talon Documentation

Release 0.2.0

Samuel Deslauriers-Gauthier, Matteo Frigo, Mauro Zucchelli

Jan 20, 2021

INSTALL AND GET STARTED

talon is a pure Python package that implements Tractograms As Linear Operators in Neuroimaging.

The software provides the `talon` Python module, which includes all the functions and tools that are necessary for **filtering** a tractogram. In particular, specific functions are devoted to:

- Transforming a tractogram into a linear operator.
- Solving the inverse problem associated to the filtering of a tractogram.
- Perform these operations on a GPU.

The package is available at Pypi and can be easily installed from the command line.

```
pip install cobcom-talon
```

Talon is a free software released under MIT license and the documentation is available on Read the Docs.

CHAPTER

ONE

GETTING HELP

The preferred way to get assistance in running code that uses `talon` is through the issue system of the Gitlab repository where the source code is available. Developers and maintainers frequently check newly opened issues and will be happy to help you.

CHAPTER

TWO

CONTRIBUTING GUIDELINES

The development happens in the `devel` branch of the Gitlab repository, while the `master` is kept for the stable releases only. We will consider only merge requests towards the `devel` branch.

2.1 Installation

Talon runs only on Python 3. The installation has the following dependencies:

- Numpy
- Scipy
- NiBabel
- PyUnLocBox
- PyOpenCL (only if you plan to exploit the GPU capabilities)

If you are an Anaconda user, you may want to create a dedicated `talon-env` environment and populate it with the right dependencies, then install talon.

```
conda env create -n talon-env -f environment.yml
pip install cobcom-talon
```

Alternatively, you can install the dependencies and talon all via `pip`.

```
pip install numpy
pip install scipy
pip install nibabel
pip install pyunlocbox
# pip install pyopencl  # uncomment for GPU capabilities

pip install cobcom-talon
```

To install talon directly from the source, clone this repository and run the standard local setup commands.

```
git clone https://gitlab.inria.fr/cobcom/talon.git
cd talon
pip install -U .
```

2.1.1 Check installation

To check that talon has been properly installed, try to import the `talon` and the `talon.cli` modules into a Python session as follows. If no error is raised, the installation has been successful.

```
>>> import talon
>>> import talon.cli
```

To further check that the GPU capabilities are active, try to import the `talon.opencl`. If no error is raised, the installation has been successful.

```
>>> import talon.opencl
```

2.1.2 For developers

If you are thinking about developing your own fork of talon, you may want to use the latest version in the `devel` branch of the repository and install it in editable mode.

```
git clone https://gitlab.inria.fr/cobcom/talon.git
cd talon
git checkout devel
pip install -e .
```

Tests

The package uses `unittest` as a testing suite. To run all the tests, execute the following command in the source's root directory.

```
python -m unittest -v
```

Test coverage can be checked with `coverage` as follows.

```
coverage run -m unittest
coverage report -m
```

Documentation

The sources of the documentation are in the `doc` folder. The compilation requires the `sphinx` package and the theme to be installed.

```
pip install sphinx
pip install sphinx_rtd_theme
```

To compile the documentation, move to the `doc` folder and run `make <format>`, where the format can be `html`, `latex` or any other sphinx-compatible format. To get a local copy of the the html documentation, run the `make html` command.

```
cd doc
make clean  # deletes results of previous compilations
make html
```

2.2 Getting started

The `talon` package, at its core, provides a way to transform a tractogram into a linear operator, or more precisely a matrix. This matrix can be used in two ways: to generate data and to explain data. In both cases, the type of the data is arbitrary and is specified by the user, not by `talon`. To quickly get you started, the following examples illustrate both use cases.

If you haven't already, start by installing `talon`. See the *Installation* section.

This short introduction is separated into 3 parts:

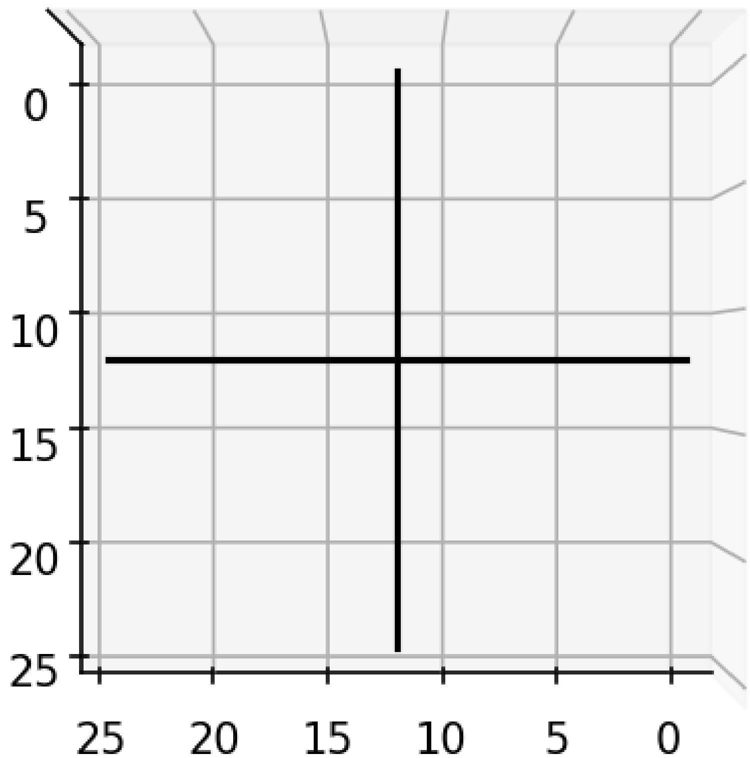
- *Building a linear operator*
- *Generating data with a linear operator*
- *Explaining data with a linear operator*

To generate data using `talon`, we need a tractogram. In general, you may use NiBabel's tools such as `nibabel.streamlines.load` to load your own tractogram. In this paragraph following paragraph we will show how to define a simple synthetic tractogram composed of two crossing streamline bundles.

```
import numpy as np
from scipy.interpolate import interp1d

# The number of voxels in each dimension of the output image.
image_size = 25

center = image_size // 2
t = np.linspace(0, 1, int(image_size / 0.1))

# Generate the horizontal and vertical streamlines.
horizontal_points = np.array([[0, center, center], [image_size - 1, center, center]])
horizontal_streamline = interp1d([0, 1], horizontal_points, axis=0)(t)

vertical_points = np.array([[center, 0, center], [center, image_size - 1, center]])
vertical_streamline = interp1d([0, 1], vertical_points, axis=0)(t)

# A tractogram is just a collection of streamlines.
tractogram = [horizontal_streamline, vertical_streamline]
```

To visualize the geometry of the streamlines, you can display them using `matplotlib`.

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure(figsize=(5, 5), dpi=150)
ax = fig.add_subplot(111, projection='3d')
ax.plot(tractogram[0][:,0], tractogram[0][:,1], tractogram[0][:,2], 'k')
ax.plot(tractogram[1][:,0], tractogram[1][:,1], tractogram[1][:,2], 'k')
ax.view_init(90,90)
ax.set_zticks([])
plt.show()
```



2.2.1 Building a linear operator

Now that we have a tractogram, we can start using `talon`. First, we will *voxelize* the tractogram by separating each streamline into voxel elements. If you are familiar with tractography, streamlines are generated by following peaks of an image. Voxelizing a tractogram is the opposite i.e. creating peaks from streamlines. In order to voxelize the tractogram, we first need to provide a list of *directions* of the possible orientations of the streamlines represented as an array of unit vectors.

```
import talon

directions = np.array([[1, 0, 0], [0, 1, 0], [0, 0, 1]], dtype=np.float)
image_shape = (image_size,) * 3
indices, lengths = talon.voxelize(tractogram, directions, image_shape)
```

Next we define how each streamline direction is projected onto the data.

```
generators = np.ones((len(directions), 1))
```

Finally, we build the linear operator A.

```
A = talon.operator(generators, indices, lengths)
```

Note that generators can be multidimensional. One way to illustrate this is to use the directions as generators.

```
G = talon.operator(directions, indices, lengths)
```

2.2.2 Generating data with a linear operator

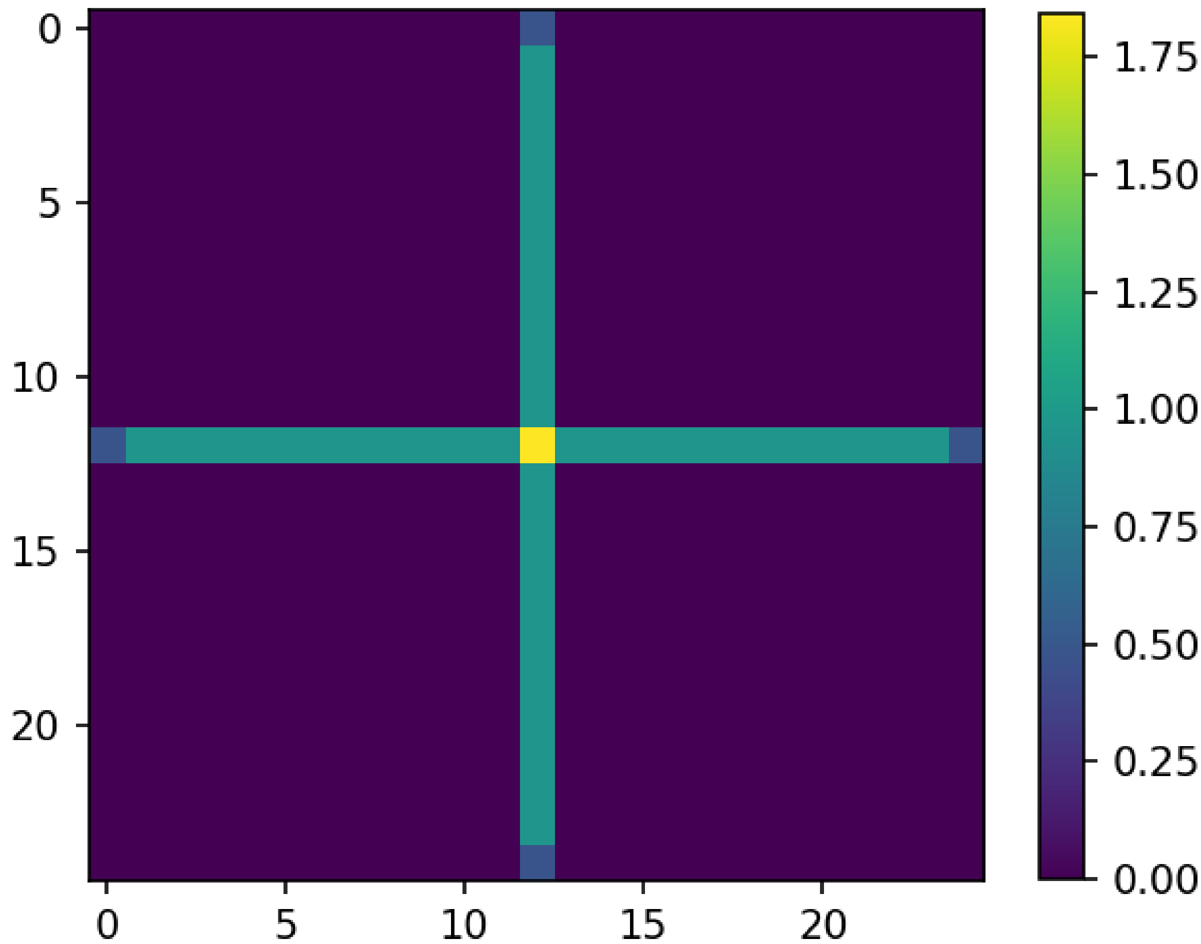
To generate data simply multiply (using the `@` operator) the linear operator by a weight vector.

