# talon Documentation

*Release 0.2.0*

**Samuel Deslauriers-Gauthier, Matteo Frigo, Mauro Zucchelli**

**Jan 18, 2021**

# CONTENTS

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

CHAPTER

ONE

# USER DOCUMENTATION

## 1.1 Installation

For now, most people are expected to use *talon* as developers and should install it by moving into the root *talon* directory and running:

```
python setup.py develop
```

To make sure *talon* was installed correctly, test it by running:

```
python -m unittest
```

You should see that a number of tests were run and no errors occurred.

## 1.2 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 4 parts:

- *Creating a test tractogram*
- *Building the linear operator*
- *Generating data with a linear operator*
- *Explaining data with a linear operator*

### 1.2.1 Creating a test tractogram

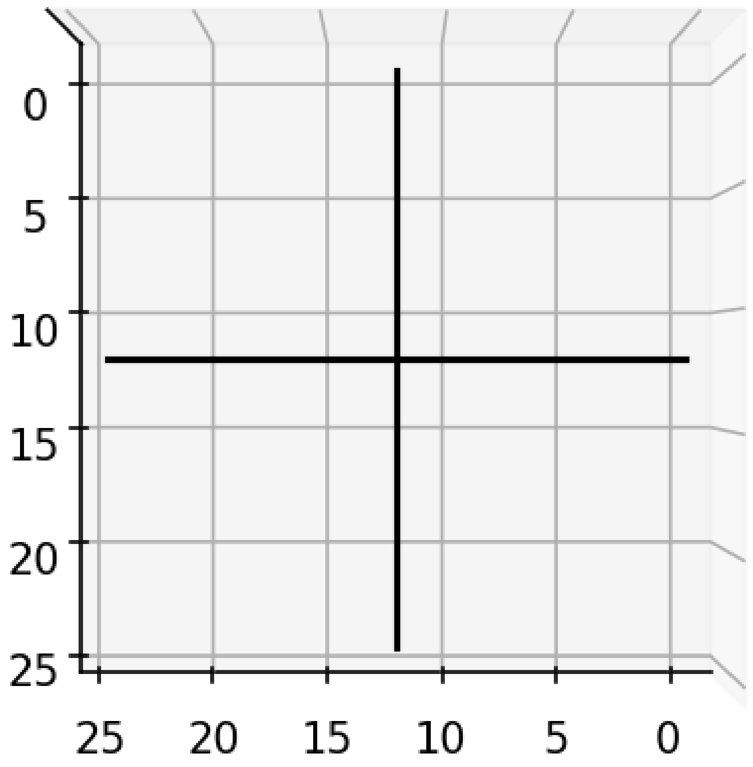
To generate data using `talon`, we need a tractogram. Here we will generate streamlines organised into a + sign.

```python
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`.

```python
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()
```

You should see the following image:



### 1.2.2 Building the 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)
```

