A beginner’s guide to shape analysis using Deformetrica

Benoit Martin Follow
Aug 23 · 13 min read

Deformetrica is a great toolbox for statistical shape analysis. This guide will provide step-by-step instructions from a clean setup to a fully running example.

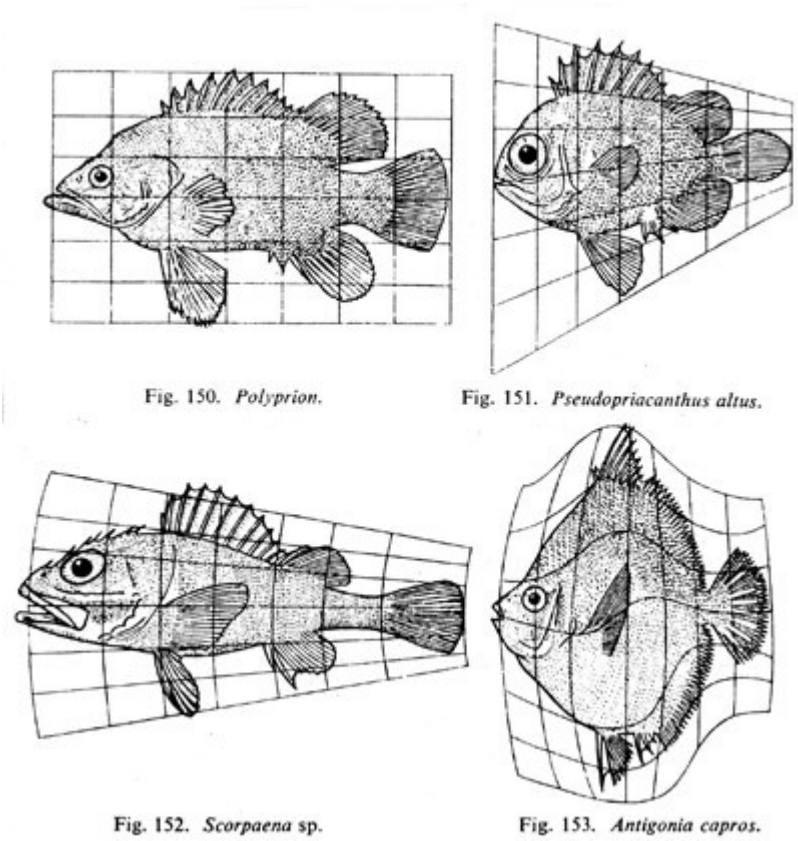


Fig. 150. *Polyprion.*

Fig. 151. *Pseudopriacanthus altus.*

Fig. 152. *Scorpaena* sp.

Fig. 153. *Antigonia capros.*

Transformation Diagrams from On Growth and Form by D’Arcy Thompson (published by Cambridge University Press in 1917)

. . .

Table of Contents

Overview

Deformetrica is an open-source software application developed by the Aramis Lab team (Inria, CNRS, Inserm, Sorbonne Université) at the Brain and Spine Institute in Paris. This software can be viewed as a toolbox containing a number of statistical analysis methods for 2D and 3D shape data. It essentially computes deformations of the 2D or 3D ambient space, which, in turn, warps any object embedded in this space, whether this object is a curve, a surface, a structured or unstructured set of points, an image, or any combination of them.

Deformetrica comes with three main applications:

- **registration**: estimates the best possible deformation between two sets of objects
- **atlas construction**: estimates an average object configuration from a collection of object sets, and the deformations from this average to each sample in the collection
- **geodesic regression**: estimates an time-series object constrained to match as closely as possible a set of observations indexed by time

Deformetrica has very little requirements about the data it can deal with.
In particular, it does *not* require point correspondence between objects!

. . .

At the time this article was written, ***Deformetrica* version 4.3.0 RC** is the latest, up-to-date version available. The following article will be based on this version.

Installing Deformetrica

Requirements: *Deformetrica* 4.3 RC is available on Linux and MacOS for Python 3.6 and 3.7. For any other configuration, it is possible to open an issue on the official repository or write a message on the official Google group.

Before continuing: make sure that Conda or Miniconda is installed before following the next steps.

Currently, *Deformetrica* is available through 2 main sources: the repository and a conda package. For an easy and simple setup it is recommended to use the conda setup method.

```
$ conda create -n deformetrica python=3.7
$ source activate deformetrica
$ conda install -c pytorch -c conda-forge -c anaconda -c
aramislab/label/rc deformetrica
```

These conda commands will create and activate an isolated conda environment based on python 3.7. Then, conda will install

deformetrica and its dependencies (this step can take a long time depending on your connection. Now would be a perfect time to grab a ☕).

Once the installation is complete, you can check that Deformetrica has successfully been installed by running the following command:

```
$ deformetrica --help
usage: deformetrica [-h]
{estimate,compute,initialize,finalize,gui}

Statistical analysis of 2D and 3D shape data.

version 4.3.0

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

command:
  {estimate,compute,initialize,finalize,gui}
```

If your console output looks like the above, congratulations 🎉 Deformetrica has successfully been installed within a dedicated conda environment!

. . .

