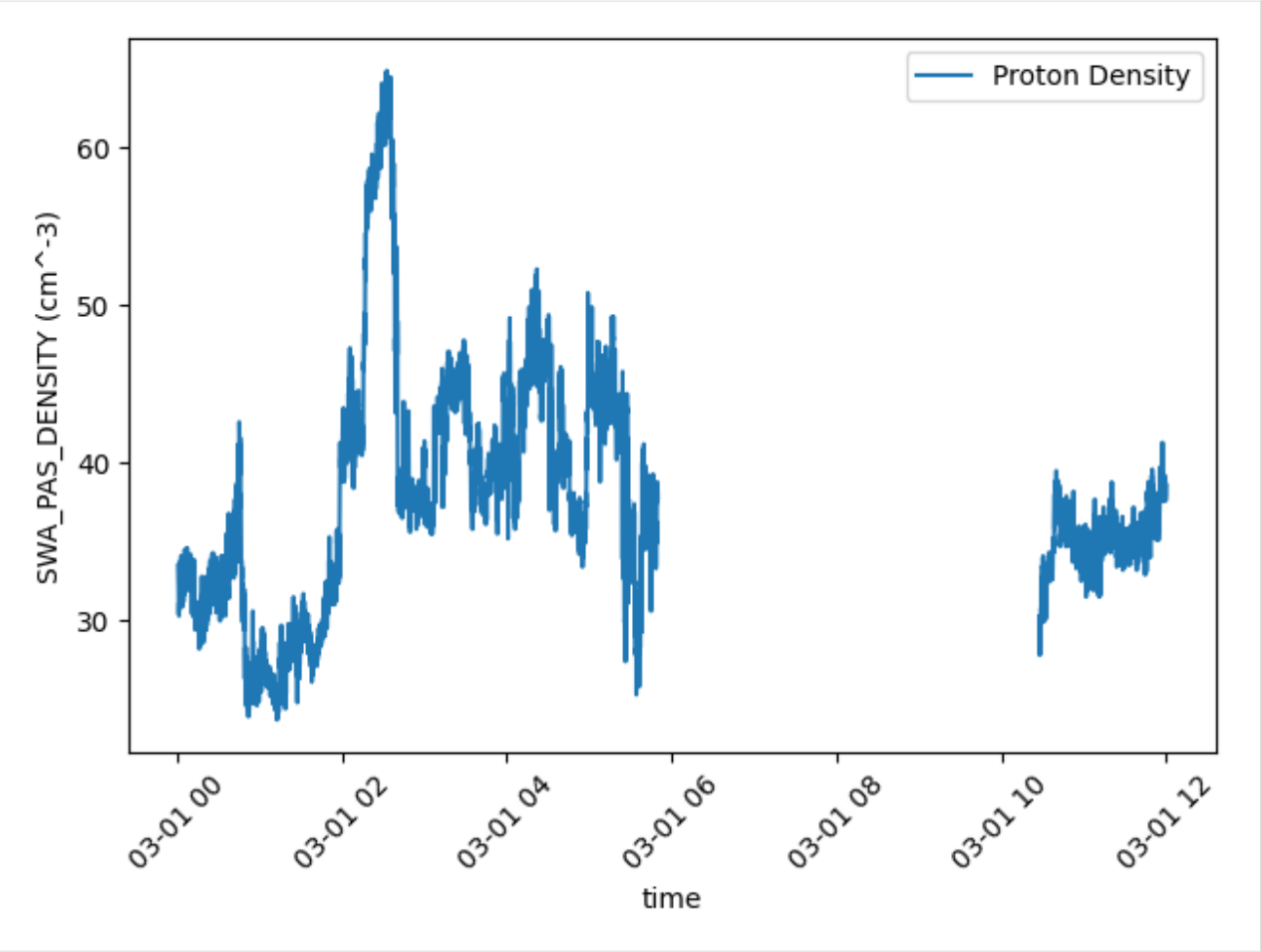
[2]:
```
solo_fgm: SpeasyVariable = spz.get_data(
    spz.inventories.tree.cda.Solar_Orbiter.SOLO.MAG.SOLO_L2_MAG_RTN_NORMAL.B_RTN,
    "2022-03-01",
    "2022-03-01T12",
)
plt.figure()
solo_fgm.plot()
plt.tight_layout()
plt.show()
```



It is quite common to have **fill values**, you can replace them by **NaN** by simply calling `replace_fillval_by_nan`

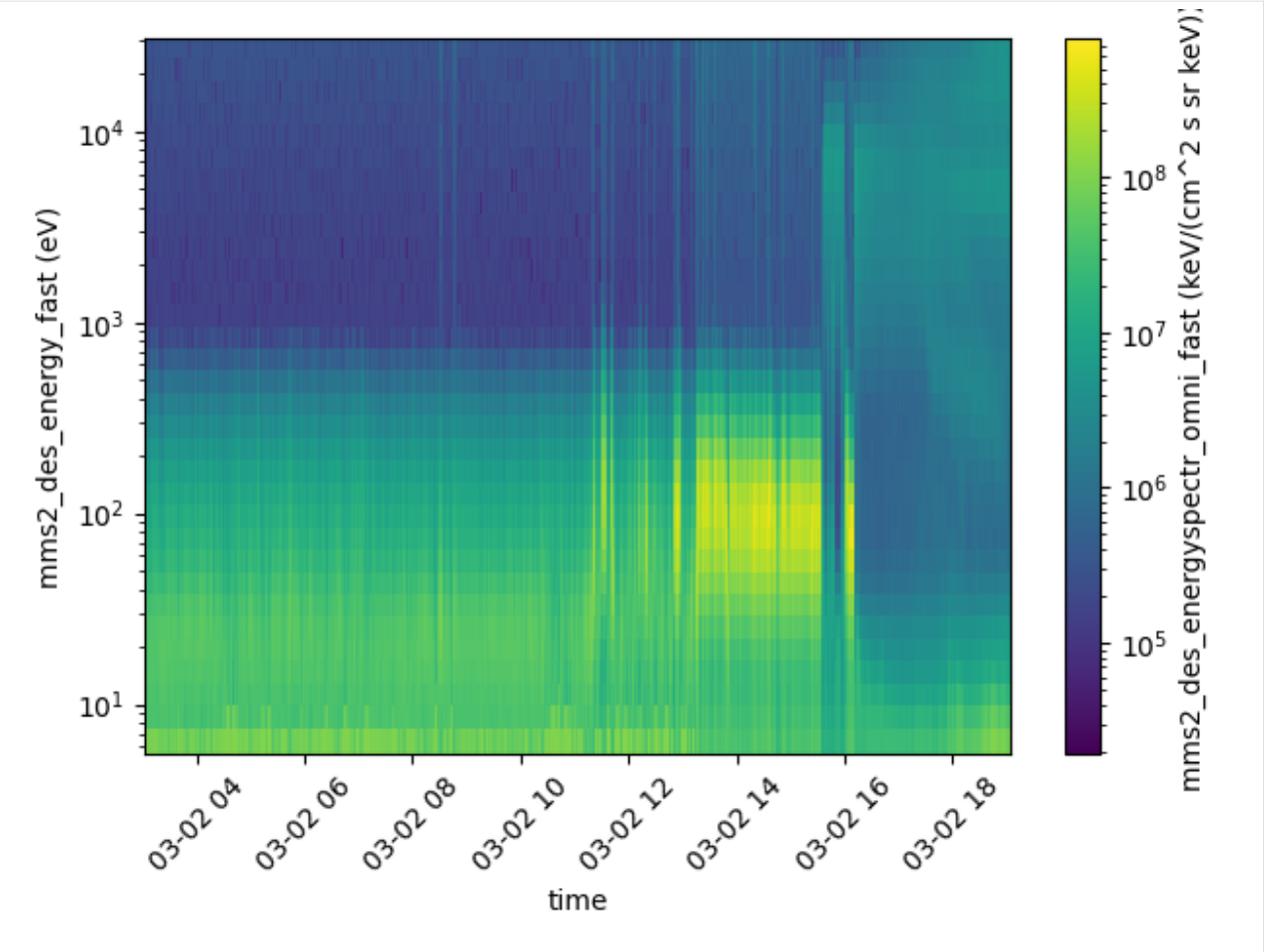
[3]:
```
swa_pas_density: SpeasyVariable = spz.get_data(
    spz.inventories.tree.cda.Solar_Orbiter.SOLO.SWA_PAS_MOM.SOLO_LL02_SWA_PAS_MOM.SWA_
↪PAS_DENSITY,
    "2022-03-01",
    "2022-03-01T12",
)
plt.figure("With 1e31 fill values")
swa_pas_density.plot()
plt.tight_layout()
plt.show()
plt.figure()
swa_pas_density.replace_fillval_by_nan().plot()
plt.tight_layout()
plt.show()
```


1e31
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
SWA_PAS_DENSITY (cm^-3)
Proton Density
03-01 00
03-01 02
03-01 04
03-01 06
03-01 08
03-01 10
03-01 12
time



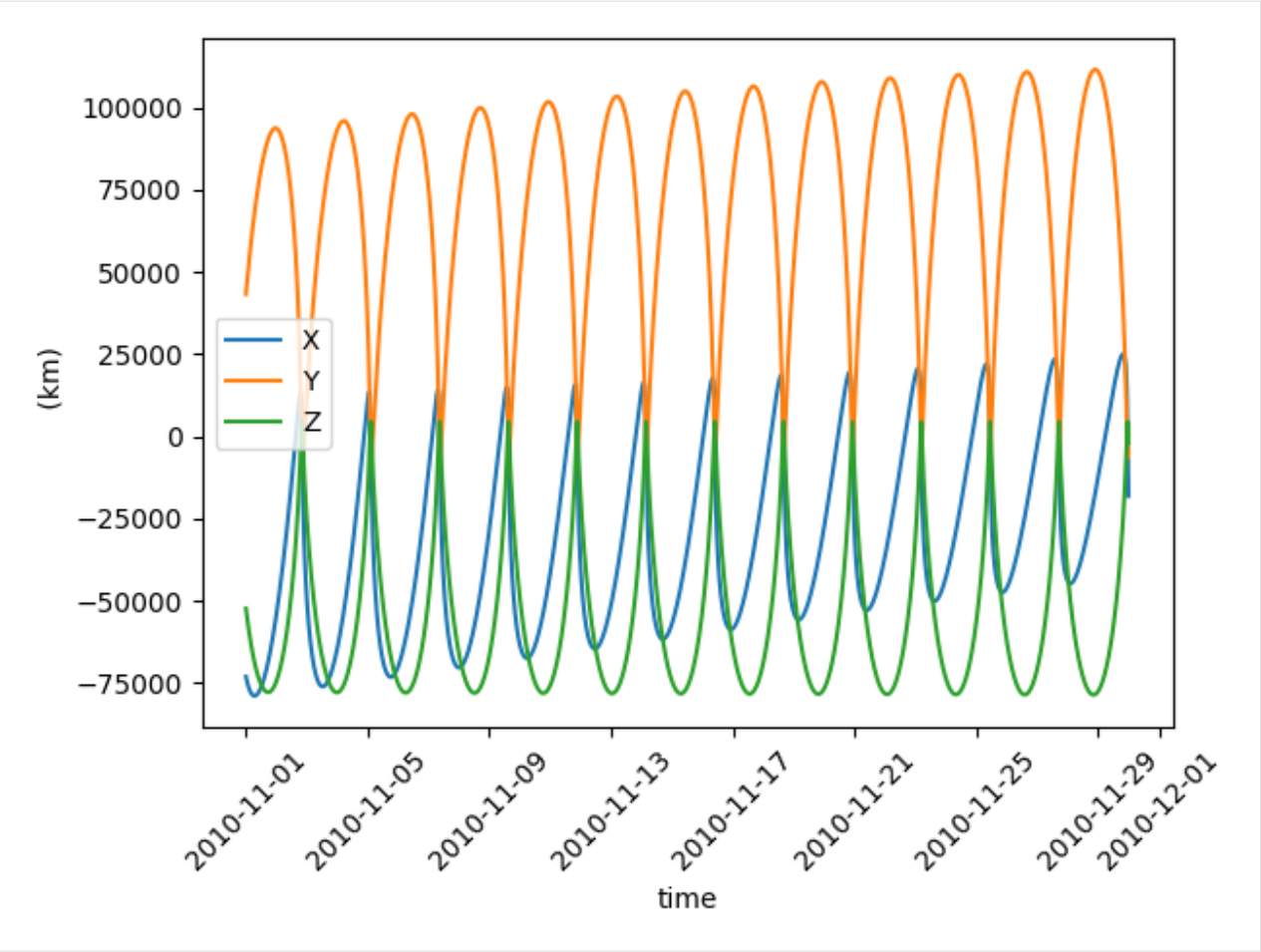
You can also get and plot spectograms

[4]:
```
mms2_des_energyspectr_omni_fast: SpeasyVariable = spz.get_data(
    spz.inventories.tree.cda.MMS.MMS2.DES.MMS2_FPI_FAST_L2_DES_MOMS.mms2_des_
↪energyspectr_omni_fast,
    "2022-03-02",
    "2022-03-03",
)
plt.figure()
mms2_des_energyspectr_omni_fast.plot(cmap="viridis")
plt.tight_layout()
plt.show()
```



4.5.4 SSCWeb Trajectory example

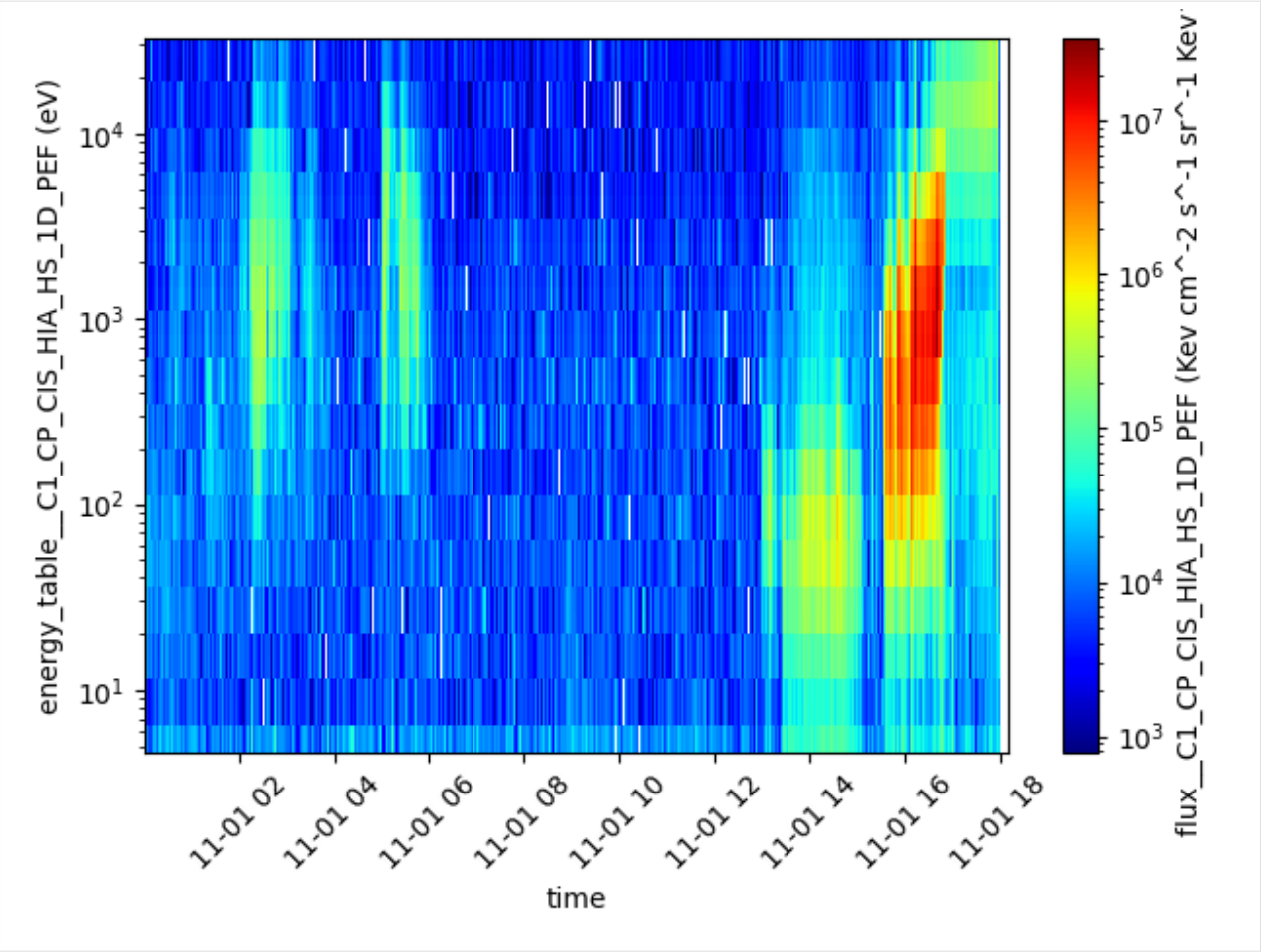
[5]:
```
plt.figure()
spz.get_data(
    spz.inventories.tree.ssc.Trajectories.cluster4, "2010-11-01", "2010-11-30"
).plot()
plt.tight_layout()
plt.show()
```



4.5.5 CSA spectrogram example

Speasy also support the Cluster Science Archive which mean you can get most Cluster 2 and Double Star products

[6]:
```
plt.figure()
spz.get_data(
    spz.inventories.tree.csa.Cluster.Cluster_1.CIS_HIA1.C1_CP_CIS_HIA_HS_1D_PEF.flux__C1_
↪CP_CIS_HIA_HS_1D_PEF,
    "2006-11-01",
    "2006-11-02",
).plot(cmap="jet")
plt.tight_layout()
plt.show()
```



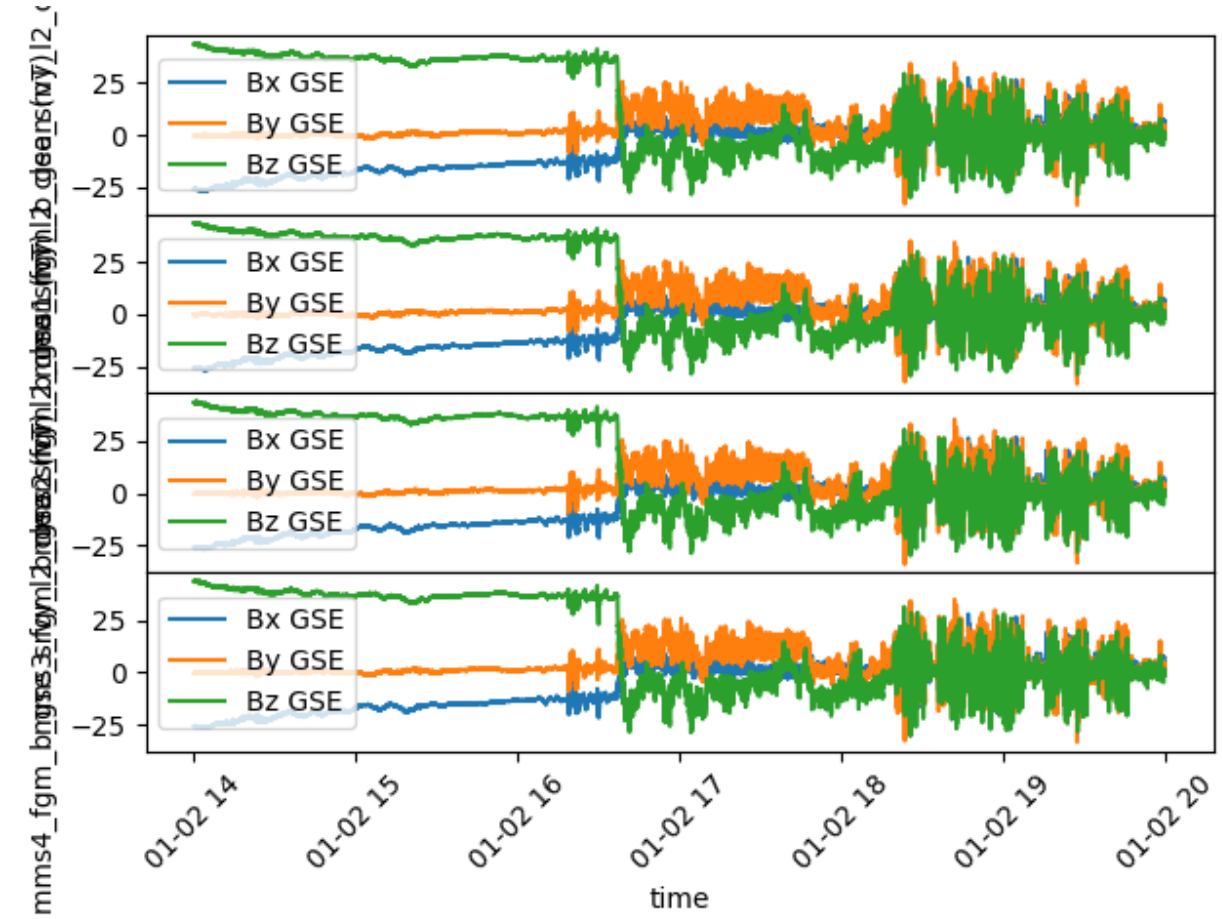
4.5.6 Speasy can download several products at once for a given interval

Speasy `get_data` function is quite flexible and can download several products at once

[7]:
```
products = [
    spz.inventories.tree.cda.MMS.MMS1.FGM.MMS1_FGM_SRVY_L2.mms1_fgm_b_gse_srvy_l2_clean,
    spz.inventories.tree.cda.MMS.MMS2.FGM.MMS2_FGM_SRVY_L2.mms2_fgm_b_gse_srvy_l2_clean,
    spz.inventories.tree.cda.MMS.MMS3.FGM.MMS3_FGM_SRVY_L2.mms3_fgm_b_gse_srvy_l2_clean,
    spz.inventories.tree.cda.MMS.MMS4.FGM.MMS4_FGM_SRVY_L2.mms4_fgm_b_gse_srvy_l2_clean,
]

#fig = plt.figure(figsize=(20, 8))
fig = plt.figure()
gs = fig.add_gridspec(4, hspace=0)
axes = gs.subplots(sharex=True, sharey=True)
mms_fgm_b_gse_srvy: List[SpeasyVariable] = spz.get_data(
    products, "2019-01-02T14", "2019-01-02T20")
for var, ax in zip(mms_fgm_b_gse_srvy, axes):
    var["Bx GSE", "By GSE", "Bz GSE"].replace_fillval_by_nan().plot(ax=ax)
plt.tight_layout()
plt.show()
```

(continues on next page)

(continued from previous page)