### 1.2.3 Generating data with a linear operator

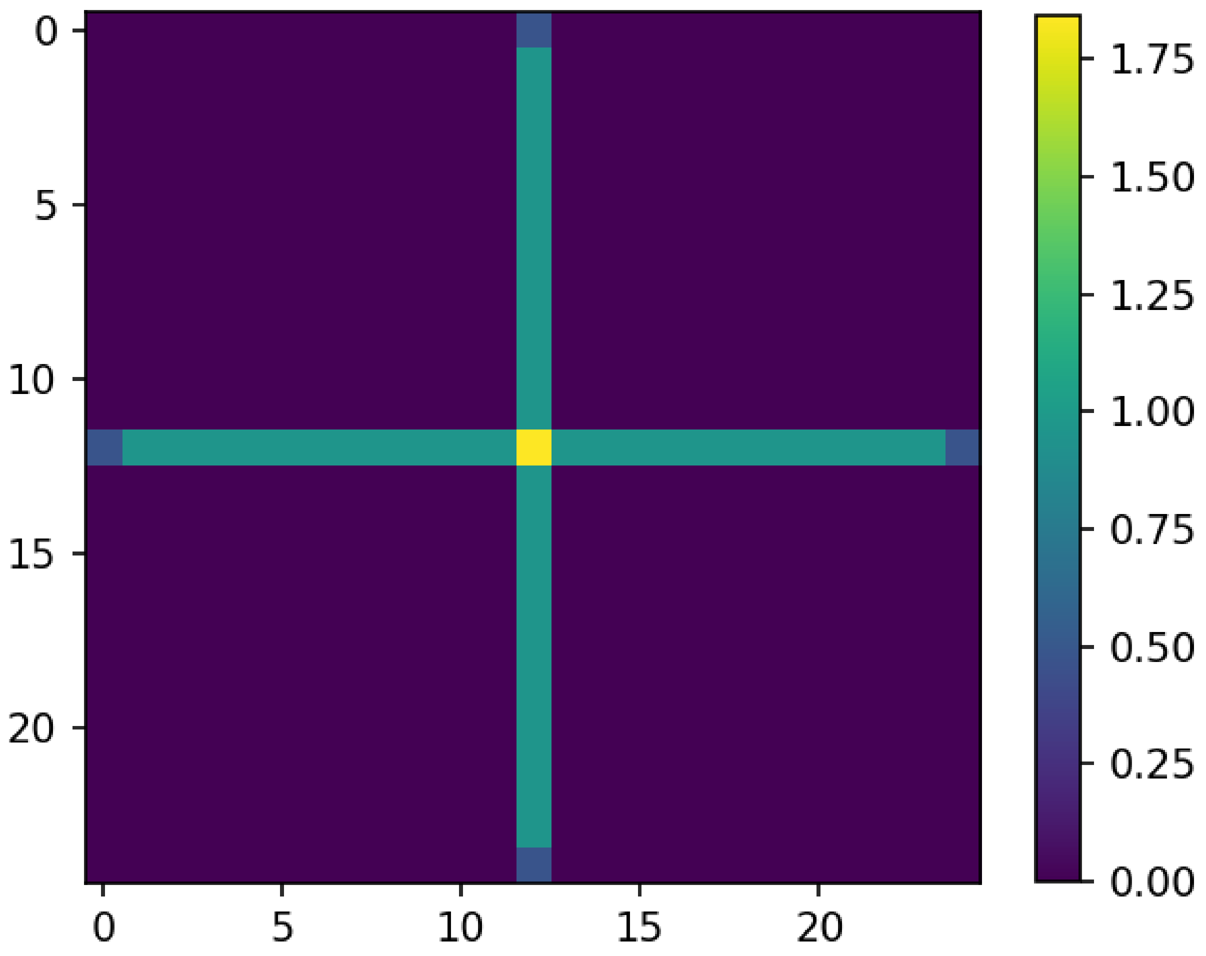
To generate data simply multiply (using the `@` operator) the linear