lagrangian-filtering Documentation

Angus Gibson

Jul 29, 2022

Documentation

lagrangian-filtering is a library for performing temporal filtering of data in a Lagrangian frame of reference. The library is written in Python, leveraging the flexible OceanParcels library for efficient Lagrangian transformation offloaded to a just-in-time compiled C kernel. The source code of this library can be found on GitHub.

CHAPTER 1

Installation

There are two ways to install the library, depending on your Python packaging ecosystem of choice. If you're unsure, the Conda approach is the most plug-and-play, getting you up and running as easily as possible.

1.1 Using Conda

If you use Conda to manage Python packages, you may run

```
$ conda install -c conda-forge -c angus-g lagrangian-filtering
```

to install this package and all its required dependencies. If the `-c angus-g` flag is placed before the `-c conda-forge` flag, a modified OceanParcels dependency is pulled in. By default, this shouldn't be required, but some experimental features may only be present in this version.

To keep things a bit cleaner, you can install *lagrangian-filtering* in its own Conda environment:

```
$ conda create -n filtering -c conda-forge -c angus-g lagrangian-filtering
```

The created environment can be activated by running

```
$ conda activate filtering
```

And deactivated by running

```
$ conda deactivate
```

1.2 Using Pip

On the other hand, you may not use Conda, or you wish to develop for *lagrangian-filtering*. In these cases, it is easier to install a modifiable version of the package in a virtual environment.

```
$ git clone https://github.com/angus-g/lagrangian-filtering
$ cd lagrangian-filtering
$ virtualenv env
$ source env/bin/activate
$ pip install -e .
$ pip install -r requirements.txt
```

This will install *lagrangian-filtering* as a development package, where changes to the files in the git repository will be reflected in your Python environment. To update *lagrangian-filtering*, run

```
$ git pull
```

In the directory into which you cloned the repository. If the *parcels* dependency has changes, running

```
$ pip install --upgrade --upgrade-strategy eager .
```

will pull changes to its corresponding git repository.

1.3 Working with Jupyter Notebooks

If you're working with Conda environments, or a regular virtual environment, it may be the case that you install *lagrangian-filtering*, but `import filtering` fails within a Jupyter notebook. This is because Jupyter doesn't know about your environment, so it's likely looking at your system Python installation instead. We can fix this by adding a new *kernel*. These instructions will be specific to pip, but you can substitute the activation and installation commands for Conda. First, make sure your environment is activated:

```
$ source env/bin/activate
```

Now install *ipykernel*

```
$ pip install ipykernel
```

You can use this package to register a new kernel for your environment:

```
$ python -m ipykernel install --user --name=filtering
```

When you're using Jupyter notebooks, you can either change to the new *filtering* kernel from the *Kernel* menu, or select *filtering* instead of "Python 3" when creating a new notebook.

CHAPTER 2

Algorithm

The *lagrangian-filtering* algorithm has a fairly straightforward goal: to take a set of observations and a velocity field on a temporally-invariant grid, and remove low-frequency components from those observations in a Lagrangian frame of reference. This operation may be useful in many contexts, such as distinguishing stationary wave signals from the sub-inertial flow on which they are imposed. In an Eulerian frame of reference, these stationary waves would be considered part of the mean flow and ignored!

The crux of the algorithm is, unsurprisingly, the transformation to a Lagrangian frame of reference. Given the velocity field, we can perform this transformation by simply advecting point particles along the flow. Interpolating any observations of interest onto the particle positions from the underlying gridded dataset gives us the discretised approximation to the Lagrangian transformation. Consider that the given velocity field may have regions of convergence and divergence. It becomes important to ensure that in particular, regions of divergent flow are spatially well-sampled.

