LAGAR

Release Alpha 0.1

Authors:
Angel Daniel Garaboa Paz
Vicente Pérez Muñuzuri,

Institution: University of Santiago de Compostela
Department: Particle Physics
Research Group: Grupo de Física No-Lineal (GFNL)

julio 10, 2019

LAGAR is a python package to perform **LAG**rangian **A**nalysis and **R**esearch of particle trajectories using analytical and/or real flow fields. This package was developed at the Non-Linear Physics Group in the University of Santiago the Compostela (USC), Spain by Dr. Angel Daniel Garaboa Paz and Prof. Vicente Pérez Muñuzuri.

This package address the transport problem in a flow velocity field using a dynamical systems approach,

d **r**/dt = **F**(**r**(t),t)

with initial conditions

r(t_0)= **r**_0.

CHAPTER 1

Authoring

Authors	Dr. Ángel Daniel Garaboa Paz and Prof. Vicente Pérez Muñuzuri
Organization	University of Santiago de Compostela
Department	Particle Physics
Research Group	Grupo de Física No-Lineal (GFNL)
Address	Campus Vida, Univ. Santiago de Compostela, Rúa de José María Suárez Núñez, s/n, 15705 Santiago de Compostela, A Coruña, Spain
Contact	angeldaniel.garaboa@usc.es
date	October-2018
status	Alpha release
revision	0000
version	0.1
copyright	GPLv3

CHAPTER 2

Plattform

Language	Python 3
OS	Linux or Windows
Requirements	Anaconda / pip

Python packages required

- xarray
- numpy
- scipy
- interpolate (conda -c conda-forge interpolate)
- tqdm (fancy progress bars)
- scikit-image (for postprocessing tools)

CHAPTER 3

Documentation

Getting Started

- *LAGAR summary*
- *Quick-overview*

3.1 LAGAR summary

LAGAR is python package to address the transport problem using a dynamical systems approach,

d **r** /dt = **F** (**r** (t),t)

with initial conditions

r (t_0)= **r** _0.

We seek for an approach where the vector **r**(t) represents the state of the particle. In case of particle trajectories and in the easiest case this vector represents the position of the particle in 2D or 3D. The LAGAR package, extend this concept to n-dimensional problems and the state vector of the particle can carry out any number of properties, making the problem extendable to a n-dimensional. The **F** functions represents the dynamical system changing the state vector on time. In the simplest case, this function is the flow field, changing only the position. In the same way, that we address the state vector, the **F** function can be any function writed by the user and extendable to any n-dimensional system.

3.2 why-LAGAR

LAGAR is a "bridge" between the transport problem and being user-friendly with a low python knowledge. It is intended to avoid the hardest part of coding and solving the ordinary diferential equations in case of real flow fields, to provide and user friendly way to create you own **F** functions and your own **r** state vectors and evaluate them using real data. Their key functionalities or advantages are:

- The evaluations of **F** function in case you are using real flow data, are done into encapsulated the called Kernels objects. They containing all the information regarding to other subfunctions or parameters, to evaluate **F** (**r** (t),t). The build of the interpolants to evaluate the **F** in case of real data fields is an automated process hidden for the user.
- The different inial conditions are done in one unique array of **n** x **m** (initial conditions x variables). It evaluates **F** function at once. This allow to evaluate arrays of millions of initial conditions using the interpolants for real data (scipy or interpolate packages) increasing the speed up in many orders of magnitude in compare with map, vectorial functions or other scipy.odesolver which solves each

initial conditions individually.

- The setup of the problem and their inputs, just require a JSON file, allowing the scripting of the code to run it with operational purposes or parametric analysis purporses.
- To add new particles (new **F** functions) to test their effects on the solutions can be done in a friendly way.
- It can write outputs in multiple file formats: netcdf, csv, vtu.
- It includes a postprocessing module to compute different measures from trajectories or solutions such as FTLE, or residence times.