In the next 2 examples we will first be running an Atlas on a set of 2D polylines representing simple skull structures using Deformetrica’s command line interface (CLI) and secondly, we will be running a Geodesic Regression using the Python application

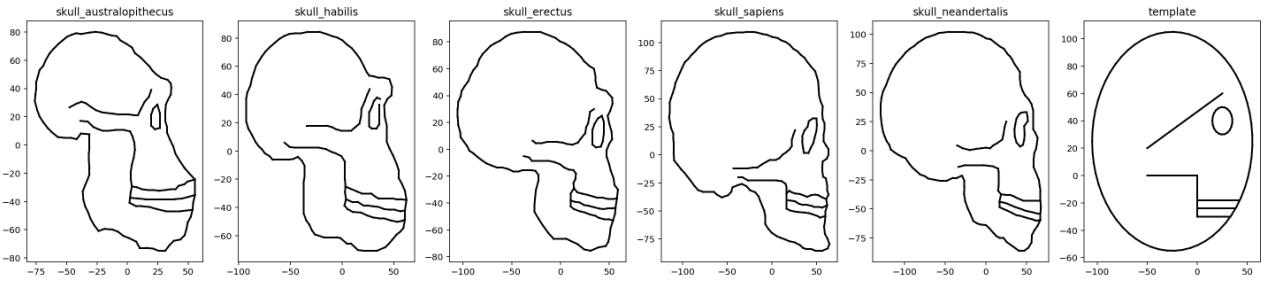
programming interface (API).

Before continuing: the example dataset should be downloaded in order to run the following code snippets. The first example will be based on the example given in the `atlas/landmark/2d/skulls` folder. The second example will be based on the `registration/landmark/3d/surprise` folder. All the following paths are relative to the example folder corresponding to the example.

Running a simple example: Deterministic Atlas

Given a family of objects, the atlas model proposes to learn a **template shape** which corresponds to a **average of the given objects**. A complete description of the mathematics for the deterministic atlas model is given on the LDDMM page from the official wiki.

Let’s start by viewing the data on which the atlas will be estimated. The skulls dataset in composed of 5 2D skull polylines and an initial template 2D polyline. Here is a matplotlib output of the data:



Plot showing the skulls dataset used as an Atlas input.

Side note: The source code used to produce the above plots is available on this Google Colab link.

Before we run the atlas estimation, we can start by checking the different configuration xml files. If you wish to skip this section, feel free to do so.

- the `model.xml` file:

```
<?xml version=”1.0"?>
<model>
 <model-type>DeterministicAtlas</model-type>

<template>
  <object id=”skull”>
   <deformable-object-type>Polyline</deformable-object-type>
   <attachment-type>varifold</attachment-type>
   <kernel-width>20</kernel-width>
   <kernel-type>keops</kernel-type>
   <noise-std>1</noise-std>
   <filename>data/template.vtk</filename>
  </object>
 </template>

 <deformation-parameters>
  <kernel-width>20</kernel-width>
```

```
  <kernel-type>keops</kernel-type>
 </deformation-parameters>
</model>
```

This file contains information on the statistical model that is to be run. In this example, the model type defines as a `DeterministicAtlas`.
The template section defines the list of objects considered in the computation. In our case, a single polyline object is used.
Finally, the deformation parameters section defines the kernel parameters that will be used to perform the deformations. Again, more information on this can be found on the LDDMM page.

Additional information on the model xml file can be found on the official wiki using the following link.

- The `data_set.xml` file:

```
<?xml version=”1.0"?>
<data-set>
 <subject id=”australopithecus”>
  <visit id=”experiment”>
   <filename
object_id=”skull”>data/skull_australopithecus.vtk</filename>
  </visit>
 </subject>

[... snip ...]

 <subject id=”sapiens”>
  <visit id=”experiment”>
   <filename
object_id=”skull”>data/skull_sapiens.vtk</filename>
  </visit>
```

```
 </subject>
</data-set>
```

This file contains the paths to all the objects that will be considered when computing the atlas. Here a relative or an absolute path can be given. Each observation is a **subject** section with a subject id. Each subject section contains a single visit which contains the list of filenames, with the id of the shape as an attribute.

Additional information on the dataset xml file can be found on the official wiki using the following link.

- the `optimization_parameters.xml` file:

```
<?xml version=”1.0"?>
<optimization-parameters>
 <optimization-method-type>GradientAscent</optimization-
method-type>
</optimization-parameters>
```

Kept to a minimum for this example, this file simply specifies the **optimization-method-type** that should be used, in this case a Gradient Ascent optimizer.

Additional information on the optimization xml file can be found on the official wiki using the following link.

. . .

Now that the configuration files have been set up, we can carry out the actual atlas estimation by running the *deformetrica estimate* command and by inputting the xml files and optionally setting the verbose setting to `INFO` as follows:

```
$ deformetrica estimate model.xml data_set.xml -p
optimization_parameters.xml -v INFO
```

This will produce a console output similar to the following:

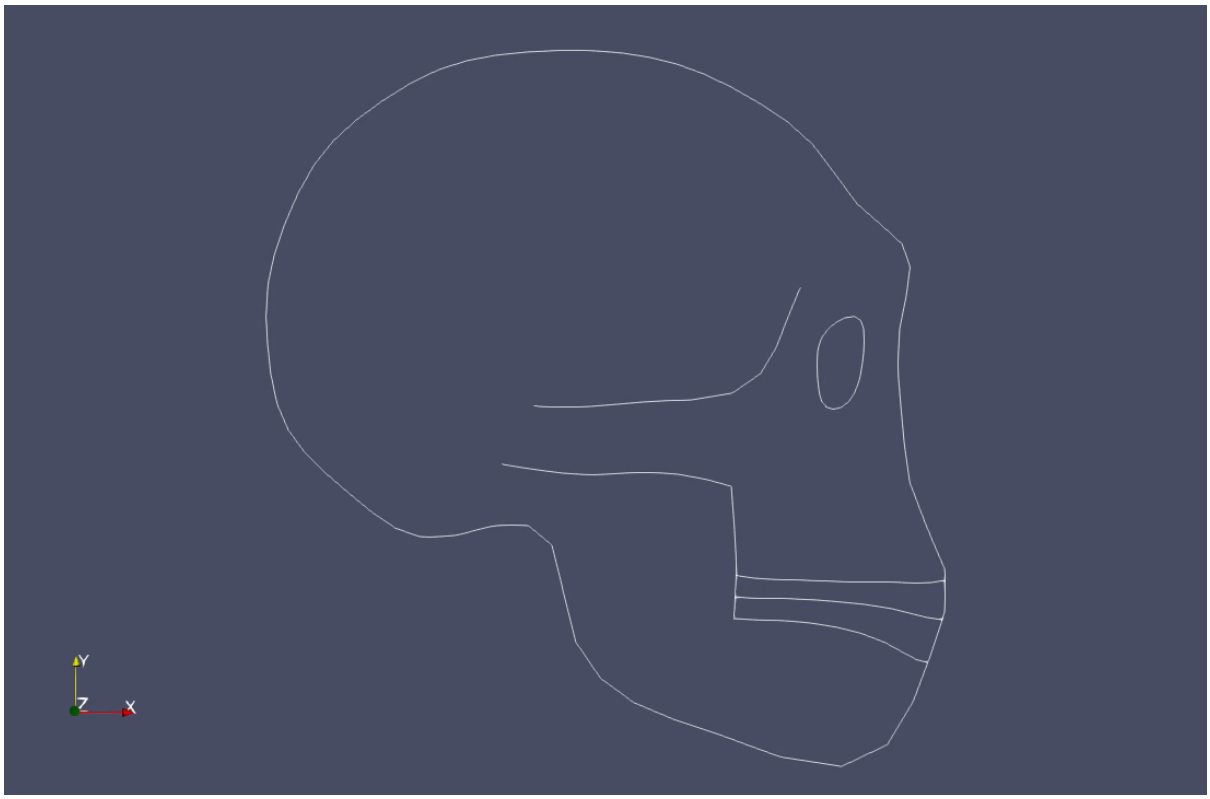
```
$ deformetrica estimate model.xml data_set.xml -p
optimization_parameters.xml -v INFO
No CUDA runtime is found, using CUDA_HOME='/usr/local/cuda'
INFO:deformetrica.__main__:No output directory defined,
using default: XXX/examples/atlas/landmark/2d/skulls/output
Logger has been set to: INFO
>> No initial CP spacing given: using diffeo kernel width of
40.0
OMP_NUM_THREADS was not found in environment variables. An
automatic value will be set.
OMP_NUM_THREADS will be set to 2
Could not set torch settings.
>> No specified state-file. By default, Deformetrica state
will by saved in file: XXX/examples/atlas/landmark/2d/skulls
/output/deformetrica-state.p.
>> Removing the pre-existing state file with same path.
>> Set of 16 control points defined.
>> Momenta initialized to zero, for 5 subjects.
>> Started estimator: GradientAscent
Compiling libKeOpstorch1a4ca06227 in XXX/.cache/pykeops-
1.1.1//build-libKeOpstorch1a4ca06227:
       formula: Sum_Reduction(Exp(-G*SqDist(X,Y)) * P,0)
       aliases: G = Pm(0,1); X = Vi(1,2); Y = Vj(2,2); P =
Vj(3,2);
       dtype  : float32
... Done.

[... snip ...]
```

```
Compiling libKeOpstorcha15fafbaa6 in XXX/.cache/pykeops-
1.1.1//build-libKeOpstorcha15fafbaa6:
       formula: Grad_WithSavedForward(Sum_Reduction((Px|Py)
* Exp(-G*SqDist(X,Y)) * (X-Y),0), Var(4,2,1), Var(5,2,0),
Var(6,2,0))
       aliases: G = Pm(0,1); X = Vi(1,2); Y = Vj(2,2); Px =
Vi(3,2); Py = Vj(4,2); Var(5,2,0); Var(6,2,0);
       dtype  : float32
... Done.
----------------------- Iteration: 0 -----------------------
>> Log-likelihood = -1.773E+05   [ attachment = -1.773E+05 ;
regularity = 0.000E+00 ]
>> Step size and gradient norm:
  5.306E-04   and   1.885E+03  [ momenta ]
----------------------- Iteration: 1 -----------------------
>> Log-likelihood = -1.755E+05   [ attachment = -1.755E+05 ;
regularity = -1.706E+00 ]
>> Step size and gradient norm:
  7.960E-04   and   1.882E+03  [ momenta ]
----------------------- Iteration: 2 -----------------------
>> Log-likelihood = -1.726E+05   [ attachment = -1.726E+05 ;
regularity = -1.067E+01 ]
>> Step size and gradient norm:
  1.194E-03   and   1.877E+03  [ momenta ]


[... snip ...]


----------------------- Iteration: 98
-----------------------
>> Log-likelihood = -1.896E+04   [ attachment = -5.672E+03 ;
regularity = -1.328E+04 ]
>> Step size and gradient norm:
  1.893E-03   and   1.514E+03  [ landmark_points ]
  2.107E-02   and   1.122E+02  [ momenta ]
----------------------- Iteration: 99
-----------------------
>> Log-likelihood = -1.889E+04   [ attachment = -5.578E+03 ;
regularity = -1.331E+04 ]
>> Step size and gradient norm:
  1.893E-03   and   1.329E+03  [ landmark_points ]
  1.054E-02   and   1.395E+02  [ momenta ]
----------------------- Iteration: 100
-----------------------
>> Log-likelihood = -1.883E+04   [ attachment = -5.646E+03 ;
regularity = -1.318E+04 ]
>> Estimation took: 02 minutes and 58 seconds
Deformetrica.__del__()
```