```
# Using a vector off all ones gives all streamlines equal weight.
x = np.ones(A.shape[1])
b = A @ x

# We can do the same thing with the multidimensional operator.
m = G @ x
```

The data vector `b` can be reshaped into an image and visualized.

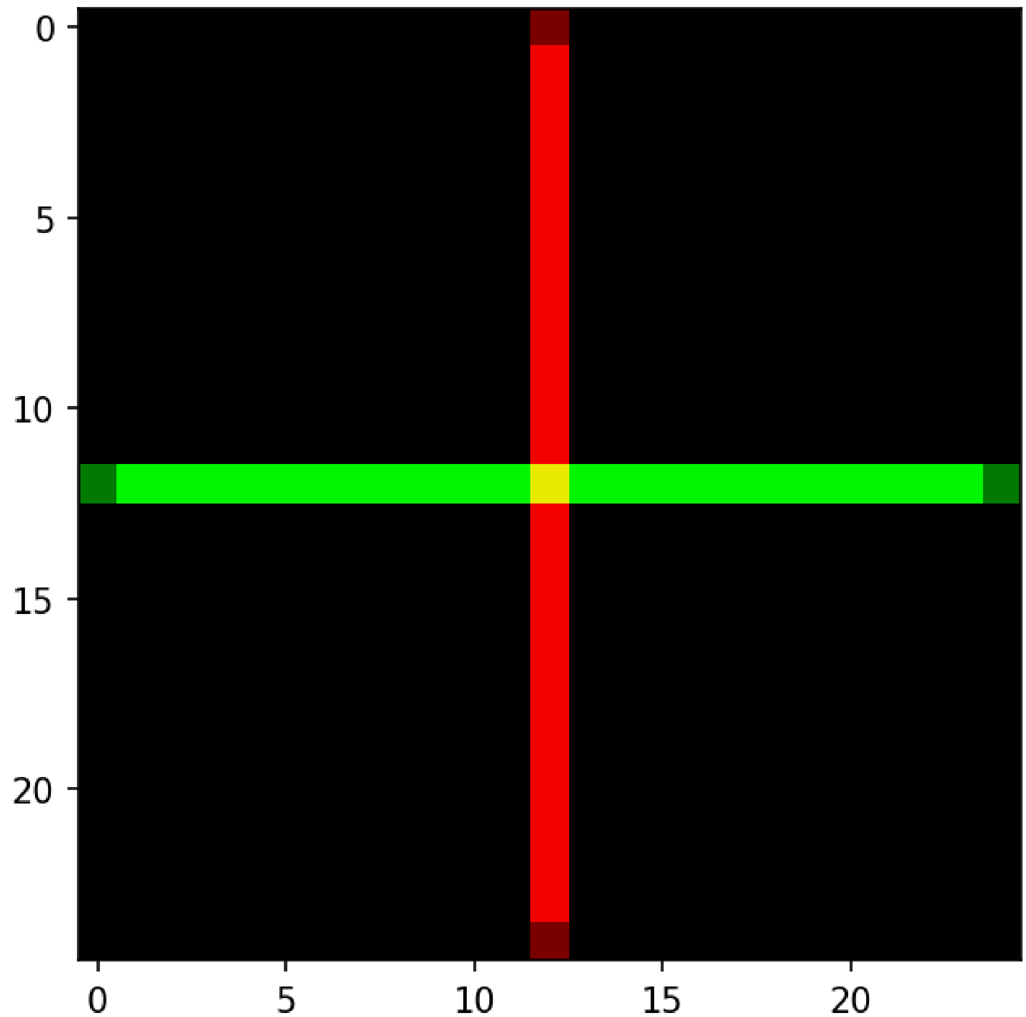
```
image = b.reshape(image_shape)

plt.figure(figsize=(5, 5), dpi=150)
plt.imshow(image[:, :, center])
plt.colorbar(shrink=0.8)
plt.show()
```

An we obtain the following image which corresponds to the streamline density.



The second data vector can also be visualized, but requires a bit more manipulation.

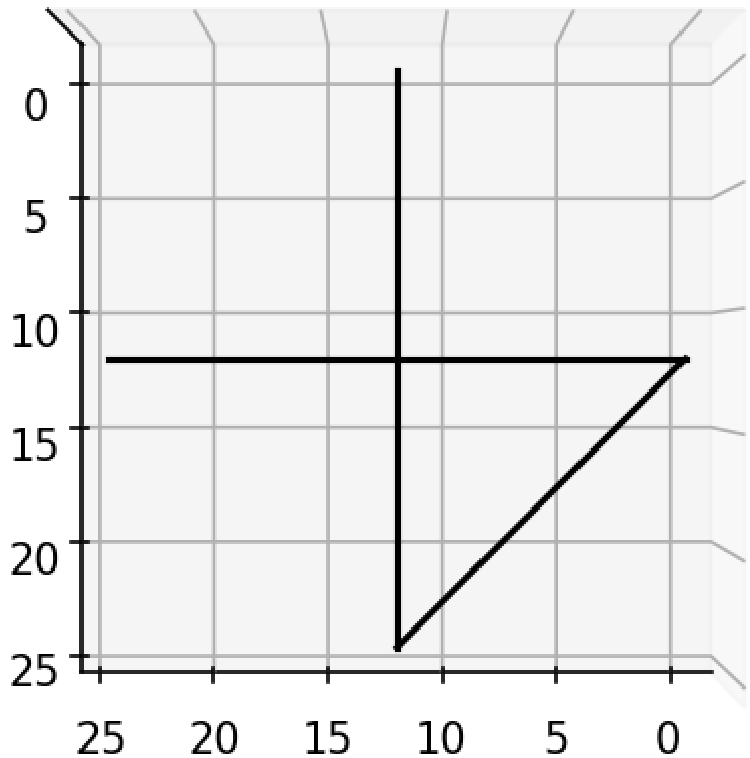
```
rgb_image = m.reshape(image_shape + (3,))

plt.figure(figsize=(5, 5), dpi=150)
plt.imshow(rgb_image[:, :, center])
plt.show()
```



2.2.3 Explaining data with a linear operator

Considering the case where an error in the tractography algorithm generates a spurious streamline in our tractogram. In the case of our example, we simply add a diagonal streamline to *tractogram*.

```
diagonal_points = np.array([[0, center, center], [center, image_size - 1, center]])
diagonal_streamline = interp1d([0, 1], diagonal_points, axis=0)(t)

tractogram.append(diagonal_streamline)

# Visualize the new tractogram.
fig = plt.figure(figsize=(5, 5), dpi=150)
ax = fig.add_subplot(111, projection='3d')
ax.plot(tractogram[0][:,0], tractogram[0][:,1], tractogram[0][:,2], 'k')
ax.plot(tractogram[1][:,0], tractogram[1][:,1], tractogram[1][:,2], 'k')
ax.plot(tractogram[2][:,0], tractogram[2][:,1], tractogram[1][:,2], 'k')
ax.view_init(90,90)
ax.set_zticks([])
plt.show()
```



Given *b*, the data generated using by the original tractogram, we can use `talon` to calculate the contribution of each streamline to the data. In order to do so, we first have to generate a *linear operator* using the new tractogram. In this case, we use also use a set of 1000 equally spaced unit vectors as *directions*.

```
directions = talon.utils.directions(1000)
generators = np.ones((len(directions), 1))
indices, lengths = talon.voxelize(tractogram, directions, image_shape)
Z = talon.operator(generators, indices, lengths)
```

What we want to find are the streamline contributions *x* which minimize

$$\frac{1}{2}||Zx - b||^2 + \Omega(x)$$

In this example it does not make sense to have streamlines with a negative contribution, therefore, $\Omega(x)$ will be set as a positivity constraint. In `talon`, we can force positivity constraint using the `talon.regularization` function.

```
positivity_constraint = talon.regularization(non_negativity=True)
```

The resulting regularization term is then given to the `talon.solve` function in order to obtain the streamlines contributions.

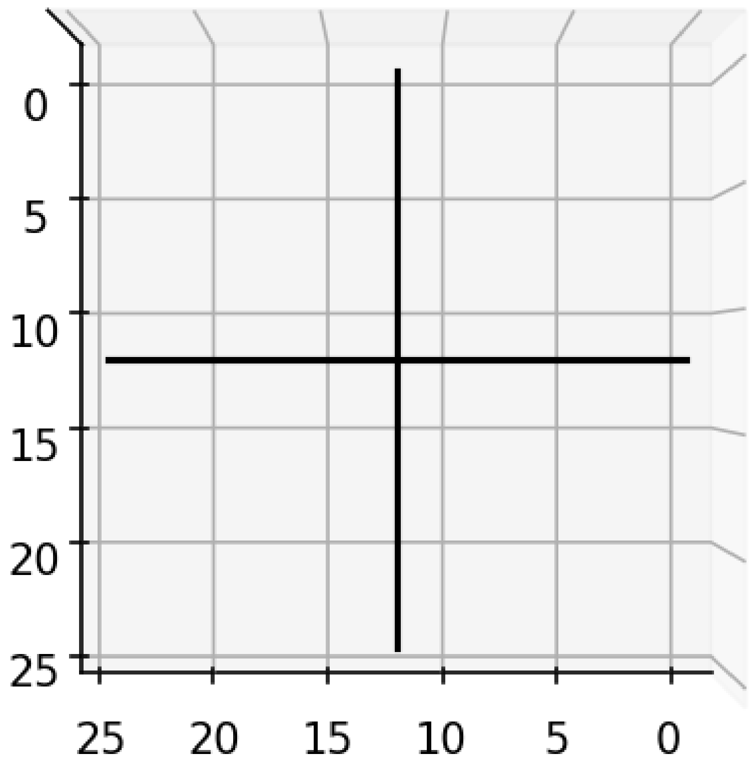
```
solution = talon.solve(Z, b, reg_term=positivity_constraint)
print('solution.x = [%.2f, %.2f, %.2f]' % tuple(solution.x))
```

```
solution.x = [1.00, 1.00, 0.00]
```

As it is possible to see, the two original streamlines contribute equally to the data while the third streamline does not contribute.

We can use the `talon` solution to filter the tractogram and visualize only the streamlines presenting a non-zero contribution.