3.3 Quick-overview

LAGAR solves the problem mentioned in the following stages:

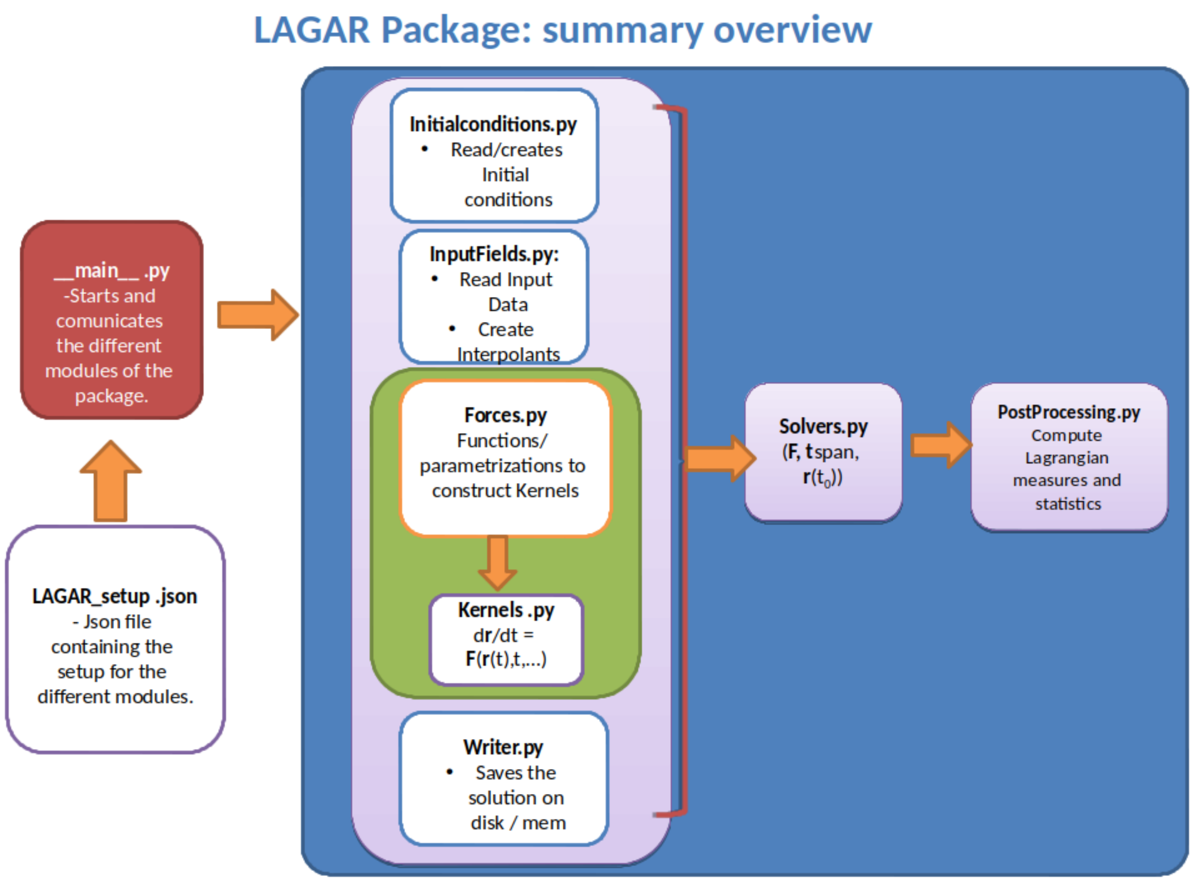
1. Read the main configuration file: `LAGAR_setup.json`.

This file constains information about:

1. Get the initial conditions to be evaluated: From file or fill domain with a regular grid.
2. Get the input files with real data fields: Regular NETcdf data are supported.
3. Get the function **F**: the name of the **F** function inside LAGAR `Kernels.py`.
4. Get the solver type: Euler and RK4.
5. Get the output format setup: vtu, csv, netcdf.
6. Stablish other optional particle parameters
    - Boundaries: Wall, drop, rebound, periodic.
    - Emissors (new initial conditions)
7. Solves the problem.
8. PostProcessing package to analyze the data.
    - Concentrations.
    - Residence Time.
    - FTLE.

