

Polynomial Reconstruction of the Magnetic Field and Current Density - Python Implementation

v1.0

1 General

This software package implements a means of 3D magnetic field and current density reconstruction from multipoint measurements in Python. In its core it represents the method, first presented by Denton et al. 2020. For the software to work as intended, it comes with additional implementations of various other functions, procedures, algorithms or methods, which can be found in `prec_utilities.py`. The implementation is done by M. Hosner¹. A minimum working example with the file name `how_to_run_it.py` is provided with the software that shows reconstruction results of the example event, also shown in Denton et al. 2020. Disclaimer: The software package is delivered *as-is*. Use at own risk. This document serves as a guide and may not be exhaustive.

2 Package Dependencies

The code is developed and run, using the packages (and their version number) given in the table below. Version numbers of some packages that differ from the ones below, may work as well, but cannot be guaranteed. Subdependencies might not be mentioned explicitly, as the given packages require them to work anyway, and will be installed together.

3 Installation and Usage

The software is intended to be used as a package. That is, it can be used in any python script with the `import` command and it is not necessary for the user to change any files to apply the

¹martin.hosner@oeaw.ac.at

Package	Version
Python	3.8.6
Numpy	1.23.0
SciPy	1.8.1
pyspedas	1.3.6
pytplot	1.7.28
matplotlib	3.6.3

software to a specific event. For installation, copy all the files (but at least the files `Spacecraft.py`, `reconst.py`, `prec_utilities.py` and `__init__.py`) into a folder on your hard drive. The name of the folder will determine the name of the package in your python script, i.e. the name you use to import the package: `import FOLDERNAME`. In the minimum working example a folder name of `prec` is assumed. For the package to be recognized correctly by your python interpreter, the folder with these files must be located in a directory, where python searches for packages. To specify such directories, you can set the environment variable `PYTHONPATH`, which contains the directories, where the interpreter additionally searches for packages, next to the default path (see also <https://docs.python.org/3/using/cmdline.html#envvar-PYTHONPATH>). For instructions on how to set environment variables, please refer to your operating system's documentation. It is strongly advised to not copy it to the default installation directory (`../site-packages/`).

An example on how to use the software is shown in the minimal working example `how_to_run_it.py` with additional commenting on what the commands do.

4 The API

4.1 class Spacecraft

Since the magnetic field is reconstructed from multipoint spacecraft measurement data, it is convenient to define a class for a spacecraft mission, that holds the later used data. As of now, the code supports the MMS mission. In addition, a virtual spacecraft mission is supported that is able to deal with simulation data. All mission classes are derived from an *abstract* Spacecraft class, from which they inherit common methods. As such, the Spacecraft class is not meant to be used directly. A spacecraft object is instantiated, using at least a time range and the `probe` identifier. Additionally, the spacecraft classes are able to store and load data in a given directory.

method `__init__(self, trange, probe, noPlasma = False)`

Input	
<code>trange</code>	(2; iterable of string or float)
<code>probe</code>	int
<code>noPlasma</code>	bool
Output	
-	

Abstract Spacecraft class initialization method. Not meant to be called directly, but it will be called by the subsequent mission initialization methods. `trange` (iterable of two time points, which can be either string or float) is the time range that is used to load data, `probe` (int) is the probe identifier and `noPlasma` (bool) can be used to manually set the electron moments to `None` which results in no current data. For benchmark uses.

method `running_avg(self, window_B = 0.001, window_plasma = 0.3)`

Input	
<code>window_B</code>	float or None
<code>window_plasma</code>	float or None
Output	
-	

It is used to smooth the data, using the `scipy.ndimage.uniform_filter1d` function. The arguments are the smoothing window duration in seconds for the magnetic field `window_B` (float) and plasma `window_plasma` (float) data, respectively. No return value, instead stores the result in `self.B_avg`, `self.ne_avg`, `self.ni_avg`, `self.ve_avg`, `self.vi_avg` and `self.r_avg` where `self.r_avg` is only a copy of `self.pos`. Vector quantities are of dimension (`len(self.t_res)`, 3) and scalars of (`len(self.t_res)`). Any of the arguments can be `None`, in that case no smoothing is done but the variables are copied as they are. For the electron and ion velocities, the spin tone is subtracted in this method.

method `resample(self, trange = None, sample_rate = 0.001, ni_density = True)`

Input	
<code>trange</code>	(2; iterable of string or float)
<code>sample_rate</code>	float
<code>ni_density</code>	bool
Output	
-	

The data is resampled to a new time vector which is defined with the `trange` (iterable of two time points, which can be either string or float) time range and `sample_rate` in seconds (float). For the method to work, it requires `running_avg` to be invoked before as the `self.*_avg` attributes are used. The function has no return value, instead the resampled quantities are stored in attributes with the appendix `*_res`. The respective quantities are the magnetic field `self.B_res`, the electron and ion densities `self.ne_res`, `self.ni_res`, the electron and ion velocity `self.ve_res`, `self.vi_res` and the S/C position `self.r_res`. The method creates an additional attribute `self.J_part` which stores the current, calculated by $J = (n_i v_i - n_e v_e) / (1.602E1)$ in [nA/m²]. If `ni_density` is `False`, it assumes the ions have the same density as the electrons. Every quantity is resampled to the common time vector `self.t_res`. If `trange = None`, the whole time vector, which with the object was initialized, is used. If the spacecraft is MMS4 and the date is later than 2018-06-07/00:00UT, then the electron moments will be set to `None` and therefore the current is `None` as well.