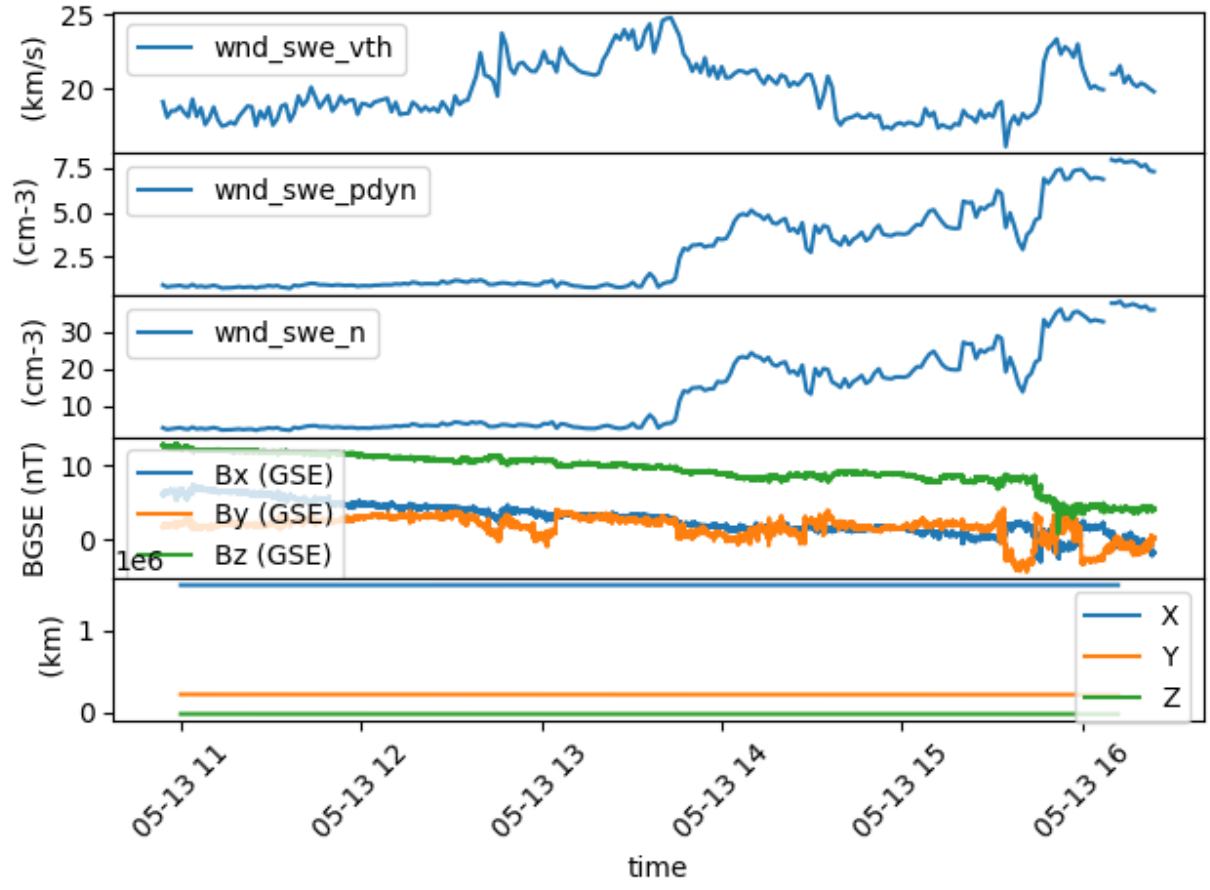
4.5.7 Several product for several dates is also supported

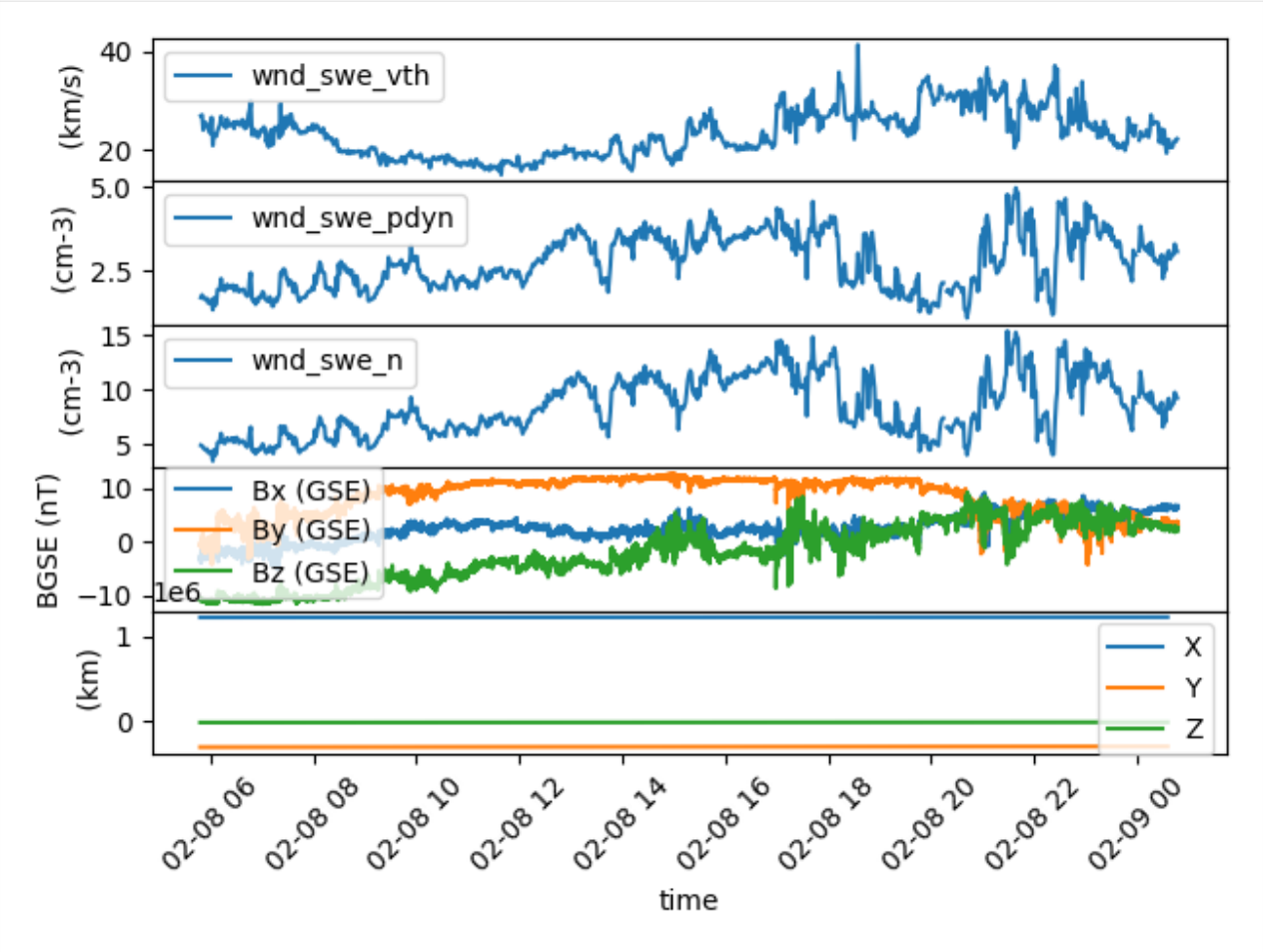
[8]:
```
products = [
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_vth,
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_pdyn,
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_n,
    spz.inventories.tree.cda.Wind.WIND.MFI.WI_H2_MFI.BGSE,
    spz.inventories.tree.ssc.Trajectories.wind,
]

data_several_dates: List[List[SpeasyVariable]] = spz.get_data(
    products,
    spz.inventories.tree.amda.TimeTables.SharedTimeTables.SOLAR_WIND.Magnetic_Clouds
)

for i in range(5):
    #fig = plt.figure(figsize=(20, 6))
    fig = plt.figure()
```

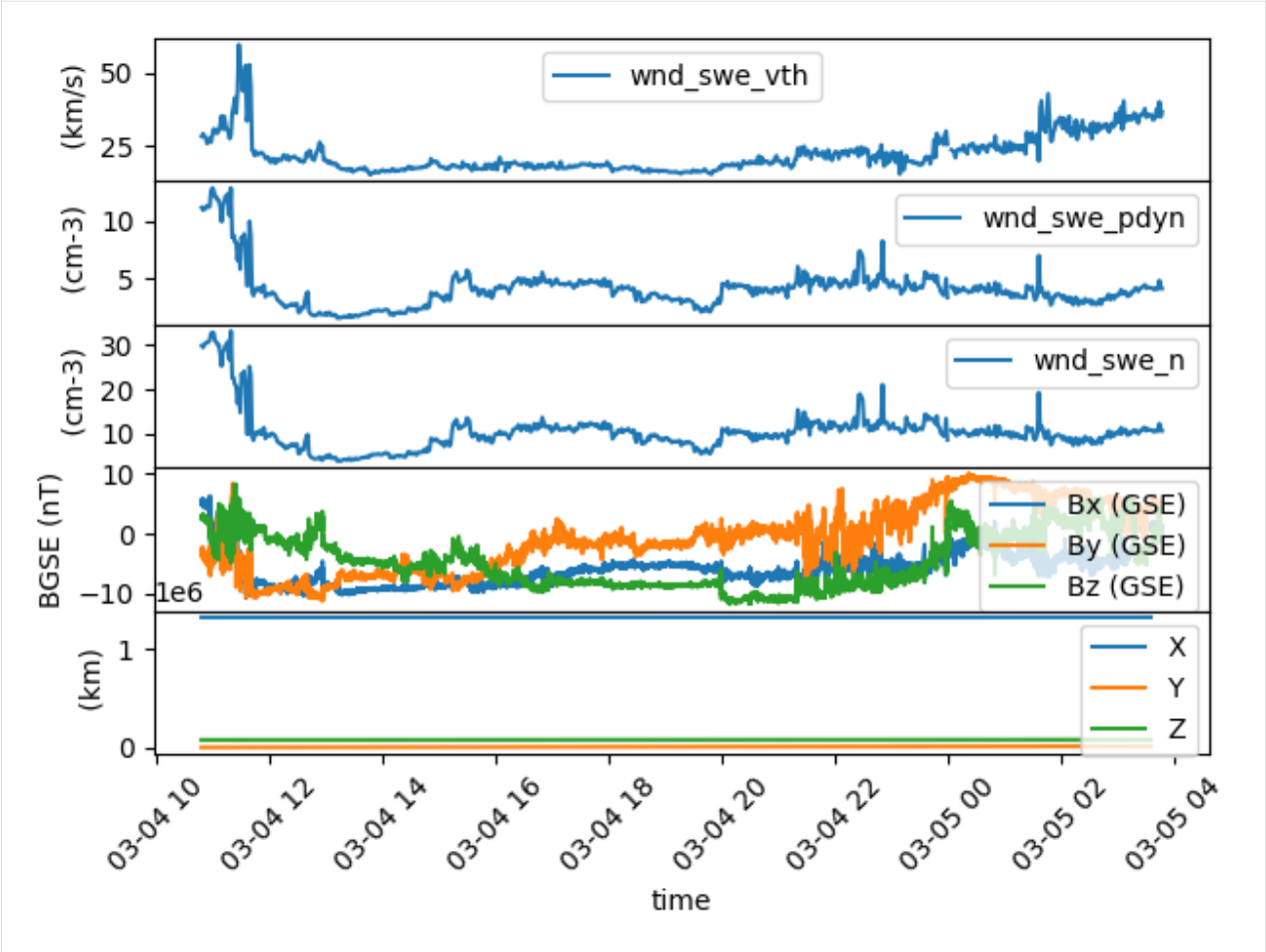
(continues on next page)

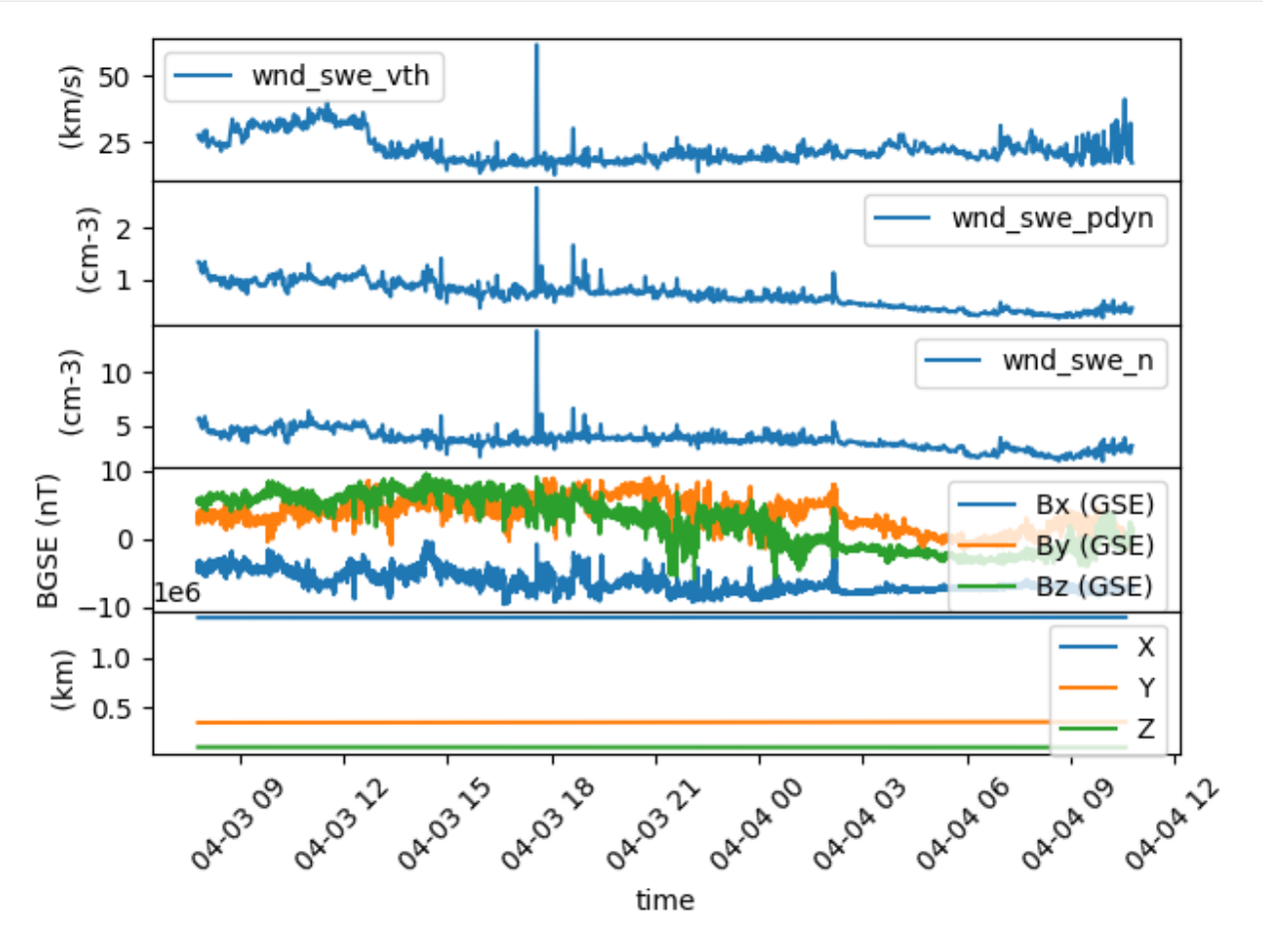
(continued from previous page)

```
    gs = fig.add_gridspec(5, hspace=0)
    axes = gs.subplots(sharex=True, sharey=False)
    for j in range(5):
        data_several_dates[j][i].plot(ax=axes[j])
    plt.tight_layout()
plt.show()
```




wnd_swe_vth
(km/s)
40
20
wnd_swe_pdyn
(cm-3)
5.0
2.5
wnd_swe_n
(cm-3)
15
10
5
Bx (GSE)
By (GSE)
Bz (GSE)
BGSE (nT)
10
0
−10
1e6
(km)
1
0
X
Y
Z
02-08 06
02-08 08
02-08 10
02-08 12
02-08 14
02-08 16
02-08 18
02-08 20
02-08 22
02-09 00
time


wnd_swe_vth
(km/s)
50
25
wnd_swe_pdyn
(cm-3)
10
5
wnd_swe_n
(cm-3)
30
20
10
BGSE (nT)
10
0
−10
1e6
Bx (GSE)
By (GSE)
Bz (GSE)
(km)
1
0
X
Y
Z
03-04 10
03-04 12
03-04 14
03-04 16
03-04 18
03-04 20
03-04 22
03-05 00
03-05 02
03-05 04
time


wnd_swe_vth
(km/s)
50
25
wnd_swe_pdyn
(cm-3)
2
1
wnd_swe_n
(cm-3)
10
5
BGSE (nT)
10
0
−10
1e6
Bx (GSE)
By (GSE)
Bz (GSE)
(km)
1.0
0.5
X
Y
Z
04-03 09
04-03 12
04-03 15
04-03 18
04-03 21
04-04 00
04-04 03
04-04 06
04-04 09
04-04 12
time

4.5.8 get_data preserves input product list shape

Speasy `get_data` function support any nesting depth of product and time ranges lists, it will always explore products first then for each product get data for each given time range

[9]:
```
data_preserve_shape: List[List[List[SpeasyVariable]]] = spz.get_data(
    [
        [
            spz.inventories.tree.cda.OMNI_Combined_1AU_IP_Data__Magnetic_and_Solar_
↪Indices.OMNI_1AU_IP_Data.IMF_and_Plasma_data.OMNI_HRO_1MIN.Beta,
            spz.inventories.tree.cda.OMNI_Combined_1AU_IP_Data__Magnetic_and_Solar_
↪Indices.OMNI_1AU_IP_Data.IMF_and_Plasma_data.OMNI_HRO_1MIN.T,
        ],
        [
            spz.inventories.tree.cda.OMNI_Combined_1AU_IP_Data__Magnetic_and_Solar_
↪Indices.OMNI_1AU_IP_Data.IMF_and_Plasma_data.OMNI_HRO_1MIN.E,
            spz.inventories.tree.cda.OMNI_Combined_1AU_IP_Data__Magnetic_and_Solar_
↪Indices.OMNI_1AU_IP_Data.IMF_and_Plasma_data.OMNI_HRO_1MIN.Pressure,
        ],
    ],
    [["2010-01-02", "2010-01-02T10"], ["2009-08-02", "2009-08-02T10"]]
```

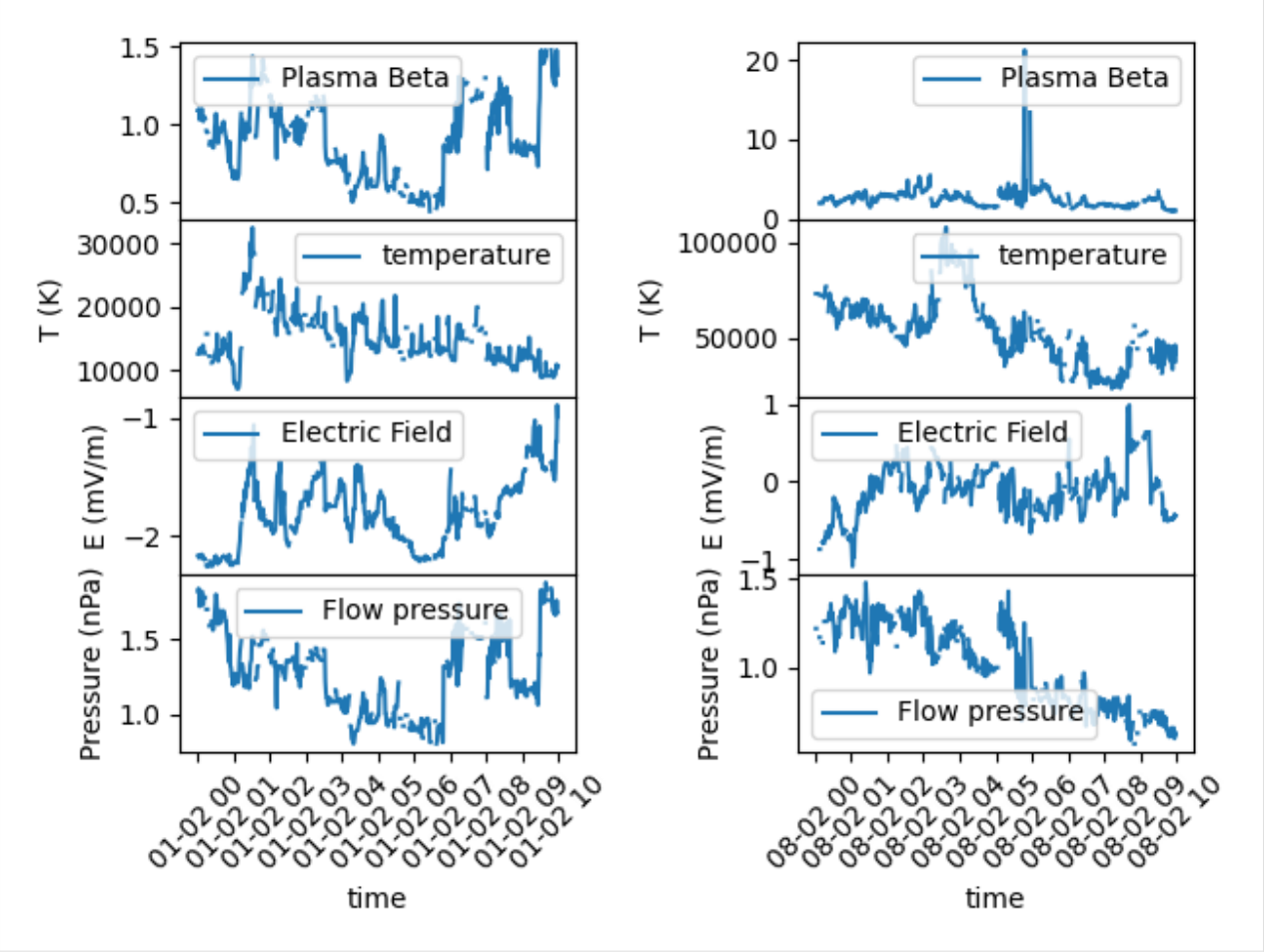
(continues on next page)

(continued from previous page)

```
)
data_preserve_shape
```

[9]:
```
[[[<speasy.products.variable.SpeasyVariable at 0x7fb63bd2d900>,
   <speasy.products.variable.SpeasyVariable at 0x7fb63bd7d440>],
  [<speasy.products.variable.SpeasyVariable at 0x7fb63bdbb940>,
   <speasy.products.variable.SpeasyVariable at 0x7fb63bde0d40>]],
 [[<speasy.products.variable.SpeasyVariable at 0x7fb63bde2080>,
   <speasy.products.variable.SpeasyVariable at 0x7fb63bde3300>],
  [<speasy.products.variable.SpeasyVariable at 0x7fb63bde85c0>,
   <speasy.products.variable.SpeasyVariable at 0x7fb63bde9780>]]]
```

[10]:
```
#fig = plt.figure(figsize=(20, 6))
fig = plt.figure()
gs = fig.add_gridspec(1, 2)
left_gs = gs[0, 0].subgridspec(4, 1, hspace=0)
right_gs = gs[0, 1].subgridspec(4, 1, hspace=0)
left_ax = left_gs.subplots(sharex=True, sharey=False)
right_ax = right_gs.subplots(sharex=True, sharey=False)

data_preserve_shape[0][0][0].replace_fillval_by_nan().plot(ax=left_ax[0])
data_preserve_shape[0][1][0].replace_fillval_by_nan().plot(ax=left_ax[1])
data_preserve_shape[1][0][0].replace_fillval_by_nan().plot(ax=left_ax[2])
data_preserve_shape[1][1][0].replace_fillval_by_nan().plot(ax=left_ax[3])

data_preserve_shape[0][0][1].replace_fillval_by_nan().plot(ax=right_ax[0])
data_preserve_shape[0][1][1].replace_fillval_by_nan().plot(ax=right_ax[1])
data_preserve_shape[1][0][1].replace_fillval_by_nan().plot(ax=right_ax[2])
data_preserve_shape[1][1][1].replace_fillval_by_nan().plot(ax=right_ax[3])
plt.tight_layout()
plt.show()
```