```
# New filtered tractogram.
filtered_tractogram = []

fig = plt.figure(figsize=(5, 5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

for i,s in enumerate(tractogram):

    # If the current streamline contributes to the data.
    if solution.x[i] > 0.0:

        # Add streamline to filtered tractogram.
        filtered_tractogram.append(s)

        # Visualize the streamline.
        ax.plot(s[:,0], s[:,1], s[:,2], 'k')

ax.view_init(90,90)
ax.set_zticks([])
plt.show()
```



2.3 CLI: Command Line Interface

Talon provides a handy command line interface that allows to filter a tractogram file and obtain the streamline coefficients in text format.

The main command is `talon`, which is installed together with the package (see *Installation*) and allows to filter and voxelize a tractogram.

```
>>> talon --help
usage: talon [-h] {filter,voxelize} ...

Tractograms As Linear Operators in Neuroimaging - command line interface
```

(continues on next page)

(continued from previous page)

```
positional arguments:
  {filter,voxelize}
    filter              Filter a tractogram using TALON.
    voxelize            Voxelize a tractogram using TALON.

optional arguments:
  -h, --help            show this help message and exit

Copyright: CoBCoM 2021.
```

2.3.1 `talon filter`

The `talon filter` command allows to filter a given tractogram as in *Solving the inverse problem*, but without the need to write any Python code.

The basic syntax that you'll have to use is

```
talon filter streamlines.tck data.nii.gz streamline_weights.txt
```

where `streamline.tck` is the tractogram to be filtered, `data.nii.gz` is what is being fit by the filtering process (we will get to that later) and `streamline_weights.txt` is the text file where the streamline weights will be saved.

Streamlines

The input tractogram must be in NiBabel-readable format, i.e., in `tck` or `trk` format. In both cases, it is required to be in RAS+ and mm space. The streamline coordinate (0,0,0) refers to the center of the voxel.

Data

The input data must be a `.nii/.nii.gz` volume registered with the tractogram. It contains the data fitted by talon. For the *volume-fraction* model used by `talon filter` it has to encode the intra-axonal volume fraction in each voxel.

Output weights

The output is a text file where the n-th row contains the weight computed for the n-th streamline.

Group sparsity regularization

The command is able to take into account the bundle organization of the streamlines. For a detailed presentation of how this is encoded as a regularization term, please refer to *Structured Sparsity*. This prior is activated by passing the option `--streamline-assignment sa.txt` to `talon solve`. The `sa.txt` file contains one row per streamline and the n-th row contains the labels of the two regions connected by the n-th streamline. For instance, a tractogram with three streamlines could correspond to the following assignment file.

```
# assignment file of subject ABC1234
3 15
7 2
15 3
```

The first row starts with #, hence will not be read by the program. Then we have a streamline connecting regions 3 and 15, a second one connecting regions 7 and 2 and a third streamline connecting regions 15 and 3. The order of the labels is ignored by the program, hence the first and the third streamlines are bundled together, while the second streamline forms another bundle.

The assignment file is typically obtained via `tck2connectome`, which is part of the Mrtrix's suite.

```
tck2connectome \
    streamlines.tck atlas.nii.gz connectome.txt \
    -out_assignment streamline_assignment.txt
```

Using GPUs

Using a GPU can significantly speed up the execution. Before attempting to use it, be sure to have PyOpenCL installed. The use of the GPU processing capabilities is triggered by the `--operator-type` option as follows.

```
--operator-type opencl
```

Other options

```
>>> talon filter --help
usage: talon filter [-h] [--operator-type {reference,fast,opencl}]
                    [--ndir number] [--allow-negative-x] [--sigma value]
                    [--streamline-assignment file] [--connectome file]
                    [--objective-relative-tolerance value]
                    [--x-absolute-tolerance value] [--maxiter count]
                    [--precomputed-indices-weights file_idx file_wei]
                    [--save-generators-indices-weights file_gen file_idx file_wei | --
↪save-operator-pickle file]
                    [--force] [--quiet | --warn | --info | --debug]
                    in_tracks in_data out_weights

Use TALON to filter a tractogram with the Volume Fraction forward model.

positional arguments:
  in_tracks             Input tractogram file in RAS+ and mm space. The
                        streamline coordinate (0,0,0) refers to the center of
                        the voxel. Must be in NiBabel-readable format (.trk or
                        .tck).
  in_data               Input data to be fitted. Serves also as reference
                        space for tractogram. Must be in NiBabel-readable
                        format (.nii or .nii.gz).
  out_weights           Output text file containing the streamline weights.

optional arguments:
  -h, --help            show this help message and exit
  --operator-type {reference,fast,opencl}
                        Type of operator to use. Default: `fast`.
  --ndir number         Number of directions for the voxelization. Default:
```

(continues on next page)

(continued from previous page)

```
                        1000.
  --precomputed-indices-weights file_idx file_wei
                        Uses the indices and weights passed as input to build
                        the linear operator. E.g. `--precomputed-indices-
                        weights <indices>.npz <weights>.npz`. The two matrices
                        must be defined on the same number of directions as
                        the ones that are used at the call of this script.
  --save-generators-indices-weights file_gen file_idx file_wei
                        Saves the linear operator as three separate files
                        `<generators>.npy <indices>.npz <weights>.npz`. All
                        types of operator can be saved in this format.
  --save-operator-pickle file
                        Saves the linear operator with pickle. Only available
                        when --operator-type is set to `reference` or `fast`.
  --force               Overwrite existing files.
  --quiet               Do not display messages.
  --warn                Display warning messages.
  --info                Display information messages.
  --debug               Display debug messages.

Solver options:
  --allow-negative-x    Disables the non negativity constraint.
  --sigma value         Sets the regularization scale parameter as in (Frigo,
                        2021). The final value of lambda is
                        `sigma*max(||At*data||/gwei)`, where sigma is the
                        passed parameter, `||At*data||` is the 2-norm of the
                        product between the transposed linear operator and the
                        data, and `gwei` is the vector of the weights
                        associated to each group of streamlines. Default: 0.0.
  --streamline-assignment file
                        Activates the group sparsity regularization by
                        specifying the node assignments of each streamline to
                        some parcellation. Typically, this file is produced by
                        the Mrtrix3 command `tck2connectome` with the option
                        `-out_assignment`. The file is expected to be in text
                        format with one row per streamline. E.g., if the first
                        row is [5, 14], the first streamline will be bundled
                        together with all the streamlines corresponding rows
                        having [5, 14] or [14, 5].
  --connectome file     Activates the FIT regularization by specifying the
                        connectivity matrix. Each streamline bundle is
                        associated to the entry in the connectivity matrix
                        corresponding to the region lables that it connects.
                        E.g., the bundle connecting regions 5 and 14 is
                        associated to the entry [5, 14] of the connectivity
                        matrix. Notice that the first row and column
                        correspond to the zero label. Must be used together
                        with `--streamline-assignment`.
  --objective-relative-tolerance value
                        Sets relative tolerance on cost function. Default:
                        1e-06.
  --x-absolute-tolerance value
                        Sets absolute tolerance on variable. Default: 1e-06.
  --maxiter count       Sets maximum number of iterations. Default: 1000.
```

2.3.2 `talon voxelize`

The `talon filter` command allows to create the *indices* and *weights* matrices that are necessary to define a talon linear operator as in *Getting started*, but without the need to write any Python code.

The basic syntax that you'll have to use is

```
talon voxelize streamlines.tck image.nii.gz indices.npz weights.npz
```

where `streamline.tck` is the tractogram to be voxelized, `image.nii.gz` is a reference image that defines the shape of the linear operator (typically the data that is going to be fitted in the filtering process) and `indices.npz` and `weights.npz` are the two COO sparse matrices that define the indices and weights of the linear operator respectively.

Streamlines

The input tractogram must be in NiBabel-readable format, i.e., in `tck` or `trk` format. In both cases, it is required to be in RAS+ and mm space. The streamline coordinate (0,0,0) refers to the center of the voxel.

Output matrices

The two COO matrices are saved in `.npz` format. If the suffix is not present in the filename, it is automatically appended.

Other options

```
>>> talon voxelize --help
usage: talon voxelize [-h] [--ndir number] [--force]
                      [--quiet | --warn | --info | --debug]
                      in_tracks in_img out_ind out_wei

Transform a tractogram into the `indices` and `weights` matrices that are used
in the definition of the linear operator used by TALON.

positional arguments:
  in_tracks      Tractogram file to be voxelized in RAS+ and mm space. The
                 streamline coordinate (0,0,0) refers to the center of the
                 voxel. Must be in NiBabel-readable format (.trk or .tck).
  in_img         Image serving as space reference. Must be in NiBabel-readable
                 format (.nii or .nii.gz).
  out_ind        Path where the indices will be saved in .npz format.
  out_wei        Path where the weights will be saved in .npz format.

optional arguments:
  -h, --help     show this help message and exit
  --ndir number  Number of directions for the voxelization. Default: 1000.
  --force        Overwrite existing files.
  --quiet        Do not display messages.
  --warn         Display warning messages.
  --info         Display information messages.
  --debug        Display debug messages.
```

2.4 Solving the inverse problem

The `talon` package, provides a way to solve the following optimization problem

$$x^* = \operatorname*{argmin}_x \frac{1}{2}\|Ax - y\|_2^2 + \Omega(x)$$

where x is a vector in $\mathbb{R}^n$, A is a linear operator from $\mathbb{R}^n \to \mathbb{R}^m$ and y is a vector in $\mathbb{R}^m$. The functional $\Omega : \mathbb{R}^n \to \mathbb{R}$ acts as regularization term and must be convex and lower semi-continuous.

The first term of the target functional is devoted to the fitting of the data vector by means of the forward linear operator A and the coefficient x_j associated to each atom of A.

2.4.1 Defining regularization term

The possible choices for the regularization term are the following.

- *Least Squares*
- *Non Negativity Constraint*
- *Structured Sparsity*
- *Structured Sparsity with Non Negativity*

Each of these regularization terms can be defined in *talon* by calling the `talon.regularization` function.

Least Squares

Whenever $\Omega(x) = 0$ for all the admissible values of x, the problem reduces to the classical Least Squares formulation. This is the default regularization term in *talon*, hence one just needs to call the `talon.regularization` function as follows.

```
regterm = talon.regularization()
```

See an example of this problem in *Solve the Least Squares problem*.

Non Negativity Constraint

To solve the Non Negative Least Squares (NNLS) problem the regularization term must be the indicator function (in the sense of convex analysis) of the first orthant, namely

$$\Omega(x) = \iota_{\geq 0}(x)$$

which is the function that takes value ∞ whenever x does not belong to the first orthant. The *talon* way to obtain such a regularization term is the following.

```
regterm = talon.regularization(non_negativity=True)
```

See an example of this problem in *Solve the Non Negative Least Squares (NNLS) problem*.

Structured Sparsity

To promote sparse solutions, define the group sparsity regularization term

$$\Omega(x) = \lambda \sum_{g \in G} w_g \|x_g\|_2$$

where λ is the regularization parameter, w_g is the weight associated to each group g, x_g is the subset of entries of x corresponding to group g and G is the list of groups. See [2011j] for a discussion on the mathematical definition of these groups.

The groups $g \in G$ must be defined as a list of lists, where each element encodes the indices that define a single group. The weights w_g associated to each group must be contained in a single numpy array of the same length as G. The following code defines three groups and some standard weight for each of them.