Note that by default the Keops kernel is used to perform deformations. This results in the compilation of a few libraries due to the inner workings of the Keops library. This step might take a few minutes but, fortunately, is only done once on the first run. By default, an **output folder** will be created in the current working directory and will contain numerous files:

- a date and time stamped log file containing the console output
- a few `.vtk` files which represent the final deformations as well as the intermediate deformation steps
- some `.txt` files which correspond to the estimated *control-points*, *momentum* and *residuals*
- a `deformetrica-state.p` file that can be used to resume an estimation

You can now import the intermediate *vtk* files into your favorite vtk viewer. Here is what an animation would look like in ParaView for the australopithecus subject:



Animation showing the estimated template (first frame) being deformed onto the target australopithecus subject (last frame).

Side note: This example can also be found running using the python API as shown in the Google Colab link.

. . .

Using the Python API: Geodesic Regression

What if *Deformetrica* is to be used within another Python application or script? We are in luck! Fortunately, Deformetrica exposes most of its functionalities in its Python API. It is as simple as importing the *deformetrica* python package.

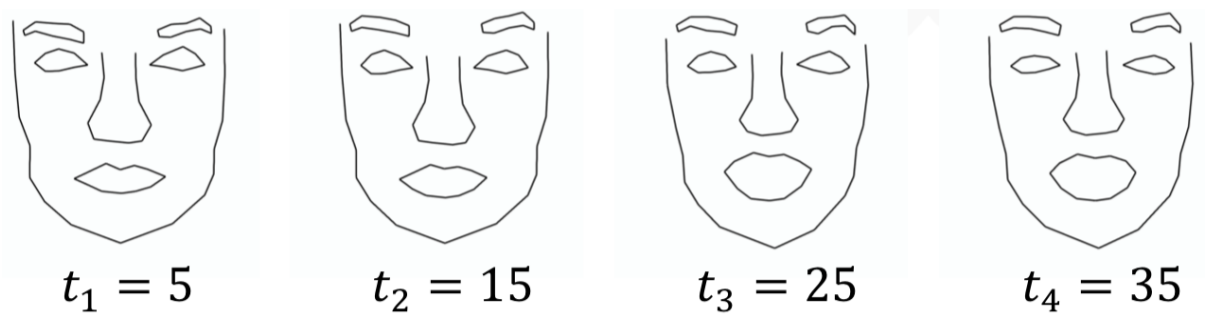
```
import deformetrica as dfca
```

. . .

The geodesic regression model will construct a shape trajectory (geodesic) that will be as close as possible to the given time-spaced targets. In practice, this will utilize the given time-spaced input data to compute a geodesic. From this, and depending on the parameters used for computation, it will be possible to construct the intermediate data that closely fits to the geodesic.

Note that this example is based on the data found in the `registration/landmark/3d/surprise/data` folder that should have previously been downloaded. This data is composed of a number of 3D polyline *vtk* files representing a surprised face.

We can first start by viewing a sample of our input data:



Sample input data for time 5, 15, 25 and 35

The following code snippet illustrates how our time-spaced input data can be used to reconstruct the missing intermediates using a Geodesic Regression model.
We first start by setting a few dictionaries that correspond to the

input dataset and template that should be used as well as the estimator options. These settings closely relate to the settings that can be found in the xml configuration files. In this example, we also take time to set an estimator callback that is used to retrieve intermediate metrics corresponding to the method’s convergence that will later be plotted using matplotlib.
Finally, a `Deformetrica` object is instantiated (setting the output directory and the verbosity) and the `estimate_geodesic_estimation` method is called.

```
import os
import numpy as np
import matplotlib.pyplot as plt
import deformetrica as dfca

data_path = "./"
data_base = os.path.join(data_path, 'regression/landmark/3d/surprise/data/')

iteration_status_dictionaries = []


def estimator_callback(status_dict):
    iteration_status_dictionaries.append(status_dict)
    return True


dataset_specifications = {
    'dataset_filenames': [
        [{'skull': os.path.join(data_base, 'sub-F001_ses-000.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-005.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-010.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-015.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-020.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-025.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-030.vtk')},
         {'skull': os.path.join(data_base, 'sub-F001_ses-035.vtk')}]],
    'visit_ages': [[0, 5, 10, 15, 20, 25, 30, 35]],
    'subject_ids': [['ses-000', 'ses-005', 'ses-010', 'ses-015', 'ses-020', '
}
template_specifications = {
    'skull': {'deformable_object_type': 'polyline',
              'noise_std': 0.0035,
              'filename': os.path.join(data_base, 'ForInitialization__Templat
              'attachment_type': 'landmark'}}
estimator_options = {'optimization_method_type': 'GradientAscent', 'max_itera
                     'convergence_tolerance': 1e-5, 'initial_step_size': 1e-6


deformetrica = dfca.Deformetrica(output_dir='output', verbosity='INFO')

deformetrica.estimate_geodesic_regression(
    template_specifications, dataset_specifications,
    estimator_options=estimator_options,
```

When using the Geodesic Regression model, 3 important parameters should be defined with care:

- the `concentration_of_time_points` which controls of the discretization step for the geodesic. The default value is 10, but it can be set to lower/higher values depending on the time scale and the desired precision
- the `t0` attribute in the model options controls the point from which the geodesic is computed. In practice, the provided initial template will be the point on the geodesic at *t0*, hence it must be initialized with care. By default *t0*'s value will be set to the mean of the observation times
- By default, the template is frozen during a geodesic regression. It is advised to keep it that way, since estimating both the template and the momenta of the regression can be ill-defined

The previously added estimator callback is extremely useful to retrieve a number of metrics that can be used to evaluate the model's convergence. In our example, a list named `iteration_status_dictionaries` is appended with updated metrics after each estimator iteration.
We can start by exploring the dictionary keys and value types that are used. The following code snippet should help us:

```
import numpy as np

def explore_dict(d):
  for k, v in d.items():
```

```
    str = "id(dict)=%i : k=%s : v=%s"%(id(d), k,
type(v).__name__)
    if isinstance(v, np.ndarray):
      str += " : %s"%(v.shape,)
    print(str)

    if isinstance(v, dict):
      explore_dict(v)

explore_dict(iteration_status_dictionaries[-1])
```

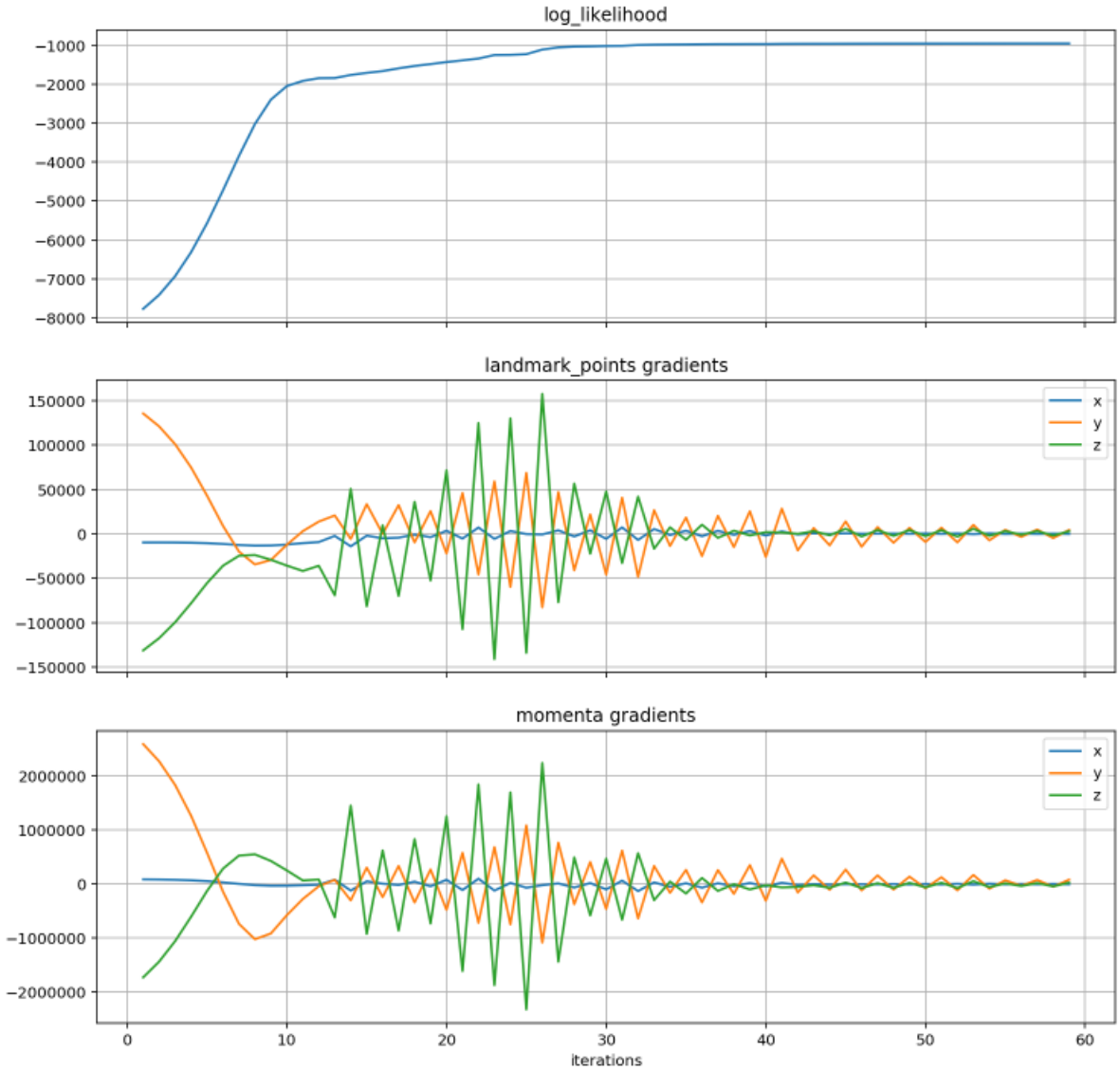
This python function will output the dictionary’s unique id as well as its containing key and value types. The output will look similar to:

```
id(dict)=139864914661808 : k='current_iteration' : v=int
id(dict)=139864914661808 : k='current_log_likelihood' :
v=float id(dict)=139864914661808 : k='current_attachment' :
v=float id(dict)=139864914661808 : k='current_regularity' :
v=float id(dict)=139864914661808 : k='gradient' : v=dict
id(dict)=139864906988712 : k='landmark_points' : v=ndarray :
(83, 3) id(dict)=139864906988712 : k='momenta' : v=ndarray :
(83, 3)
```

From this console output, we now have a better understanding of the inner structures. The metrics that are of interest are the `log_likelihood` and the two `gradient` numpy *ndarrays*. These metrics will give some insight on the method’s overall convergence. Let’s now plot these values using matplotlib’s subplot functionality. The following code snippet will plot 3 line graphs representing the estimator's iteration against the `log_likelihood` , the `landmark_points` gradients and the `momenta` gradients respectively.