4.2 class MMS(Spacecraft)

This class is meant to be initialized by the user. It inherits the methods from the `Spacecraft` class.

method `__init__(self, trange, probe, load = None, noPlasma = False)`

Input	
<code>trange</code>	(2; iterable of string or float)
<code>probe</code>	int
<code>load</code>	string or None
<code>noPlasma</code>	bool
Output	
-	

Used to initialize a MMS spacecraft. In addition to the Spacecraft `__init__` method, it has a key-word argument `load` (string) that can be a file path to a saved MMS data object. (See `save_data` method).

method `reload_data(self)`

Input	
-	
Output	
-	

Is used to reload data for that spacecraft by using the pyplot interface and overwrites data that is stored in that class. As the time range, the `trange` variable is used. No return values

method `save_data(self, path)`

Input	
<code>path</code>	string
Output	
-	

Stores the currently loaded tplot variables of the relevant data in a tplot savefile at the given path, using the pyplot interface. If the path does not exist, it will be created. These file can be restores using the `reload_data` method to load the data. Caution: This method does not save the actual stored data but the content of the tplot variables, i.e. if the data is changed, it won't be stored.

4.3 class Virtual

The `Virtual` spacecraft class is similar to the Spacecraft/MMS class, however does not inherit from the Spacecraft class due to simulation data being different from in-situ data. Since for the reconstruction only the magnetic field, position and current is needed, the initialization is slightly different. The `Virtual` class holds the following methods

@staticmethod load(path, trange, probe, noPlasma = False)

Input	
path	(2; iterable of string or float)
probe	int
noPlasma	bool
Output	
Virtual	

This method is invoked directly from the **Virtual** class, rather than from an object instance. It returns a **Virtual** object instance. The parameters are as above.

method __init__(self, trange, probe, B, j, r, noPlasma = False)

Input	
trange	(2; iterable of string or float)
probe	int
B	(2; list or tuple)
j	(2; list or tuple)
r	(2; list or tuple)
noPlasma	bool
Output	
-	

Initialization method for a **Virtual** spacecraft. **probe** (int) is the probe identifier, **B** the magnetic field (tuple, with **B[0]** being a time vector and **B[1]** being the data vector with dimension (t,3)). Similar to **B**, the variables for the position **r**, and the current **j**, must be given. This data should be like in-situ data, i.e. the data that a spacecraft would measure over time when it flies through the simulation domain. It defines the attributes for the magnetic field **self.fgm_t**, **self.fgm** (and aliases thereof, which are called **self.B_t**, **self.B**), for the current **self.J_t**, **self.J**, and the position **self.pos_t**, **self.pos**, and has no return value.

method save_data(self, path)

Input	
path	string
Output	
-	

Similar to the **Spacecraft** classes, it stores data in tplot variables and saves it at **path**. Restore it with the **load** staticmethod (see above).

method running_avg(self, window_B = 0.001, window_plasma = 0.3)

Input	
window_B	float or None
window_plasma	float or None
Output	
-	

This method is similar to the `Spacecraft/MMS` class. The input parameters are the same as above. It defines the attributes for magnetic field `self.B_avg`, for the current `self.J_avg`, and position `self.r_avg`. The time vector for this attributes are not changed, i.e. they have the same dimension in time as the non-smoothed data. There is no return value. If any of the values is `None`, no smoothing is performed.

method `resample(self, trange = None, sample_rate = 0.001)`

Input	
<code>trange</code>	(2; iterable of string or float) or None
<code>sample_rate</code>	float
Output	
-	

The smoothed data is resampled to a time vector and sample rate, that can be passed as arguments. The argument `trange` (list with two time entries (float, string)) defines the time range. If `trange = None`, the whole time vector, which with the object was initialized, is used. `sample_rate` (float) is the sample rate in seconds. For the method to work, it requires `running_avg` to be invoked before as the `self.*_avg` attributes are used.