We get around this problem of ensuring the entire domain is well-sampled by a comprehensive set of particle advections: for each time slice t of the source data, one particle is seeded at every grid point. By running the particle advection both forwards and backwards in time, a timeseries of the Lagrangian transformation is obtained for each grid point. Considering this timeseries, we know that by construction, the domain is well sampled at time t. Assuming that the timeseries is sufficiently long to filter sub-inertial frequencies, we can easily compute the Lagrangian-transformed, high-pass filtered observation at time t. Note that we don't make use of the filtered timeseries at any times preceeding or following t. The reason for this is threefold:

1. we don't know whether these fields are spatially well-sampled
2. if they are well-sampled, we would have to run an unstructured interpolation algorithm to reconstruct the data on the source grid
3. due to the finite sampling window, these times are more likely to be impacted by ringing at the ends of the timeseries

Bearing in mind the aforementioned points, the forward-backward particle advection must then be performed at every time slice t, to obtain filtered data only at that slice. Indeed, the times at the beginning and end of the source data may be skipped as well, since there may not be enough data to filter the desired frequencies without ringing.

2.1 Implementation

The above description breaks down the algorithm at a high level. The implementation follows quite naturally:

1. use OceanParcels to perform forward and backward particle advection at each time slice
2. construct a custom sampling kernel to record observations at particle locations during advection
3. high-pass filter the Lagrangian-transformed observations
4. save the resultant filtered fields to disk in a convenient format

The heavy lifting is done by OceanParcels, which combines the expressive nature of Python with the performance of machine code by compiling the advection and sampling kernels to a C library on the fly, giving a vast speedup over native Python. To handle large datasets, the *lagrangian-filtering* library uses a custom fork of OceanParcels that uses a structure-of-arrays representation for particle data. This allows for efficient, vectorised access to the particle data on the Python side, which tends to bottleneck performance. Advection is also parallelised using OpenMP, to take advantage of systems with multiple CPU cores.

CHAPTER 3

Specifying Input Data

When we construct a *LagrangeFilter* object, the `filenames`, `variables` and `dimensions` arguments are passed straight into OceanParcels. There are some examples of how these arguments should be constructed in the OceanParcels tutorial, but we will summarise some of the important takeaways here.

3.1 Filenames dictionary

The `filenames` argument is more properly called the `filenames_or_dataset` argument in the *LagrangeFilter* initialiser. We'll start by describing the more common usecase, providing filenames, rather than a dataset. In all cases where you provide filenames, the files should be in the NetCDF format. In the most simple case, all your data is in a single file:

```
filenames = "data_file.nc"
```

The filenames can contain wildcard characters, for example:

```
filenames = "data_directory/output*/diags.nc"
```

If your variables are in separate files, you can pass a dictionary:

```
filenames = {
  "U": "u_velocity.nc",
  "V": "v_velocity.nc",
  "rho": "diags.nc",
}
```

Finally, you can pass a dictionary of dictionaries, separating the files containing latitude, longitude, depth and variable data. This is particularly useful when your data is on a B- or C-grid, as *detailed below*. The format of the dictionaries follows, noting that the `depth` entry is not required if you're only using two-dimensional data:

```
filenames = {
  "U": {"lat": "mask.nc", "lon": "mask.nc", "depth": "depth.nc", "data": "u_velocity.
↪nc"},
```

(continues on next page)

(continued from previous page)

```
  "V": {"lat": "mask.nc", "lon": "mask.nc", "depth": "depth.nc", "data": "v_velocity.
↪nc"},
}
```

As an alternative to passing filenames, an `xarray` dataset can be given to the `filenames_or_dataset` argument. This is probably more useful when using synthetic data, without requiring that it first be written to a file.

3.2 Variables dictionary

OceanParcels uses particular names for the velocity components and dimensions of data. These names may differ from those actualy used within your files. The first bridge between these two conventions is the `variables` dictionary. This is a map between a variable name used within OceanParcels, and the name within the data files themselves. Note that if you have extra data beyond just the velocity components, it still requires an entry in `variables`.

```
variables = {"U": "UVEL", "V": "VVEL", "P": "PHIHYD", "RHO": "RHOAnoma"}
```

This mapping defines the usual `U` and `V` velocity components, and the additional `P` and `RHO` variables, named `PHIHYD` and `RHOAnoma` in the source data files, respectively.