3.3.1 Flow-Diagram

The scheme of the package is as follows:


LAGAR Package: summary overview
__main__ .py
-Starts and comunicates the different modules of the package.
LAGAR_setup .json
- Json file containing the setup for the different modules.
Initialconditions.py
• Read/creates Initial conditions
InputFields.py:
• Read Input Data
• Create Interpolants
Forces.py
Functions/ parametrizations to construct Kernels
Kernels .py
dr/dt = F(r(t),t,...)
Writer.py
• Saves the solution on disk / mem
Solvers.py
(F, tspan, r(t$_0$))
PostProcessing.py
Compute Lagrangian measures and statistics

In order to config and initialize the simulations, there a main file:

`LAGAR_setup.json` - `LAGAR Setup`_: Json file with dictionaries containing the setup that will be passed to LAGARCore.

Other important files inside the package are:

1. `InputFields.py` - *LAGAR.InputFields()*: It contains different routines, to read input files provided, read the data fields and create the interpolants to evalute the **F** function inside `Kernels.py`. At the moment, it can read NETCDF data format using xarray package.To create the interpolants, it uses two packages: scipy.RegularGridInterpolator or interpolate (conda -c conda-forge interpolate). The first one, allows to use structured but non-regular spaced meshes the second one is a numba-based interpolant and it just allow to use structured and regular-spaced meshes.
2. `Kernels.py`- *LAGAR.Kernels()*: It contains different **F** functions. This kernels must be fed with all the information provided from the preprocessing stage, together with other parameters to feed the **F** function. In future releases, It will contain new **F** functions to solve the motion of new particles.
3. `Solvers.py`- *LAGAR.Solvers()*: It contains the ODE solvers: Euler and RK4. More solvers, will be added in upcoming releases.
4. `PostProcessing.py` . *LAGAR.PostProcessing()* : It contains functions to compute different Lagrangian measures and statistics from the trajectories obtained.

Additional files.

`IdealFlows.py` - *`LAGAR.IdealFlows()`*: It contains some analytical flows.

3.4 Installation

At this moment LAGAR does not contain an automatic way to be installed. We are currently working on it. Once it is finished it should be installed using:

pip install git+https://github.com/DanielGaraboaPaz/LAGAR.git

At this development stage, it can only be installed manually (by the "hard" way). You can download the master.zip from the github reporsitory:

unzip it and add the LAGAR folder to the PYTHONPATH manually.

If you have access to tar.gz build. You can install it with:

```
$ pip install LAGAR-0.1.tar.gz
```

Please check that dependencies are satisfied. We strongly recomend the use of pip and Anaconda to install them: - numpy - xarray - scipy - pandas - interpolate (from conda-forge interpolate, to use Numba Interpolants) - scikit-image (in your are going to use postprocessing) - json - tqdm

3.5 how to run LAGAR

To run LAGAR, once you installed run the following command:

```
$ python -m LAGAR LAGAR_setup.json 1
```

The LAGAR has two inline input arguments, the first one is the json setup. The second one, is optional and control the number of chunks to divide the simulation in case that you want to perform longs integration with large data fields. (This will be deprecated in future releases)

CHAPTER 4

User Guide

4.1 Setup Init

Core of the LAGAR package. This init script reads and connects the different inputs and outputs of the different modules. It works as a template and it can be modified.

4.1.1 Reading the config file

It reads the JSON file described in `LAGAR Setup`_ passed as a argument from the command line,

```
$ python LAGAR_init.py LAGAR_setup.json
```

4.2 Setup json template

Before running, the first step is to configure the `LAGAR_setup.json`. This file contains different dictionaries to be read by the `LAGAR_init.py` - `LAGAR Init`_ script. Each dictionary inside the JSON file configures a module.

4.2.1 Initial Conditions

The first dictionary controls the creation of initial conditions by `LAGAR.InitialConditions.InitialConditions()`:

```
"r0":{
"names":["lon","lat","depth"],
"ranges":[[-8.95,-8.65],[42.10,42.40],[8,9]],
"step":[0.01,0.01,1]
},
```

The keyword :term:`r0', controls de initial conditions for the differentes variable names :term: 'names'. The :term:`ranges', specify the limits over each variable name in order to create a set of points with the resolution specified in :term:`step'. This, generates a square, cube or hipercube of points of initial conditions.

4.2.2 Inputs

The input dictionary controls the reading of external files with *LAGAR.InputFields()*. To work properly, the file must contain regular meshes in NetCDF format. The reading is made using xarray so it admits the OpenDAP protocol to open files through thredds servers.