4.4 class `Prec`

In this class the actual reconstruction is implemented. It takes a list of previously defined `Spacecraft` (i.e. `MMS` or `Virtual`) objects and a time range, which should be used to calculate the reconstruction, LMN system etc. of not otherwise stated. In the following, the writing `Spacecraft` explicitly means the abstract spacecraft class, whereas the writing `Spacecraft` means a spacecraft object (or class) in general and can be either an `MMS` or `Virtual` object.

method `__init__(self, tbeg, tend, *sc)`

Input	
<code>tbeg</code>	float
<code>tend</code>	float
<code>sc</code>	Spacecraft
Output	
-	

Initialization method. The parameters `tbeg` (float) and `tend` (float) define `self.trange` (tuple) that is used for all operations, like determining the LMN system, reconstruction, etc. if not otherwise specified at the respective position. The `*sc` accepts an arbitrary amount of `Spacecraft` objects and stores it in `self.sc` (list). For the reconstruction, the data from the spacecraft in `self.sc` will be used. Whereas for the MMS mission, it is obvious that there are only for `Spacecraft` at maximum, for virtual spacecraft in a simulation it might be useful to have more than four. The method additionally defines the attribute `self.f` (float) that is later used as a scaling factor for the FPI-based current. It also defines the attribute `self.J_fit_modif = 1` (float), that additionally scales the contribution of the current to the cost function. It is experimental legacy code and should not be changed.

method `save_data(self, path)`

Input	
<code>path</code>	string
Output	
-	

Wrapper method for the Spacecraft `save_data` method, that stores the data of all the probes at ones. For the `path` argument, see the `save_data` method of the Spacecraft classes.

method `data_prep(self, window_B = 0.001, window_plasma = 0.3)`

Input	
<code>window_B</code>	float or None
<code>window_plasma</code>	float or None
Output	
-	

A wrapper method that passes on the arguments to, and calls the Spacecraft's `running_avg` and `resample` methods on all the Spacecraft in `self.sc`. For the arguments description, see the respective Spacecraft method's section.

method `calf_f(self, external = None)`

Input	
<code>external</code>	float or None
Output	
-	

This method calculates the correction factor `self.f` (float) that is introduced to adjust for magnitude differences between the plasma particle based current and the curlometer-based current (Dunlop et al. 2002). With the parameter `external` (float), the attribute `self.f` can be set and the calculation is ignored. The calculation is performed according to

$$f = \text{median} \left(\frac{\sum_t J_{\text{part},t} J_{\text{curl},t}}{\sum_t J_{\text{part},t}^2} \right), \quad (1)$$

Where $J_{\text{part},t}$ denotes the t -th time point of the particle based current and $J_{\text{curl},t}$ the t -th time point of the curlometer-based current. Then f is taken as the correction factor `self.f`. Since the factor f is calculated for each spacecraft individually at this point, `self.f` is still of length `len(self.sc)` but the mean is applied at another point (see `transform_data` method below).

method `define_LMN(self, trange = None, external = None, raw = False)`

Input	
<code>trange</code>	(2; iterable of string or float) or None
<code>external</code>	(3, 3) <code>np.array</code> or None
<code>raw</code>	bool
Output	
-	

Here the boundary layer coordinate system LMN is defined. It can either be calculated or an external system can be used. If an external LMN system should be used, a (3, 3) `np.array` must be passed to `external`, where the column vectors represent L, M and N, respectively. If `external` is `None` then the LMN system is calculated based on the Minimum Directional Derivative (MDD) (Shi et al. 2005) and Minimum Variance Analysis (MVA) (Sonnerup and Cahill 1967; Sonnerup and Scheible 1998) method, as described in (Denton et al. 2016; Denton et al. 2020) and also summarized in this section. The `trange` argument (length-two iterable of float/string) can be passed to specify a time range that should be used to determine the coordinate system. The method defines `self.LMN (3, 3; np.array)` attribute that holds the LMN system. For the LMN calculation, first the magnetic field vectors of all Spacecraft are concatenated component-wise, and the MVA is applied to that. If `raw` is True, the un-resampled and un-smoothed magnetic field data are used for the calculation of LMN. We take the eigenvector of largest corresponding eigenvalue as the L direction. Second, we apply the Minimum Directional Derivative method Shi et al. 2005 but we sum up the matrices from each time point and the eigenvectors and eigenvalues from that summed matrix. The eigenvector with largest corresponding eigenvalue is projected onto the plane, that is perpendicular to the L vector, rescaled to unity and used as the N vector. The M vector is the third vector that completes the right-handed system via $L \times M = N$. The method additionally defines the attributes `self.B_sc_LMN` and `self.J_sc_LMN (len(self.t_res), 3, len(self.sc); np.array)`, with the dimension time, LMN components, spacecraft, i.e. the magnetic field and current density in LMN components for each spacecraft. The method has no return value.

method `get_lmn(self, t = None, external = None)`

Input	
<code>t</code>	(*,) <code>np.array</code> or None
<code>external</code>	(3, 3) <code>np.array</code> or None
Output	
-	