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


x = np.arange(1, iteration_status_dictionaries[-1]
['current_iteration']+1)


figsize=12
fig, (ax1, ax2, ax3) = plt.subplots(3, 1, figsize=
(figsize,figsize))
plt.xlabel('iterations')


ax1.plot(x, [it_data['current_log_likelihood'] for it_data
in iteration_status_dictionaries], label='log_likelihood')
ax1.title.set_text('log_likelihood')
ax1.grid(True)


ax2.plot(x, [np.sum(it_data['gradient']['landmark_points'],
axis=0) for it_data in iteration_status_dictionaries])
ax2.legend(['x', 'y', 'z'])
ax2.title.set_text('landmark_points gradients')
ax2.grid(True)


ax3.plot(x, [np.sum(it_data['gradient']['momenta'], axis=0)
for it_data in iteration_status_dictionaries])
ax3.legend(['x', 'y', 'z'])
ax3.title.set_text('momenta gradients')
ax3.grid(True)


plt.show()
```

The following plots are produced:



Multiple plots showing the estimator's iteration vs the log-likelihood, the landmark-points gradients and the momenta gradients respectively

As can be seen from the above line graphs, there clearly seems to be 3 distinct phases that can be identified:

1. **From iteration 1 and 10**: the `log_likelihood` curve is very steep as most of the Gradient Ascent optimization steps are very beneficial. The initial optimization step size can be modified using the `initial_step_size` parameter. This will have an impact on the size of each optimization steps that will be taken

by the optimizer.

2. **From iteration 11 and 35**: the Gradient Ascent method manages to further optimize the `log_likelihood` by aggressively varying the gradients for the y and z dimensions.