For each input that we want to read from external files, it reads the dict used in *inputs* with the characteristics of the data passed, using the *LAGAR.InputFields()* module. We must copy the following block of code to prepare the interpolant in the form F(r(t),t) for each field that we want to make an interpolant, including the mask field.

The different dictionaries inside the dataset correspond to the different fields to be read. Here, we are reading the flow velocity field and the mask field.

The keyword dims_dataset, specify the dimensions of the dataset in the order to be read. That is, the dataset will be transposed to fit in the order chose.

The keyword dims_fields, specify the dimensions of each field to be read, and dims_order, specifies the order of the dimensions read, and change if it is necessary.

why should I use this?

In some datasets, the order of the dimensions is not increasing and turns into an error at the time of creating the interpolants.

Finally, the keyword, fields, specifies the variables to be read from the file. In case that we want to read a vector field such as in the example used below, we should pass the fields in a list.

```
"input" :{
        "velocity":{
            "file_name":"MOHID_Vigo_20180723_0000.nc4",
            "dims_dataset":["lon","lat","depth","time"],
            "dims_fields":["lon","lat","depth","time"],
            "dims_order":[1,1,-1,1],
            "fields":["u","v","w"]
            },
        "mask":{
            "file_name":"MOHID_Vigo_20180723_0000.nc4",
            "dims_dataset":["lon","lat","depth","time"],
            "dims_fields":["lon","lat","depth"],
            "dims_order":[1,1,-1,1],
            "fields":["mask"]
            }
        },
```

4.2.3 Solvers

This part controls the selection of the solver in *LAGAR.Solvers()*. The :term: “time_range”, are the time limits to perform the integration (in the example below, it is in seconds) and :term: “dt”, controls the time step.

```
"solver":{
        "solver":"RK4",
        "time_range" : [1,120600],
        "dt": 3600
        },
```

4.2.4 Kernels

This part control the selection of kernel inside the kernels module *LAGAR.Kernels()*.

```
"kernel":{
        "kernel": "LagrangianSpherical2D"
        },
```

4.2.5 Outputs

The ouput dictionary controls the following aspects:

```
"output" :{
        "store_type": "Mem",
        "path_save": "./VIGO/",
        "pattern":"t_",
        "type":".nc",
        "var_names":["lon_t","lat_t","depth_t"],
        }
```

The :term:"store_type" has two available options, "Mem" or "Disk". In the first, case the steps or solution from solvers, can be stored into a big array in ram memory and then it is turned into a .nc or CSV file. If the computation is heave on memory and we desire to store the data on disk, we can save intermediate steps into multiple binary files. The place where these files are saved is set by the terms :term:"path_save" and :term:"pattern".

The :term:`var_names' controls the names used in the output for the saved variables in the case that we want to store it as CSV or NetCDF. In the case of, we want to store it as nc files, these names must be different from dimensions in the :term:`r0', to avoid name collision.

CHAPTER 5

Code Guide

- *LAGAR package*

5.1 LAGAR package

5.1.1 Submodules

5.1.2 LAGAR.Forces module

This module contains the different forces and processes that are going to be used in the kernels construction. More processes will be added in future releases.

`LAGAR.Forces.`**`AnisotropicDiffusion`** (*r_p, D, Delta_t*)

It computes the velocity due to diffusion.

Args:

- r_p (float): Position of the particle [space_units].
- D (float): Diffusion rate or diffusion speed [space_units/time_units].
- Delta_t (float): Timestep to consider the jump due to difusion [time_units].

Returns:

float: Difusion Velocity [space_units/time_units]

`LAGAR.Forces.`**`Buoyancy`** (*rho_p, rho_f*)

This function computes the force exerted due to density diferences between the particle and the medium.

Args:

- rho_p (float): Density of the particle [kg/m^3].
- rho_f (float): Density of the fluid [kg/m^3].

Returns:

- float: Buoyancy net force on the particle [m/s^2].

`LAGAR.Forces.`**`Coriolis`** (*r_p, v_p*)

This function computes the force that acts on objects that are in motion relative to a rotating reference frame.

Args:

- r_p (float): Position of the particle [º].

Returns:

- float: Coriolis net force on the particle.