```
groups = [[0, 2, 5], [1, 3, 4, 6], [7, 8, 9]]
weights = np.array([1.0 / len(g) for g in groups])
```

Ones the groups, the weights and the regularization parameter are defined, the regularization term can be initialized as follows.

```
print('Regularization parameter: {}'.format(the_lambda))
print('Number of groups: {}'.format(len(groups)))
print('Number of weights: {}'.format(len(weights)))

regterm = talon.regularization(regularization_parameter=the_lambda,
                               groups=groups, weights=weights)
```

See an example of this problem at *Solve the Group Sparsity problem*.

Notice that the standard ℓ_1 regularization is a particular case of structure sparsity where there is only one group containing all the admissible indices. Assuming that these indices are $0 \ldots n$, the following line of code defines the problem for classical ℓ_1 regularization.

```
groups = [list(range(n))]
```

See an example of this problem at *Solve the Lasso problem* and *Solve the Non Negative Lasso problem*.

Structured Sparsity with Non Negativity

To add the Non Negativity constraint to the Structured Sparsity regularization we just need to set the `non_negativity` flag as `True` during the initialization of the regularization term.

```
regterm = talon.regularization(regularization_parameter=the_lambda,
                               groups=groups, weights=weights,
                               non_negativity=True) # here it is
```

See an example of this problem at *Solve the Non Negative Group Sparsity problem*.

2.4.2 Computing the solution

The function devoted to the computation of the solution of the inverse problem is the `talon.solve` function. It can be called as follows.

```
linear_operator = # build linear operator
data = # define the data to fit
reg_term = # initialize the regularization term as above

solution = talon.solve(linear_operator=linear_operator,
                       data=data,
                       reg_term=regterm)
```

The optimization problem is solved with the FISTA+BT algorithm proposed by Beck and Teboulle in [2009b].

See the API documentation for the description of the supplementary optional parameters.

The `talon.solve` function is a wrapper of the `pyunlocbox.solvers.solve` function.

2.4.3 Reading the result

The result of the optimization problem is given as a `scipy.optimize.OptimizeResult` object, which is a dictionary with the following fields.

- `x`: estimated solution.
- **`status`: attribute of `talon.solve.ExitStatus` enumeration. If** status < 1, the algorithm didn't converge properly.
- `message`: string explaining reason for termination.
- `fun`: value of the objective function at the minimizer.
- `nit`: number of performed iterations
- `reg_param`: value of the regularization parameter, if employed.

2.4.4 Examples

Build the ground truth tractogram with two bundles of fibers.

```
import matplotlib.pyplot as plt
import numpy as np
import talon

from mpl_toolkits.mplot3d import Axes3D
from scipy.interpolate import interp1d

# Set seed for reproducibility
np.random.seed(1992)

# The number of voxels in each dimension of the output image.
image_size = 25
center = image_size // 2

n_points = int(image_size / 0.01)
t = np.linspace(0, 1, n_points)
```

(continues on next page)

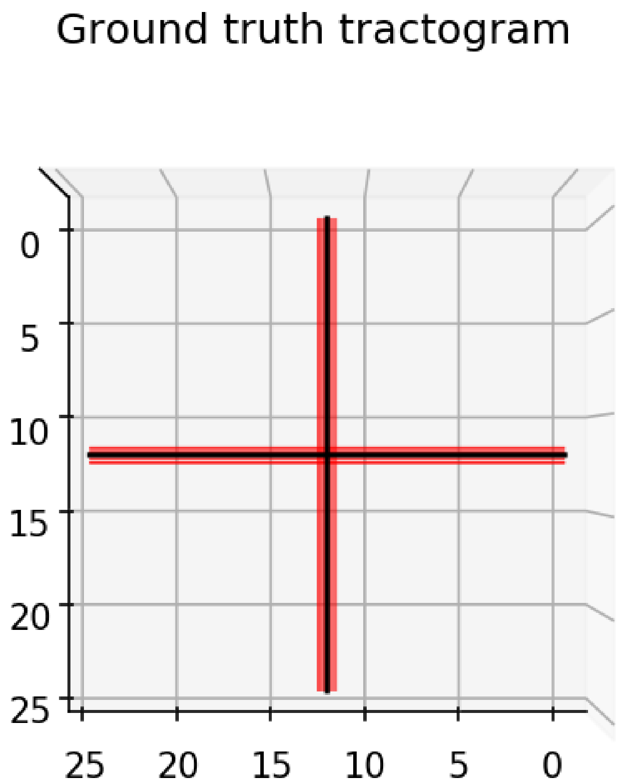
(continued from previous page)

```
# Generate the ground truth tractogram.
tractogram = []
n_streamlines_per_bundle = 50

horizontal_points = np.array([[0, center, center],
                              [image_size - 1, center, center]])
horizontal_streamline = interp1d([0, 1], horizontal_points, axis=0)(t)

for k in range(n_streamlines_per_bundle):
    new_streamline = horizontal_streamline.copy()
    new_streamline[:,1] += (np.random.rand(1) - 0.5)
    tractogram.append(new_streamline)

vertical_points = np.array([[center, 0, center],
                            [center, image_size - 1, center]])
vertical_streamline = interp1d([0, 1], vertical_points, axis=0)(t)

for k in range(n_streamlines_per_bundle):
    new_streamline = vertical_streamline.copy()
    new_streamline[:,0] += (np.random.rand(1) - 0.5)
    tractogram.append(new_streamline)
```

Show the ground truth tractogram.

```
fig = plt.figure(figsize=(5, 5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

for streamline in tractogram:
    ax.plot(streamline[:,0], streamline[:,1], streamline[:,2], 'r',
            linewidth=0.1)

ax.plot(horizontal_streamline[:,0],
        horizontal_streamline[:,1],
        horizontal_streamline[:,2], 'k')
ax.plot(vertical_streamline[:,0],
        vertical_streamline[:,1],
        vertical_streamline[:,2], 'k')
ax.view_init(90,90)
ax.set_zticks([])
plt.title('Ground truth tractogram')
plt.show()
```

You should see the following image:



Generate the corresponding linear operator and the streamline density.

```
directions = talon.utils.directions(1000)
generators = np.ones((len(directions), 1))
image_shape = (image_size,) * 3
indices, lengths = talon.voxelize(tractogram, directions, image_shape)
linear_operator = talon.operator(generators, indices, lengths)

data = linear_operator @ np.ones(linear_operator.shape[1], dtype=np.float64)
image = data.reshape(image_shape)
```

Plot the density of the ground truth streamlines

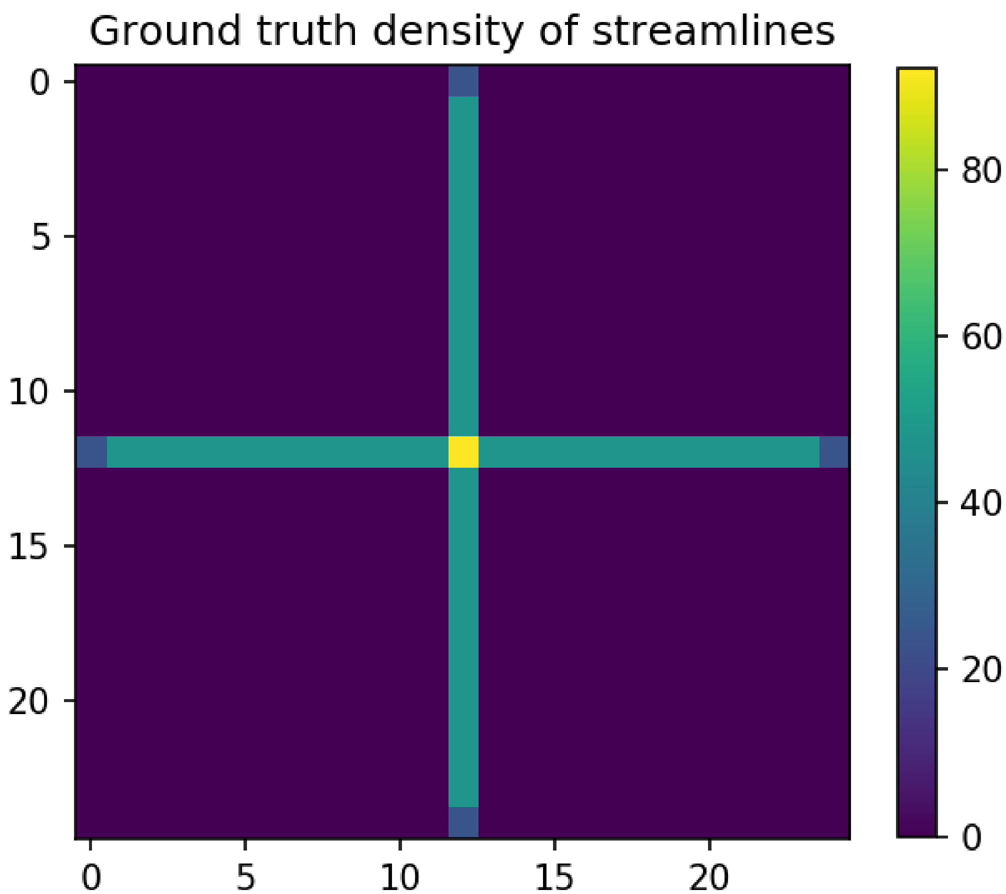
```
plt.figure(figsize=(5, 5), dpi=150)
plt.imshow(image[:, :, center])
```

(continues on next page)

(continued from previous page)

```
plt.colorbar(shrink=0.8)
plt.title('Ground truth density of streamlines')
plt.show()
```

You should see the following image:



Add a diagonal bundle of false positives.

```
diagonal_points = np.array([[0, center, center],
                            [center, image_size - 1, center]])
diagonal_streamline = interp1d([0, 1], diagonal_points, axis=0)(t)

for k in range(n_streamlines_per_bundle):
```

(continues on next page)

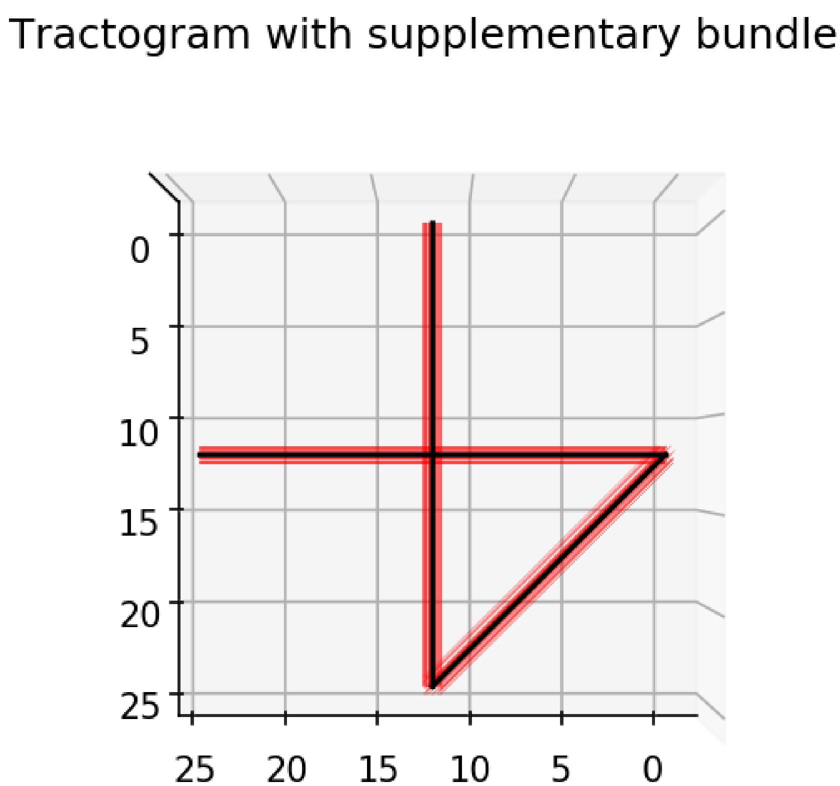
(continued from previous page)

```
    new_streamline = diagonal_streamline.copy()
    new_streamline[:,0] += (np.random.rand(1) - 0.5)
    new_streamline[:,1] += (np.random.rand(1) - 0.5)
    tractogram.append(new_streamline)
```

Visualize the new tractogram.