Calculates the time-dependent *lmn* coordinate system. The argument `t` (`np.array` of float) determines the times, when the vector should be calculated. If `None`, then `self.trange` is used as `t`. A constant external *lmn* system can be used, when passed as the `external` argument as (3,3; `np.array`), with the vectors as column vectors, similar as for the `get_LMN` method. If no external system is used, the vectors are calculated at each time point using the MDD (Shi et al. 2005) method. The method calculates a matrix for each time point, that represents the *lmn* vectors as column vectors in two ways: `self.e_lmn (len(t), 3, 3; np.array of float)` represents the coordinate system in GSE coordinates, and `self.e_lmn_ (len(t), 3, 3; np.array of float)` represents the *lmn* system in LMN coordinates. According to the notation in Denton et al. 2020 the *l* component represents the intermediate gradient direction, *n* the maximum gradient direction and *m* the minimum gradient direction.

method `GSE_to_LMN(self, v, mode = 'GSE_to_LMN')`

Input	
<code>v</code>	$(*, 3, n)$ or $(3, *)$ or $(*, 3)$ <code>np.array</code>
<code>mode</code>	string
Output	
<code>transformed_vec</code>	<code>np.array</code> with same shape as <code>v</code>

Convenience method to transform data from GSE to LMN and vice versa. If no LMN system has been defined yet, the method calls `self.get_LMN()`. The argument `v` is the input vector. It expects a `np.array` and can deal with various dimensions: First, the dimension can either be $(t, 3)$ or $(3, t)$, e.g. the usual (or transposed) tplot-style dimensions and it determines which axis corresponds to the components. Second, it can be $(t, 3, n)$, e.g. you have many time points and various spacecraft all in one array. Here, t stands for the amount of time points and n for different points in the grid at one certain time point, which can be e.g. different s/c locations. The `mode` (string) argument determines the direction of transformation and can either be 'GSE_to_LMN' or 'LMN_to_GSE' for back-transformation. The method returns the transformed quantity in the same shape as `v`.

method `GSE_to_lmn(self, v, t = None, mode = 'GSE_to_lmn')`

Input	
<code>v</code>	$(*, 3, n)$ or $(3, *)$ <code>np.array</code>
<code>t</code>	<code>np.array</code> or None
<code>mode</code>	string
Output	
<code>transformed_vec</code>	<code>np.array</code> with same shape as <code>v</code>

Similar to `GSE_to_LMN` but performs the calculation with the time-dependent *lmn* system. The argument `v` (`np.array`) is expected to be of shape $(3,n)$ or $(t,3,n)$. The argument `t` (`np.array`) is an alternative time vector at which the *lmn* system should be calculated, before the transform is applied to `v`. The size of `t` must therefore be the same as the time dimension of `v`. The method returns the transformed quantity in the same shape as `v`. The `mode` (string) argument can be either 'GSE_to_lmn' to transform from GSE to *lmn* or 'lmn_to_GSE' for the respective back-transformation from *lmn* to GSE.

method `transform_data(self, times = None)`

Input	
<code>times</code>	iterable of float or string or None
Output	
-	

This method prepares and interpolates all the data that is needed prior to the actual reconstruction job. It is called automatically during the reconstruction and does not need to be called by the user. With the argument `times` (`np.array` of float), a time vector can be specified in case the reconstruction is not performed at all times. If `None` then the data is prepared for all times `self.t_res`.

0	B_l	$\partial B_l/\partial n$	$\partial B_l/\partial l$	$\partial B_l/\partial m$	$\partial B_l^2/\partial n^2$	$\partial B_l^2/\partial n\partial l$	$\partial B_l^2/\partial l^2$	$\partial B_l^2/\partial n\partial m$	$\partial B_l^2/\partial l\partial m$
1	B_m	$\partial B_m/\partial n$	$\partial B_m/\partial l$	$\partial B_m/\partial m$	$\partial B_m^2/\partial n^2$	$\partial B_m^2/\partial n\partial l$	$\partial B_m^2/\partial l^2$	$\partial B_m^2/\partial n\partial m$	$\partial B_m^2/\partial l\partial m$
2	B_n	$\partial B_n/\partial n$	$\partial B_n/\partial l$	$\partial B_n/\partial m$	$\partial B_n^2/\partial n^2$	$\partial B_n^2/\partial n\partial l$	$\partial B_n^2/\partial l^2$	$\partial B_n^2/\partial n\partial m$	$\partial B_n^2/\partial l\partial m$
	0	1	2	3	4	5	6	7	8

Table 1: The indices of the corresponding solution vector. The first dimension runs over the time points (not shown here, as it depends on the user input), the second dimension runs over lmn (3) and the third dimension runs over the derivative variables (9).

method Q3D(self, times = None, constr = None)

Input	
times	iterable of float or string or None
constr	list
Output	
calculate_grid	function
calculate_grid_J	function

In this function the actual Full Quadratic Model reconstruction is implemented. Here the model coefficients at the given time points are determined and functions to calculate the magnetic field (`calculate_grid`) and current density (`calculate_grid_J`) at arbitrary points in space are generated and returned. The input parameter `times` (list) contains time points where the model should be solved. Note: this is only the time when the model is solved, but the solution can be evaluated at different time points. In this case the solution vector will be interpolated to that time point and may therefore be inaccurate. If `None` then all time points of `self.t_res` are used (can take a long time). Via the parameter `constr` (list) additional constraints for the minimize function can be passed. Each entry must be a dictionary of the form, further described in the `scipy.optimize.minimize` documentation. The overall structure is as follows:

1. the data is interpolated and the lmn system is determined at the respective time points, by calling `transform_data`.
2. for each time point the model function and the constraint functions are generated via a closure, named `generate_model`. The `generate_model` function is called at each time point and returns, again, several functions; the cost function itself, called `model_system(P)` that is minimized, and the constraint functions (i.e. the divergence terms) `divB_const(P)`, `divB_l(P)`, `divB_m(P)` and `divB_n(P)`. They all take the vector `P` (27; `np.array`) as an input argument, which represents the minimization parameters.
3. The cost function is minimized, using the `scipy.optimize.minimize`, which in turn uses a constrained linear least-squares solver. When the result is found, it will be stored in `self.result_Q3D` (`len(self.t_rec)`, 3, 9; `np.array` with float entries), i.e. the result of the minimization is stored for each time point.
4. Based on this solution two functions are generated and returned, that allow to reconstruct the magnetic field and current density at a given grid point and time point. The functions are named `calculate_grid(grid, t)` and `calculate_grid_J(grid, t)`. This allows to evaluate the magnetic field and current not at fixed locations but at arbitrary points.

The first argument is `grid`, which is $(3, n; \text{np.array})$, where the first dimension is X,Y,Z in GSE and the second dimension iterates over different grid points n . That is, every point at which the solution should be evaluated, must be concatenated. To evaluate the solution on a grid, simply create a meshgrid to obtain the X, Y, Z values of a grid and then flatten the arrays and put it together to a $(3, \text{nr_grid_points}^3; \text{np.array})$.

The second argument `t` is the time point to evaluate. Note that if `t` is not in `times`, the function interpolates the solution vector to the given time point and is therefore inaccurate for time points at which the solution was not calculated.

The solution vector `self.result_Q3D` contains the parameters of the Taylor expansion for each time point in `times`. It is of shape $(\text{len}(\text{times}), 3, 9)$. The first dimension runs over the time points. The remaining two dimensions are of shape $(3, 9)$ and contain the derivatives according to Tab.1. The physical dimensions of the parameters are nT, nT/km and nT/km^2 , respectively.

method `RQ3D(self, times = None, constr = None)`

Input		
<code>times</code>	iterable of float or string or None	
<code>constr</code>		list
Output		
<code>calculate_grid</code>		function
<code>calculate_grid_J</code>		function

In this function the Reduced Quadratic model reconstruction is implemented. It is implemented the same as the Q3D model, except the solution vector is called `self.result_RQ3D`.

method `QLm3D(self, times = None, constr = None)`

Input		
<code>times</code>	iterable of float or string or None	
<code>constr</code>		list
Output		
<code>calculate_grid</code>		function
<code>calculate_grid_J</code>		function

In this function the Quadratic model with linear m-dependence is implemented. It is implemented the same as the Q3D model, except the solution vector is called `self.result_QLm3D`.

method `show_list(self, reconst = {'model': 'RQ3D', 't': None}, units = None, orientation = 'LN', dist = 3., plane = 0, clim = {}, show_current_sheet = False, fig_format = {}, save = None, streamplot_config = {}, strmln_kwargs = {}, noplot = False, show_outflow = False)`

Input	
<code>reconst</code>	dict
<code>units</code>	string or None
<code>orientation</code>	string
<code>dist</code>	float
<code>plane</code>	float/list
<code>climit</code>	dict
<code>show_current_sheet</code>	bool
<code>fig_format</code>	dict
<code>save</code>	bool
<code>streamplot_config</code>	dict
<code>strmln_kwargs</code>	dict
<code>noplot</code>	bool
<code>show_outflow</code>	bool
Output	
<code>reconstructed_field</code>	<code>np.array</code>

This method is one of the most important and is the main interface for the user when it comes to performing the actual reconstruction. It is essentially a wrapper that creates the desired grid, performs the reconstruction according to the passed arguments and creates the figures and returns the reconstructed values. It is called `show_list` because it takes a list of time points, when the reconstruction should be performed and displayed. The input variables will be discussed in the following. Also, see the doc string in the code for reference.