4.5.9 Some SpeasyVariable tricks

Attributes

A SpeasyVariable is close to a pandas DataFrame, it has few attributes, most of them are extracted from underlying CDF or CSV files generated by corresponding web-services:

[11]:
```
solo_fgm: SpeasyVariable = spz.get_data(
    spz.inventories.tree.cda.Solar_Orbiter.SOLO.MAG.SOLO_L2_MAG_RTN_NORMAL.B_RTN,
    "2022-03-01",
    "2022-03-01T12",
)
print("=============================================")
print(f"Name:          {solo_fgm.name}")
print(f"Columns:       {solo_fgm.columns}")
print(f"Values Unit:   {solo_fgm.unit}")
print(f"Memory usage:  {solo_fgm.nbytes} Bytes")
print(f"Axes Labels:   {solo_fgm.axes_labels}")
print("---------------------------------------------")
print(f"Meta-data:     {solo_fgm.meta}")
print("---------------------------------------------")
```

(continues on next page)

(continued from previous page)

```
print(f"Time Axis:    {solo_fgm.time[:3]}")
print("-------------------------------------------")
print(f"Values:       {solo_fgm.values[:3]}")
print("===========================================")
```

```
===========================================
Name:         B_RTN
Columns:      ['B_r', 'B_t', 'B_n']
Values Unit:  nT
Memory usage: 11059947 Bytes
Axes Labels:  ['time']
-------------------------------------------
Meta-data:    {'DIM_SIZES': [3], 'VALIDMIN': [-10000000000.0], 'UNITS': 'nT', 'TENSOR_
↪ORDER': '1', 'SI_CONVERSION': '1.0E-9>T', 'FIELDNAM': 'B RTN', 'VAR_TYPE': 'data',
↪'VALIDMAX': [10000000000.0], 'CATDESC': 'Magnetic field vector in RTN coordinates',
↪'SCALETYP': 'linear', 'COORDINATE_SYSTEM': 'RTN', 'SCALEMIN': [-80000.0], 'DEPEND_0':
↪'EPOCH', 'DETECTOR': 'PRI>Primary Sensor', 'LABL_PTR_1': ['B_r', 'B_t', 'B_n'],
↪'DISPLAY_TYPE': 'time_series', 'REPRESENTATION_1': 'REP1_B_RTN', 'FILLVAL': [nan],
↪'FORMAT': 'f11.4', 'SCALEMAX': [80000.0]}
-------------------------------------------
Time Axis:    ['2022-03-01T00:00:00.002997368' '2022-03-01T00:00:00.128001719'
 '2022-03-01T00:00:00.252996012']
-------------------------------------------
Values:       [[-1.60646439e-01 -6.34788084e+00  8.59271908e+00]
 [-9.50906351e-02 -5.76544285e+00  8.89126110e+00]
 [ 5.36612468e-03 -5.06133699e+00  9.56754112e+00]]
===========================================
```

Methods

SpeasyVaraible class is not meant to be as fully-featured as pandas DataFrame or Xarray DataArray but still has few useful features

- you can make a (deep)copy of a SpeasyVariable:

[12]:
```
solo_fgm2 = solo_fgm.copy()
solo_fgm2 is solo_fgm, solo_fgm == solo_fgm2
```

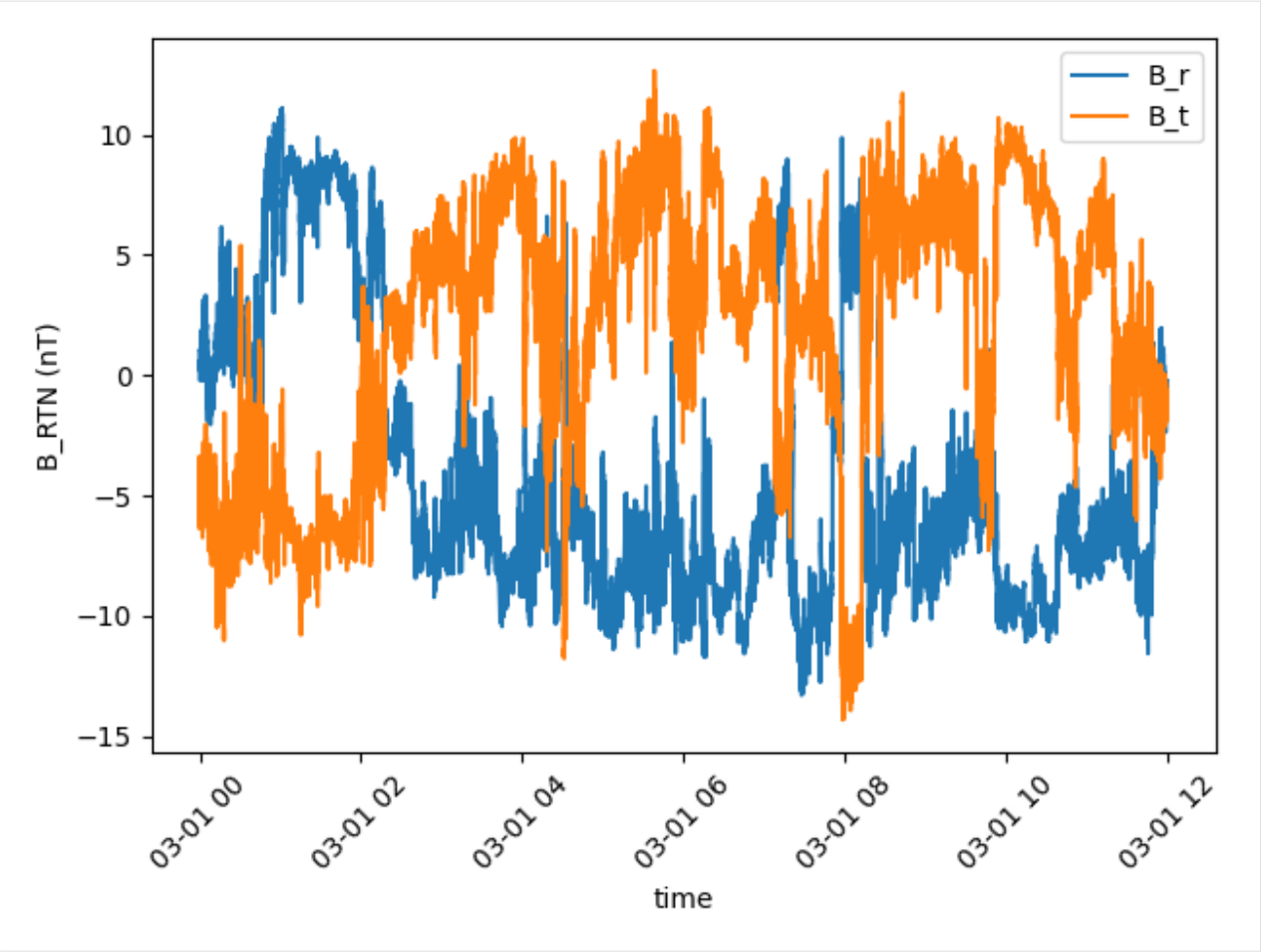
[12]:
```
(False, True)
```

- you can build a new variable with only a subset of columns of a given variable

[13]:
```
solo_fgm2 = solo_fgm.filter_columns(["B_r", "B_t"])
print(solo_fgm.columns, solo_fgm2.columns)
```

```
['B_r', 'B_t', 'B_n'] ['B_r', 'B_t']
```

[14]:
```
plt.figure("Variable with filtered columns")
solo_fgm.filter_columns(["B_r", "B_t"]).plot()
plt.tight_layout()
plt.show()
```



- you can export simple time-series to Astropy Tables or pandas DataFrames

[15]:
```
solo_fgm.to_astropy_table()[:10].show_in_notebook()
```

[15]:
```
<IPython.core.display.HTML object>
```

[17]:
```
solo_fgm.to_dataframe().head()
```

[17]:
```
                                 B_r       B_t        B_n
2022-03-01 00:00:00.002997368 -0.160646 -6.347881   8.592719
2022-03-01 00:00:00.128001719 -0.095091 -5.765443   8.891261
2022-03-01 00:00:00.252996012  0.005366 -5.061337   9.567541
2022-03-01 00:00:00.378000363  0.158163 -4.390467   9.742784
2022-03-01 00:00:00.503004714  0.322980 -3.519104  10.105829
```

- you can slice a SpeasyVariable by index or time

[18]:
```
solo_fgm[:5].to_dataframe()
```

[18]:
```
                                 B_r       B_t        B_n
2022-03-01 00:00:00.002997368 -0.160646 -6.347881   8.592719
2022-03-01 00:00:00.128001719 -0.095091 -5.765443   8.891261
2022-03-01 00:00:00.252996012  0.005366 -5.061337   9.567541
2022-03-01 00:00:00.378000363  0.158163 -4.390467   9.742784
```

(continues on next page)

(continued from previous page)

```
2022-03-01 00:00:00.503004714  0.322980 -3.519104  10.105829
```

[19]:
```
solo_fgm[np.datetime64("2022-03-01T00:00:00"): np.datetime64("2022-03-01T00:00:00.6")].
↪to_dataframe()
```

[19]:
```
                                B_r       B_t        B_n
2022-03-01 00:00:00.002997368 -0.160646 -6.347881   8.592719
2022-03-01 00:00:00.128001719 -0.095091 -5.765443   8.891261
2022-03-01 00:00:00.252996012  0.005366 -5.061337   9.567541
2022-03-01 00:00:00.378000363  0.158163 -4.390467   9.742784
2022-03-01 00:00:00.503004714  0.322980 -3.519104  10.105829
```

[20]:
```
solo_fgm[datetime(2022,3,1): datetime(2022,3,1,microsecond=600000)].to_dataframe()
```

[20]:
```
                                B_r       B_t        B_n
2022-03-01 00:00:00.002997368 -0.160646 -6.347881   8.592719
2022-03-01 00:00:00.128001719 -0.095091 -5.765443   8.891261
2022-03-01 00:00:00.252996012  0.005366 -5.061337   9.567541
2022-03-01 00:00:00.378000363  0.158163 -4.390467   9.742784
2022-03-01 00:00:00.503004714  0.322980 -3.519104  10.105829
```

- you can convert values into Astropy Quantity:

[21]:
```
solo_fgm[:5].unit_applied().values
```

[21]:
```
[[−0.16064644, −6.3478808, 8.5927191], [−0.095090635, −5.7654428, 8.8912611], [0.0053661247, −5.061337, 9.5675411], [0.15
```

The following section was generated from `docs/examples/solo_epd.ipynb`

4.6 Solar Orbiter HET data

4.6.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output
    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.6.2 For all users:

[ ]:
```
import speasy as spz
%matplotlib widget
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook

import matplotlib
import matplotlib.pyplot as plt

import pandas as pd
import numpy as np
```

[3]:
```
start = "2020-11-28T00:00:00"
stop = "2020-12-05T00:00:00"
```

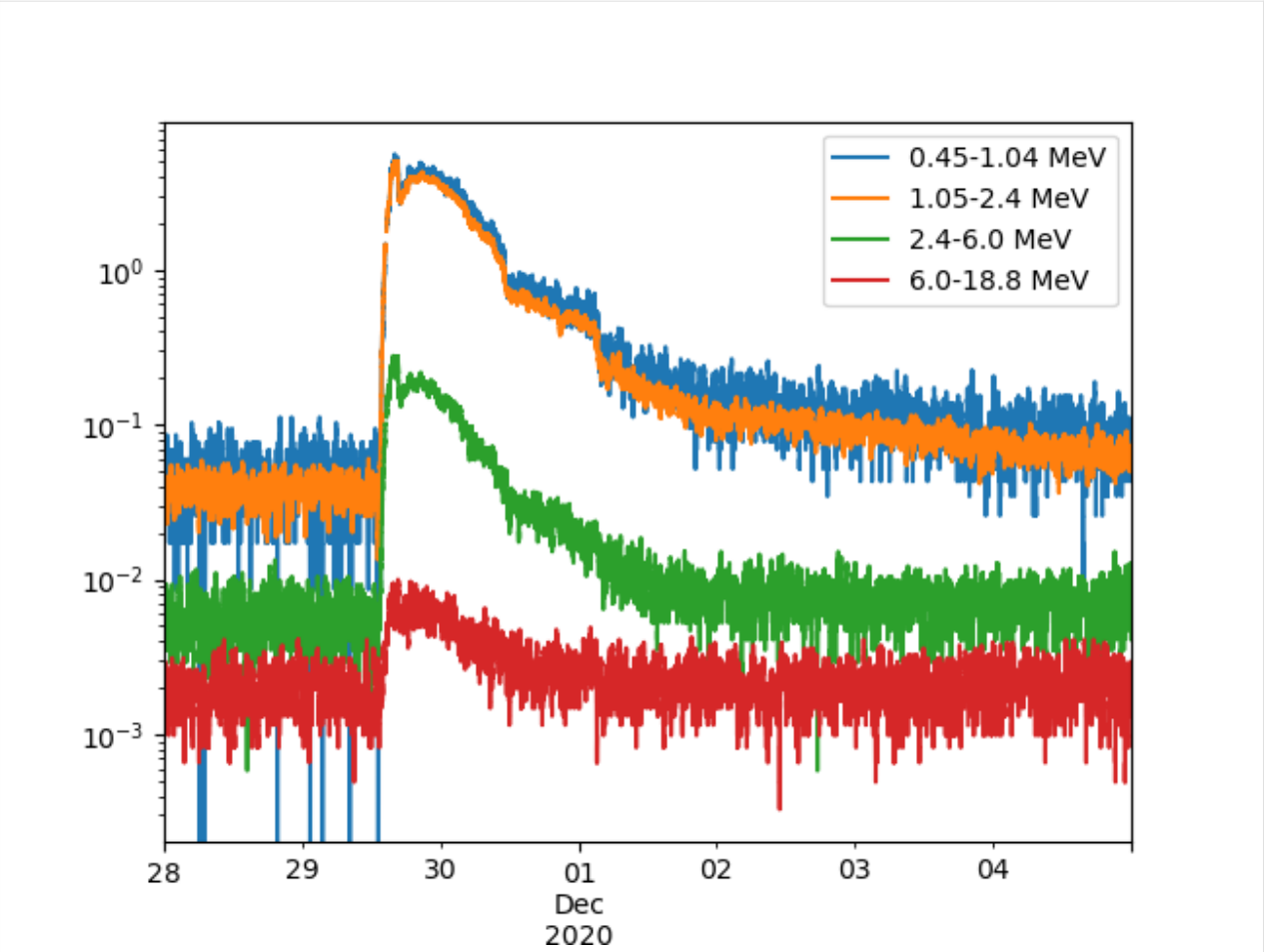
4.6.3 Solar Orbiter electron flux

[7]:
```
solo_eflux:spz.SpeasyVariable = spz.get_data("amda/solo_het_omni_eflux", start, stop)
```

[8]:
```
print(np.unique(np.diff(solo_eflux.time), return_counts=True))
```

```
(array([1000000000, 1001000000], dtype='timedelta64[ns]'), array([603798,   1000]))
```

[9]:
```
eflux_df = solo_eflux.to_dataframe()
# resample to 1s
eflux_df = eflux_df.resample("1s").ffill()
eflux_df.rolling(600).mean().plot()
plt.yscale("log")
```



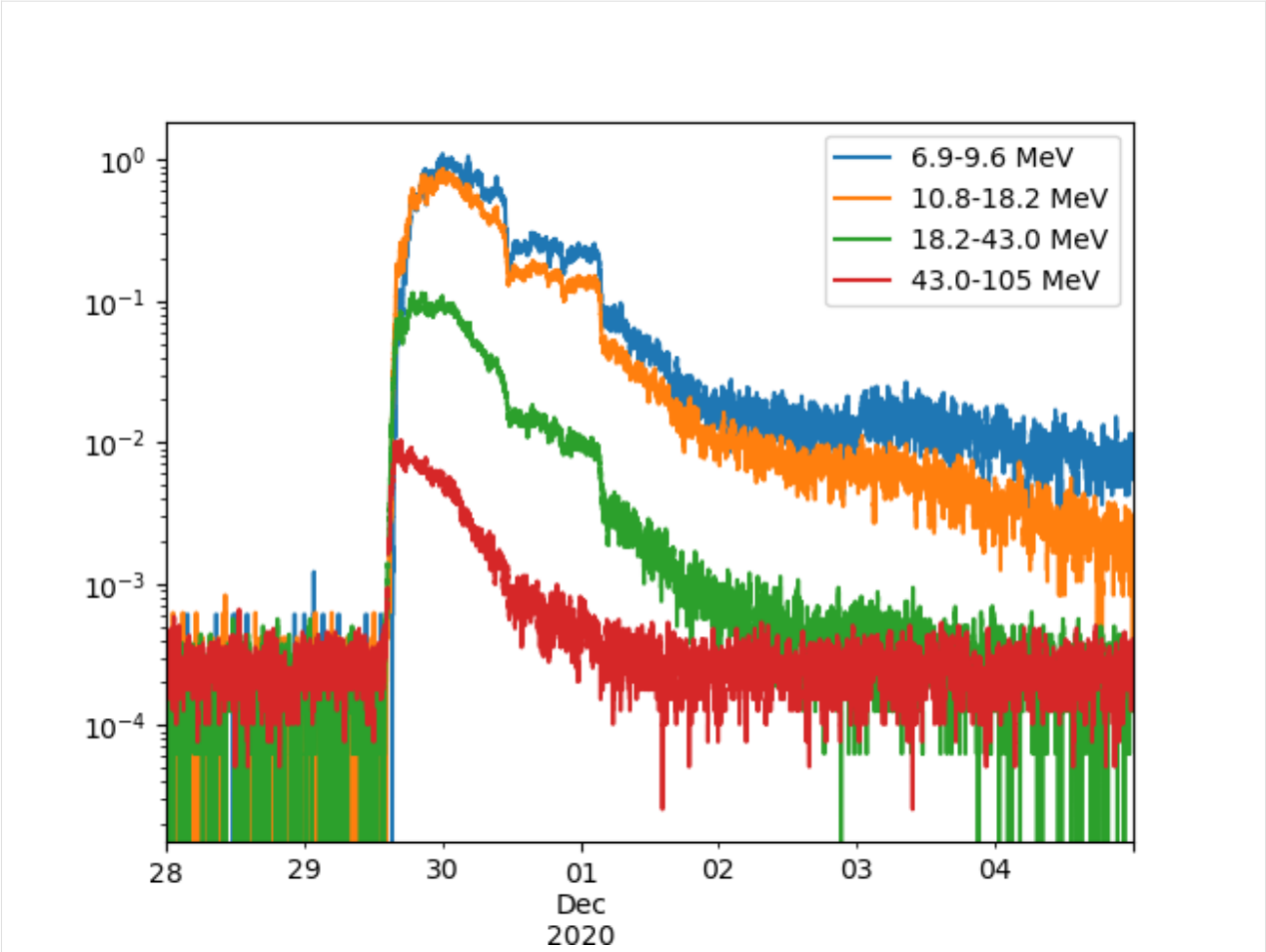
4.6.4 Solar Orbiter proton flux

[14]:
```
solo_hflux:spz.SpeasyVariable = spz.get_data("amda/solo_het_omni_hhcflux", start, stop)
```

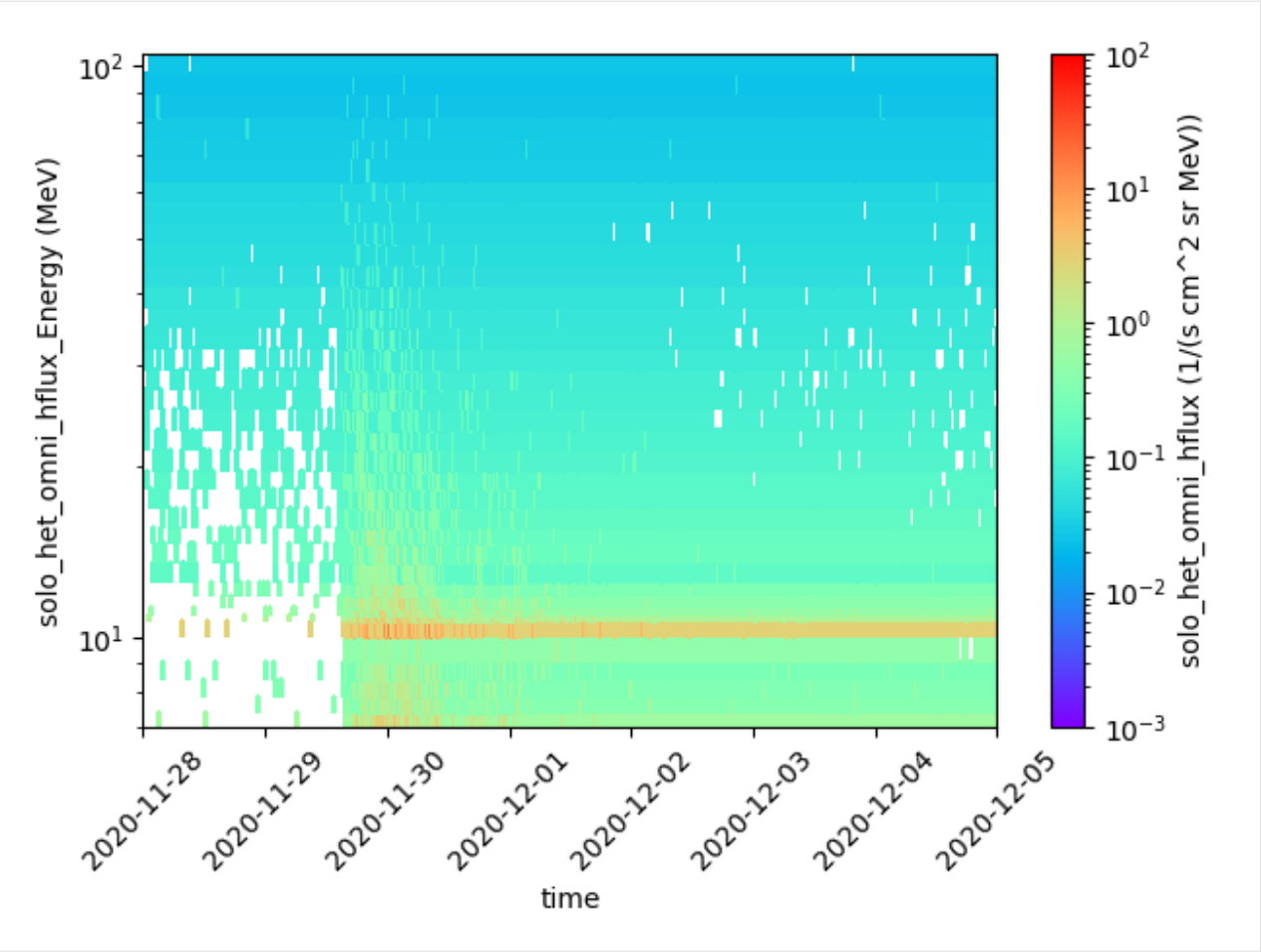
[15]:
```
print(np.unique(np.diff(solo_eflux.time), return_counts=True))
```

```
(array([1000000000, 1001000000], dtype='timedelta64[ns]'), array([603798,   1000]))
```