```
fig = plt.figure(figsize=(5, 5), dpi=150)
ax = fig.add_subplot(111, projection='3d')

for streamline in tractogram:
    ax.plot(streamline[:,0], streamline[:,1], streamline[:,2], 'r', linewidth=0.1)

ax.plot(horizontal_streamline[:,0],
        horizontal_streamline[:,1],
        horizontal_streamline[:,2], 'k')
ax.plot(vertical_streamline[:,0],
        vertical_streamline[:,1],
        vertical_streamline[:,2], 'k')
ax.plot(diagonal_streamline[:,0],
        diagonal_streamline[:,1],
        diagonal_streamline[:,2], 'k')

ax.view_init(90,90)
ax.set_zticks([])
plt.title('Tractogram with supplementary bundle')
plt.show()
```

You should see the following image:



Define the linear operator of the tractogram.

```
indices, lengths = talon.voxelize(tractogram, directions, image_shape)
linear_operator = talon.operator(generators, indices, lengths)
```

Solve the Least Squares problem

```
solution = talon.solve(linear_operator=linear_operator, data=data,
                       verbose='NONE')

print('\nLeast Squares solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0:n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle])/
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2*n_streamlines_per_bundle:3*n_streamlines_per_bundle])/
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following.

```
Least Squares solution
Success: True
Status: ExitStatus.ABSOLUTE_TOLERANCE_X
Exit criterion: XTOL
Number of iterations: 145
Average coefficient of horizontal streamlines: 0.9999996764340565
Average coefficient of vertical streamlines: 0.9999996573175529
Average coefficient of diagonal streamlines : 4.908558143242968e-06
Value at minimizer: 7.0157355592255e-07
```

Solve the Non Negative Least Squares (NNLS) problem

```
reg_term = talon.regularization(non_negativity=True)
solution = talon.solve(linear_operator=linear_operator, data=data,
                       reg_term=reg_term, verbose='NONE')

print('\nNNLS solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0:n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle])/
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2*n_streamlines_per_bundle:3*n_streamlines_per_bundle])/
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following.

```
NNLS solution
Success: True
Status: ExitStatus.ABSOLUTE_TOLERANCE_X
Exit criterion: XTOL
Number of iterations: 25
Average coefficient of horizontal streamlines: 0.9999991567472424
Average coefficient of vertical streamlines: 0.9999991568721199
Average coefficient of diagonal streamlines : 5.0072499918376545e-06
Value at minimizer: 3.620593044727195e-07
```

Solve the Lasso problem

```
regpar = 1.0 # regularization parameter a.k.a. the lambda in the formula
groups = []
groups.append([k for k in range(0, len(tractogram))])

weights = np.array([1.0 / np.sqrt(len(g)) for g in groups])

reg_term = talon.regularization(groups=groups, weights=weights,
                                regularization_parameter=regpar)

solution = talon.solve(linear_operator=linear_operator, data=data,
                       reg_term=reg_term, verbose='NONE')
print('\nLasso solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0: n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2 * n_streamlines_per_bundle: 3 * n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following:

```
Lasso solution
Success: True
Status: ExitStatus.RELATIVE_TOLERANCE_COST
Exit criterion: RTOL
Number of iterations: 93
Average coefficient of horizontal streamlines: 0.9999926298816814
Average coefficient of vertical streamlines: 0.9999925070704963
Average coefficient of diagonal streamlines : -2.1995490196016877e-05
Value at minimizer: 0.8165122997013363
```

Solve the Non Negative Lasso problem

```
reg_term = talon.regularization(non_negativity=True,
                                groups=groups, weights=weights,
                                regularization_parameter=regpar)

solution = talon.solve(linear_operator=linear_operator, data=data,
                       reg_term=reg_term, verbose='NONE')
print('\nNon Negative Lasso solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0: n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2 * n_streamlines_per_bundle: 3 * n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following:

```
Non Negative Lasso solution
Success: True
Status: ExitStatus.RELATIVE_TOLERANCE_COST
Exit criterion: RTOL
Number of iterations: 23
Average coefficient of horizontal streamlines: 0.9999914147578718
Average coefficient of vertical streamlines: 0.9999914603196133
Average coefficient of diagonal streamlines : 4.482209580050452e-06
Value at minimizer: 0.8164938196507543
```

Solve the Group Sparsity problem

```
groups = []
groups.append([k for k in range(0, n_streamlines_per_bundle)]) # horizontal
groups.append([k for k in range(n_streamlines_per_bundle,
              2 * n_streamlines_per_bundle)]) # vertical
groups.append([k for k in range(2 * n_streamlines_per_bundle,
              3 * n_streamlines_per_bundle)]) # diagonal

weights = np.array([1.0 / np.sqrt(len(g)) for g in groups])

reg_term = talon.regularization(groups=groups, weights=weights,
                                regularization_parameter=regpar)

solution = talon.solve(linear_operator=linear_operator, data=data,
                       reg_term=reg_term, verbose='NONE')
print('\nGroup Sparsity solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
```

(continues on next page)

(continued from previous page)

```
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0: n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2 * n_streamlines_per_bundle: 3 * n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following:

```
Group Sparsity solution
Success: True
Status: ExitStatus.RELATIVE_TOLERANCE_COST
Exit criterion: RTOL
Number of iterations: 64
Average coefficient of horizontal streamlines: 0.9999821712768615
Average coefficient of vertical streamlines: 0.9999823618643954
Average coefficient of diagonal streamlines : 2.2318881330827924e-05
Value at minimizer: 2.000096258909371
```

Solve the Non Negative Group Sparsity problem

```
reg_term = talon.regularization(groups=groups, weights=weights,
                                non_negativity=True,
                                regularization_parameter=regpar)

solution = talon.solve(linear_operator=linear_operator, data=data,
                       reg_term=reg_term, verbose='NONE')
print('\nNon Negative Group Sparsity solution')
print('Success: {}'.format(solution['success']))
print('Status: {}'.format(solution['status']))
print('Exit criterion: {}'.format(solution['message']))
print('Number of iterations: {}'.format(solution['nit']))

x = solution['x']
print('Average coefficient of horizontal streamlines: {}'.format(
      np.sum(x[0: n_streamlines_per_bundle])/n_streamlines_per_bundle))
print('Average coefficient of vertical streamlines: {}'.format(
      np.sum(x[n_streamlines_per_bundle:2*n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Average coefficient of diagonal streamlines : {}'.format(
      np.sum(x[2 * n_streamlines_per_bundle: 3 * n_streamlines_per_bundle]) /
      n_streamlines_per_bundle))
print('Value at minimizer: {}\n'.format(sum(solution['fun'])))
```

The output should be the following:

```
Non Negative Group Sparsity solution
Success: True
```

(continues on next page)

(continued from previous page)

```
Status: ExitStatus.RELATIVE_TOLERANCE_COST
Exit criterion: RTOL
Number of iterations: 22
Average coefficient of horizontal streamlines: 0.9999825264666186
Average coefficient of vertical streamlines: 0.9999825878147537
Average coefficient of diagonal streamlines : 0.0
Value at minimizer: 1.9999822314331122
```

References

2.5 Concatenating linear operators

It is possible to concatenate linear operators in a way that imitates the `numpy.concatenate` function. The only concatenations that are allowed are in the vertical and horizontal directions.

The `talon.concatenate` function requires an iterable containing the linear operators to concatenate and the axis along which they have to be concatenated.

The following code shows the correct syntax to concatenate two linear operators A and B vertically and horizontally:

```
V = talon.concatenate((A, B), axis=0) # vertical (default)
H = talon.concatenate((A, B), axis=1) # horizontal
```

which correspond to the following

$$V = \begin{bmatrix} A \\ B \end{bmatrix} \qquad H = \begin{bmatrix} A & B \end{bmatrix}.$$

2.5.1 Examples

Build a tractogram with two crossing bundles of fibers and the corresponding linear operator.

```
import numpy as np
import talon

from scipy.interpolate import interp1d

# Set seed for reproducibility
np.random.seed(1992)

# The number of voxels in each dimension of the output image.
image_size = 25
center = image_size // 2

n_points = int(image_size / 0.01)
t = np.linspace(0, 1, n_points)

streamlines_per_bundle = 50

def generate_crossing_tractogram():
    tractogram = []

    horizontal_points = np.array([[0, center, center],
```

(continues on next page)

(continued from previous page)

```
                                  [image_size - 1, center, center]])
    horizontal_streamline = interp1d([0, 1], horizontal_points, axis=0)(t)

    for k in range(streamlines_per_bundle):
        new_streamline = horizontal_streamline.copy()
        new_streamline[:,1] += (np.random.rand(1) - 0.5)
        tractogram.append(new_streamline)

    vertical_points = np.array([[center, 0, center],
                                [center, image_size - 1, center]])
    vertical_streamline = interp1d([0, 1], vertical_points, axis=0)(t)

    for k in range(streamlines_per_bundle):
        new_streamline = vertical_streamline.copy()
        new_streamline[:,0] += (np.random.rand(1) - 0.5)
        tractogram.append(new_streamline)
    return tractogram

cross_tractogram = generate_crossing_tractogram()
directions = talon.utils.directions(1000)
generators = np.ones((len(directions), 1))
image_shape = (image_size,) * 3
indices, lengths = talon.voxelize(cross_tractogram, directions, image_shape)

A = talon.operator(generators, indices, lengths)
```

Vertical concatenation

If multiple features for each streamline are encoded in different linear operators we can concatenate different linear operators vertically. If A encodes the linear operator for the set of streamlines α and generators G_1 and B encodes the linear operator for the same streamlines but with generators G_2, instead of rebuilding the linear operator from scratch we can concatenate A and B vertically to obtain the same result.

```
G2 = np.random.rand(len(directions), 5) # New generators
B = talon.operator(G2, indices, lengths)

V = talon.concatenate((A,B), axis=0)

print('Shape of A: {}'.format(A.shape))
print('Shape of B: {}'.format(B.shape))
print('Shape of V: {}'.format(V.shape))
print('Check: {} + {} = {}'.format(A.shape[0], B.shape[0], A.shape[0] + B.shape[0]))
```

Notice that the `axis=0` argument is redundant since it is the default.