- **reconst** is a dictionary that is used to set up the reconstruction. It specifies the used model, grid size, time points, etc. This dict contains the fields (all of type string)
 - **'model'**, string, can be **'Q3D'** (full quadratic model), **'QLm3D'** (quadratic model with linear m-dependence), and **'RQ3D'** (reduced quadratic model).
 - **'t'**, list, time points (of any type that is accepted by `pyspedas.time_double()`), when the reconstruction should be performed. The function therefore creates subplots for each entry of **'t'**.
 - **'B'**, function, Instead of the **'model'** string there can also be a function passed directly (with the same signature as `calculate_grid`). If **'B'** is given, **'model'** is ignored. *Note: Not really necessary and was added for testing.*
 - **'oop'**, string, stands for out-of-plane and specifies, which physical quantity should be displayed by the color code. Can either be **'B'** to display the magnetic field, or **'j'** to display the current density.
 - **'cc_comp'**, int, stands for color-coded component and specifies, which component (L,M,N) should be displayed as the background color, can be either 0 (L component), 1 (M component, default) or 2 (N component). This allows for all kinds of variations and doesn't have to be the out-of-plane component.
 - **'kwargs'**, dictionary, any additional arguments that should be passed to the reconstruction function (such as e.g. custom constraints)
 - **'res'**, int, the grid resolution.

- **units**, float, the units in which the spatial axes will be displayed in. If None, the average spacecraft separation is used.
- **orientation**, string, specifies the components that are shown in the figures. The first letter is the horizontal axis, the second one the vertical one. For example 'LN' will display the reconstruction result with the L component on the horizontal axis and N component on the vertical axis. Accepts all combinations of L,M,N. An axis can be reversed with a minus sign, e.g. 'L-N', '-NM', '-L-M' to reverse the second axis, first, or both.
- **dist**, float, the subfigure limits in units of **units**.
- **plane**, float/list, the cut in the out-of-plane direction. If e.g. the LN plane is shown, then this determines the shown M coordinate. In case it is a float, the same plane is shown for every panel. In case it is a list (or **np.array**) (of floats), the entries correspond to the coordinate of each subfigure. Consequently, the length must then match the length of **t**.
- **climit**, dict, color-code configuration dictionary. It will be passed to the **ax.pcolormesh** method and therefore can contain fields which are accepted by that method. It additionally contains 'vmin' and 'vmax' to set the color-code limits and 'cmap' to choose the colormap.
- **show_current_sheet**, bool, enables a search algorithm to find where the magnetic field reverses in L and N to identify an X-point and O-lines in the reconstructed pictures. The X-point is displayed as an orange X, the O-line is represented by a red diamond. For the internal algorithm the transition region between positive/negative L is smoothed with a spline to avoid errors due to a too coarse grid. Along that reversing line, a reversal in N is searched. Depending on the how the sign changes the point is identified as an X- or O-line. The X-line position is stored in the variable **self.cs_pos** (**len(t)**, 2; **np.array**), with the first index running over the time points, the second one over L, N. It additionally creates the variables **self.cs_fit_params** (**len(t)**, 2; **np.array**) that contains fit parameters of a linear function fit (first entry is slope, second one the offset), and **self.t_cs** and **self.n_cs** that both have the shape (**len(t)**, 2; **np.array**) and represent the unit vector of the tangential and normal direction of the current sheet in LN coordinates. It is obvious that these require a more or less geometrically "straight and level" current sheet to be reasonable since it is determined by a linear fit.
- **fig_format**, dict, holds keys for configuring the figure such as the geometry, size, alignment of subfigures, etc. The dictionary holds keys (all string):
 - 'columns', int, controls how the subfigures are aligned by passing the number of columns of the subfigure grid.
 - 'units_label', string, what is displayed as the axis label.
 - 'sc_pos', bool, If True, a second color bar is displayed on the left of the figure that indicates the spacecraft out-of-plane coordinate. The otherwise white boundary if the spacecraft symbol will then be coloured according to its out-of-plane coordinate relative to the panel out-of-plane coordinate.
 - 'fig_size', tuple with floats, The overall figure size in inches.

- `'wspace'`, `'top'`, `'bottom'`, `'right'`, (float), will be passed to `ax.subplots_adjust` and controls the padding width between columns, distance from the figure top, bottom and right border, respectively.
- `'save'`, dict, if not None, the figure will be saved according to the passed parameters. The dict should contain a field `'fname'`, containing the file path. It can contain additional keyword arguments, which will be passed to the `fig.savefig` method.
- `streamplot_config`, dict, can contain additional configuration parameters for the `streamplot` routine.
- `strmln_kwargs`, dict, additional keyword arguments that will be passed to the custom field line highlighting function. See the documentation for `streamline2D_symmetric` for what can be passed.
- `noplot`, bool, if True, then no figure will be drawn but only the underlying calculation will be performed.
- `show_outflow`, bool, initiates an algorithm to find separatrix field lines in a reconnection geometry and calculates opening angles based on geometric considerations.

The method has the additional feature of highlighting single field lines by clicking onto a desired location in a panel while holding the `'b'` key on the keyboard. The field line in this particular subfigure will be highlighted in orange, based on integrating forward and backward starting from the clicked point.