[16]:
```
hflux_df = solo_hflux.to_dataframe()
# resample to 1s
hflux_df = hflux_df.resample("1s").ffill()
hflux_df.rolling(600).mean().plot()
plt.yscale("log")
```



[17]:
```
plt.figure()
pas_omni:spz.SpeasyVariable = spz.get_data("amda/solo_het_omni_hflux", start, stop)
pas_omni.plot(cmap='rainbow', vmin=1e-3, vmax=1e2, edgecolors="face")
plt.tight_layout()
```



The following section was generated from `docs/examples/alfvenic.ipynb`

4.7 Multiscale views of an Alfvenic slow solar wind:

4.7.1 3D velocity distribution functions observed by the Proton-Alpha Sensor of Solar Orbiter

4.7.2 Louarn et al., 2021

4.7.3 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output
    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.7.4 For all users:

[ ]:
```
import speasy as spz
%matplotlib widget
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook

import matplotlib
import matplotlib.pyplot as plt

import pandas as pd
import numpy as np
from datetime import datetime, timedelta
```

Define the observation dates

[2]:
```
start = "2020-07-14T10:00:00"
stop  = "2020-07-15T06:00:00"
```

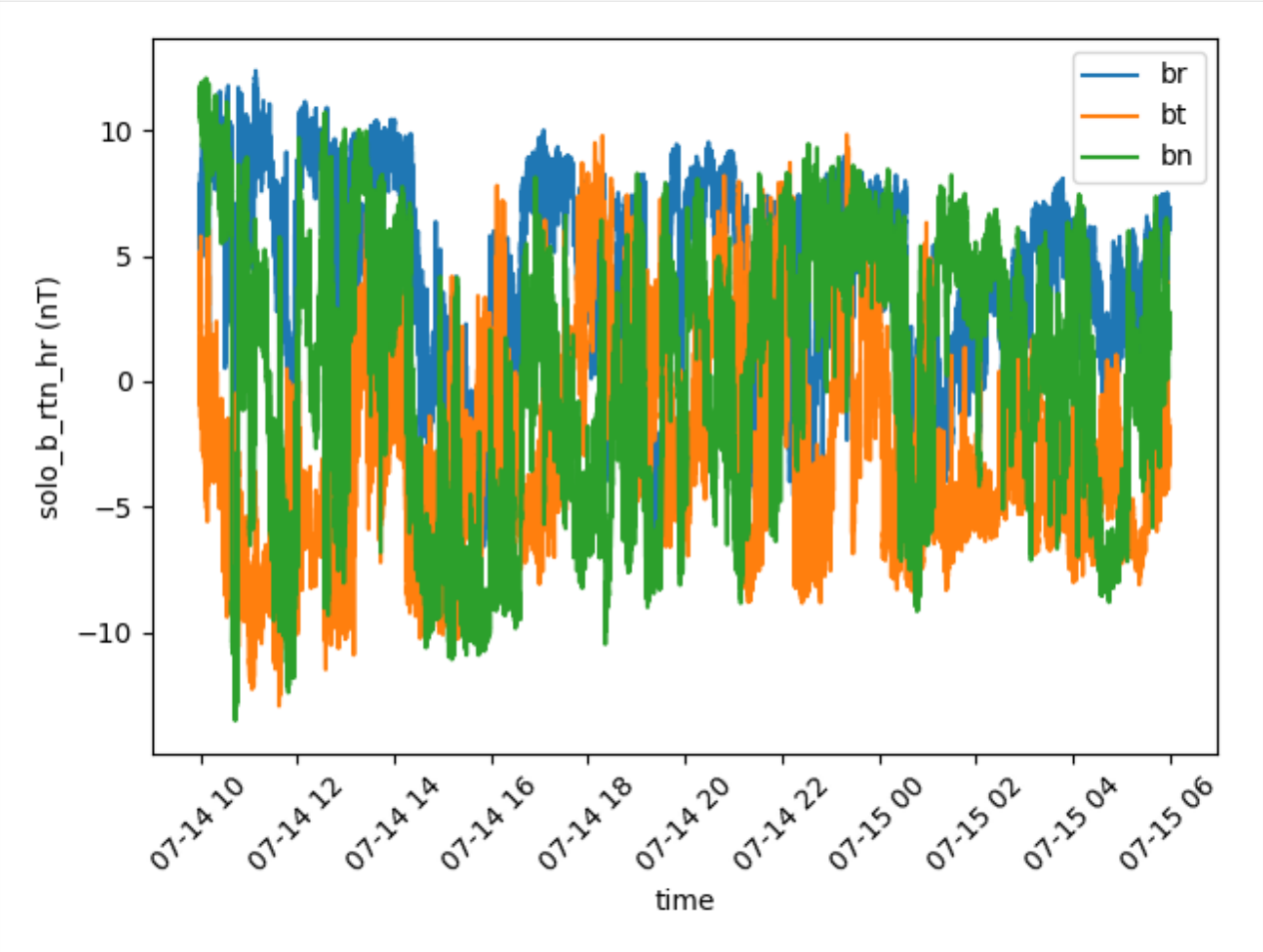
4.7.5 Magnetic field data

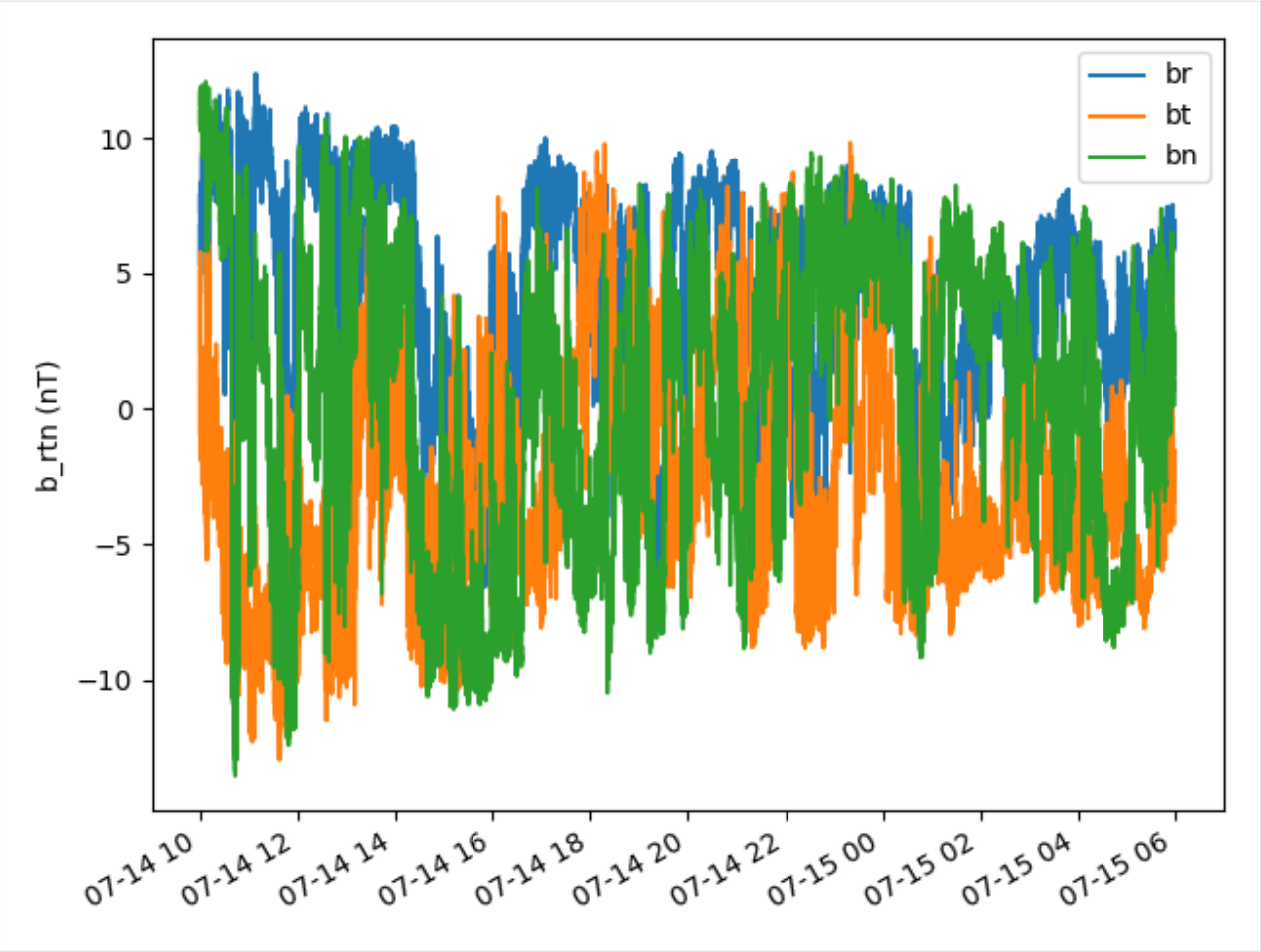
Magnetic field measurements are stored under the `amda/solo_b_rtn_hr` parameter. We save the `SpeasyVariable` object as `b_rtn_hr` for later use.

[10]:
```
b_rtn_hr:spz.SpeasyVariable = spz.get_data("amda/solo_b_rtn_hr", start, stop)
```

We can easily check the data by using the `SpeasyVariable.plot` method.

[5]:
```
plt.figure()
b_rtn_hr.plot()
plt.tight_layout()
# equivalent to
b_rtn_hr.to_dataframe().plot(ylabel=f"b_rtn ({b_rtn_hr.unit})")
plt.tight_layout()
```


br
bt
bn
10
5
0
−5
−10
solo_b_rtn_hr (nT)
07-14 10
07-14 12
07-14 14
07-14 16
07-14 18
07-14 20
07-14 22
07-15 00
07-15 02
07-15 04
07-15 06
time



Transform the variable to a `pandas.DataFrame` object using the `to_dataframe` method. `to_dataframe` accepts a `datetime_index` argument (default:`False`) indicating if time values should be converted to `datetime` objects.

Let's store the magnetic field in a dataframe called `b_rtn_df`.

[6]:
```
b_rtn_df = b_rtn_hr.to_dataframe()
b_rtn_df.describe()
```

[6]:

	br	bt	bn
count	576009.000000	576009.000000	576009.000000
mean	4.436778	-3.022162	0.313650
std	3.676364	4.013111	5.414090
min	-7.779764	-12.940836	-13.529315
25%	1.998192	-5.914904	-4.331481
50%	4.978280	-3.851054	1.152826
75%	7.296504	-0.396841	4.847765
max	12.380941	9.837873	12.080406

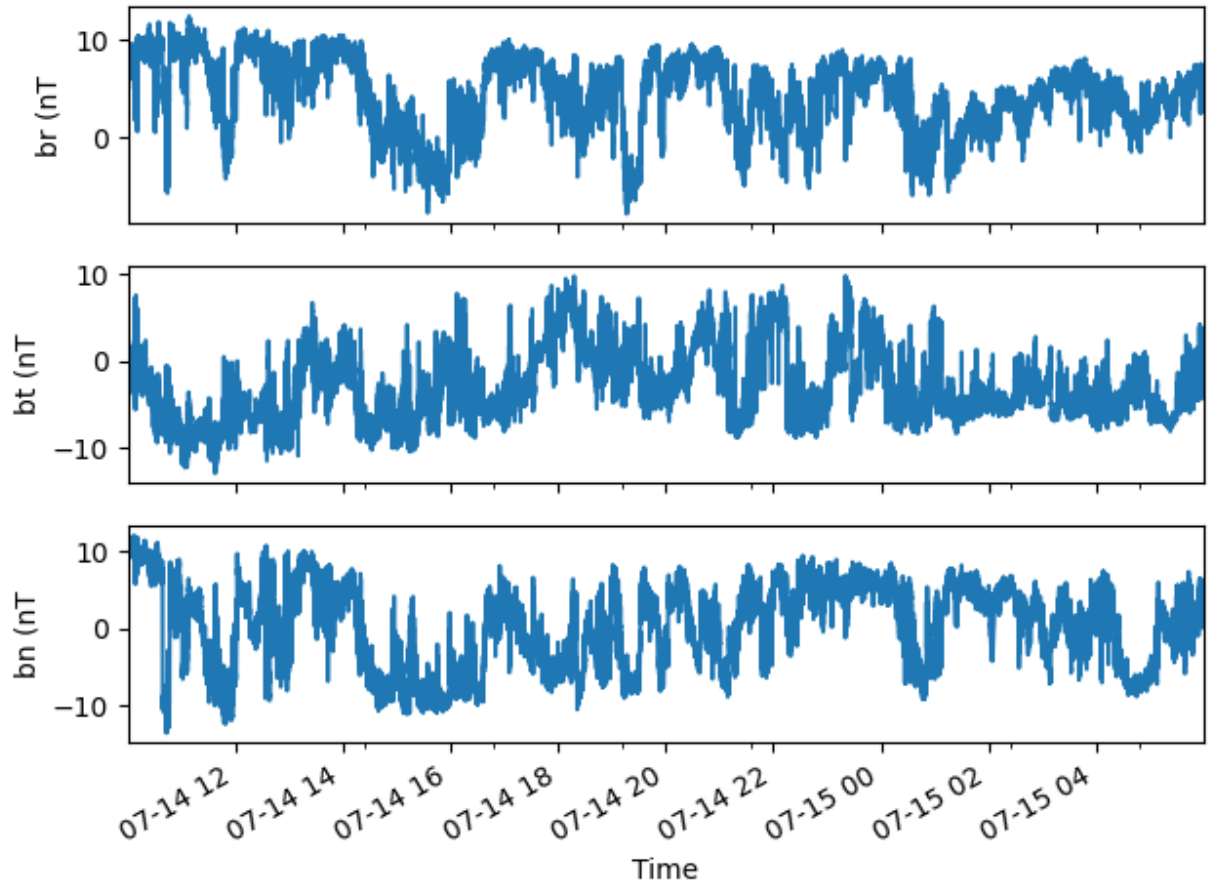
And make the figure a bit more pleasant.

[7]:
```
titles = b_rtn_hr.columns
units = b_rtn_hr.unit
fig, ax = plt.subplots(3,1, sharex=True)
for i,name in enumerate(b_rtn_df.columns):
```

(continues on next page)

(continued from previous page)

```
    b_rtn_df[name].plot(ax=ax[i])
    ax[i].set_ylabel(f"{titles[i]} ({units}")
plt.xlim([b_rtn_df.index[0],b_rtn_df.index[-1]])
ax[2].set_xlabel("Time")
plt.tight_layout()
```



4.7.6 Velocity data

PAS proton velocity data is available on AMDA and its identifier is `pas_momgr1_v_rtn`. We set the data aside under the `v_rtn` variable.

[11]:
```
v_rtn:spz.SpeasyVariable = spz.get_data("amda/pas_momgr1_v_rtn", start, stop)
```

Create a dataframe object.

[12]:
```
v_rtn_df = v_rtn.to_dataframe()
v_rtn_df.describe()
```

[12]:

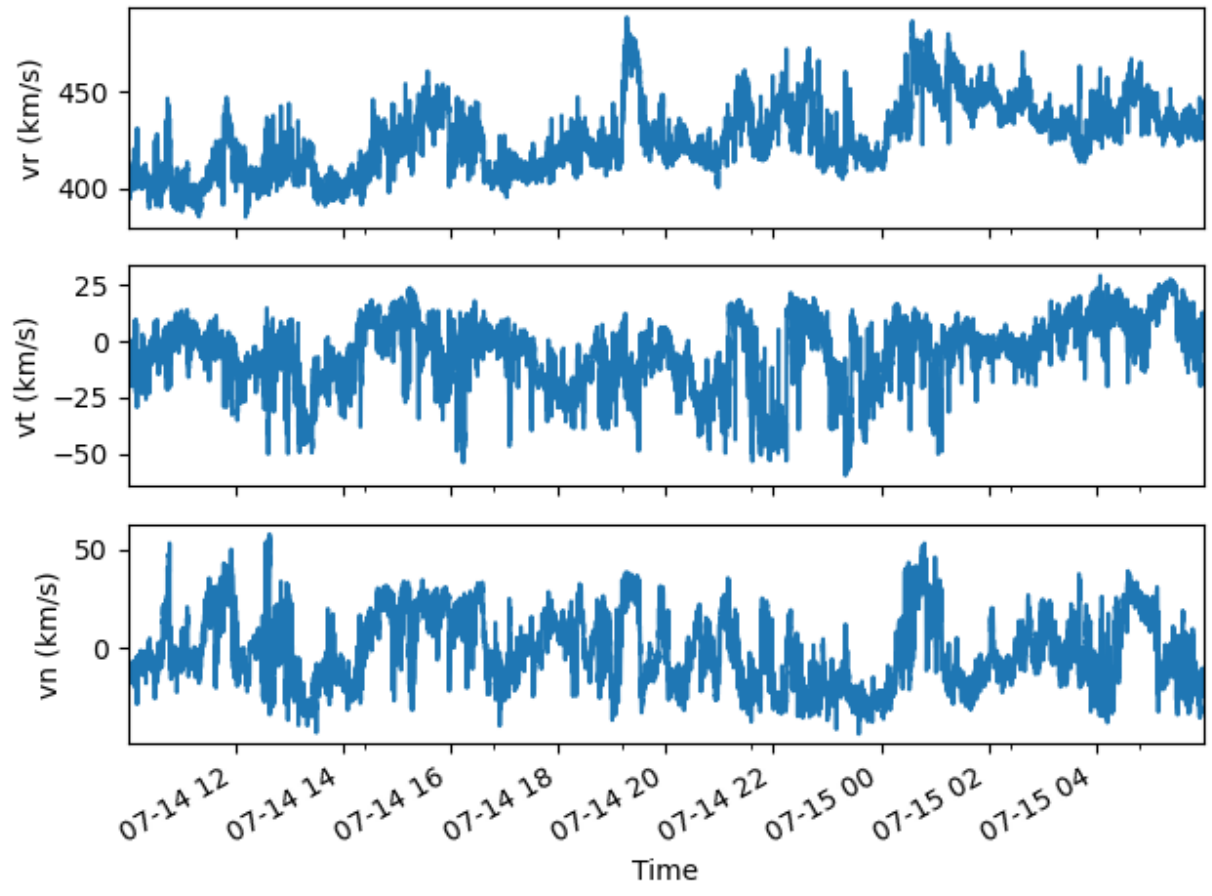
	vr	vt	vn
count	27514.000000	27514.000000	27514.000000
mean	426.249600	-6.334826	0.317864
std	17.954962	15.367386	20.181820

(continues on next page)

(continued from previous page)

```
min      384.966644    -59.670586    -43.584286
25%      412.822838    -16.342590    -16.186169
50%      424.106384     -3.483476     -2.348698
75%      438.325470      4.522001     16.442065
max      488.915253     29.120104     58.001766
```

[13]:
```
titles = v_rtn.columns
units = v_rtn.unit
fig, ax = plt.subplots(3,1, sharex=True)
for i,name in enumerate(v_rtn_df.columns):
    v_rtn_df[name].plot(ax=ax[i])
    ax[i].set_ylabel(f"{titles[i]} ({units})")
plt.xlim([v_rtn_df.index[0],v_rtn_df.index[-1]])
plt.xlabel("Time")
plt.tight_layout()
```



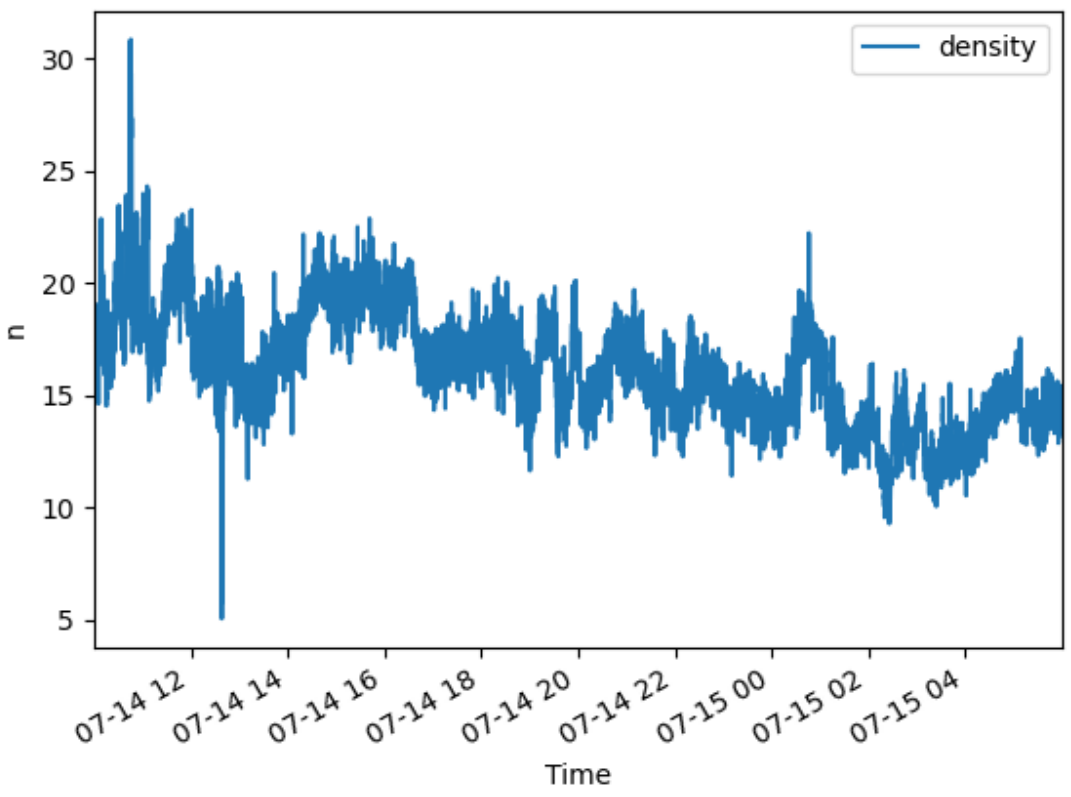
4.7.7 Density

PAS proton density is identified by `pas_momgr_n`.