The output should be the following:

```
Shape of A: (15625, 100)
Shape of B: (78125, 100)
Shape of V: (93750, 100)
Check: 15625 + 78125 = 93750
```

Horizontal concatenation

One (but not the only) reason to concatenate two linear operators horizontally is to add a set of streamlines to the system. If A encodes the linear operator for the set of streamlines α and C for set β, instead of rebuilding the linear operator from scratch we can concatenate A and C horizontally to obtain the same result.

```
def generate_diagonal_tractogram():
    tractogram = []
    diagonal_points = np.array([[0, center, center],
                                [center, image_size - 1, center]])
    diagonal_streamline = interp1d([0, 1], diagonal_points, axis=0)(t)

    for k in range(streamlines_per_bundle):
        new_streamline = diagonal_streamline.copy()
        new_streamline[:,0] += (np.random.rand(1) - 0.5)
        new_streamline[:,1] += (np.random.rand(1) - 0.5)
        tractogram.append(new_streamline)
    return tractogram

diag_tractogram = generate_diagonal_tractogram()
indices, lengths = talon.voxelize(diag_tractogram, directions, image_shape)

C = talon.operator(generators, indices, lengths) # diagonal
```

The concatenation of the two linear operators is performed as follows:

```
H = talon.concatenate([A, C], axis=1)
print('Shape of A: {}'.format(A.shape))
print('Shape of C: {}'.format(C.shape))
print('Shape of H: {}'.format(H.shape))
```

The output should be the following:

```
Shape of A: (15625, 100)
Shape of C: (15625, 50)
Shape of H: (15625, 150)
```

The matrix multiplication and transposition operations work as usual:

```
x = H @ np.random.rand(H.shape[1])
y = H.T @ np.random.rand(H.shape[0])

print('Shape of x: {}'.format(x.shape))
print('Shape of y: {}'.format(y.shape))
```

The output should be the following:

```
Shape of x: (15625,)
Shape of y: (150,)
```

2.6 Create linear operator from volume

It may be interesting to create linear operators that describe a single contribution for each voxel as in a volume mask. This can be encoded as follows:

$$\begin{bmatrix} w_1 \cdot \mathbf{g} & & & \\ & w_2 \cdot \mathbf{g} & & \\ & & \ddots & \\ & & & w_n \cdot \mathbf{g} \end{bmatrix}$$

where $\mathbf{g}$ is the generator used for every voxel and w_j is the value of the mask at voxel j. Only the voxels exhibiting non-zero value are considered.

To build such a linear operator, one just needs to provide a three-dimensional ndarray to the *talon.diagonalize* function.

2.6.1 Example

Let us build a toy volume of dimension 2-by-2-by-2 with values from 0 to 7.

```
import numpy as np
values = np.arange(2 ** 3).astype(np.float64)

mask = values.reshape((2, ) * 3)
print(mask)
```

Output:

```
[[[0. 1.]
  [2. 3.]]

 [[4. 5.]
  [6. 7.]]]
```

To diagonalize the volume, call the corresponding *talon* function.

```
import talon
indices, weights = talon.diagonalize(mask)
```

The considered generator is vector $g = [1, 10]^T$.

```
generators = np.array([[1.0, 10.0]])
linear_operator = talon.operator(generators, indices, weights)
```

Check the output:

```
print(linear_operator.todense())

[[ 0.  0.  0.  0.  0.  0.  0.]
 [ 0.  0.  0.  0.  0.  0.  0.]
 [ 1.  0.  0.  0.  0.  0.  0.]
 [10.  0.  0.  0.  0.  0.  0.]
 [ 0.  2.  0.  0.  0.  0.  0.]
 [ 0. 20.  0.  0.  0.  0.  0.]
 [ 0.  0.  3.  0.  0.  0.  0.]
 [ 0.  0. 30.  0.  0.  0.  0.]
```

(continues on next page)

(continued from previous page)

```
[ 0.   0.   0.   4.   0.   0.   0.]
[ 0.   0.   0. 40.   0.   0.   0.]
[ 0.   0.   0.   0.   5.   0.   0.]
[ 0.   0.   0.   0. 50.   0.   0.]
[ 0.   0.   0.   0.   0.   6.   0.]
[ 0.   0.   0.   0.   0. 60.   0.]
[ 0.   0.   0.   0.   0.   0.   7.]
[ 0.   0.   0.   0.   0.   0. 70.]]
```

2.7 Functions

talon.**concatenate**(*operators*, *axis=0*)

Concatenate a sequence of linear operator along axis 0 or 1.

This method defines the object that acts as the concatenation of the linear operators contained in the list/tuple *operators* along the chosen *axis*. The syntax is consistent with the one of *np.concatenate*.

Parameters

- **operators** – list or tuple of LinearOperator objects to be concatenated in the same axis.
- **axis** – int direction in which we want to concatenate the LinearOperator or Concatenated-LinearOperator objects that we want to concatenate. Vertical concatenation is obtained for *axis = 0* and horizontal concatenation is obtained for *axis = 1* as in np.concatenate. (Default: 0)

Returns

the concatenated linear operator.

Return type *talon.core.ConcatenatedLinearOperator*

talon.**diagonalize**(*mask*)

Returns the matrices used to create a linear operator from a mask

This functions transforms a volume mask into the weights and indices components that are necessary to build a linear operator. It is assumed that the all the voxels in the mask will share a common generator. The indexed generator is therefore unique, corresponds to index zero, and is weighted by the value contained in the mask at the specific voxel.

Parameters **mask** – np.ndarray with three dimensions that contains the weight to be associated to each voxel. Only voxels with non-zero weight are considered.

Returns

tuple of length 2 containing

- **index_sparse** [diagonal scipy.sparse matrix with a shape of (n, m)] where n is the number of voxels of the volume and m in the number of voxels of the mask.
- **weight_sparse** [diagonal scipy.sparse matrix with a shape of (n, m)] containing the value of the mask at each non-zero voxel in the same fashion as `index_sparse`.

Raises

- **TypeError** – If the the mask is not a numpy.ndarray.
- **ValueError** – If the mask does not have three dimensions.

`talon.operator`(*generators*, *indices_of_generators*, *weights*, *operator_type='fast'*)

Create a LinearOperator object.

This method defines the object that describes the linear operator by means of its fundamental components. These components are a set of generators, a table that encodes the non-zero entries of the operator and indexes the proper generator in each entry and another table that encodes the weight applied to each called generator in the linear operator.

Each block of entries of the linear operator A is given by

$$A[k \cdot i \dots k \cdot (i+1), j] = g_{T_{i,j}} \cdot w_{i,j}$$

where *k* is the length of the generators, *T* is the table of indices and *w* is the table of weights.

Parameters

- **`generators`** – np.array where each row is a generator.
- **`indices_of_generators`** – COO sparse matrix that tells which generator is called where in the linear operator.
- **`weights`** – COO sparse matrix that encodes the weight applied to each generator indexed by indices_of_generators. It has the same dimension as indices_of_generators.
- **`operator_type`** (`optional`) – string Operator type to use. Accepted values are `'fast'`, `'opencl'`, and `'reference'`. The latter is intended to be used only for testing purposes. (default = *fast*).

Returns the wanted linear operator.

Return type *talon.core.LinearOperator*

Raises **`ValueError`** – If *reference_type* is not `'fast'` or `'reference'`.

`talon.regularization`(*non_negativity=False*, *regularization_parameter=None*, *groups=None*, *weights=None*)

Get regularization term for the optimization problem.

By default this method returns an object encoding the regularization term

$$\Omega(x) = 0.$$

If *regularization_parameter*, *groups* and *weights* are all not None it returns the structured sparsity regularization.

$$\Omega(x) = \lambda \sum_{g \in G} w_g \|x_g\|_2$$

where λ is *regularization_parameter*, w_g is the entry of *w* associated to g, x_g is the subset of entries of *x* encoded by the indices of g and G is the list of groups.

If non_negativity is True it adds the non-negativity constraint to the regularization term.

$$\Omega(x) \leftarrow \Omega(x) + \iota_{\geq 0}(x).$$

Parameters

- **non_negativity** – boolean (default = False)
- **regularization_parameter** – float. Must be >= 0 (default = None)
- **groups** – list of lists where each element encodes the indices of the streamlines belonging to a single group. (default = None).

  E.g.: `groups = [[0,2,5], [1,3,4,6], [7,8,9]]`.
- **weights** – ndarray of the same length as *groups*. Weight associated to each group. (default = None)

Returns

instance of one between

- `talon.optimize.NoRegularization`;
- `talon.optimize.NonNegativity`;
- `talon.optimize.StructuredSparsity`;
- `talon.optimize.NonNegativeStructuredSparsity`.

Raises

- **ValueError** – If weights and groups do not have the same length.
- **ValueError** – If regularization_parameter < 0 .

`talon.`**solve**(*linear_operator: talon.core.LinearOperator, data: numpy.ndarray, reg_term: Optional[talon.optimization.RegularizationTerm] = None, cost_reltol: float = 1e-06, x_abstol: float = 1e-06, max_nit: int = 1000, x0: Optional[numpy.ndarray] = None, verbose: str = 'LOW'*) → scipy.optimize.optimize.OptimizeResult

Fit the solution.

This routine finds the *x* that solves the problem

$$\min_x 0.5\|Ax - y\|^2 + \Omega(x)$$

where *x* is the vector of coefficients to be retrieved, *A* is the linear operator, *y* is the data vector and Ω is defined as in `talon.regularization`.

Parameters

- **linear_operator** – linear operator endowed with the @ operation.
- **data** – ndarray of data to be fit. First dimension must be compatible with the second of *linear_operator*.
- **reg_term** – regularization term defined by talon.regularization. (default: $\Omega(x) = 0.0$)
- **cost_reltol** – float relative tolerance on the cost (default = 1e-6).
- **x_abstol** – float mean abs tolerance on the variable (default = 1e-6).
- **max_nit** – int maximum number of iterations (default = 1000).
- **x0** – ndarray starting value for the optimization. The length must be the equal to the second dimension of *linear_operator*. (default=zeros)
- **verbose** – {'NONE', 'LOW', 'HIGH', 'ALL'} The log level : `'NONE'` for no log, `'LOW'` for resume at convergence, `'HIGH'` for info at all solving steps, `'ALL'` for all possible outputs, including at each steps of the proximal operators computation (default='LOW').

Returns

dictionary with the following fields

- x : estimated minimizer of the cost function.
- status : attribute of talon.optimization.ExitStatus enumeration.
- message : string that explains the reason for termination.
- fun : evaluation of each term at the minimizer.
- nit : number of performed iterations.
- reg_param: value of the regularization parameter.

Return type scipy.optimize.OptimizeResult

`talon.`**`voxelize`**(*streamlines*, *vertices*, *image_shape*, *step=0.04*)

Transform a tractogram into the matrices that are necessary to build a linear operator.