4.5 Utility functions

For the reconstruction to work as intended, it comes with various additional utility functions. Although implemented by the author, they may rely on other packages and/or may contain intellectual property of others (e.g. they implement algorithms, initially developed by others).

function `ninterp(xnew, x, y, **kwargs)`

Input	
<code>xnew</code>	iterable of new x vector
<code>x</code>	iterable of old x vector
<code>y</code>	iterable of old y array
Output	
<code>resampled_array</code>	<code>(len(xnew), *y.shape[1:]) np.array</code>

Recursively calls itself to interpolate `y` to a new `x` vector `xnew`. The recursion allows efficient interpolation of arbitrarily-shaped arrays. Every dimension is interpolated using `numpy.interp`, to which additional `**kwargs` can be passed. It is interpolated along the first dimension.

function minvar(input)

Input	
input	(n, 3) np.array
Output	
ew	(np.array ; 3) the eigenvalues in descending order of magnitude
ev	(np.array ; 3, 3) associated eigenvectors as column vectors
transformed_data	np.array ; shape is same as input

Custom implementation of the Minimum Variance analysis (MVA) Sonnerup and Cahill 1967; Sonnerup and Scheible 1998. Returns eigenvalues, eigenvectors and the transformed input vector into the minimum variance system.

function MDD(data, pos, mode = 'each')

Input	
data	list, containing the s/c data
pos	list, containing the s/c position
mode	string or array
Output	
ew	np.array ; (shape depends on 'mode') the eigenvalues in descending order of magnitude
ev	np.array ; (shape depends on 'mode') associated eigenvectors as column vectors

Calculates the Minimum Directional Derivative based on Shi et al. 2005. The **data** input is a list that contains the data of each spacecraft. Each list entry should be a **np.array** of shape (timepoints, data-dimension). The **pos** input is similar but contains the spacecraft position. It is therefore of shape (timepoints, 3). The **mode** keyword determines how the timepoints are treated. If **mode** is 'each', the MDD system is calculated for each time point separately and the output will be of shape (timepoints, 3) and (timepoints, 3, 3), respectively. If **mode** is 'event' then the MDD matrices of each time point will be summed up and the eigenvalues and eigenvectors are returned. The output shape is then (3,) and (3,3) respectively. The **mode** keyword can also be an array of int entries. In that case the MDD is calculated similar to 'event', but it splits the whole data in subintervals, split at the indices, given by the array **mode**. The shape is then (subintervals, 3) and (subintervals, 3, 3), respectively.

function norm(V, axis = 1)

Input	
V	np.array
axis	int
Output	
norm	(shape depends on input) np.array

Calculates the norm of a vector **V**. If the dimension of **V** is 1, the output shape is (1,). If the dimension of **V** is larger than 1, the norm is calculated along **axis** and the shape of the output is the same as **V**, except reduced by the **axis** dimension.

function normV(V, axis = 1)

Input	
V	np.array
axis	int
Output	
norm	(same shape as V) np.array

Calculates the unit vector of V, assuming the dimensions of the vector to be in the second dimension.

function normM(V, axis = 1)

Input	
M	np.array
Output	
output	(same shape as M) np.array

Convenience function, that calculates the unit vectors from column vectors of a matrix for each time point. The matrix M is assumed to have the dimensions (timepoints, rows, columns). Output is the same shape as M.

function rotate(v)

Input	
v	(m, n, 3) or (m, 3) np.array
Output	
output	(same shape as v) np.array

Rotates a time dependent vector or a set of vectors (like e.g. a coordinate system) v such that they always point more parallel to the positive directions than anti-parallel. E.g. if a time-dependent coordinate system, with its first unit vector being roughly parallel to X GSM but sometimes also anti-parallel, it will be rotated (or rather flipped) such that the first unit vector's X component is always positive and the other vectors are rotated too, such that the chirality is conserved (e.g. if it was right-handed before, it will still be righthanded after the operation).

function streamline2d_symmetric(f, start, endif, step_len = 1.,
max_step = 'both', close_error = False)

Input	
v	function
start	(2,) np.array
endif	([xmin, xmax], [ymin, ymax]) tuple
step_len	float
max_step	int on None
close_error	bool
Output	
output	list

Calculates a streamline by stepwise following a 2D vector field, defined by f. The function f

must have the signature $f(\mathbf{r}) \rightarrow \mathbf{V}$, where \mathbf{r} is a (2,) `np.array`, representing the position of where to evaluate the vector field, and must return the vector \mathbf{V} , which is the vector field value at that position, as a (2,) `np.array` object. The parameter `start` is a (2,) `np.array` and defines the starting point of the streamline. The `endif` argument is a tuple and defines boundary values in each direction, where the algorithm stops for that direction, if the field line encounters any of those values. If not None, the `max_step` determines the maximum amount of steps that the algorithm performs before terminating. Depending on the vector field and `step_len` it could otherwise take a very long time. The parameter `close_error` determines if an error is raised if the field line comes close to itself within a step length (to e.g. prevent intersecting itself). The function returns a list of (2,) `np.array`s, each entry representing a point of the streamline.

function time_to_xaxis(t_vec)