[14]:
```
sw_n:spz.SpeasyVariable = spz.get_data("amda/pas_momgr_n", start, stop)
sw_n_df = sw_n.to_dataframe()
sw_n_df.describe()
```

[14]:
```
            density
count  27514.000000
mean      16.025642
std        2.405489
min        5.053784
25%       14.319734
50%       15.890954
75%       17.525525
max       30.859621
```

[16]:
```
sw_n_df.plot()
plt.xlim([sw_n_df.index[0], sw_n_df.index[-1]])
plt.ylabel("n")
plt.xlabel("Time")
plt.show()
```



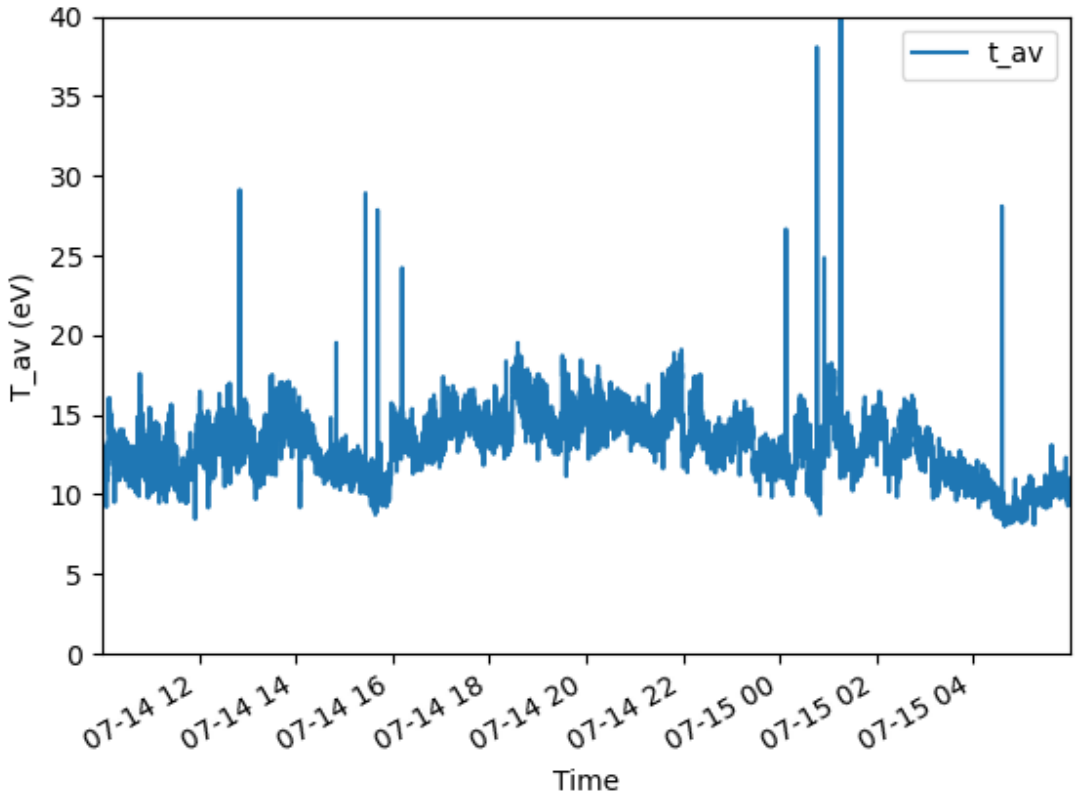
4.7.8 Proton temperature

Proton temperature is identified by `pas_momgr_tav`.

[17]:
```
tav:spz.SpeasyVariable = spz.get_data("amda/pas_momgr_tav", start, stop)
tav_df = tav.to_dataframe()
tav_df.describe()
```

[17]:
```
                t_av
count  27514.000000
mean      12.799194
std        2.060786
min        8.017986
25%       11.346625
50%       12.981331
75%       14.296211
max       52.293873
```

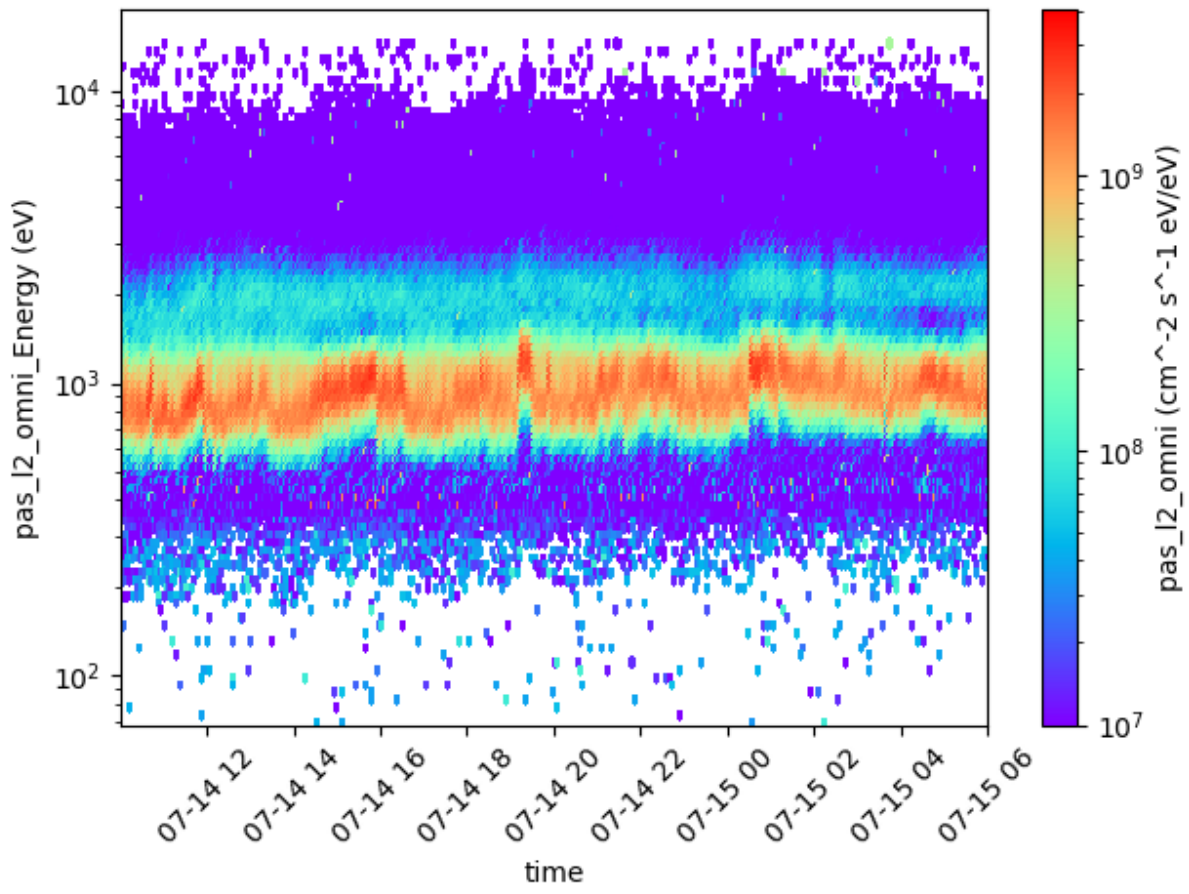
[18]:
```
tav_df.plot()
plt.xlabel("Time")
plt.ylabel("T_av (eV)")
plt.xlim([tav_df.index[0], tav_df.index[-1]])
plt.ylim([0,40])
plt.show()
```



4.7.9 Proton differential energy flux

Proton differential energy flux is `pas_l2_omni`.

[19]:
```
plt.figure()
pas_l2_omni=spz.get_data("amda/pas_l2_omni", start, stop)
pas_l2_omni.plot(cmap='rainbow', edgecolors="face", vmin=1e7)
plt.tight_layout()
```



4.7.10 Correlation of velocity and magnetic field fluctuations

Study of the correlation between `v_rtn` and `b_rtn`.

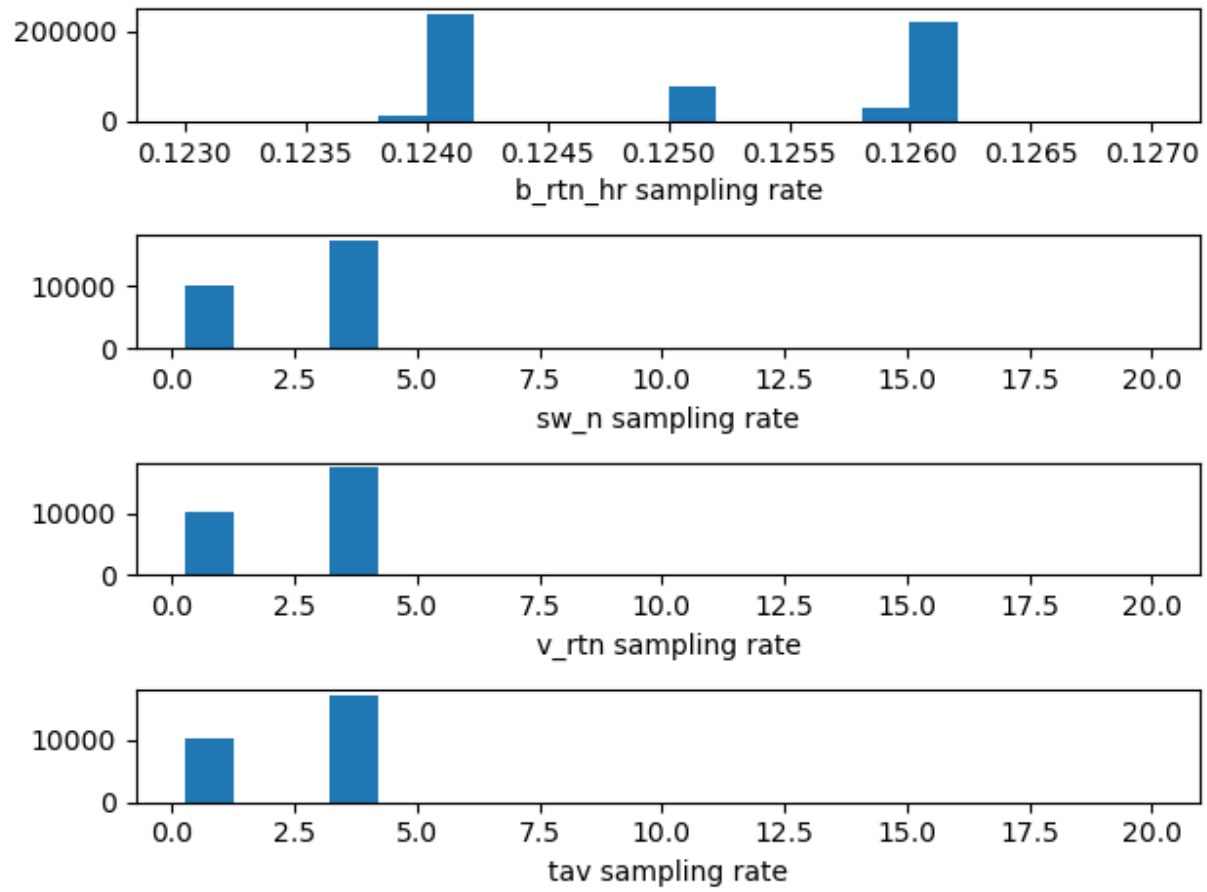
We will need to perform operations on data that are not regularly sampled as represented on the figure below.

[20]:
```
def to_epoch(datetime64_values):
    return (datetime64_values).astype(np.float64)/1e9

fig, ax = plt.subplots(4,1)
ax[0].hist(np.diff(to_epoch(b_rtn_hr.time)), bins=20)
ax[0].set_xlabel("b_rtn_hr sampling rate")
ax[1].hist(np.diff(to_epoch(sw_n.time)), bins=20)
ax[1].set_xlabel("sw_n sampling rate")
```

(continues on next page)

(continued from previous page)

```
ax[2].hist(np.diff(to_epoch(v_rtn.time)), bins=20)
ax[2].set_xlabel("v_rtn sampling rate")
ax[3].hist(np.diff(to_epoch(tav.time)), bins=20)
ax[3].set_xlabel("tav sampling rate")
plt.tight_layout()
```



Resample the data every second. Create a new dataframe object `df_1s` which will contain all the data.

[21]:
```
b_rtn_1s = b_rtn_df.resample("1S").ffill()
sw_n_1s = sw_n_df.resample("1S").ffill()
v_rtn_1s = v_rtn_df.resample("1S").ffill()

df_1s = b_rtn_1s.merge(sw_n_1s, left_index=True, right_index=True)
df_1s = df_1s.merge(v_rtn_1s, left_index=True, right_index=True)
df_1s = df_1s.dropna()

df_1s.describe()
```

[21]:

	br	bt	bn	density	vr	\
count	71996.000000	71996.000000	71996.000000	71996.000000	71996.000000	
mean	4.436458	-3.022399	0.313455	16.235723	425.954116	
std	3.676020	4.012876	5.413990	2.426260	17.762155	
min	-7.641881	-12.710729	-13.472364	5.478008	384.966644	

(continues on next page)

(continued from previous page)

```
25%        2.005486     -5.914948     -4.322630     14.484841    412.498413
50%        4.976464     -3.851966      1.154816     16.096893    424.748138
75%        7.297559     -0.395437      4.848405     17.809422    437.517120
max       12.322162      9.772242     11.992266     30.859621    488.915253

                 vt            vn
count  71996.000000  71996.000000
mean      -5.499633     -1.954408
std       14.873811     18.220244
min      -59.670586    -43.281979
25%      -14.777957    -16.209478
50%       -3.206085     -4.703673
75%        4.816283     12.270930
max       29.120104     56.761192
```

Compute

$$b_{\text{rtn}} = \frac{B_{\text{rtn}}}{(\mu_0 n m_p)^{1/2}}$$

The column names `br,bt,bn` are already taken, we will use `b_r,b_t,b_n`.

[22]:
```
m_p = 1.67e-27
mu_0 = 1.25664e-6
# you can also use
#from scipy import constants as cst
#print(cst.m_p, cst.mu_0, cst.Boltzmann)

N = df_1s.shape[0]

b = (df_1s[["br","bt","bn"]].values /
             (np.sqrt(mu_0*m_p*1e6*df_1s["density"].values.reshape(N,1)))*1e-12)
colnames = ["b_r","b_t","b_n"]

## In case the correction is wrong this worked
#b = (b_rtn_1s[sw_n_1s.index[0]:].values / \
#          (np.sqrt(mu_0 * sw_n_1s.values * 1e6 * m_p) ) *1e-12)

b = pd.DataFrame(data=b, index=df_1s.index, columns=colnames)
df_1s = df_1s.merge(b, right_index=True, left_index=True)
df_1s = df_1s.dropna()

df_1s[["b_r","b_t","b_n"]].describe()
```

[22]:
```
                b_r           b_t           b_n
count  71996.000000  71996.000000  71996.000000
mean      24.435180    -16.308684      2.815561
std       19.742385     21.555876     28.862056
min      -39.813521    -64.716665    -68.406051
25%       10.977414    -32.669705    -22.717015
50%       27.979350    -21.045612      6.404938
75%       40.156327     -2.147645     27.531269
max       69.329297     53.754876     81.635840
```

Compute the fluctuations for the velocity and the magnetic field (1h can be adjusted depending on the event):

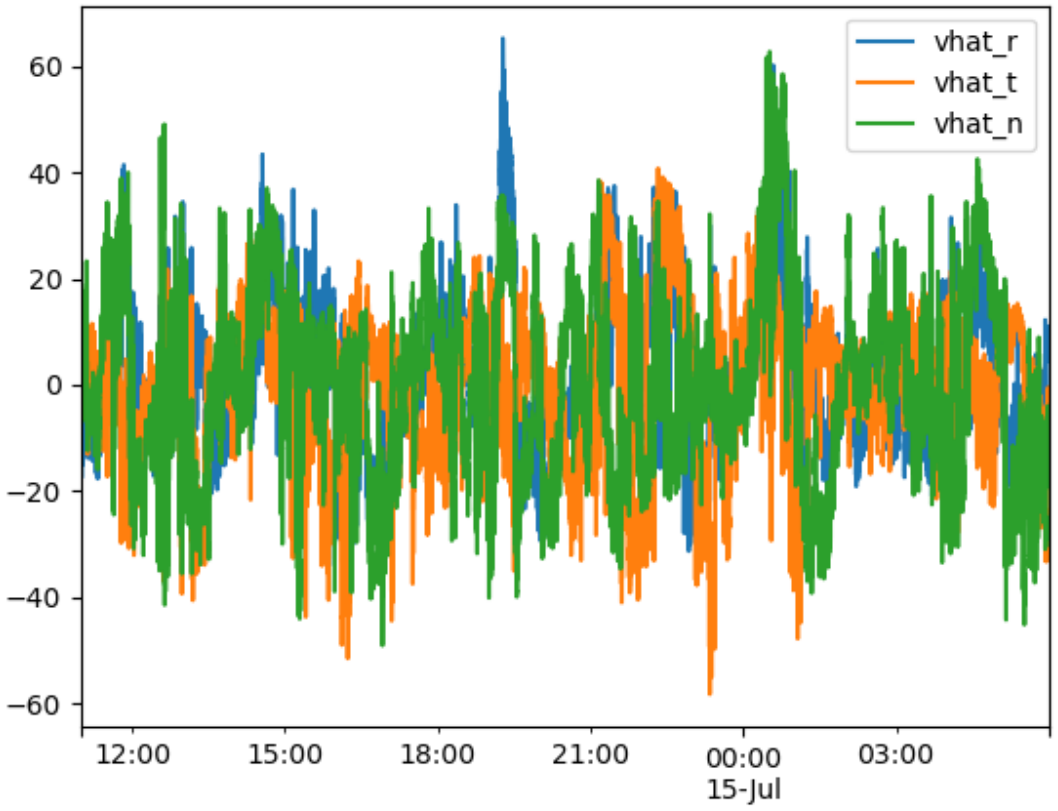
$$\hat{v} = v - \langle v \rangle_{1\text{h}}$$

and

$$\hat{b} = b - \langle b \rangle_{1\text{h}}$$

[23]:
```
vhat = df_1s[["vr","vt","vn"]] - df_1s[["vr","vt","vn"]].rolling(3600).mean()
colmap = {n1:n2 for n1,n2 in zip(vhat.columns,
                                 ["vhat_r","vhat_t","vhat_n"])}
vhat = vhat.rename(columns=colmap)
df_1s = df_1s.merge(vhat, right_index=True, left_index=True)
df_1s = df_1s.dropna()

df_1s[["vhat_r","vhat_t","vhat_n"]].plot()
```

[23]:
```
<Axes: >
```



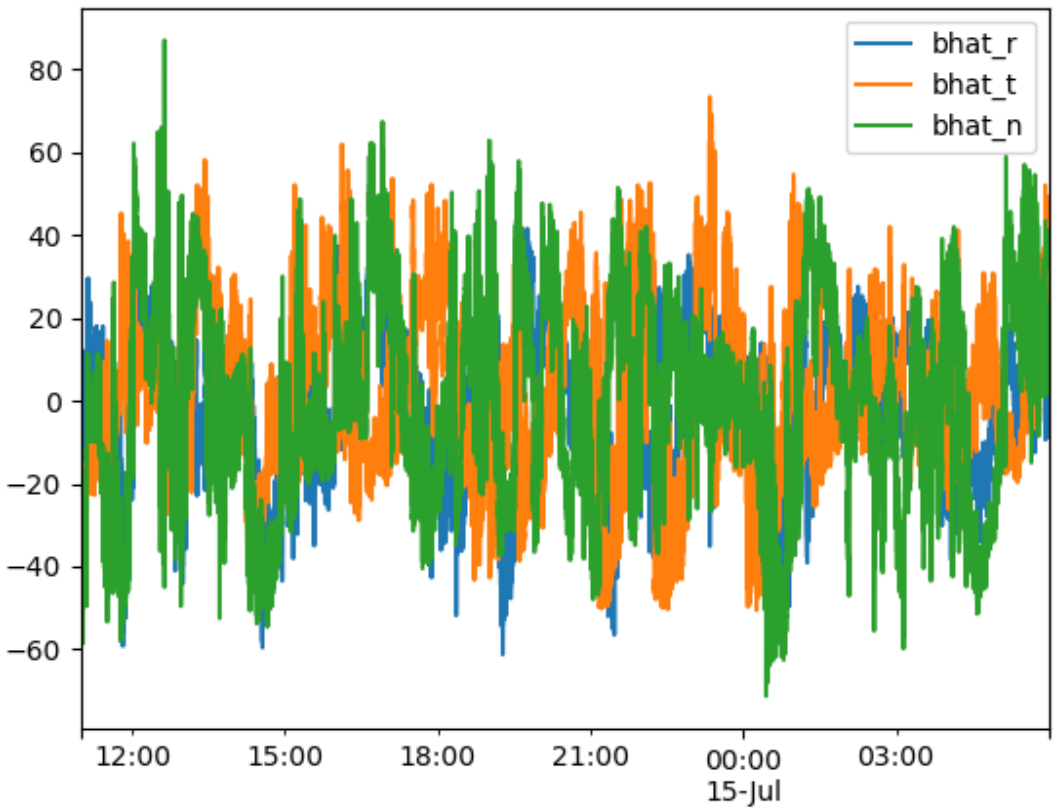
[24]:
```
bhat = b - b.rolling(3600).mean()
colmap = {n1:n2 for n1,n2 in zip(bhat.columns,
                                 ["bhat_r","bhat_t","bhat_n"])}
```

(continues on next page)

(continued from previous page)

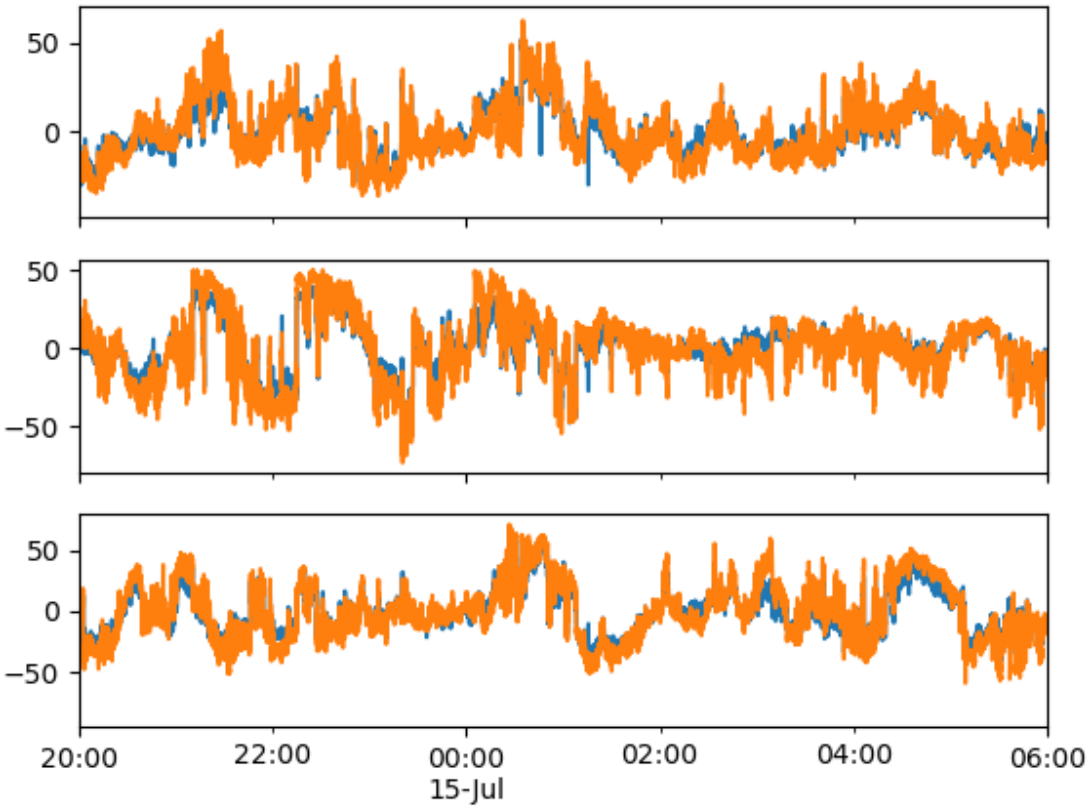
```
bhat = bhat.rename(columns=colmap)
df_1s = df_1s.merge(bhat, right_index=True, left_index=True)
df_1s = df_1s.dropna()

df_1s[["bhat_r","bhat_t","bhat_n"]].plot()
```