Parameters

- **`streamlines`** – list of streamlines in voxel space. The coordinates of each voxel are assumed to point at the center of the voxel itself.
- **`vertices`** – Nx3 np.array, vertices of an unit sphere in which we sample the streamlines direction.
- **`image_shape`** – tuple, final shape of the mask image.
- **`step`** – double, streamlines interpolation step.

Returns

tuple of length 2 containing

- **index_sparse** [(voxel x streamlines) scipy.sparse matrix containing] for each voxel and fiber the index of the vertices that it is closest to the streamline direction in that voxel.
- **length_sparse** [(voxel x streamlines) scipy.sparse matrix containing] for each voxel and fiber the length of the streamline in that voxel.

Raises `ValueError` – If the streamlines are not in voxel space.

`talon.`**`zeros`**(*shape: Tuple[int, int], dtype: type = <class 'numpy.float64'>*) → *talon.core.LinearOperator*

Create a zero filled linear operator

Returns a zero filled talon linear operator. Useful in combination with talon.concatenate.

Parameters

- **`shape`** – The shape of the linear operator.
- **`dtype`** (*`optional`*) – The datatype of the linear operator.

Returns A talon linear operator filled with zeros.

`talon.utils.`**`check_pattern_iw`**(*indices_of_generators: scipy.sparse.coo.coo_matrix*, *weights: scipy.sparse.coo.coo_matrix*) → None

Check if the sparsity pattern of the indices and the weights are the same.

This function performs a complete check on the sparsity pattern of the *indices_of_generators* and the *weights* matrices. If the two matrices have a different number of non-empty entries or the non-empty entries are in different locations, it raises an error.

If the two matrices came out of *talon.voxelization*, this check is not necessary.

Note: This function is **very** expensive in terms of memory and time.

Parameters

- **indices_of_generators** – sp.coo_matrix of the indices.
- **weights** – sp.coo_matrix of the weights.

Raises ValueError – If *weights* and *indices_of_generators* don't have the same sparsity pattern.

talon.utils.**concatenate_giw**(*giws: Iterable*, *axis: int = 0*) → tuple

Concatenates generators, indices, and weights along an existing axis

The indices and weights are concatenated along the supplied axis and the generators along axis 1. The generators must have the same number of columns. The indices and weights must have the same shape, except in the dimension corresponding to axis.

Parameters

- **giws** – An iterable of (generator, indices, weights) to concatenate e.g. [(g1, i1, w1), (g2, i2, w2)].
- **axis** – The axis along which the indices and weights will be joined. Default is 0.

Returns The concatenated generators, indices, and weights.

talon.utils.**directions**(*number_of_points: int = 180*) → numpy.ndarray

Get a list of 3D vectors representing the directions of the fibonacci covering of a hemisphere of radius 1 computed with the golden spiral method. The z coordinate of the points is always strictly positive.

Parameters number_of_points – number of points of the wanted covering (default=180)

Returns

number_of_points x 3 array with the cartesian coordinates of a point of the covering in each row.

Return type ndarray

Raises ValueError – if `number_of_points <= 0`.

References

https://stackoverflow.com/questions/9600801/evenly-distributing-n-points-on-a-sphere/44164075#44164075

talon.utils.**mask_data**(*data: numpy.ndarray*, *linear_operator: talon.core.LinearOperator*) → numpy.ndarray

Mask the data using the mask that covers only the entries that are affected by the linear operator. This prevents numerical errors in the solution of the optimization problem.

Parameters

- **data** – np.ndarray one dimensional array that contains the data to mask.
- **linear_operator** – LinearOperator object that contains the self.data_mask attribute to be used as a mask.

Returns np.ndarray with the same dimension as data where the entries corresponding to the False entries of the mask have been set to zero.

2.8 Classes

2.8.1 Linear Operator

class talon.core.**AbstractLinearOperator**(**args*, ***kwargs*)

Abstract class for all linear operators

This abstract class defines the interface that all linear operators in talon must implement.

property T

Returns the transpose of the linear operator.

convert_x(*x*)

Converts x so that it can be used on the right of a dot product.

This method converts x so that it has the right dimensions and type to be used as a right operand of a dot product with a linear operator. That is, it asserts that A @ x will work. Raises exceptions if the input cannot be converted to the correct format.

Parameters **x** – The vector to test.

Returns A numpy array that can be used in the dot product.

Return type x

Raises

- **TypeError** – If x is not a numpy array.
- **ValueError** – If the length of x does not match the number of columns of the linear operator.

abstract property data_mask

Returns the mask to apply to the data to keep only the entries covered by the linear operator.

property dtype

Returns the data type of the linear operator

abstract property shape

Returns the shape of the matrix.

abstract property todense

Returns a dense matrix representation of the linear operator.

abstract property transpose

Returns the transpose of the linear operator.

CPU

class talon.core.**LinearOperator**(**args*, ***kwargs*)

__init__(*generators*, *indices_of_generators*, *weights*)

Linear operator that maps tractography to signal space. The linear operator can be used to compute products with a vector.

Parameters

- **generators** – np.array where each row is a generator.
- **indices_of_generators** – COO sparse matrix that tells which generator is called where in the linear operator.

- **weights** – COO sparse matrix that encodes the weight applied to each generator indexed by indices_of_generators. It has the same dimension as indices_of_generators.

Raises

- **TypeError** – If *generators* is not a numpy ndarray of float.
- **TypeError** – If *indices_of_generators* is not a COO scipy matrix.
- **TypeError** – If *weights* is not a COO scipy matrix of float64.
- **ValueError** – If *weights* does not have the same dimension as indices_of_generators.

property columns

Returns the indices of the nonzero columns.

Type int

property data_mask

Returns the mask to apply to the data to keep only the entries covered by the linear operator.

property generator_length

length of each generator (constant across generators).

Type int

property generators

Returns the generators of the linear operator.

Type np.ndarray

property indices

Returns the generator indices.

Type np.ndarray

property nb_atoms

Number of atoms (columns) in the linear operator.

Type int

property nb_data

Number of data points.

Type int

property nb_generators

Number of generators.

Type int

property rows

Returns the indices of the nonzero rows.

Type int

property shape

Shape of the linear operator.

The shape is given by the number of rows and columns of the linear operator. The number of rows is equal to the number of data points times the length of the generators. The number of columns is equal to the number of atoms.

Type tuple of int

todense()

Return the dense matrix associated to the linear operator.

Note: The output of this method can be very memory heavy to store. Use at your own risk.

Returns full matrix representing the linear operator.

Return type ndarray

property transpose

the transpose of the linear operator.

Type _TransposedLinearOperator

property weights

The weights of the nonzero elements

Type np.ndarray

class talon.fast.**LinearOperator**(**args*, ***kwargs*)

__init__(*generators*, *indices_of_generators*, *weights*)

A LinearOperator that computes products quickly.

The FastLinearOperator class implements a linear operator optimized to compute matrix-vector products quickly. It is single threaded and written in pure Python, which makes it a good default choice for linear operators.

Parameters

- **generators** – np.array where each row is a generator.
- **indices_of_generators** – COO sparse matrix that tells which generator is called where in the linear operator.
- **weights** – COO sparse matrix that encodes the weight applied to each generator indexed by indices_of_generators. It has the same dimension as indices_of_generators.

Raises

- **TypeError** – If generators is not a numpy ndarray of float64.
- **TypeError** – If indices_of_generators is not a COO scipy matrix.
- **TypeError** – If weights is not a COO scipy matrix of float64.
- **ValueError** – if weights does not have the same dimension as indices_of_generators.
- **ValueError** – if weights and indices_of_generators don't have the same sparsity pattern.

property transpose

Returns the transpose of the linear operator.

class talon.core.**ConcatenatedLinearOperator**(**args*, ***kwargs*)

__init__(*operators*, *axis*)

Concatenated LinearOperator object

The ConcatenatedLinearOperator class implements the vertical or horizontal concatenation of LinearOperator objects.

Parameters

- **`operators`** – list or tuple of LinearOperator objects to be concatenated in the same axis.
- **`axis`** – int direction in which we want to concatenate the LinearOperator or ConcatenatedLinearOperator objects that we want to concatenate. Vertical concatenation is obtained for *axis = 0* and horizontal concatenation is obtained for *axis = 1* as in np.concatenate. (Default: 0)

Raises

- **`TypeError`** – If any element of *operator* is not an instance of LinearOperator or ConcatenatedLinearOperator.
- **`TypeError`** – If *operators* is not a list or a tuple.
- **`ValueError`** – If *axis* is not 0 or 1.
- **`ValueError`** – If *operators* is an empty list or tuple.
- **`ValueError`** – If the operators do not have compatible dimensions.

`property axis`
axis in which the concatenation was performed.

Type int

`property data_mask`
Returns the mask to apply to the data to keep only the entries covered by the linear operator.

`property operators`
list of concatenated operators.

Type list

`property shape`
Shape of the concatenated linear operator.

Type tuple of int

`todense()`
Return the dense matrix associated to the linear operator.

Note: The output of this method can be very memory heavy to store. Use at your own risk.

Returns full matrix representing the linear operator.

Return type ndarray

`property transpose`
transpose of the linear operator.

Type TransposedConcatenatedLinearOperator

GPU

class talon.opencl.**LinearOperator**(**args*, ***kwargs*)

__init__(*generators*, *indices_of_generators*, *weights*, *chunk_size=100000000*)

Linear operator implemented with OpenCL

A linear operator that has a sparse vector structure. The product between this operator and a vector is implemented using OpenCL.

Parameters

- **generators** – np.array where each row is a generator.
- **indices_of_generators** – COO sparse matrix that tells which generator is called where in the linear operator.
- **weights** – COO sparse matrix that encodes the weight applied to each generator indexed by indices_of_generators. It has the same dimension as indices_of_generators.
- **chunk_size** – The product is computed by splitting the linear operator into chunks. This parameter determines the approximate chunk size. Reducing this value reduces the amount of memory required on the device.

Raises

- **TypeError** – If *generators* is not a numpy array of float.
- **TypeError** – If *indices_of_generators* is not a COO scipy matrix.
- **TypeError** – If *weights* is not a COO scipy matrix of float64.
- **ValueError** – If *weights* does not have the same dimension as indices_of_generators.
- **ValueError** – If *weights* and *indices_of_generators* don't have the same sparsity pattern.

property dtype

Returns the data type of the linear operator

todense() → numpy.ndarray

Return the dense matrix associated with the linear operator.

Note: The output of this method can be very memory heavy to store. Use at your own risk.

Returns Full matrix representing the linear operator.

property transpose

transpose of the linear operator.

Type TransposedFastLinearOperator

2.8.2 Regularization Term

class talon.optimization.**RegularizationTerm**(*regularization_parameter: float*)

__init__(*regularization_parameter: float*)
Abstract base class for all regularization terms

The optimization problem solved by *talon* is

$$\min_x 0.5\|Ax - y\|^2 + \Omega(x)$$

where Ω is a regularization term. This class is the base for all concrete implementations of this term.

Parameters **regularization_parameter** – float The scaling factor of the regularization term. Must be a float greater or equal to zero.