operator by a weight vector.

```
# 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.

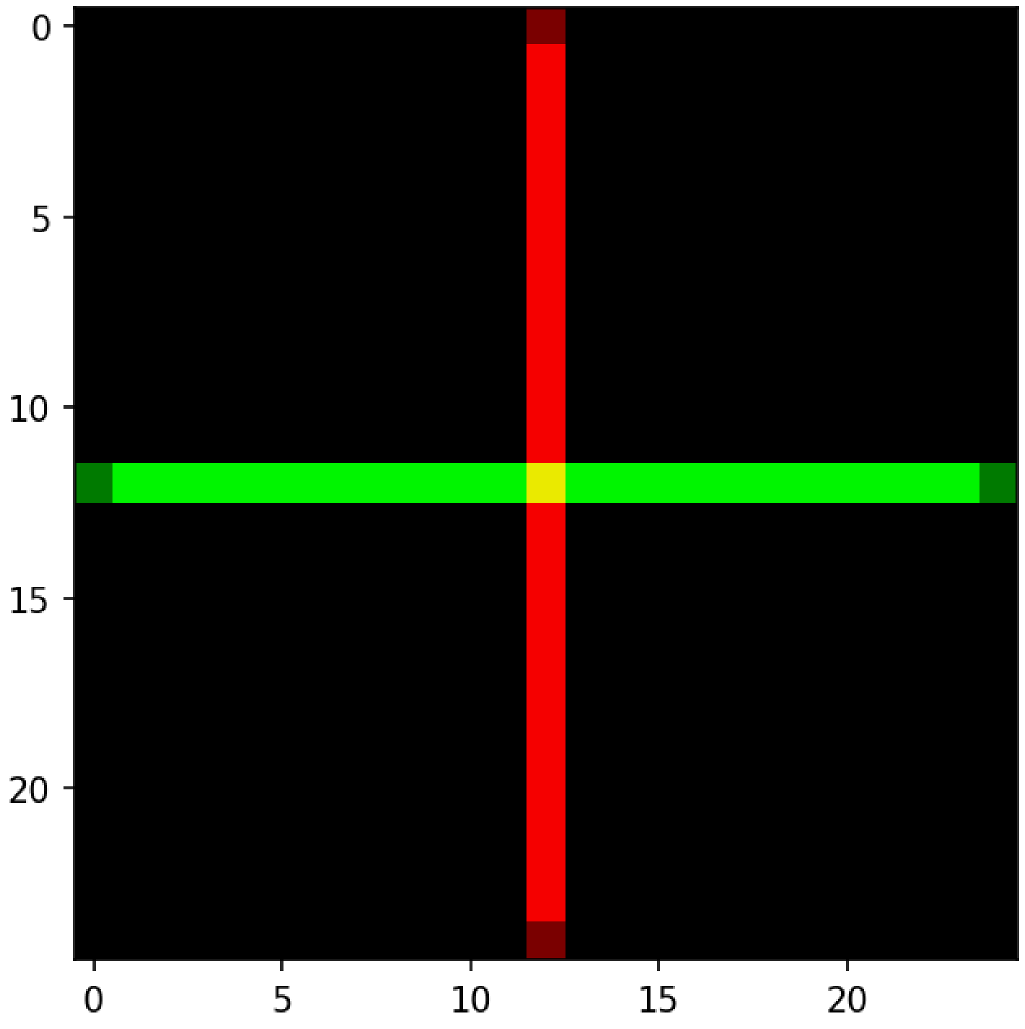
```
image = b.reshape(image_shape)