[24]:
```
<Axes: >
```



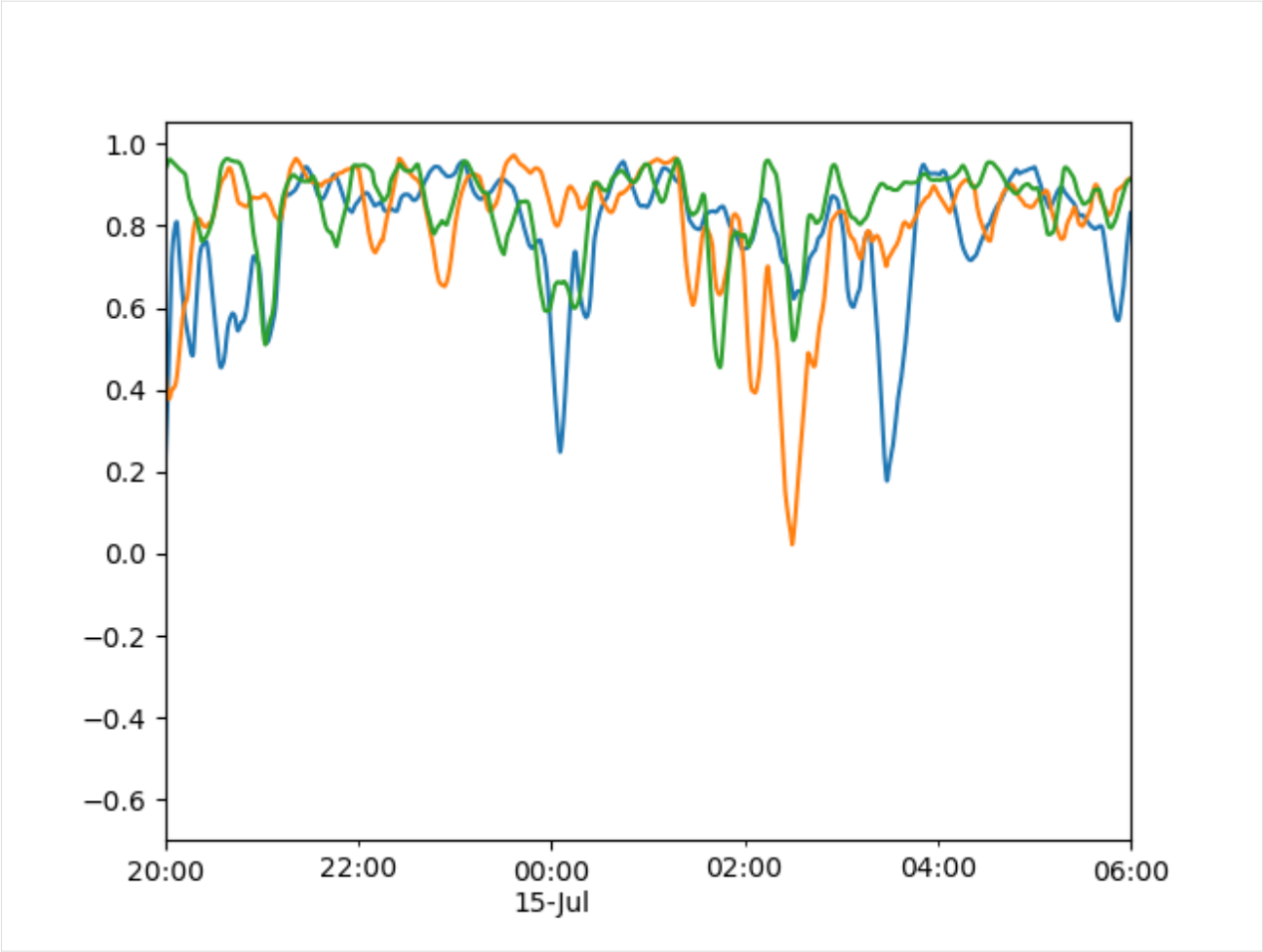
[25]:
```
fig, ax = plt.subplots(3,1,sharex=True)
for i in range(3):
    vhat.iloc[:,i].plot(ax=ax[i], label="v")
    (-bhat).iloc[:,i].plot(ax=ax[i], label="b")
t0 = datetime(2020,7,14,20)
plt.xlim([t0, t0+timedelta(hours=10)])
```

[25]:
```
(1594756800.0, 1594792800.0)
```



[26]:
```
fig, ax = plt.subplots(1,1,sharex=True)
for i in range(3):
    #(-bhat).iloc[:,i].rolling(600).corr(v_rtn_1s.iloc[:,i]).rolling(600).mean().
↪plot(ax=ax)
    (-bhat).iloc[:,i].rolling(600).corr(vhat.iloc[:,i]).rolling(600).mean().plot(ax=ax)
plt.xlim([t0, t0+timedelta(hours=10)])
```

[26]:
```
(1594756800.0, 1594792800.0)
```



As a complement, you can also compute the alfvénicity :

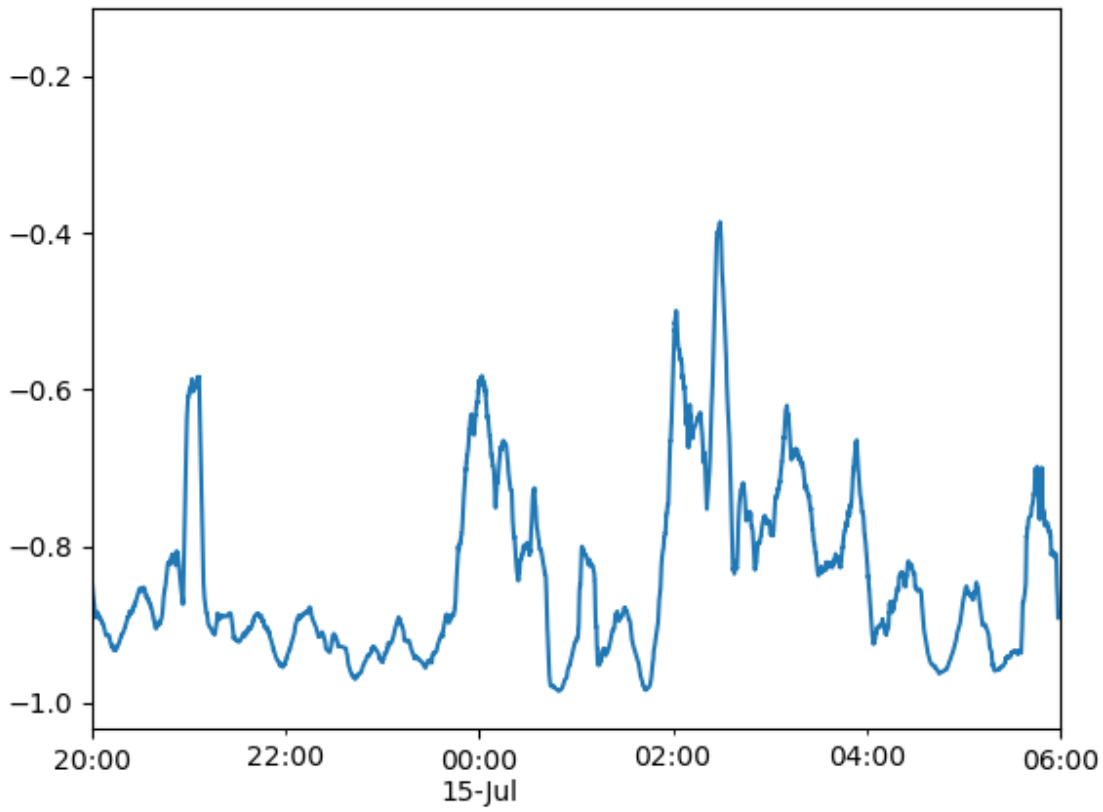
$$\sigma = \frac{2\hat{b}.\hat{v}}{(\hat{b}.\hat{b} + \hat{v}.\hat{v})}$$

and plot it :

[27]:
```
alfveniciry = sss=2*(bhat.bhat_r*vhat.vhat_r+bhat.bhat_t*vhat.vhat_t+bhat.bhat_n*vhat.
↪vhat_n)/(bhat.bhat_r*bhat.bhat_r+bhat.bhat_t*bhat.bhat_t+bhat.bhat_n*bhat.bhat_n+vhat.
↪vhat_r*vhat.vhat_r+vhat.vhat_t*vhat.vhat_t+vhat.vhat_n*vhat.vhat_n)
```

[28]:
```
fig, ax = plt.subplots(1,1,sharex=True)
sss.rolling(600).mean().plot(ax=ax)
plt.xlim([t0, t0+timedelta(hours=10)])
```

[28]:
```
(1594756800.0, 1594792800.0)
```



4.7.11 Fourier Transform

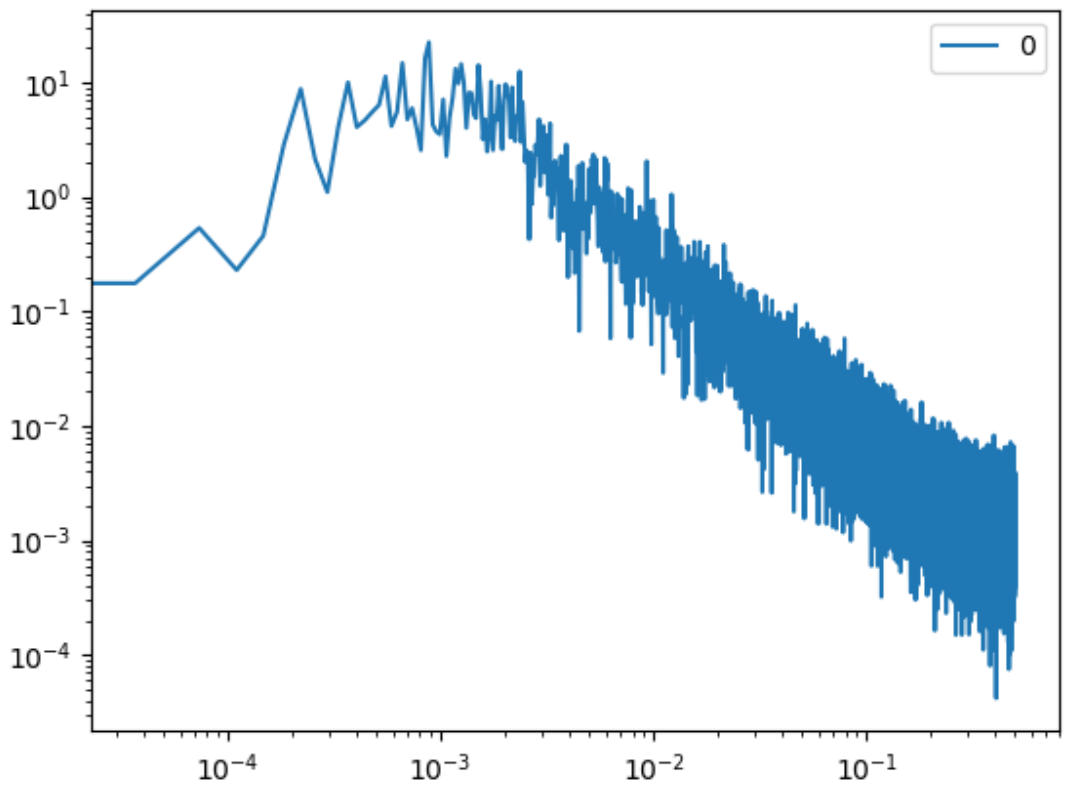
Some of the calculations that follow are a bit slow.

[29]:
```
#from numpy.fft import fft, fftfreq, rfft, rfftfreq
from scipy.fft import fft, fftfreq, rfft, rfftfreq

# v rtn
x = v_rtn_df - v_rtn_df.rolling(int(1200 / 4)).mean()
x = x.interpolate()
x = x.dropna()#x.fillna(0)

N = x.shape[0]

sp_v = fft(x.values, axis=0)
sp_v = np.abs(sp_v[:N//2]) * 2./N
sp_v = np.sum(sp_v**2, axis=1)

sp_v_freq = fftfreq(x.shape[0], 1.)
sp_v_freq = sp_v_freq[:N//2]
```

(continues on next page)

(continued from previous page)

```
sp_v = pd.DataFrame(index=sp_v_freq, data=np.absolute(sp_v))
sp_v = sp_v.dropna()

fig, ax = plt.subplots(1,1)
sp_v.plot(ax=ax)

plt.xscale("log")
plt.yscale("log")
```



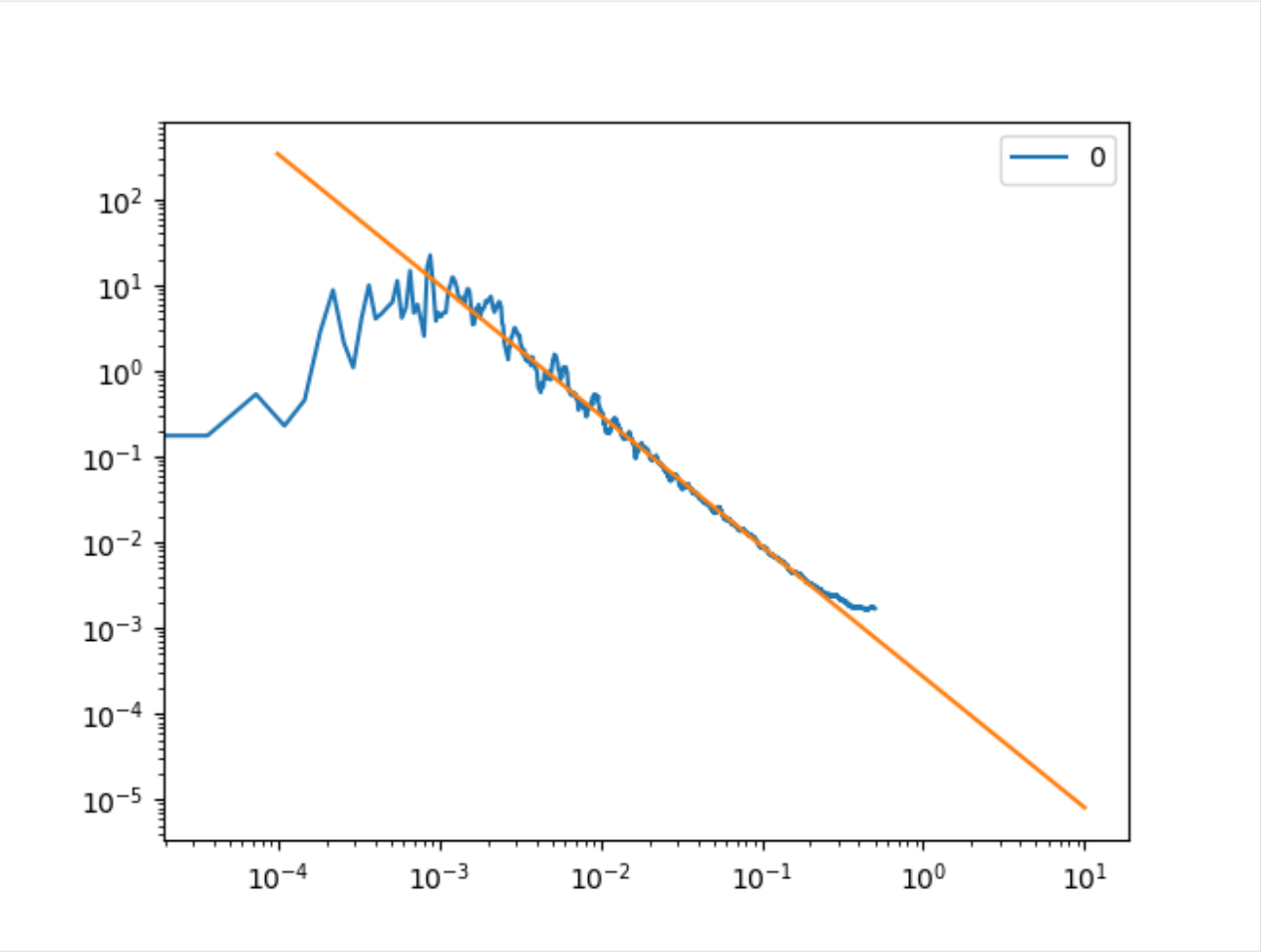
The data is noisy and we need to filter it. For each frequency f in the domain we take the mean magnitude over $[(1-\alpha)f, (1+\alpha)f[$. α is set to 4%

[30]:
```
def f(r, x, alpha=.04):
    fmin,fmax = (1.-alpha)*r[0], (1.+alpha)*r[0]
    indx = (x[:,0]>=fmin) & (x[:,0]<fmax)
    if np.sum(indx)==0:
        return np.nan
    return np.mean(x[indx,1])
def proportional_rolling_mean(x, alpha=.04):
    X = np.hstack((x.index.values.reshape(x.index.shape[0],1), x.values))
    Y = np.apply_along_axis(f, 1, X, X)
    return pd.DataFrame(index=X[:,0], data=Y)
```

Add a linear fit to the the spectrum in the proper frequency range. From Kolmogorov theory we expect a -1.6 spectral index

[31]:
```
from sklearn.linear_model import LinearRegression
sp_v_mean = proportional_rolling_mean(sp_v)
sp_v_mean = sp_v_mean.bfill()

sp_v_mean.plot()

sp_v_mean_log = pd.DataFrame(data =
                             np.hstack((np.log(sp_v_mean[1e-3:2e-1].index.values).
↪reshape(sp_v_mean[1e-3:2e-1].shape[0],1),
                                        np.log(sp_v_mean[1e-3:2e-1].values))))
sp_v_mean_log = sp_v_mean_log.replace([np.inf, -np.inf], np.nan)
sp_v_mean_log = sp_v_mean_log.dropna()

# linear regression data
tt = sp_v_mean_log.values[:,0].reshape((-1,1))
xx = sp_v_mean_log.values[:,1].reshape((-1,1))

# fit the linear regression
model = LinearRegression()
model.fit(tt, xx)

print(f"Coef : {model.coef_[0,0]}")

xx = np.log(np.linspace(1e-4, 1e1, 100))
yy = xx*model.coef_[0,0]+model.intercept_[0]
plt.plot(np.exp(xx), np.exp(yy))

plt.xscale("log")
plt.yscale("log")
#plt.xlim([1e-4,1e1])
#plt.ylim([1e-4,1e8])
```

```
Coef : -1.5250478779805297
```



Magnetic field FFT

The magnetic field data are sampled at a higher rate (8 Hz) than the velocity (.25 Hz). Let's merge both those parameters into a single dataframe `df_hr`.

[32]:
```
df_hr = b_rtn_df.merge(sw_n_df, right_index=True, left_index=True, how="outer")
df_hr = df_hr.interpolate()

# resample the data to a rate of 4Hz
df_hr_8hz = df_hr.resample("125ms").ffill()

# get b
N = df_hr_8hz.shape[0]
b_ = (df_hr_8hz[["br","bt","bn"]].values /
             (np.sqrt(mu_0*m_p*1e6*df_hr_8hz["density"].values.reshape(N,1)))*1e-12)
colnames = ["b_r","b_t","b_n"]
b_ = pd.DataFrame(data = b_, columns=colnames,index=df_hr_8hz.index)
bhat_ = b_ - b_.rolling(1200*8).mean()
colnames={n1:n2 for n1,n2 in zip(bhat_.columns, ["bhat_r","bhat_t","bhat_n"])}
bhat_=bhat_.rename(columns=colnames)
```

(continues on next page)

(continued from previous page)

```
df_hr_8hz = df_hr_8hz.merge(b_, right_index=True, left_index=True)
df_hr_8hz = df_hr_8hz.merge(bhat_, right_index=True, left_index=True)
df_hr_8hz.describe()
```

[32]:

	br	bt	bn	density	\
count	575999.000000	575999.000000	575999.000000	575972.000000	
mean	4.436757	-3.022156	0.313615	16.235375	
std	3.676386	4.013093	5.414093	2.412064	
min	-7.779764	-12.940836	-13.529315	5.053784	
25%	1.998236	-5.914867	-4.331347	14.491481	
50%	4.978395	-3.851042	1.152890	16.111278	
75%	7.296612	-0.396753	4.847986	17.776652	
max	12.380941	9.837873	12.080406	30.799139	

	b_r	b_t	b_n	bhat_r	\
count	575972.000000	575972.000000	575972.000000	566373.000000	
mean	24.436116	-16.305073	2.819573	-0.086668	
std	19.744158	21.552018	28.857625	13.870057	
min	-40.092490	-64.840088	-67.248313	-65.886914	
25%	10.941165	-32.676037	-22.728208	-7.706674	
50%	27.989464	-21.068695	6.383892	0.582921	
75%	40.175065	-2.162044	27.546302	7.786346	
max	69.421091	54.575628	86.514038	54.931340	

	bhat_t	bhat_n
count	566373.000000	566373.000000
mean	-0.041386	-0.322127
std	15.108742	20.535992
min	-72.876711	-88.599835
25%	-8.761896	-13.434541
50%	-0.534475	-0.450718
75%	9.109271	11.819825
max	72.324271	87.932526

Compute b FFT.

[33]:
```
df_hr_8hz = df_hr_8hz.fillna(0)
```

[34]:
```
# b rtn
x = df_hr_8hz[["bhat_r","bhat_t","bhat_n"]]

sp_b = fft(x.values, axis=0)
sp_b = sp_b[:N//2] * 2./N
sp_b = np.sum(sp_b**2, axis=1)

sp_b_freq = rfftfreq(x.shape[0], .125)
sp_b_freq = sp_b_freq[:N//2]

sp_b = pd.DataFrame(index=sp_b_freq, data=np.absolute(sp_b))
```

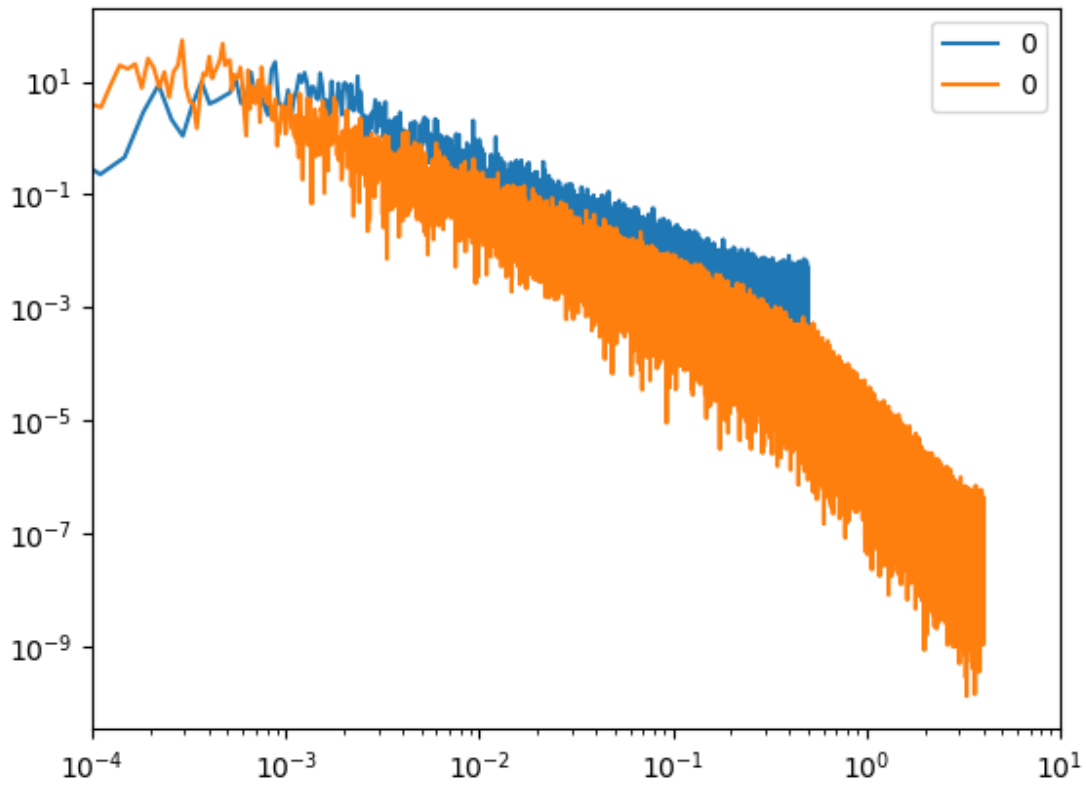
[35]:
```
fig, ax = plt.subplots(1,1)
```

(continues on next page)

(continued from previous page)

```
sp_v.plot(ax=ax)
sp_b.plot(ax=ax)
plt.xscale("log")
plt.yscale("log")
plt.xlim([1e-4,1e1])
```

[35]:
```
(0.0001, 10.0)
```



Compute rolling mean. Warning: This cell can take a couple minutes to execute.