`LAGAR.Forces.`**`Diffusion`** (*r_p, v_f, w_f, D, Delta_t*)

Computes the velocity due to diffuisión effect .

Args:

- r_p (float): Position of the particle [space_units].
- v_f (float): Velocity of the flow field at the particle position [m/s].
- w_f (TYPE): Desription.
- D (float): Diffusion rate or diffusion speed [m/s] or [space_units/time_units].
- Delta_t (float): Timestep to consider the jump due to difusion [s] or [time_units].

Returns:

- float: Difusion Velocity [m/s] or [space_units/time_units]

`LAGAR.Forces.`**`MoralesDiffusion`** (*r_p, D, Delta_t, v_f, w_f*)

It computes the velocity due to diffusion.

Args:

- v_f (float): Velocity of the flow field at the particle position [m/s].
- w_f (float): Velocity of the wf field at the particle position [m/s].
- r_p (float): Position of the particle [space_units].
- D (float): Diffusion rate or diffusion speed [space_units/time_units].
- Delta_t (float): Timestep to consider the jump due to difusion [time_units].

Returns:

float: Difusion Velocity [space_units/time_units]

`LAGAR.Forces.`**`Stokes`** (*v_p, v_f, rho_p, rho_f, nu_f, R*)

This function computes the force exerted by viscosity when a perfect sphere (without impurities) moves through a viscous fluid in a laminar flow.

Args:

- v_f (float): Velocity of the flow field at the particle position [m/s].
- v_p (float): Velocity of the particle [m/s].
- rho_p (float): Density of the particle [kg/m^3].
- rho_f (float): Density of the fluid [kg/m^3].
- nu_f (float): Kinematic viscosity of the fluid in [m^2/s].

- R (float): Radius of the particle [m].

Returns:
-float: Stokes force on the particle [m/s^2]

`LAGAR.Forces.`**`TerminalVelocity`** (*v_f, v_p, R, rho_f, rho_p, g, nu*)

Args:
v_f (float): Velocity of the flow field at the particle position [m/s]. v_p (float): Velocity of the particle [m/s]. R (float): Radius of the particle [m]. rho_f (float): Density of the fluid [kg/m^3]. rho_p (float): Density of the particle [kg/m^3]. g (TYPE): Description nu (TYPE): Description

Returns:
TYPE: Description

`LAGAR.Forces.`**`Windage`** (*w_p, A, W*)

Computes the influenced exerted by the wind flow over the floating particle and it is directly linked to the buoyancy of the object.

Args:

- w_p (float): Surface wind field at particle position [m/s].
- A (float): Area of the particle above the surface [m^2].
- W (float): Area of the particle below the surface [m^2].

Returns:

- float: Velocity wind influence on the particle [m/s].

5.1.3 LAGAR.IdealFlows module

This module contains ideal flows, to use them as velocity fields inside the different kernels.

class `LAGAR.IdealFlows.`**`ABC`** (*A, B, C*)

Bases: `object`
It generates a ABC flow object.
The Arnold–Beltrami–Childress (ABC) flow is a three-dimensional incompressible velocity field which is an exact solution of Euler's equation.
Attributes:

- A (float): A parameter.
- B (float): B parameter.
- C (float): C parameter.

`F` (*r, t*)

This method evaluates the velocity at the given postion.

Args:
r (float): Particle position.
t (float): time (not needed).

Returns:
float: Velocity of the flow field at the particle postion r

class `LAGAR.IdealFlows.`**`DoubleGyre`** (*omega*, *eps*, *A*)

Bases: `object`

It generates a two-dimensional flow consisting of two vortices rotating in a conterwise way inside a box with dimensions x=[0,2] and y=[0,1]

Attributes:

A (float): Factor to control the magnitude of the velocity.
eps (float): Factor to control the how far the line separating the gyres moves to the left or right.
omega (float): frequency of oscillation.

`F` (*r*, *t*)

This method evaluates the velocity at the given postion.

Args:

r (float): Particle position.
t (float): Time (not needed).

Returns:

float: Velocity of the flow field at the particle postion r.

class `LAGAR.IdealFlows.`**`Lorenz`** (*A*, *B*, *C*)

Bases: `object`

The Lorenz model is simplified mathematical model for atmospheric convection widely used to observe the chaotic behavior.