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.

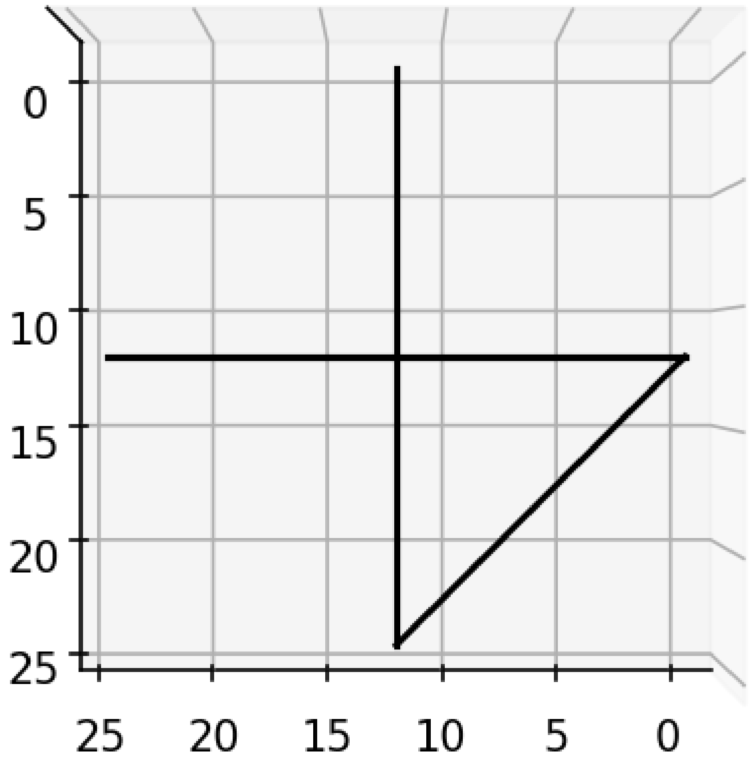
```
rgb_image = m.reshape(image_shape + (3,))

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



### 1.2.4 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.

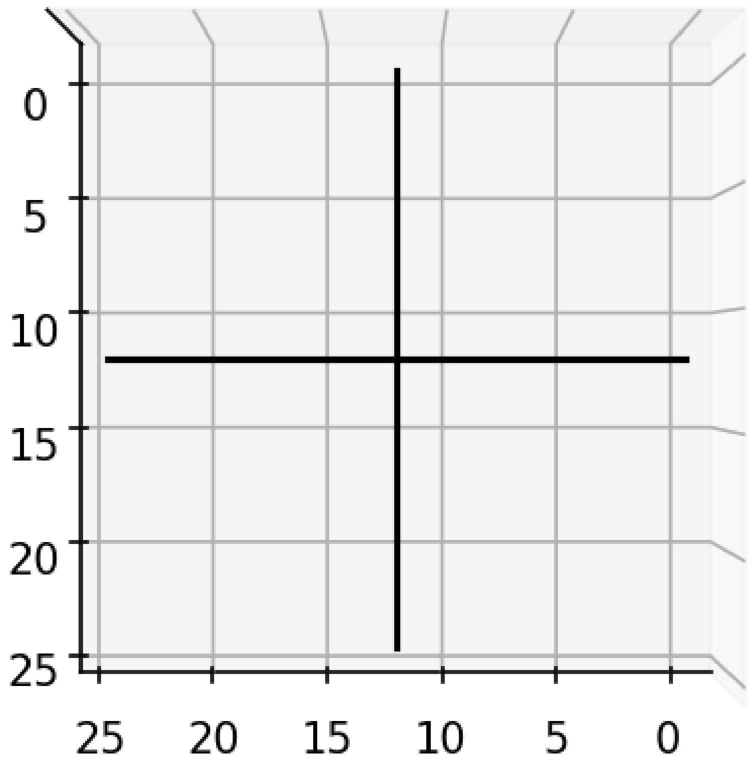
```
solution = talon.solve(Z, b, reg_term=positivity_constraint)
print('solution.x = [%.2f, %.2f, %.2f]' % tuple(solution.x))
```

```
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()
```



## 1.3 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 $A$.

### 1.3.1 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 $\infty$ 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 $\lambda$ 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 $\ell_1$ regularization is a particular case of structure sparsity where there is only one group containing all the admissible indices. Assuming that these indices are $0 \dots n$, the following line of code defines the problem for classical $\ell_1$ 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*.

## 1.3.2 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.

### 1.3.3 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.

### 1.3.4 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)

# 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()
```

(continues on next page)