3.3 Dimensions dictionary

The other bridge between conventions relates to the dimensions of the data. There are two considerations here: first is to simply inform OceanParcels of the latitude, longitude, depth and time dimensions within the data. However, the second consideration is to redefine the data locality of the variables, which is required when using *B- or C-grid interpolation*.

If all data is on the same grid, i.e. Arawaka A-grid, `dimensions` can be a single dictionary mapping the OceanParcels dimension names `lat`, `lon`, `time` and `depth` to those found within the data files. As before, `depth` isn't required for two-dimensional data. However, if your data is three-dimensional and you're choosing a single depth-level with the index mechanism below, `depth` must still be present in the `dimensions` dictionary.

```
dimensions = {"lon": "X", "lat": "Y", "time": "T", "depth": "Zmd000200"}
```

It is also possible to separately specify the dimensions for each of the variables defined in the `variables` dictionary. This is often used when variables have different spatial staggering.

```
dimensions = {
  "U":   {"lon": "xu_ocean", "lat": "yu_ocean", "time": "time"},
  "V":   {"lon": "xu_ocean", "lat": "yu_ocean", "time": "time"},
  "RHO": {"lon": "xt_ocean", "lat": "yt_ocean", "time": "time"},
}
```

3.4 Index dictionary

In some cases, we might want to restrict the extent of the data that OceanParcels sees. This is different from using *`seed_subdomain()`* to use the full domain for advection, but restrict the domain size used for filtering. This functionality is most useful considering that we perform filtering in two-dimensional slices: if we provide a full three-dimensional data file, we may run into some problems. Instead of requiring a pre-processing step to split out separate vertical levels, we can tell OceanParcels to consider only a particular level by its index through the `indices`

dictionary. This is an optional argument to the *LagrangeFilter* initialiser. For example, to use only the surface data (for a file where the indices increase downwards):

```
indices = {"depth": [0]}
```

3.5 B- and C-grid data

Compared to the Arakawa A-grid, where all variables are collocated within a grid cell, the different variables are staggered differently in the B- and C-grid conventions. In particular, on a B-grid, velocity is defined on cell edges, and tracers are taken as a cell mean. This means that velocity is interpolated bilinearly, as you may expect. The behaviour with three-dimensional data is more complicated, but we will not discuss this because the filtering library is aimed at two-dimensional slices.

OceanParcels assumes that C-grid velocity data is constant along faces. The U component is defined on the eastern face of a cell, and the V component on the northern face. To interpolate in this manner, OceanParcels needs the grid information for velocities to refer to the *corner* of a cell. Perhaps confusingly, this means that although U and V are staggered relative to each other, they need to have the same grid information in `dimensions`. OceanParcels assumes the NEMO grid convention, where `U[i, j]` is on the cell edge between corners `[i, j-1]` and `[i, j]`. Similarly, `V[i, j]` is on the edge between corners `[i-1, j]` and `[i, j]`. If your data doesn't follow this convention, new coordinate data will need to be generated in order to work correctly. More detail is available in the indexing documentation.

3.6 Output grid

The underlying *algorithm* involves seeding particles at all gridpoints in order to sample the fields of interest. With the potential staggering mentioned above in mind, this could mean running the filtering advection with three times the number of points. Additionally, we can specify variables on arbitrary grids to be sampled by the velocity data, which could increase the advection time and memory consumption further. Instead, we anticipate a given filtering workflow will seed particles on a single grid, leveraging interpolation for other staggering schemes.