Attributes:

A (float): Parameter proportional to Prandtl number.
B (float): Parameter proportional to Raileygh number.
C (float): Physical dimension of the layer.

`F` (*r*, *t*)

This method evaluates the velocity at the given postion.

Args:

r (float): Particle position.
t (float): Time (not needed).

Returns:

float: Velocity of the flow field at the particle postion r.

5.1.4 LAGAR.Kernels module

This module contains different kernels or macro F functions from drdt=F(r(t),t) ODE systems. The idea of these Kernels is to encapsulate inside an object all the parameters and functions required to evaluate the particle properties that r(t) involve in time, in such a way that finally we only get a function F, where the only parameters needed to evaluate are the time-dependent ones. This allows us, to send this function F to be evaluated with the Solver package. New kernels containing different properties will be added in the future.

class `LAGAR.Kernels.`**`Lagrangian`** (*json_dict*)

Bases: `object`

This class provides pure Lagrangian kernels. The particle just follows the local velocity flow field.

Attributes:

-Vi (function): The interpolant function, to evaluate the local flow velocity field V(r(t),t).

F (*r*, *t*)
Model Kernel function to send to the solver module.

Args:
r (array): Array containing the position of the particles [space_units]. t (float): Time instant [time_units]

Returns:
array: Array containing the velocity componentes evaluted. [space_units/time_units]

class `LAGAR.Kernels.`**`LagrangianSpherical2D`** (*json_dict*)
Bases: `object`

This class provides Lagrangian Kernels, for spherical (lat,lon) integrations, considering a perfectly spherical earth with a 6370Km radius. The particle just follows the local flow velocity field.

Attributes:

- m_to_deg (float): Conversion from [m/s] to [degrees/s].
- Vi (function): The interpolant function, to evaluate the local flow velocity field V(r(t),t).

F (*r*, *t*)
Model Kernel function to send to the solver module.

Args:

- r (array): Array containing the position of the particles [space_units].
- t (float): Time instant [time_units].

Returns:

- array: Array containing the velocity componentes evaluted [space_units/time_units].

5.1.5 LAGAR.PostProcessing module

class `LAGAR.PostProcessing.`**`FTLE`** (*alias*, *spherical*, *model_input*, *grid_shape*, *integration_time_index*)
Bases: `object`

Attributes:

- alias (dict): Dictionary to link the variable outputs of your lagrangian model with the standard x,y,z variables to compute the FTLE.
- ds (xarray.dataset): Netcdf input file in xarray.dataset format.
- ds_output (xarray.dataset): Netcdf input file in xarray.dataset format.
- grid_coords_labels (list): Internal coords names used by MYCOASTFTLE module.
- grid_shape (list): Dimensions of the grid used to perform the advection (optional).
- model_input (string): Name of the Lagrangian model used.
- spherical (boolean): True/false if the model is in cartesian or spherical coordinates.
- time (ds.Datarray):Datarray of 'time' aliases positions from netcdf input.
- vars_labels (list): Private variables names used by MYCOASTFTLE module.
- x (xarray.Datarray): Datarray of 'x' aliases positions from netcdf input.
- x0 (xarray.Datarray): Datarray of 'x0' grid coords.

- y (xarray.Datarray): Datarray of ‘y’ aliases positions from netcdf input.
- y0 (xarray.Datarray): Datarray of ‘y0’ grid coords.
- z (xarray.Datarray): Datarray of ‘z’ aliases positions from netcdf input.
- z0 (xarray.Datarray):Datarray of ‘z0’ grid coords.

`explore_ftle_timescale` ( )

It computes the FTLE for all timesteps instead of a given one. The output produced will help you to explore the timescale of the deformation in order to infer the attributes for LCS and FTLE computation.

Args:

- to_dataset (bool, optional): By default, it added the computed FTLE field,

to the output dataset.

Returns:
self(MYCOASTFTLE): ds_output with FTLE computed for all timesteps.

`get_ftle` (*to_dataset=True*)

It computes the FTLE (Finite Time Lyapunov Exponents) using the Cauchy Green finite time deformation tensor described, Shadden (2005).