[36]:
```
sp_v_rm = proportional_rolling_mean(sp_v)
sp_b_rm = proportional_rolling_mean(sp_b)
```

Do a linear regression on the magnetic field spectrum for frequencies in $[10^{-3}, 2 \times 10^{-1}]$ and $[2 \times 10^{-1}, 10^{1}]$.

[37]:
```
sp_b_rm_log = sp_b_rm[1e-3:2e-1]
sp_b_rm_log["f"] = np.log(sp_b_rm_log.index)
sp_b_rm_log["sp"] = np.log(sp_b_rm[1e-3:2e-1].values)

sp_b_rm_log = sp_b_rm_log.replace([np.inf, -np.inf], np.nan)
sp_b_rm_log = sp_b_rm_log.dropna()
```

(continues on next page)

(continued from previous page)

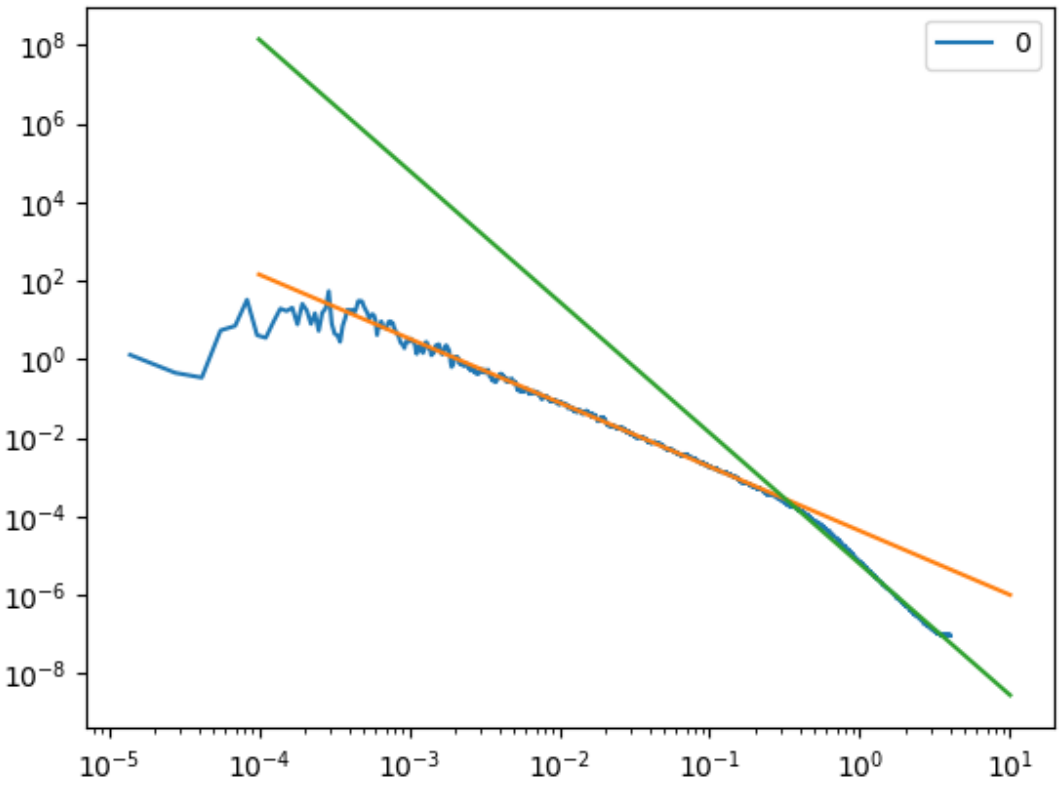
```
sp_b_rm_log_2 = sp_b_rm[2e-1:]
sp_b_rm_log_2["f"] = np.log(sp_b_rm_log_2.index)
sp_b_rm_log_2["sp"] = np.log(sp_b_rm[2e-1:].values)

sp_b_rm_log_2 = sp_b_rm_log_2.replace([np.inf, -np.inf], np.nan)
sp_b_rm_log_2 = sp_b_rm_log_2.dropna()

# fit the linear regressions
model1 = LinearRegression()
model1.fit(sp_b_rm_log.values[:,1].reshape((-1,1)),
           sp_b_rm_log.values[:,2].reshape((-1,1)))

model2 = LinearRegression()
model2.fit(sp_b_rm_log_2.values[:,1].reshape((-1,1)),
           sp_b_rm_log_2.values[:,2].reshape((-1,1)))

xx = np.log(np.linspace(1e-4, 1e1, 100))
yy = xx*model1.coef_[0,0]+model1.intercept_[0]
yy2 = xx*model2.coef_[0,0]+model2.intercept_[0]

sp_b_rm.plot()
plt.plot(np.exp(xx), np.exp(yy))
plt.plot(np.exp(xx), np.exp(yy2))



plt.xscale("log")
plt.yscale("log")
```

```
/tmp/ipykernel_17669/1550647204.py:2: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  sp_b_rm_log["f"] = np.log(sp_b_rm_log.index)
/tmp/ipykernel_17669/1550647204.py:3: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  sp_b_rm_log["sp"] = np.log(sp_b_rm[1e-3:2e-1].values)
/tmp/ipykernel_17669/1550647204.py:10: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  sp_b_rm_log_2["f"] = np.log(sp_b_rm_log_2.index)
/tmp/ipykernel_17669/1550647204.py:11: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
```

(continues on next page)

(continued from previous page)

```
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_
↪guide/indexing.html#returning-a-view-versus-a-copy
  sp_b_rm_log_2["sp"] = np.log(sp_b_rm[2e-1:].values)
```



[38]:
```
print(model1.coef_[0,0],model2.coef_[0,0])
```

```
-1.63289684621718 -3.341639196329613
```

[39]:
```
fig, ax = plt.subplots(1,1)
sp_v_rm.plot(ax=ax)
sp_b_rm.plot(ax=ax)
# linear interpolations
plt.plot(np.exp(xx), np.exp(yy))
plt.plot(np.exp(xx), np.exp(yy2))

plt.xscale("log")
plt.yscale("log")
plt.legend()
```

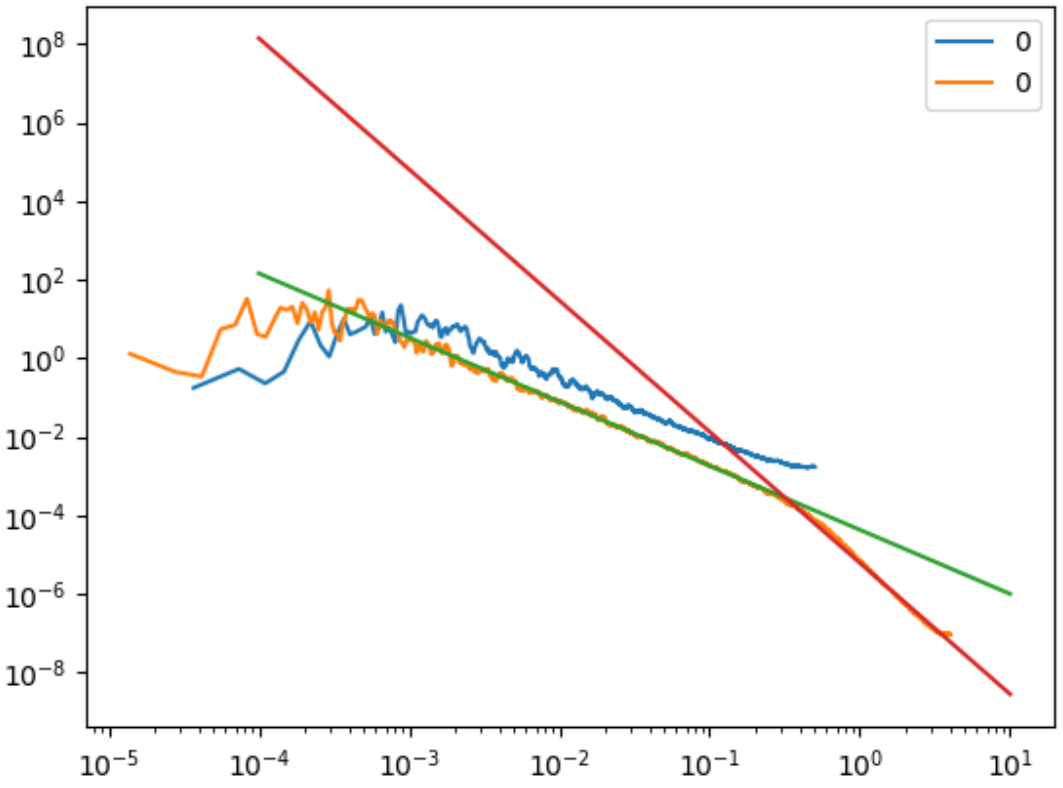
```
/tmp/ipykernel_17669/1627177063.py:1: RuntimeWarning: More than 20 figures have been␣
↪opened. Figures created through the pyplot interface (`matplotlib.pyplot.figure`) are␣
```

(continues on next page)

(continued from previous page)

```
↪retained until explicitly closed and may consume too much memory. (To control this␣
↪warning, see the rcParam `figure.max_open_warning`). Consider using `matplotlib.pyplot.
↪close()`.
  fig, ax = plt.subplots(1,1)
```

[39]:
```
<matplotlib.legend.Legend at 0x7f6dcf1a6ea0>
```



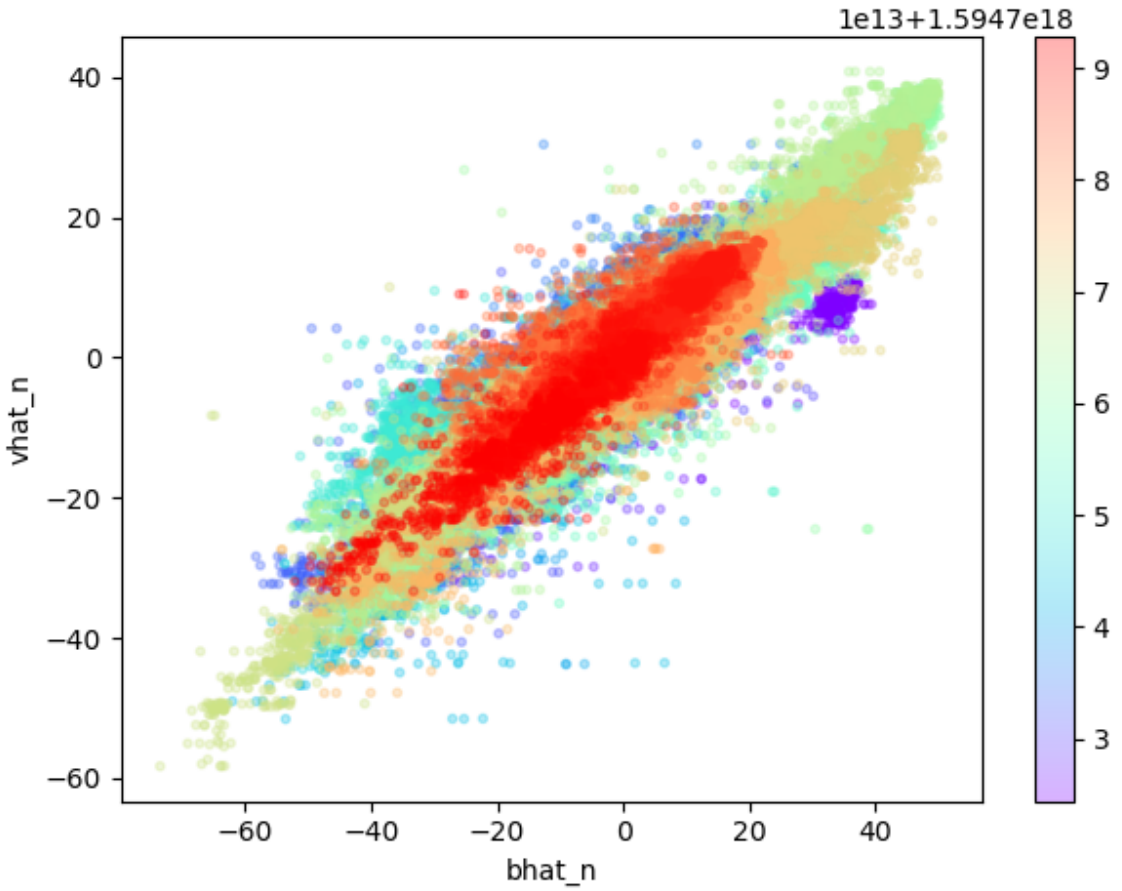
4.7.12 Distribution in b-V space

[40]:
```
import matplotlib
cmap = matplotlib.cm.rainbow.copy()
cmap.set_bad('White',0.)


fig, ax = plt.subplots(1,1)
coord = "rtn"

cs=ax.scatter(-df_1s["bhat_t"], df_1s["vhat_t"],
              c=df_1s.index, cmap=cmap, marker=".", alpha=.3)
ax.set_xlabel(f"bhat_{coord[i]}")
ax.set_ylabel(f"vhat_{coord[i]}")
```

(continues on next page)

(continued from previous page)

```
plt.colorbar(cs,ax=ax)
plt.tight_layout()
plt.show()
```



The following section was generated from `docs/examples/Filtering.ipynb`

4.8 Scipy filters compatibility

4.8.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
try:
    from google.colab import output

    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.8.2 For all users:

[1]:
```
import speasy as spz
from speasy.core.any_files import any_loc_open
from speasy.core.cdf import load_variable
from speasy.signal.filtering import sosfiltfilt
import numpy as np
%matplotlib widget
mms1_products = spz.inventories.tree.cda.MMS.MMS1
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
from scipy import signal
```

4.8.3 Loading data

Let's use the MMS1 EDP burst data from the CDAWeb to test the filtering.

[2]:
```
mms1_edp_hmfe_par_epar_brst_l2 = load_variable( "mms1_edp_hmfe_par_epar_brst_l2", "https:
↪//cdaweb.gsfc.nasa.gov/pub/data/mms/mms1/edp/brst/l2/hmfe/2021/07/mms1_edp_brst_l2_
↪hmfe_20210701020013_v2.0.0.cdf")
```

[3]:
```
sampling_frequency = 1e9/float(mms1_edp_hmfe_par_epar_brst_l2.time[1]-mms1_edp_hmfe_par_
↪epar_brst_l2.time[0])
sampling_frequency
```

[3]:
```
65535.09404285995
```

4.8.4 Filtering with scipy.signal

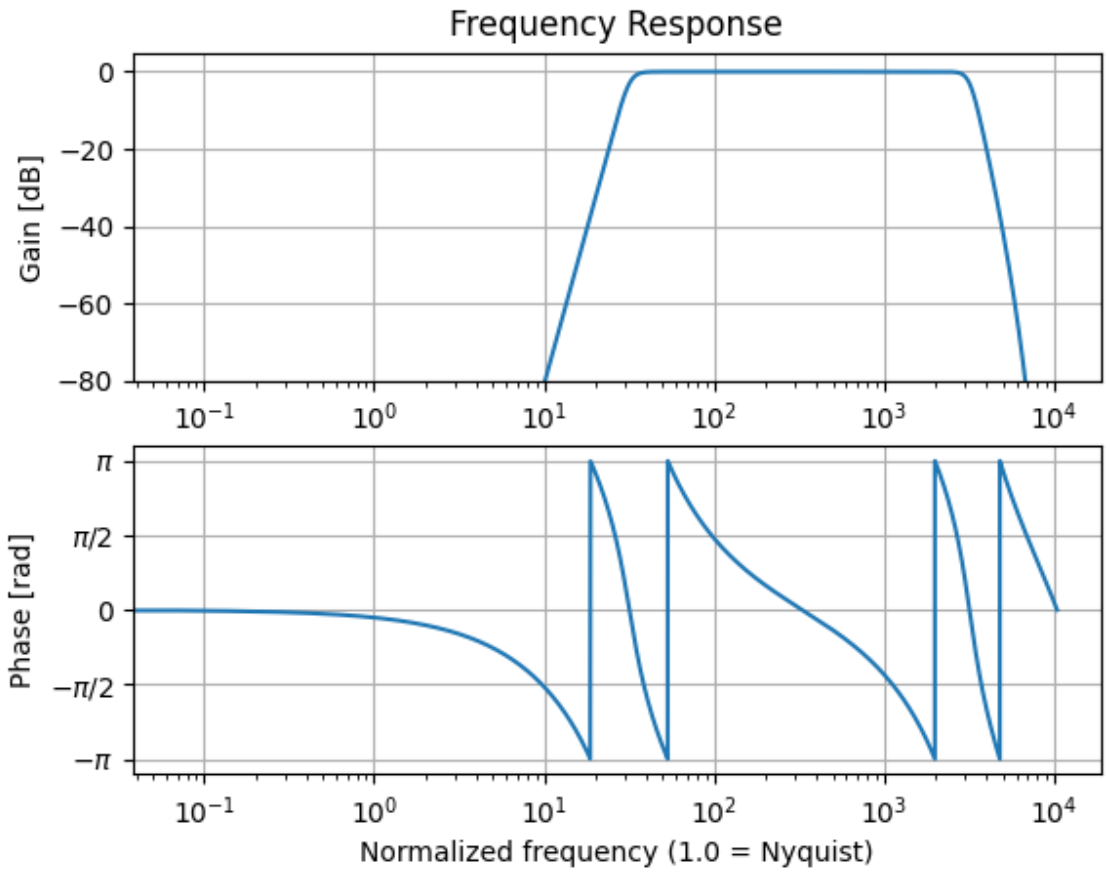
Let's design a time domain bandpass filter with scipy.signal cutting off frequencies below 100 Hz and above 10 kHz.

[4]:
```
sos = signal.iirfilter(N=8, Wn=[100, 10000], rs=80, btype='bandpass',output='sos',␣
↪fs=sampling_frequency)
w, h = signal.sosfreqz(sos, worN=150000, fs=sampling_frequency)
plt.figure()
plt.subplot(2, 1, 1)
db = 20*np.log10(np.maximum(np.abs(h), 1e-5))
plt.plot(w/np.pi, db)
plt.ylim(-75, 5)
plt.grid(True)
plt.yticks([0, -20, -40, -60, -80])
plt.semilogx()
plt.ylabel('Gain [dB]')
plt.title('Frequency Response')
plt.subplot(2, 1, 2)
plt.plot(w/np.pi, np.angle(h))
plt.grid(True)
plt.yticks([-np.pi, -0.5*np.pi, 0, 0.5*np.pi, np.pi],
           [r'$-\pi$', r'$-\pi/2$', '0', r'$\pi/2$', r'$\pi$'])
plt.ylabel('Phase [rad]')
```

(continues on next page)

(continued from previous page)

```
plt.xlabel('Normalized frequency (1.0 = Nyquist)')
plt.semilogx()
plt.show()
```



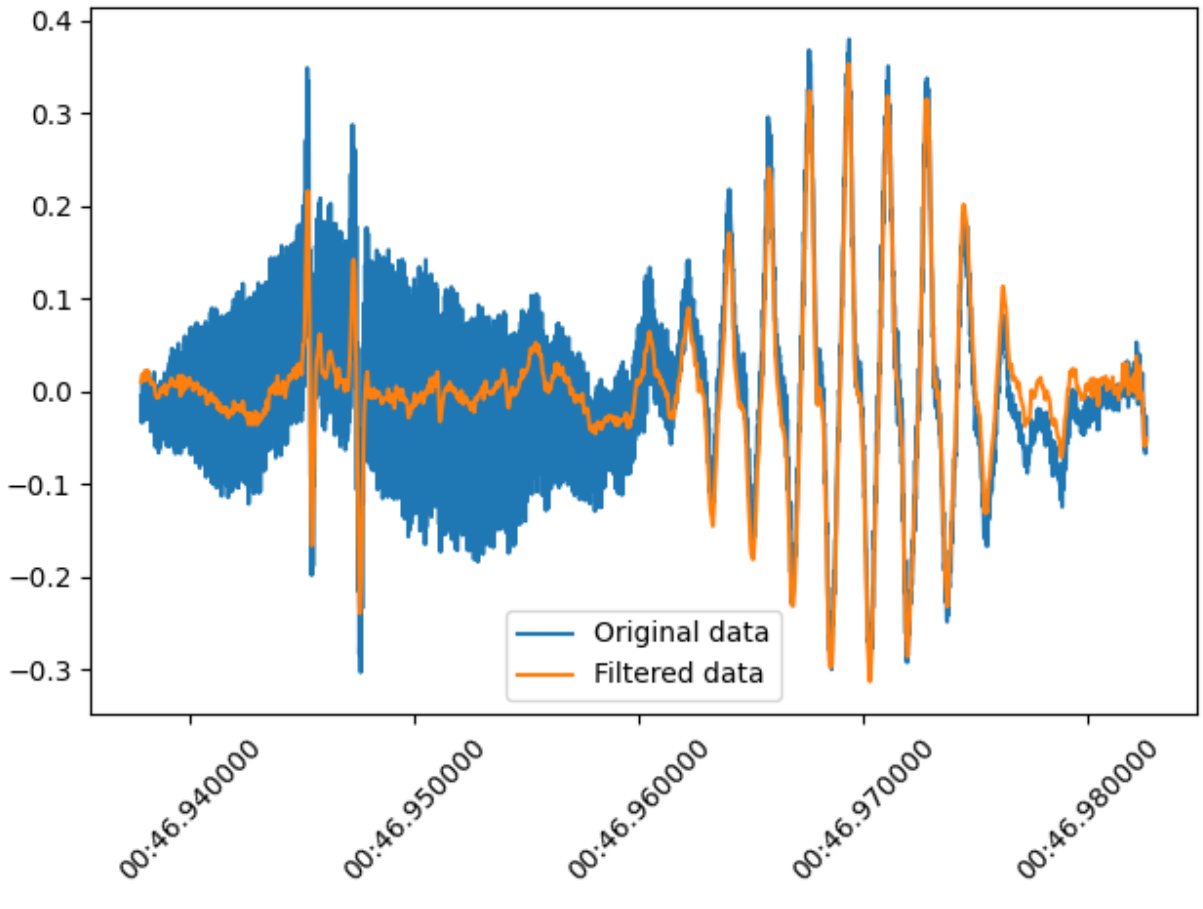
4.8.5 Speasy variables filtering

We can directly filter the data using the sos filter designed with scipy.signal.