3. **From iteration 36 to the end**: the `log_likelihood` is very slowly being optimized until it reaches the pre-defined `convergence_tolerance` threshold.

It is also worth noticing that the gradients have a close to 0 value on the x dimension. This is due to the nature of our input data: the given polylines mostly differ on the y and z dimensions.

As with the CLI, several output files are generated and stored within an output folder (see the previous section for more details). These files can be imported into a *vtk* viewer. Here is an animation built using ParaView:

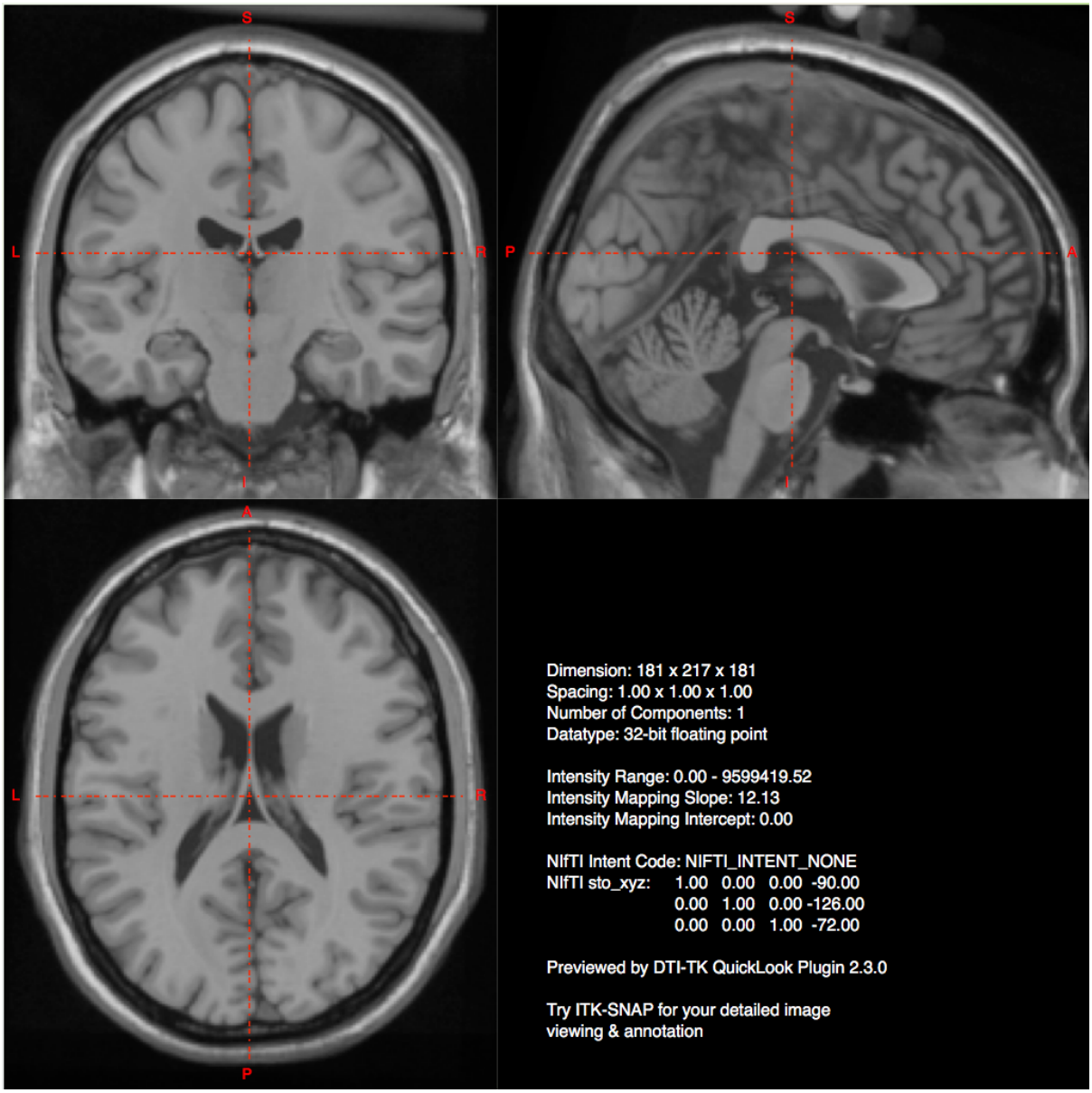
Geodesic regression animation generated from ParaView

Note: This example can also be found at this Google Colab link.

. . .

Using the command line tool: Registration

In a more medical context, *Deformetrica* can be used on to perform a Registration. In practice, the following example will be using a set of 3D T1 weighted Magnetic Resonance Images (MRIs). A full-resolution (7 million voxels) MR image registration can be done in 2–3 minutes, with a low GPU memory footprint. Note that these computations are extremely intensive due to the nature of the data. It is advised to accelerate the computation using a CUDA capable GPU.



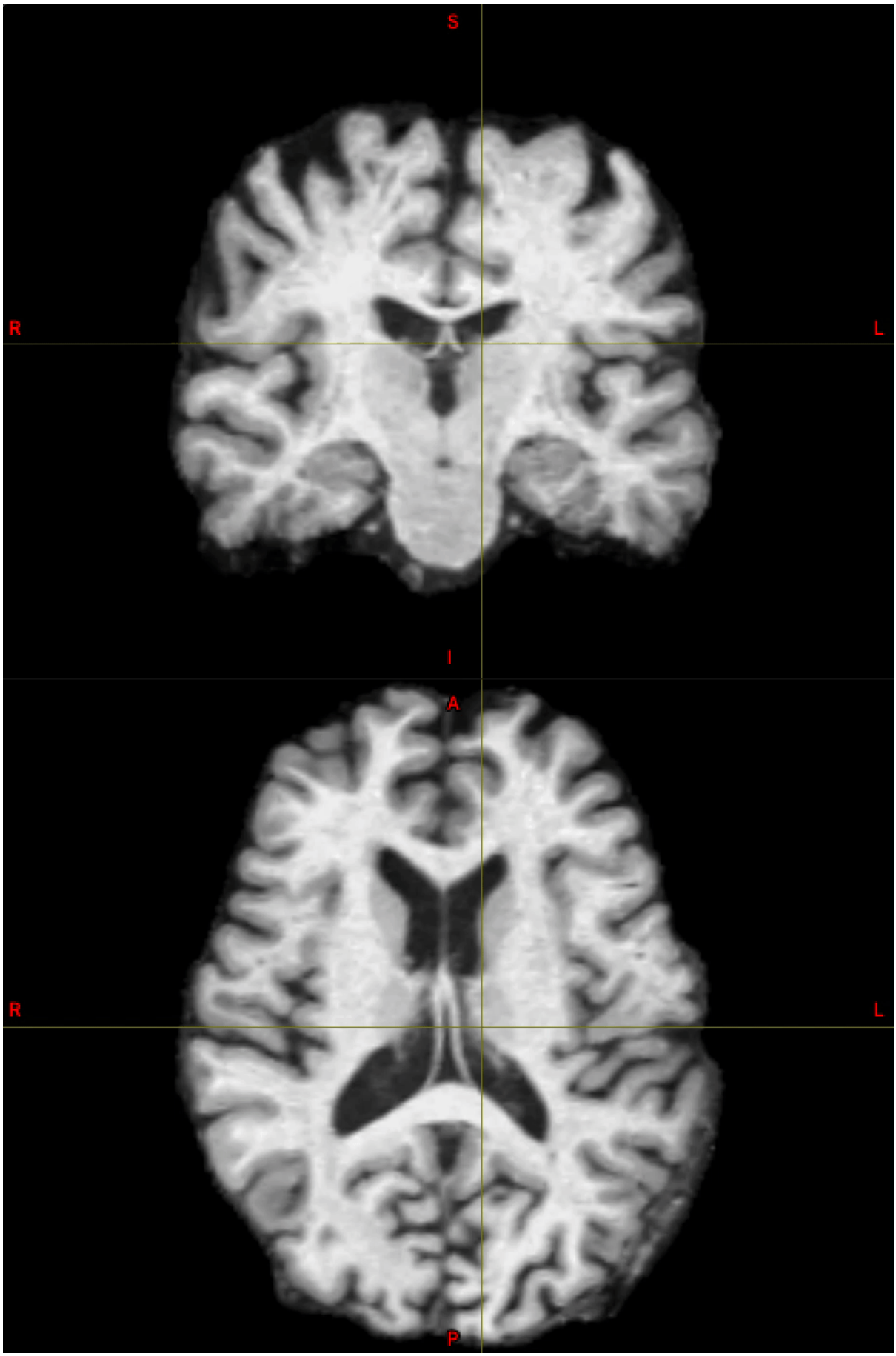
Example MR Image with a resolution of 181 x 217 x 181

Using the previously downloaded example files given in the `registration/images/3d/brains` folder, we only have to run the following command:

```
$ deformetrica estimate model.xml data_set.xml -p
optimization_parameters.xml -v INFO
```

This will produce an `output` folder containing numerous output files.

The following animation illustrates the result of the previously run command:



3D MRI Registration

. . .

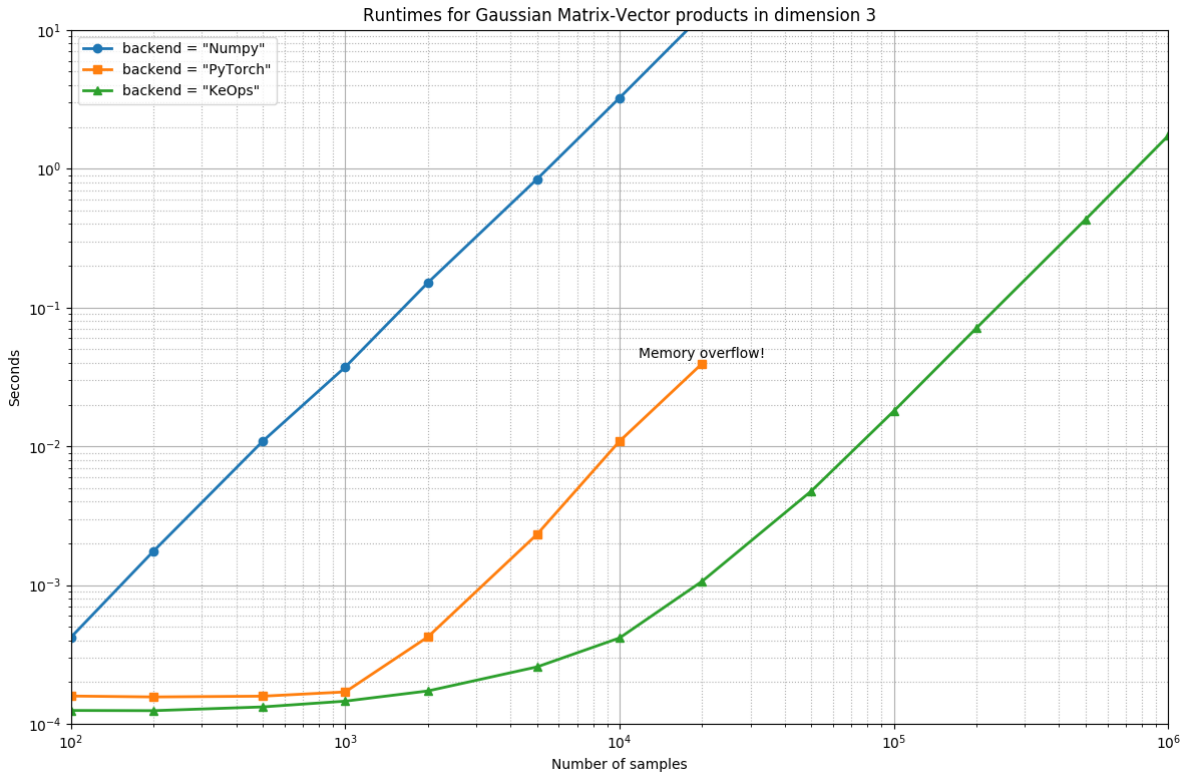
Final thoughts

Extra features

As was shown in the previous sections, *Deformetrica* is a versatile shape analysis tool that can be used as stand-alone software using its CLI or embedded within a Python application using the dedicated API. For the sake of simplicity, only 3 models were shown in this article. Note that many other models are available: Bayesian Atlas, Affine Atlas, Longitudinal Atlas to name a few. Also, it is possible to use the `compute` command to run a standalone “*shoot*” or a “*parallel transport*”.

Kernels

It is important to note that many optimizer and core model options are configurable. One of these configurable options is the computation kernel that can be used to perform the deformations. Two kernels are available: a PyTorch and a PyKeops version. Both can be used interchangeably without impacting the end-result. It is, however, advised to use the Keops kernel when dealing with large-data as it has a better memory footprint.



Numpy vs PyTorch vs Keops

Scaling up Gaussian convolutions on 3D point clouds - KeOps

Let's compare the performances of PyTorch and KeOps on simple Gaussian RBF kernel products, as the number of samples...

www.kernel-operations.io

When defining the kernel to use, a crucial parameter is the ***kernel-size***. This parameter is data-specific and will define how precise the deformation will be. Too big a value will result in coarser deformations as opposed to too low a value that will result in more fine-grained deformations but at a much higher computational cost.

CUDA acceleration

At any moment, if available, the computations can be done on a CUDA device. By default, *Deformetrica* will detect if a CUDA device is available on the system, if so, the kernel operations will be conducted on the GPU. It is also possible to use the `gpu-mode` option to manually define what mode will be used. This option can take the values: `Auto`, `Full`, `None` or `Kernel`.

. . .

Source code availability

The complete source code for the above examples can be found hosted on Google Colab. This enables a simple and fast deployment for testing of the said examples:

- Deterministic Atlas
- Geodesic Regression

. . .

References

- [Deformetrica]: Official website
- [Keops library]: Official website
- [Durrleman et al. 2014]: Morphometry of anatomical shape complexes with dense deformations and sparse parameters
- [Gori et al. 2017]: Bayesian atlas model

- [Fishbaugh et al. 2017]: Geodesic regression model
- [Louis et al. 2018]: Parallel transport model
- [Bône et al. 2018a]: Longitudinal atlas model
- [Bône et al. 2018b]: Summary of the functionalities available in Deformetrica

Python Statistical Learning