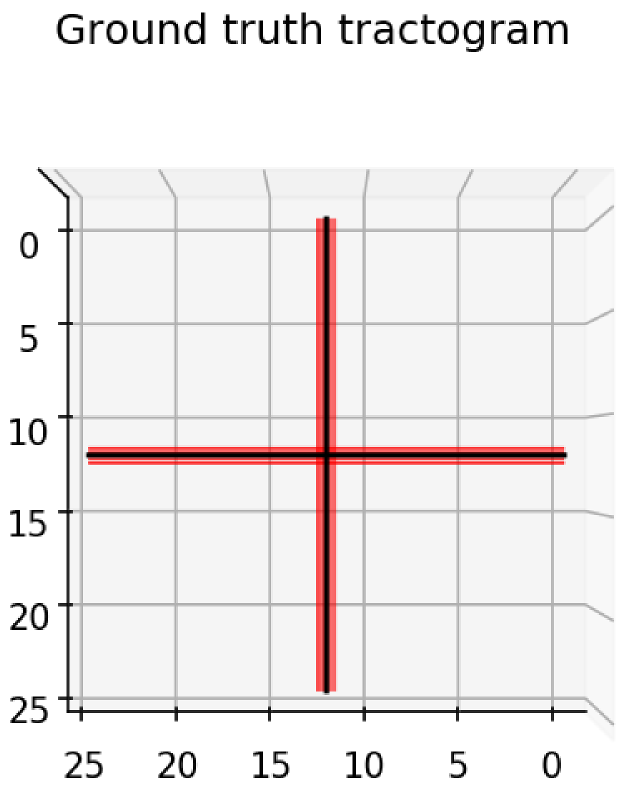
(continued from previous page)

```
    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

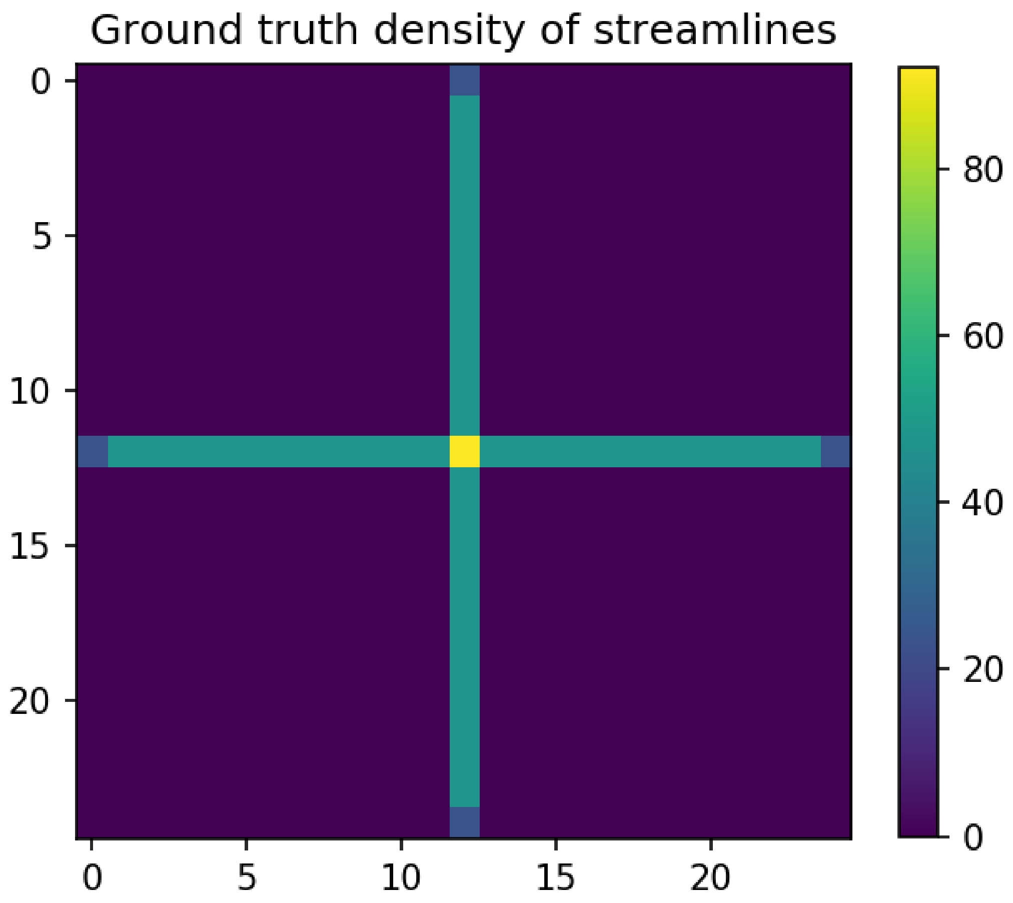
```
plt.figure(figsize=(5, 5), dpi=150)
plt.imshow(image[:, :, center])
```

(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)

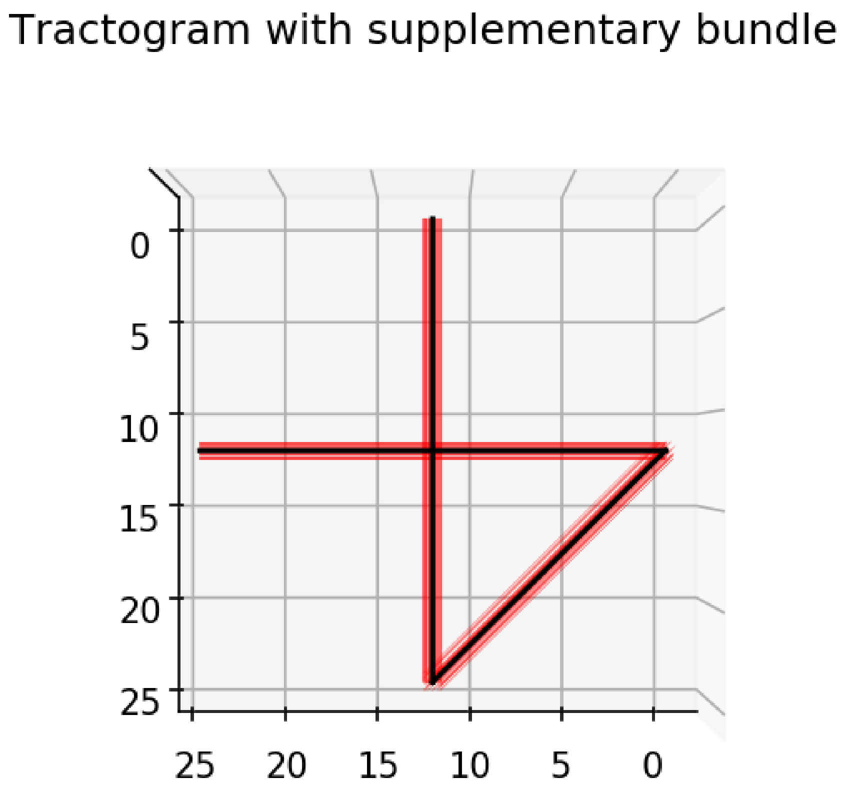
(continued from previous page)

```
    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: 0.99999141147578718
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

## 1.4 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

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

### 1.4.1 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 $\alpha$ 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 $\alpha$ and $C$ for set $\beta$, 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,)
```

## 1.5 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.

### 1.5.1 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.]]
```

CHAPTER

TWO

# API DOCUMENTATION

## 2.1 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

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

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'` 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 $\lambda$ 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*, *data*, *reg_term=None*, *cost_reltol=1e-06*, *x_abstol=1e-06*, *max_nit=1000*, *x0=None*, *verbose='LOW'*)

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.utils.`**`directions`**(*number_of_points=180*)

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

## 2.2 Classes

### 2.2.1 LinearOperator

**class** talon.core.**LinearOperator**(*generators*, *indices_of_generators*, *weights*)

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

**property columns**

Returns the indices of the nonzero columns.

**Type** int

**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

### 2.2.2 ConcatenatedLinearOperator

**class** talon.core.**ConcatenatedLinearOperator**(*operators*, *axis*)

**__init__**(*operators*, *axis*)
Concatenated LinearOperator object

The ConcatenatedLinearOperator class implements the vertical or horizontal concatenation of LinearOperator objects. It is endowed with the multiplication operation (@).

**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 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

# 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

# INDEX