EPFL

LCAV

Optional project in Computer Science - report

LinvPy - A Python package for linear inverse problems

Authors:
Guillaume Beaud
Marta Martinez-Camara

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

June 9, 2016

Contents

1 Goal **2**

2 What do we need? Specifications **2**

3 Solution: Python package **2**

3.1 Open source 3

3.2 Architecture 3

3.2.1 Master-slave paradigm 3

3.2.2 Modularity 4

3.3 Documentation and distribution 5

3.4 Easy to contribute 5

Abstract

The aim of this project is to make a clean, extensible and user-friendly Python package to solve some linear inverse problems.

1 Goal

- Make some of the algorithms proposed by Marta easy to get and easy to use.
- Design and implement a framework to build python packages, that can be reused to implement different Python packages.
- Target users: researchers and engineers working in the industry.

2 What do we need? Specifications

To make our code attractive to the users, we need...

- **Reliable and well tested code**. This is the first condition to have a reliable code.
- **Easy to get, easy to install, easy to use**. The users do not want to spend time trying to get the code and make it work in their local machines. This process has to be fast and easy. Also, they do not want a code for which you need a master before you use it. It has to be user-friendly.
- **Excellent documentation**. We need to explain as clearly as possible how to use our code, and what the code does.
- **Open source**. We want to reach as many users as we can. A condition for that is to make our code open source.
- **Easy to contribute**. We think that we perform better together. That is why we want to motivate people to contribute to our package with new functionalities and ideas.

3 Solution: Python package

A Python package is a solution that fits our specifications. Also, it is very powerful from the pedagogical point of view. By building it, we not only learnt how to make a Python package, also it helped us to understand the most frequently used algorithms to solve linear inverse problems, and it helped us a lot to design the architecture. We started with a very simple package, fully working but with very basic functionalities, and we developed it until we reached the requirements that we wanted.

Also, while we were building the package, we realized that the whole framework that we were developing could be useful to build other packages. Thus, we added a tutorial to the project that explains how to use this tool.

3.1 Open source

The package has been built using open software tools:

- Python
- GitHub
- Sphinx
- MathJax

Also, our code is available and transparent to everybody, everybody can get it for free, and contribute to it.

3.2 Architecture

LinvPy's architecture is based on two major principles : high modularity and the master-slave paradigm.

3.2.1 Master-slave paradigm

This is the principle that lower level functions don't know what happens at the upper level; each function must take care of its job but have no conscience of the reason why they are called. This ensures a very strict hierarchy and no possible interference between lower level functions because the master (the upper function) controls everything thus all the intelligence of the program comes from the top hierarchy and keeps low level functions totally blind.

For example, all the loss functions are only working on scalars; although they are meant to operate on vectors and matrices, they are programmed for a simple scalar. To apply the loss function to an array, you need to use the function

```
array_loss(input, loss_function, clipping)
```

which will take care of evaluating if the input is a scalar, a vector or a matrix and will vectorize the loss function to apply it correctly to all the cells of the input. This way the loss function doesn't care about the structure it is operating on, this choice is made by the upper function which is array_loss. The same principle applies between all different-level functions in LinvPy, which we describe in the next subsection.

3.2.2 Modularity

Modularity is key when dealing with complex algorithms. In LinvPy, functions are separated in three levels : low level, mid level and high level. The higher the level, the closer to the user we are. In each level there are groups of functions that share the same signatures (input:output) and are used for the same type of problems, like objects that share the same interface in Java.

$\boxed{\textit{LOW LEVEL}}$

loss functions :

```
rho_huber(), psi_huber(), rho_bisquare(), psi_bisquare(),
rho_cauchy(), psi_cauchy(), rho_optimal(), psi_optimal()
```

They all have the same signature : `loss_function(scalar, clipping=default_value)` and operate on simple scalars.

weighting functions :

```
weights(), array_loss()
```

They have the same signature `weighting(input, loss_function, clipping)` (+ nmeasurements=None for weights()) and return a weighted form of the input.

regularizations / solvers :

```
least_squares(), tikhonov_regularization(), lasso()
```

They have the signature `regularization(A,y,lambda)` (no lambda for least squares) and output a vector x solution of A,y under a the corresponding regularization.

$\boxed{\textit{MEDIUM LEVEL}}$

iterative solver :

```
irls(A, y, loss_function, clipping=None, scale=None, lamb=0, initial_x=None,
regularization=tikhonov_regularization, ...)
```

This is the backbone of the architecture; the link between all low level functions and the higher-level functions. It articulates the different elements together : the loss function and the regularization function. This is the first level where different elements are combined in an intelligent way (master-slave paradigm).

It is very important to note that both the regularization and the loss function are passed by their <u>reference</u> so any user can put his/her own function inside without editing the code, as long as the functions match the signature of their group, described above.

HIGH LEVEL

M/Tau estimators :

```
basictau(), fasttau(), irls()
```

`irls()` is by default the M-estimator; if nothing is specified it works as such, that's why it is both high and medium level because it can be used from the user's side as the M-estimator (high level) but is also slave of the tau estimator and is the only link between the latter and the other functions (regularizations and loss). High level functions use all sub level functions in an intelligent way and know what they are doing : master side of the master-slave paradigm. They also have the same signatures :

```
estimator(A, y, loss_function, clipping, ninitialx, maxiter=100, nbest=1,
initialx=None, b=0.5, regularization=tikhonov_regularization, lamb=0)
returns x_hat
```

Once again all functions are passed by their references so adding new loss or regularization functions can be done from the user's side and don't need the code to be edited. It is also very easy to extend LinvPy for the future developers : simply create new loss/reg functions respecting the signatures and then pass their reference in the top-level functions (M/Tau estimators) and the functions will be passed to all sub-level functions and work perfectly all along the process in a dynamic way.

3.3 Documentation and distribution

To build the documentation, we used two tools:

- Sphinx
- MathJax

The online documentation on ReadTheDocs : http://linvpy.readthedocs.io/en/latest/
GitHub repository : https://github.com/GuillaumeBeaud/linvpy
PyPi repository to install : https://pypi.python.org/pypi/linvpy

3.4 Easy to contribute

Instructions to contribute are found here : http://linvpy.readthedocs.io/en/latest/#contribute