Args:

- T (float, optional): If the dataset has no time attribute in datetimeformat,

you can provide the advection time. - timeindex (int, optional): Index time to select the to compute the FTLE. By default is the last time step of the Lagrangian advection. - to_dataset (bool, optional): By default, it added the computed FTLE field, to the output dataset

Returns:
self: Added the FTLE_forward or FTLE_backward field to the ds_output.

`get_input` (*netcdf_file*)

`set_alias` ( )

This method set the alias names in case that you use you own dictionary for your own Lagrangian model.
Args:

Example:

- **YourLagrangianModel.nc**
  output_variables: longitude_t, latitude_t, depth_t, t
- **alias_dict:**
  alias_dictionary={‘x’:’longitude_t’,’y’:latitude_t’,’depth’:’z_t’,’time’:’t’}

`transform_input` ( )

This functions turns the input dataset comming from lagrangian model input into a compatible structured dataset to compute FTLE. New models will be available in the future.

Args:

- self(MYCOAST_FTLE) : MYCOAST_FTLE instance

Returns:

- self(MYCOAST_FTLE): Return a MYCOAST_FTLE instance ready for FTLE computation.

class `LAGAR.PostProcessing.`**`GridMeasures`** (*alias*, *model_input*, *bins_option*, *nbins*, *integration_-time_index*)

Bases: `object`

`get_concentrations` (*to_dataset=True*)

It computes the raw residence time. This function counts the timesteps that a particle spetn in a box. It doesn't matter if it is just a particle or a bunch of them.

Args:

timeindex (TYPE, optional): Description bins_option (str, optional): Description nbins (int, optional): Description to_dataset (bool, optional): Description

Returns:

TYPE: Description

`get_input` (*netcdf_file*)

`get_netcdf` (*file_output*)

`get_residence_time` (*to_dataset=True*)

It computes the raw residence time. This function counts the timesteps that a particle spetn in a box. It doesn't matter if it is just a particle or a bunch of them.

Args:

-timeindex (TYPE, optional): Description bins_option (str, optional): Description nbins (int, optional): Description to_dataset (bool, optional): Description

Returns:

TYPE: Description

`set_alias` ( )

This method set the alias names in case that you use you own dictionary for your own Lagrangian model.
Args:

Example:

- **YourLagrangianModel.nc**
  output_variables: longitude_t, latitude_t, depth_t, t
- **alias_dict:**
  alias_dictionary={'x':'longitude_t','y':latitude_t','depth':'z_t','time':'t'}

class `LAGAR.PostProcessing.`**`LCS`** (*lag_ftle*, *eval_thrsh='infer'*, *ftle_thrsh='infer'*, *area_thrsh=100*, *nr_neighb=8*, *ridge_points=False*, *to_dataset=True*)

Bases: `object`

`get_lcs` (*to_dataset=True*)

Extract points that sit on the dominant ridges of FTLE 2D data A ridge point is defined as a point, where the gradient vector is orthogonal to the eigenvector of the smallest (negative) eigenvalue of the Hessian matriv (curvature). The found points are filtered to extract only points on strong FTLE separatrices. Therefore points are only taken, if:

1. FTLE value is high at the point's position
2. the negative curvature is high

Args:

- eval_thrsh (str or float, optional): scalar. Selects zones with small Hessian eigenvalue smaller than eval_thrsh. use 'infer' to obtain the threshold from the 95 percentile of the data of the FTLE field.
- FTLE_thrsh (str or float, optional): scalar. Selects connected areas (with 4 or 8 neighbors) larger than area_thrsh. use 'infer' to obtain the threshold from the 95 percentile of the data of the FTLE field.
- area_thrsh (float, optional): scalar. Selects connected areas (with 4 or 8 neighbors) larger than area_thrsh
- nr_neighb (int, optional): scalar. Connection neighbours (4 or 8)
- ridge_points (bool, optional): x0,y0 exact ridge poisition if 1. (matrix coordinates)
- to_dataset (bool, optional): Logical mask for ridges in the FTLE field LCS_forward and LCS_backward to the outputted dataset.