[5]:
```
filtered_mms1_edp_hmfe_par_epar_brst_l2 = sosfiltfilt(sos, mms1_edp_hmfe_par_epar_brst_
↪l2)
```

4.8.6 Plotting the results

[6]:
```
start_time = np.datetime64("2021-07-01T02:00:46.9378", "ns")
stop_time = np.datetime64("2021-07-01T02:00:46.9826", "ns")
filtered_mms1_edp_hmfe_par_epar_brst_l2 = filtered_mms1_edp_hmfe_par_epar_brst_l2[start_
↪time:stop_time]
mms1_edp_hmfe_par_epar_brst_l2 = mms1_edp_hmfe_par_epar_brst_l2[start_time:stop_time]

plt.figure()
plt.plot(mms1_edp_hmfe_par_epar_brst_l2.time, mms1_edp_hmfe_par_epar_brst_l2.values,␣
↪label="Original data")
plt.plot(filtered_mms1_edp_hmfe_par_epar_brst_l2.time, filtered_mms1_edp_hmfe_par_epar_
↪brst_l2.values, label="Filtered data")
plt.legend()
plt.xticks(rotation=45)
plt.tight_layout()
```



The following section was generated from docs/examples/Resampling.ipynb

4.9 Resampling and Interpolation example

4.9.1 Only for Google Colab users:

[ ]:
```
%pip install --upgrade ipympl speasy
```

[ ]:
```
try:
    from google.colab import output

    output.enable_custom_widget_manager()
except:
    print("Not running inside Google Collab")
```

4.9.2 For all users:

[1]:
```
import speasy as spz
from speasy.signal.resampling import interpolate
import numpy as np
%matplotlib widget
mms1_products = spz.inventories.tree.cda.MMS.MMS1
# Use this instead if you are not using jupyterlab yet
#%matplotlib notebook
import matplotlib.pyplot as plt
```

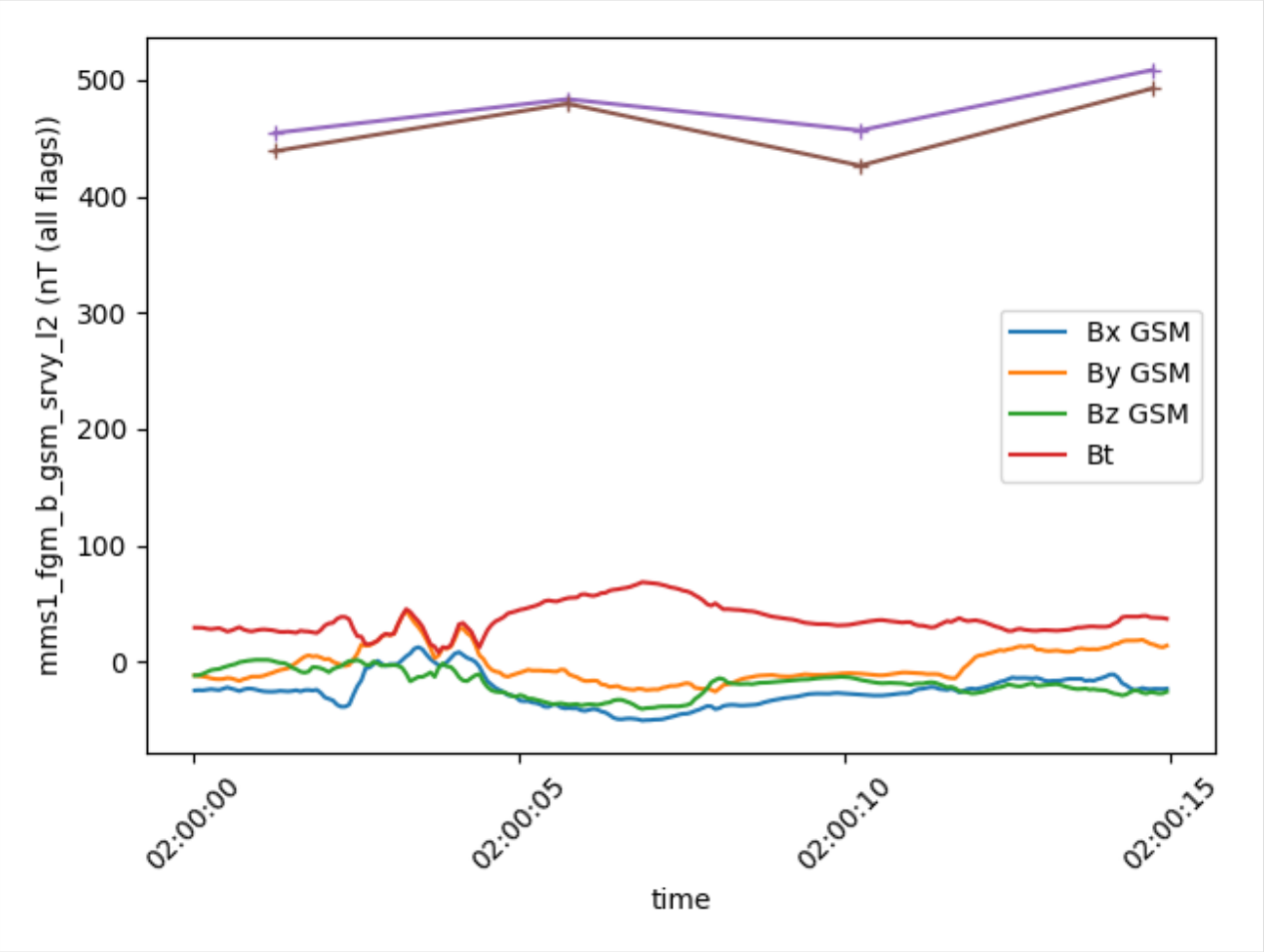
4.9.3 let's get some data with different time resolutions

[2]:
```
b, Tperp, Tpara = spz.get_data(
        [
            mms1_products.FGM.MMS1_FGM_SRVY_L2.mms1_fgm_b_gsm_srvy_l2,
            mms1_products.DIS.MMS1_FPI_FAST_L2_DIS_MOMS.mms1_dis_tempperp_fast,
            mms1_products.DIS.MMS1_FPI_FAST_L2_DIS_MOMS.mms1_dis_temppara_fast
        ],
        '2017-01-01T02:00:00',
        '2017-01-01T02:00:15'
    )
```

[3]:
```
np.diff(b.time)[:3], np.diff(Tperp.time), np.diff(Tpara.time)
```

[3]:
```
(array([62500852, 62500852, 62500852], dtype='timedelta64[ns]'),
 array([4500024000, 4500029000, 4500024000], dtype='timedelta64[ns]'),
 array([4500024000, 4500029000, 4500024000], dtype='timedelta64[ns]'))
```

[4]:
```
plt.figure()
ax = b.plot()
plt.plot(Tperp.time, Tperp.values, marker='+')
plt.plot(Tpara.time, Tpara.values, marker='+')
plt.tight_layout()
```



4.9.4 Now we can interpolate the data to the magnetic field time resolution

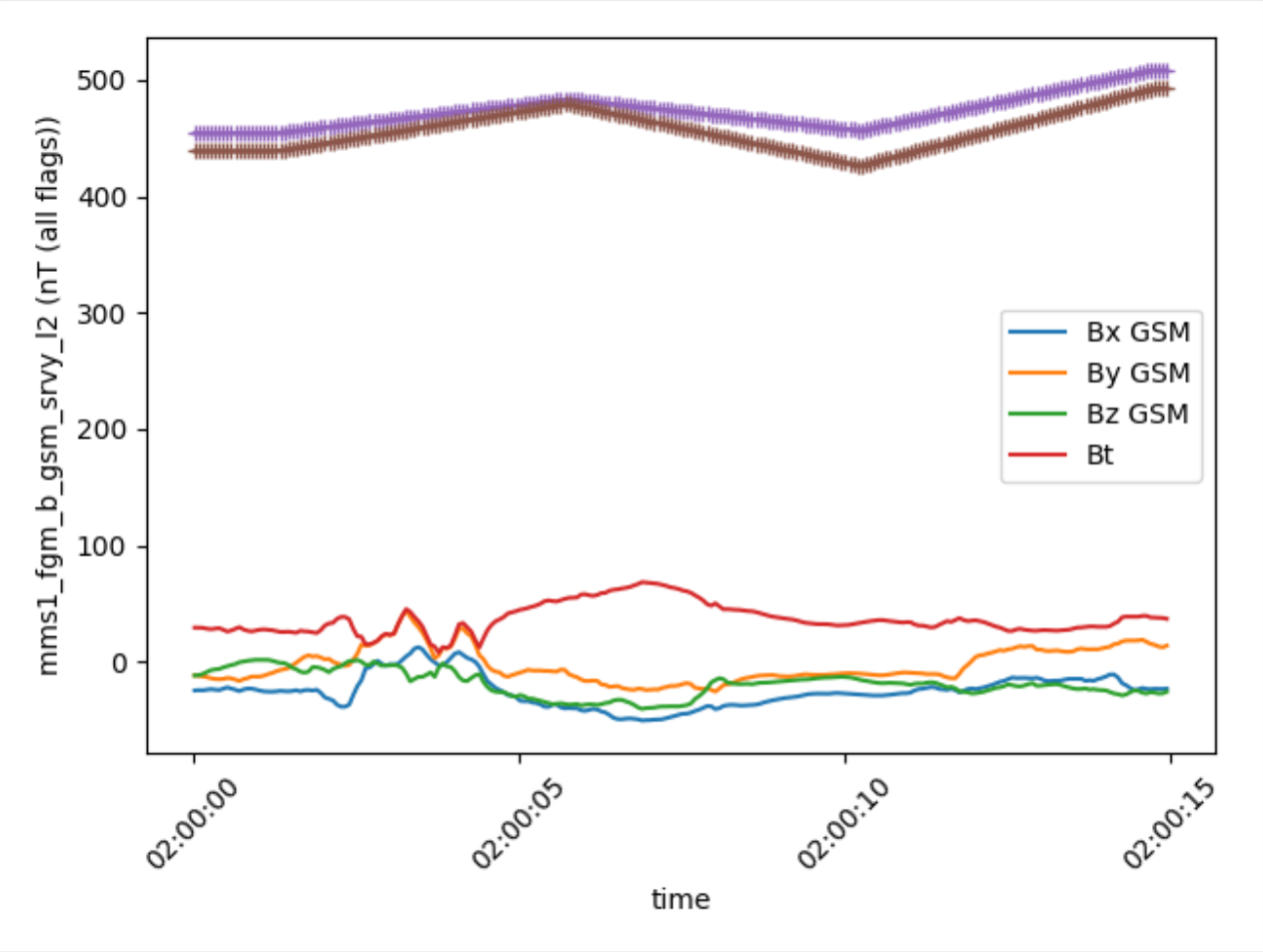
The interpolate function will interpolate the data to the time of the first argument.

[5]:
```
Tperp_interp, Tpara_interp = interpolate(b, [Tperp, Tpara])
```

[6]:
```
b.time[:2], Tperp_interp.time[:2], Tpara_interp.time[:2]
```

[6]:
```
(array(['2017-01-01T02:00:00.011774444', '2017-01-01T02:00:00.074275296'],
       dtype='datetime64[ns]'),
 array(['2017-01-01T02:00:00.011774444', '2017-01-01T02:00:00.074275296'],
       dtype='datetime64[ns]'),
 array(['2017-01-01T02:00:00.011774444', '2017-01-01T02:00:00.074275296'],
       dtype='datetime64[ns]'))
```

[7]:
```
plt.figure()
ax = b.plot()
plt.plot(Tperp_interp.time, Tperp_interp.values, marker='+')
plt.plot(Tpara_interp.time, Tpara_interp.values, marker='+')
plt.tight_layout()
```


500
400
300
200
100
0
mms1_fgm_b_gsm_srvy_l2 (nT (all flags))
Bx GSM
By GSM
Bz GSM
Bt
02:00:00
02:00:05
02:00:10
02:00:15
time

CHAPTER

FIVE

SPEASY DEVELOPER DOCUMENTATION

5.1 Indices and tables

- genindex
- modindex
- search

CHAPTER

SIX

HISTORY

6.1 1.2.7 (2024-04-17)

- Always check if a cache entry is None before slicing it by @jeandet in https://github.com/SciQLop/speasy/pull/127

6.2 1.2.6 (2024-04-17)

- Emergency release because sscweb Json schema has changed by @jeandet

6.3 1.2.5 (2024-04-17)

- Add python3.12 on ci by @jeandet in https://github.com/SciQLop/speasy/pull/126
- If last cache fragment is None then don't slice it by @jeandet in https://github.com/SciQLop/speasy/pull/125

6.4 1.2.4 (2024-03-12)

- [AMDA]Handles cases where timeRestriction is after stop_date by @jeandet in https://github.com/SciQLop/speasy/pull/124

6.5 1.2.3 (2024-02-22)

- Fixes https://github.com/SciQLop/speasy/issues/119 by @jeandet in https://github.com/SciQLop/speasy/pull/120
- Add support for AMDA restricted products by @jeandet in https://github.com/SciQLop/speasy/pull/118
- Automatically disable web services if they are not available by @jeandet in https://github.com/SciQLop/speasy/pull/112

6.6 1.2.2 (2023-11-28)

- Fixes https://github.com/SciQLop/speasy/issues/110, returns None instead of crash when there is no file on server by @jeandet in https://github.com/SciQLop/speasy/pull/111

6.7 1.2.1 (2023-11-07)

- Fixes non ISTP compliant files axis merging by @jeandet in https://github.com/SciQLop/speasy/pull/109

6.8 1.2.0 (2023-10-31)

- Fix old version code example in README.md by @jgieseler in https://github.com/SciQLop/speasy/pull/93
- Cdaweb and others archives direct file access by @jeandet in https://github.com/SciQLop/speasy/pull/89
- Drops Python 3.7 support and adds Python 3.11 by @jeandet in https://github.com/SciQLop/speasy/pull/97
- Switch to PyCDFpp 0.6+ by @jeandet in https://github.com/SciQLop/speasy/pull/100
- [AMDA] Uses CDF_ISTP as default by @jeandet in https://github.com/SciQLop/speasy/pull/101
- [Cache] Always use with transact(): statement with by @jeandet in https://github.com/SciQLop/speasy/pull/102
- Increase tests code coverage by @jeandet in https://github.com/SciQLop/speasy/pull/103
- Make more obvious to user that Speasy doesn't support downloading a whole dataset at once with some WS by @jeandet in https://github.com/SciQLop/speasy/pull/106
- [AMDA] Switch to https by @jeandet in https://github.com/SciQLop/speasy/pull/108
- Readme improvments by @jeandet in https://github.com/SciQLop/speasy/pull/104

6.9 1.1.2 (2023-06-01)

- New Speasy logo! by @jeandet in https://github.com/SciQLop/speasy/pull/84
- Switches readme to Markdown and removes lgtm badges (deprecated) by @jeandet in https://github.com/SciQLop/speasy/pull/85
- Reduces requests size for MMS big burst products on CDAWeb by @jeandet in https://github.com/SciQLop/speasy/pull/86
- Handles cases where labels are missing in CDAWeb generated files by @jeandet in https://github.com/SciQLop/speasy/pull/88
- Fixes AMDA CSV parser where derived parameters attributes gets overwritten by base param by @jeandet in https://github.com/SciQLop/speasy/pull/87
- Fixes #90: Uses output format value from config as fallback when requesting data from proxy for AMDA by @jeandet in https://github.com/SciQLop/speasy/pull/91

6.10 1.1.1 (2023-04-06)

- Fixes bug in v1.1.0 where AMDA CDF requests were not correctly written in cache.

6.11 1.1.0 (2023-04-06)

- Adds badges and links to Google Colab by @jeandet in https://github.com/SciQLop/speasy/pull/82
- better figure by @nicolasaunai in https://github.com/SciQLop/speasy/pull/83
- Adds bits for CDF support with AMDA server by @jeandet in https://github.com/SciQLop/speasy/pull/77

6.12 1.0.5 (2022-12-22)

- Drop LegacyVersion usage, fixes #78 by @jeandet in https://github.com/SciQLop/speasy/pull/79
- Replaces np.float by np.float64 since it was removed in numpy 1.24 by @jeandet in https://github.com/SciQLop/speasy/pull/81

6.13 1.0.4 (2022-12-05)

- [AMDA] Fix broken user product detection
- [AMDA] Add WS entry point in config
- Add tolerance for network failures
- Add option to disable webservices
- Fix cache issue with some CDF files

6.14 1.0.3 (2022-10-18)

- correct typo in README.rst
- uses cache setting also when loading inventory from proxy
- Matplotlib was accidentally working with DataContainer instead of Numpy array
- Amda csv read hardening
- also replace comma in dynamic inventory names

6.15 1.0.2 (2022-10-07)

- fixes regression on CSA inventory
- fixes rare issue on variable merge

6.16 1.0.1 (2022-10-06)

- several documentation improvements
- SpeasyVaraible can be sliced with numpy.datetime64
- comparing SpeasyVaraible with NaNs works as expected now (ignore NaNs)
- fixes cda inventory issue where some datasets were missing
- speasy loading time reduction by only downloading inventory from proxy if it has changed

6.17 1.0.0 (2022-09-25)

This is the first stable release of Speasy, this means that some part of the API won't change until next major release, they will only get bug fixes or backward compatible enhancements. Since last release, a lot of new features has landed:

- now Speasy fully support AMDA, CDAWeb, SSCWeb and CSA web-services which represent around 55000 products.
- for CSA and CDAWeb uses CDF file format thanks to pycdfpp and PyISTP speeds up download and allow 2D+ data handling
- for each web-service Speasy provides an inventory of available products
- for each web-service except SSCWeb, Speasy automatically discard outdated data from local cache
- get_data function has evolved to accept many complex combination of products and time intervals
- get_data function is now part of the stable API of Speasy
- on disk cache loading algorithm has been improved and is now at least 10x faster
- (unstable) plotting API is under heavy rework and will continue to evolve in next releases but already support spectrogram plots and handles as much as possible information such as axes label or units
- by default Speasy proxy is enabled (for new fresh installs)
- SpeasyVariable object has been rewritten to better handle ND data and provide nice slicing features

From now upcoming releases will mostly fix bugs, extend plotting API and follow web-services evolution.

6.18 0.10.0 (2022-02-03)

- Adds support for all AMDA products, even private ones
- Adds support for AMDA credentials
- Adds dynamic inventory for AMDA and SSC
- Adds possibility to set config values from ENV
- Drops Python 3.6 support and adds 3.10
- New API documentation using numpydoc
- New user documentation using numpydoc
- Most code examples are tested with doctest
- Renames SSCWeb module get_orbit to get_trajectory

6.19 0.9.1 (2021-11-25)

- Fix AMDA module bug #24 downloading multidimensional data fails

6.20 0.9.0 (2021-07-29)

- Adds SPWC migration tool
- Rename SpwcVariable to SpeasyVariable

6.21 0.8.3 (2021-07-28)

- Package renamed from SPWC to SPEASY
- Some doc and CI improvements

6.22 0.8.2 (2021-04-20)

- sscweb trajectories are always in km

6.23 0.8.1 (2021-04-18)

- Fixes minimum request duration for sscweb

6.24 0.8.0 (2021-04-18)

- Full support for trajectories and 0.2 proxy version

6.25 0.7.2 (2020-11-13)

- ccsweb/proxy: Fix missing coordinate system parameter

6.26 0.7.1 (2020-11-13)

- Fix project URL on PyPi

6.27 0.7.0 (2020-11-13)

- SSCWEB support to get satellites trajectories.
- Few bug fixes.
- Totally disabled cdf support for now.

6.28 0.1.0 (2019-12-07)

- First release on PyPI.

CHAPTER

SEVEN

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/SciQLop/speasy/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

Space Physics WebServices Client could always use more documentation, whether as part of the official Space Physics WebServices Client docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/SciQLop/speasy/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

7.2 Get Started!

Ready to contribute? Here's how to set up *SPEASY* for local development.

1. Fork the *SPEASY* repo on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/speasy.git
```

3. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv speasy
$ cd speasy/
$ python setup.py develop
```

4. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ make lint
$ make test-all
```

   To get flake8 and tox, just pip install them into your virtualenv.

6. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

7. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The pull request should work for Python from 3.6 to 3.9, and for PyPy. Check https://github.com/SciQLop/speasy/actions and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ py.test tests.test_speasy
```

7.5 Deploying

A reminder for the maintainers on how to deploy. Make sure all your changes are committed (including an entry in HISTORY.rst). Then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
```

GH Actions will then deploy to PyPI if tests pass.

CHAPTER

EIGHT

CREDITS

8.1 Development Lead

- Alexis Jeandet <alexis.jeandet@member.fsf.org>

8.2 Contributors

8.2.1 AMDA Webservice

- Alexandre Schulz <alexandre.schulz@irap.omp.eu>
- Benjamin Renard <benjamin.renard@irap.omp.eu>

Speasy is an open source Python client for Space Physics web services such as CDAWEB or AMDA. Most space physics data analysis starts with finding which server provides which dataset then figuring out how to download them. This can be difficult specially for students or newcomers, Speasy try to remove all difficulties by providing an unique and simple API to access them all. Speasy aims to support as much as possible web services and also cover a maximum of features they propose.

CHAPTER

NINE

QUICKSTART

Installing Speasy with pip (*more details here*):

```
$ python -m pip install speasy
# or
$ python -m pip install --user speasy
```

Getting data is as simple as:

```
import speasy as spz
ace_mag = spz.get_data('amda/imf', "2016-6-2", "2016-6-5")
```

Where amda is the webservice and imf is the product id you will get with this request.

Using the dynamic inventory this can be even simpler:

```
import speasy as spz
amda_tree = spz.inventory.data_tree.amda
ace_mag = spz.get_data(amda_tree.Parameters.ACE.MFI.ace_imf_all.imf, "2016-6-2", "2016-6-
↪5")
```

Will produce the exact same result than previous example but has the advantage to be easier to manipulate since you can discover available data from your favourite Python environment completion such as IPython or notebooks (might not work from IDEs).

This also works with SSCWEB, you can easily download trajectories:

```
import speasy as spz
sscweb_tree = spz.inventory.data_tree.ssc
solo = spz.get_data(sscweb_tree.Trajectories.solarorbiter, "2021-01-01", "2021-02-01")
```

More complex requests like this one are supported:

```
import speasy as spz
products = [
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_vth,
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_pdyn,
    spz.inventories.tree.amda.Parameters.Wind.SWE.wnd_swe_kp.wnd_swe_n,
    spz.inventories.tree.cda.Wind.WIND.MFI.WI_H2_MFI.BGSE,
    spz.inventories.tree.ssc.Trajectories.wind,
]
intervals = [["2010-01-02", "2010-01-02T10"], ["2009-08-02", "2009-08-02T10"]]
data = spz.get_data(products, intervals)
```

CHAPTER

TEN

FEATURES

- Simple and intuitive API (spz.get_data to get them all)
- Pandas DataFrame like interface for variables
- Quick functions to convert a variable to a Pandas DataFrame
- Local cache to avoid repeating twice the same request
- Can take advantage of SciQLop dedicated proxy as a community backed ultra fast cache
- Full support of AMDA API
- Can retrieve time-series from AMDA, CDAWeb, CSA, SSCWeb

CHAPTER

ELEVEN

EXAMPLES

See *here* for a complete list of examples.

Go to developers doc