Input	
t_vec	float or string
Output	
output	(len(t_vec),) <code>np.array</code>

Converts time from MMS data, given either as string or float, to datetime objects, that can directly be passed to matplotlib's plot commands, which then automatically displays them in the correct time stamp format.

function makemask(A, L = None, U = None)

Input	
A	<code>np.array</code>
L	float or None
U	float or None
Output	
output	(same shape as A) <code>np.array</code>

Returns a `np.array` with boolean entries. The respective entries are True if the entry in A at the respective indices is between (incl.) L and (excl.) U, otherwise False. Can be used to mask (i.e. cut) certain values of a vector to only use values within a certain range.

function jcurlom(*input)

Input	
*input	(t,3 3) <code>np.array</code> or two lists
Output	
output	(t, 3) <code>np.array</code>

Calculates the current density based on the "Curlometer" technique (Dunlop et al. 2002). The `input` must either be a `np.array` of the shape (t, 3, 3), which represents the vector gradient (the jacobian) or two lists. In the latter case the first entry must be a list of four (t, 3) `np.array`s of the magnetic field data, one for each spacecraft, and the second one again a list of four (t, 3) `np.array`s with the position information of each spacecraft. That is, `input` = B, R with B = [B_MMS1, B_MMS2, B_MMS3, B_MMS4] and R = [R_MMS1, R_MMS2, R_MMS3, R_MMS4].

function `closest2d(a, b)`

Input	
<code>a</code>	(n,2) <code>np.array</code>
<code>b</code>	(m,2) <code>np.array</code>
Output	
<code>out</code>	tuple

The input arrays `a` and `b` represent curves in 2D. It returns the indices of those points of the two arrays that are closest to each other in euclidean distance.

function `colorize(values, cmap = 'jet', norm = 'lin', limits = None)`

Input	
<code>values</code>	(n,) <code>np.array</code>
<code>cmap</code>	(m,2) string
<code>norm</code>	(m,2) string
<code>limits</code>	(2,) floats or None
Output	
<code>rgba</code>	(n,) <code>np.array</code>
<code>sm</code>	<code>matplotlib.cm.ScalarMappable</code>

Convenience function that associates the data `values` to corresponding `rgba` values of a given colormap (`cmap`) and returns the `matplotlib.cm.ScalarMappable` object. See the `matplotlib.cm` documentation for more information.

function `from_static_box(t, field, r_field, rbeg, rend)`

Input	
<code>t</code>	(t,) <code>np.array</code>
<code>field</code>	(npoints, quantity_dim) <code>np.array</code>
<code>r_field</code>	(npoints, 3) <code>np.array</code>
<code>rbeg</code>	(3,) <code>np.array</code>
<code>rend</code>	(3,) <code>np.array</code>
Output	
<code>out</code>	tuple

Creates a time series signal from a 3D box filled with data, e.g. a simulation result. It simulates a spacecraft that flies through a 3D grid of data and returns the signal as it would be measured by the spacecraft. The vector `t` is the time vector, the `field` and `r_field` parameters are the values at the grid points of the measured quantity and the physical position of the grid points in x,y,z, respectively (the grid is flattened). The parameters `rbeg` and `rend` are the beginning and ending positions of the trajectory in the grid in x,y,z respectively.

Iterator class `Combinations`

Input	
<code>List</code>	list
Output	
<code>self</code>	<code>Combinations</code>

Iterator that returns all possible 2-combinations of the entries in `List`. If two same entries appear twice in the list, they are treated as unique, i.e. it is not checked whether two entries refer to the same object in memory. Treats (x,y) as (y,x): no double combinations.

References

- Denton, R. E. et al. (2016). “Motion of the MMS spacecraft relative to the magnetic reconnection structure observed on 16 October 2015 at 1307 UT”. In: *Geophysical Research Letters* 43 (11). DOI: 10.1002/2016GL069214.
- Denton, R. E. et al. (2020). “Polynomial reconstruction of the reconnection magnetic field observed by multiple spacecraft”. In: *Journal of Geophysical Research: Space Physics* 125. DOI: 10.1029/2019JA027481.
- Dunlop, M. W. et al. (2002). “Four-point Cluster application of magnetic field analysis tools: The Curlometer”. In: *Journal of Geophysical Research: Space Physics* 107 (A11). DOI: 10.1029/2001JA005088.
- Shi, Q. Q. et al. (2005). “Dimensional analysis of observed structures using multipoint magnetic field measurements: Application to Cluster”. In: *Geophysical Research Letters* 32. DOI: 10.1029/2005GL022454.
- Sonnerup, B. U. Ö. and L. J. Cahill (1967). “Magnetopause structure and attitude from Explorer 12 observations”. In: *Journal of Geophysical Research* 72 (1), pp. 171–183. DOI: 10.1029/JZ072i001p00171.
- Sonnerup, B. U. Ö. and M. Scheible (1998). *Minimum and Maximum Variance Analysis*. Vol. 1. ISSI Scientific Report Series, pp. 185–220.