Example:

Define variables eval_thrsh = -0.005; # Selects zones with small Hessian eigenvalue smaller than eval_thrsh FTLE_thrsh = 0.07; # Selects zones with FTLE larger than FTLE_thrsh area_thrsh = 10; # Selects connected areas (with 4 or 8 neighbors) larger than area_thrsh nr_neighb = 8; # conection neighbours (4 or 8)

Returns:

combined_bin: logical mask for ridges in FTLE field x0: Positions of points on FTLE ridges y0: Positions of points on FTLE ridges

5.1.6 LAGAR.InputFields module

Created on Fri Feb 22 17:12:31 2019

@author: daniel

class `LAGAR.InputFields.`**`InputFields`**

Bases: `object`

This module reads NetCDF files from local disk or online thredds servers through OPENDAP protocol using the xarray module.

Attributes:

- ds (xr.Dataset): xarray dataset format of the NetCDF, with the dimensions ordered and

transported according to JSON file "dims_order" and "dims_dataset" values.

- grid (list): Get the grid structure of the dataset and turns it into numerical

values in order to build up the interpolant functions.

- interpolants (list): List of the interpolant functions of the variables in the "fields" JSON file.

F (*r*, *t*)
Evaluates the function using the previously computed interpolants. It can be a scalar or vectorial function.

Args:

- r (array): np.array with the array of points
- t (float): Time.

Returns:

- array: Value the funtion evaluated at F(r,t).

F_s (*r*)
Evaluates the static function using the previously computed interpolants. It can be a scalar or vectorial function.

Args:

- r (array): np.array with the array of points
- t (float): Time.

Returns:

- array: Value the funtion evaluated at F(r,t).

get_ds ( )
Reads the NetCDFs, with the dimensions ordered and transposed according to JSON file "dims_order" and "dims_dataset" values.

Returns:
xarray Dataset header

get_grid ( )

get_info (*json_dict*)

get_interpolants (*method='linear'*)
Get the interpolant functions using the scipy engine or the interpolation.splines engine. The second option requires the installation of this package with pip. It provides a very faster interpolation algorithm, however, the data points should be regularly spaced. The scipy is more flexible, allowing not equally spaced points in a dimension, but it is slower.

Args:

- numba_interpolant (bool, optional): Select scipy or interpolation engine.
- method (str, optional): Nearest, linear, spline.

get_mfds ( )
Reads a ordered list of netcdf, with the dimensions ordered and transposed according to JSON file "dims_order" and "dims_dataset" values.

Returns:
xarray Dataset header

update_interpolant (*time_slice*)

class `LAGAR.InputFields.`**`MultipleInputFields`** (*input_fields*)

Bases: `object`

`F` (*r*, *t*)

Evaluates the function using the previously computed interpolants. It can be a scalar or vectorial function.

Args:

- r (array): np.array with the array of points
- t (float): Time.

Returns:

- array: Value the funtion evaluated at F(r,t).

`check_domains` (*r*)

5.1.7 LAGAR.Solvers module

This module contains different ODE solvers. The idea is to keep the same input structure than others ODE solvers from scipy packages. More solvers will be added in the future.

class `LAGAR.Solvers.`**`Solvers`** (*initial_condition*, *kernel*, *writer*, *boundaries*, *emissor*)

Bases: `object`

`Euler` (*f*, *tspan*, *r0*, *mem_or_disk='Mem'*, *pattern=None*)

A fixed step Euler solver.

Args:

- f (function): Kernel ODE function to solve.
- tspan (float): Time steps to solve the ODE.
- r0 (float): Initial conditions.

Returns:

- array: The solution at each point evaluated every timestep.

`RK4` ( )

A fixed step Runge-Kutta 4 order solver.

Args:

- f (function): Kernel ODE function to solve.
- tspan (float): Time steps to solve the ODE.
- r0 (float): Initial conditions.

Returns:

- array: The solution at each point evaluated every timestep.

`step` ( )

A fixed step Runge-Kutta 4 order solver.

Args:

- f (function): Kernel ODE function to solve.
- tspan (float): Time steps to solve the ODE.
- r0 (float): Initial conditions.

Returns:

- array: The solution at each point evaluated every timestep.

5.1.8 Module contents

CHAPTER 6

Help and Reference

6.1 Support

The easiest way to get help with the project is to open an issue on Github.

- *Index*
- *Module Index*
- *Search Page*