Raises

- **TypeError** – If the regularization parameter is not a float and cannot be converted to a float.
- **ValueError** – If the regularization parameter is negative.

property groups
Get the group structure associated to the regularization term.

Returns: list List of lists of streamline indices.

property non_negativity
Get the activation of the non-negativity constraint.

Returns: bool True if the non-negativity constraint is employed, False otherwise.

property regularization_parameter
Get the regularization parameter.

Returns: float The value of the regularization parameter.

property weights
Get the weight associated to each group.

Returns: np.ndarray 1-dimensional numpy array with one weight per group.

class talon.optimization.**NoRegularization**

__init__()
Instantiates the zero-valued regularization term.

$$\Omega(x) = 0$$

class talon.optimization.**NonNegativity**

__init__()
Instantiates the non-negativity constraint regularization function.

$$\Omega(x) = \iota_{\geq 0}(x)$$

class talon.optimization.**StructuredSparsity**(*regularization_parameter: float*, *groups: list*, *weights: numpy.ndarray*)

__init__(*regularization_parameter: float*, *groups: list*, *weights: numpy.ndarray*)
Instantiates the structured sparsity regularization term.

$$\Omega(x) = \lambda \cdot \sum_{g \in G} w_g \cdot \|x_g\|_2$$

Parameters

- **regularization_parameter** – float Value of the regularization parameter.
- **groups** – list List of lists of streamline indices.
- **weights** – np.ndarray 1-dimensional numpy array with one weight per group.

class talon.optimization.**NonNegativeStructuredSparsity**(*regularization_parameter*, *groups*, *weights*)

__init__(*regularization_parameter*, *groups*, *weights*)
Instantiates the non-negative structured sparsity regularization term.

$$\Omega(x) = \iota_{\geq 0}(x) + \lambda \cdot \sum_{g \in G} w_g \cdot \|x_g\|_2$$

Parameters

- **regularization_parameter** – float Value of the regularization parameter.
- **groups** – list List of lists of streamline indices.
- **weights** – np.ndarray 1-dimensional numpy array with one weight per group.

class talon.optimization.**ExitStatus**(*value*)
Exit criteria of the optimization routine.

2.9 CLI module

These functions are available at the `talon.cli` module, which must be imported separately.

```
# import talon
import talon.cli
```

2.9.1 Utils

talon.cli.utils.**add_ndir_to_input**(*parser: argparse.ArgumentParser*)
This function adds the number of directions as input argument to a parser.

The `--ndir` argument is added to `parser`. The argument takes as input an integer which by default is equal to 1000.

Parameters **parser** – argparse.ArgumentParser Argument parser.

talon.cli.utils.**add_verbosity_and_force_to_parser**(*parser: argparse.ArgumentParser*)
This function adds the verbosity and force parameters to a parser.

After calling this method, the input parser will have the following boolean arguments.

- `--force`
- `--quiet`
- `--warn`
- `--info`
- `--debug`

Parameters **parser** – argparse.ArgumentParser Argument parser.

`talon.cli.utils.`**`assignment_to_mapping`**(*fpath: str*, *undirected: bool = True*) → collections.defaultdict

This function creates a mapping object from a streamline assignment file.

Parameters

- **fpath** – str Path to the file whose rows contain the assignment of each streamline. E.g., if the n-th row is '5 17', the n-th streamline is assigned to regions 5 and 17. The region labels must be integer values and separated by a blank space. Lines starting with # are skipped.
- **undirected** – bool True if the mapping must be undirected, False otherwise.

Returns

defaultdict Dictionary with keys being pairs of regions connected by streamlines and values being the list of streamline indices of those streamlines connecting the corresponding regions.

`talon.cli.utils.`**`check_can_write_file`**(*fpath: str*, *force: bool = False*)

Check if a file can be written.

The function checks if the file already exists, the user has the permission to write it, overwriting can be forced and, if the file does not exist, if the parent directory exists and is writable.

Parameters

- **fpath** – str path of the file to be checked.
- **force** – bool True if the file can be overwritten, False otherwise.

Raises

- **FileExistsError** – if the file exists and can not be overwritten.
- **PermissionError** – if the file esists and the user does not have the permission to write it.
- **PermissionError** – if the file does not exist, the parent directory exists and the user does not have the permission to write a file in it.
- **FileNotFoundError** – if file does not exist and the parent directory does not exist.

`talon.cli.utils.`**`mapping_to_groups_weights`**(*mapping: collections.defaultdict*, *connectome: Optional[numpy.ndarray] = None) -> (<class 'list'>, <class 'numpy.ndarray'>*)

Extract the streamline groups and weights from a mapping object.

Groups are lists of streamline indices that form a bundle. Weights are defined as follows: let N_g be the number of streamlines in group g, and let c_g be the connectivity between the regions linked by streamline bundle g. Each group g is then associated to a weight equal to

$$w_g = [N_g \cdot (1 + c_g)]^{-1}.$$

Parameters

- **mapping** – defaultdict dictionary with keys being pairs of regions connected by streamlines and values being the list of streamline indices of those streamlines connecting the corresponding regions.
- **connectome** – np.ndarray connectivity matrix to be employed. The first row and column correspond to the zero label.

Returns

tuple of length 2

- list of groups
- 1-dimensional np.ndarray with one weight per group

`talon.cli.utils.`**`parse_verbosity`**(*args: argparse.Namespace*)

This function applies the wanted verbosity level in logging specified by the parsed arguments given in input.

The default level is logging.WARNING.

Parameters **args** – argparse.Namespace Namespace parsed from inputs.

`talon.cli.utils.`**`setup_parser`**(***kwargs*) → argparse.ArgumentParser

Setup TALON argument parser.

This function returns an argparse.ArgumentParser object with `allow_abbrev=True` and `add_help=True`.

Parameters **kwargs** – dict dictionary that will be passed to the constructor of argparse.ArgumentParser.

Returns argparse.ArgumentParser object with the passed options plus `allow_abbrev=True` and `add_help=True`

2.9.2 Commands

Filter

`talon.cli.commands.filter.`**`run`**(*in_tracks: str, in_data: str, out_weights: str, force: bool, ndir: int, precomputed_indices_weights: List[str], save_generators_indices_weights: List[str], save_operator_pickle: str, operator_type: str, streamline_assignment: str, connectome: str, sigma: float, allow_negative_x: bool, maxiter: int, objective_relative_tolerance: float, x_absolute_tolerance: float, **kwargs*)

Parameters

- **in_tracks** – str Input tractogram file in RAS+ and mm space. The streamline coordinate (0,0,0) refers to the center of the voxel. Must be in NiBabel-readable format (.trk or .tck).
- **in_data** – str Input data to be fitted. Serves also as reference space for tractogram. Must be in NiBabel-readable format (.nii or .nii.gz).
- **out_weights** – str Output text file containing the streamline weights.
- **force** – bool True if the file can be overwritten, False otherwise.
- **ndir** – int Number of directions for the voxelization.

- **precomputed_indices_weights** – List Uses the indices and weights passed as input to build the linear operator. The two matrices must be defined on the same number of directions (ndir) as the ones that are used at the call of this script.
- **save_generators_indices_weights** – List Saves the linear operator as three separate files.
- **save_operator_pickle** – str Saves the linear operator with pickle. Only available when *operator_type* is 'fast' or 'reference'.
- **operator_type** – str Type of operator to use. Default: *fast*. Choiches: 'reference', 'fast', 'opencl'.
- **streamline_assignment** – str Path to the file whose rows contain the assignment of each streamline. E.g., if the n-th row is '5 17', the n-th streamline is assigned to regions 5 and 17. The region labels must be integer values and separated by a blank space. Lines starting with # are skipped.
- **connectome** – str Path to the connectivity matrix to be employed in txt format. The first row and column correspond to the zero label.
- **sigma** – float Sets the regularization scale parameter as in (Frigo, 2021). The final value of lambda is *sigma*max(||At*data||/gwei)*, where sigma is the passed parameter, ||*At*data*|| is the 2-norm of the product between the transposed linear operator and the data, and *gwei* is the vector of the weights associated to each group of streamlines.
- **allow_negative_x** – bool Disables the non negativity constraint.
- **maxiter** – int Sets maximum number of iterations. Default: 1000.
- **objective_relative_tolerance** – float Sets relative tolerance on cost function. Default: 1e-6.
- **x_absolute_tolerance** – float Sets absolute tolerance on variable. Default: 1e-6.

Voxelize

talon.cli.commands.voxelize.**run**(*in_tracks*, *in_img*, *out_ind*, *out_wei*, *force*, *ndir*, ***kwargs*)

This function reads tractogram files and writes the corresponding indices and weights files.

Parameters

- **in_tracks** – str Tractogram file to be voxelized in RAS+ and mm space. The streamline coordinate (0,0,0) refers to the center of the voxel. Must be in NiBabel-readable format (.trk or .tck).
- **in_img** – str Image serving as space reference. Must be in NiBabel-readable format (.nii or .nii.gz).
- **out_ind** – str Path where the indices will be saved in .npz format.
- **out_wei** – str Path where the weights will be saved in .npz format.
- **force** – bool True if the file can be overwritten, False otherwise.
- **ndir** – Number of directions for the voxelization.

2.10 How to cite talon

If you use talon in your research, please cite the package in the following format.

> First Author, Second Author, Third Author. TALON: Tractograms As Linear Operators in Neuroimaging. CoBCoM, 2021.

```
@misc{cobcomtalon,
      author = {Author, First and Author, Second and Author, Third},
      title = {TALON: Tractograms As Linear Operators in Neuroimaging},
      howpublished = {CoBCoM},
      year = {2021}
     }
```

2.11 List of Contributors

Talon was conceived in the ATHENA Project Team at Inria Sophia Antipolis - Méditerranée. The package was initially developed within the Computational Brain Connectivity Mapping (CoBCoM) project by:

- Samuel Deslauriers-Gauthier, ATHENA Project Team, Inria Sophia Antipolis - Méditerranée.
- Matteo Frigo , ATHENA Project Team, Inria Sophia Antipolis - Méditerranée.
- Mauro Zucchelli , ATHENA Project Team, Inria Sophia Antipolis - Méditerranée.

The project is currently maintained by:

- Samuel Deslauriers-Gauthier, ATHENA Project Team, Inria Sophia Antipolis - Méditerranée.
- Matteo Frigo , ATHENA Project Team, Inria Sophia Antipolis - Méditerranée.

2.12 License

2.13 Funding

The development of talon was funded by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (ERC Advanced Grant agreement No 694665: CoBCoM - Computational Brain Connectivity Mapping).

European Research Council

Established by the European Commission

BIBLIOGRAPHY

[2009b] Beck, Amir, and Marc Teboulle. “A fast iterative shrinkage-thresholding algorithm for linear inverse problems.” SIAM journal on imaging sciences 2.1 (2009): 183-202.

[2011j] Jenatton, Rodolphe, et al. “Proximal methods for hierarchical sparse coding.” Journal of Machine Learning Research 12.Jul (2011): 2297-2334.

PYTHON MODULE INDEX

t

INDEX