By default, the first grid defined within the OceanParcels `FieldSet` will be used for seeding the filtering particles, and therefore as the final location of the filtered data. Usually, this will be the `U` velocity field, but the *set_particle_grid()* method can be used to modify this after creation of the filtering object. This looks up a field by name from OceanParcels, and as such needs to be called with a variable in the keys of the `variables` dictionary, as opposed to the variable name within your data files. Using the example variable data from before, to set particle seeding and output on the *rho* grid:

```
variables = {"U": "UVEL", "V": "VVEL", "P": "PHIHYD", "RHO": "RHOAnoma"}
f = LagrangeFilter(...)
f.set_particle_grid("RHO")
```

CHAPTER 4

The Filter Object

Aside from the parameters we pass to the *LagrangeFilter* initialiser controlling the particle advection portion of the algorithm (described in *Specifying Input Data*), there are also some parameters to do with filtering itself.

4.1 Times and Windows

To accurately capture enough data to filter the low-frequency components of the signal, there is the concept of the *window*. This is the length of time on either side of a sample over which particles are advected. This window is allowed to optionally be *uneven*, which allows filtering to be performed, even if there isn't a full timeseries on either side of a sample. This happens at the edges of data availability, or at in individual particle level when particles run into topography or out of the domain. The threshold for including particles without a full trajectory is determined by the `minimum_window` parameter.

Control over the temporal resolution within the filtering window is given by the `advection_dt` parameter. This defaults to a 5-minute advection timestep, but this may need to be adjusted depending on the spatial and temporal resolution of the input data. Ideally, this parameter is set to some divisor of the input data timestep, allowing for well-resolved particle paths.

4.2 Filter types

There are a couple of types of filters available for attenuating the low-frequency component of particle trajectories. The default filter is used when `highpass_frequency` is specified, which gives a 3dB cutoff over the entire domain using a 4th-order Butterworth filter (as obtained by `scipy.signal.butter()`).

There are other filters available in the *filtering.filter* module, such as one that performs the filtering in frequency space (may give a sharper cutoff, at the expense of possible ringing), or that allows variation of the cutoff frequency over the domain. Alternatively, the default *Filter* can be constructed with a different order, and as a highpass, lowpass, or bandpass filter, depending on the required application.

If an alternate filter is constructed, it can be attached to the *LagrangeFilter*:

```
ff = filtering.LagrangeFilter(...)
f = filtering.filter.Filter(...)
ff.inertial_filter(f)
```

CHAPTER 5

Exploring Lagrangian Data

While the main aim of the *lagrangian-filtering* library is the filtering of the Lagrangian-transformed data, it can be useful to work with the transformed data in an exploratory capacity. Suppose we have a new dataset on which we'd like to perform the filtering. Two choices need to be made straight away: what is the high-pass cutoff frequency, and what is a sensible advection timestep? Certainly prior experience and an idea of the source model parameters like Coriolis parameter and timestep are useful, however directly interrogating the data may lead to better results.

Under the hood (see also the *algorithm* description), the Lagrangian transformation can be performed in isolation with `advection_step()`. For example, to compute the mean Lagrangian velocity, which could then be used to compute a spectrum for determining an ideal cutoff frequency:

```
f = LagrangeFilter(...)
data = f.advection_step(time)
mean_u = np.mean(data["var_U"][1], axis=1)
```

5.1 Using analysis functions

As an alternative to using ad-hoc explorations as above, there are predefined functions available to give more robust and efficient ways to interrogate your data. For example, a mean kinetic energy spectrum over all particles could be computed at a specified time using `power_spectrum()`:

```
from filtering import analysis
f = LagrangeFilter(..., sample_variables["U", "V"], ...)
spectra = analysis.power_spectrum(f, time)
ke_spectrum = spectra["var_U"] + spectra["var_V"]
```

5.2 Eulerian filtering

It may be useful to compare Lagrangian-filtered data to Eulerian-filtered data, i.e. simply take the time series at each point, and apply the usual highpass filtering. To make the most direct comparison, this can be easily achieved within

the same framework as the Lagrangian filtering. After constructing the filtering object, change the particle kernel to have only the sampling component. In effect, this deletes the advection component of the kernel, leaving a purely Eulerian filtering pipeline.

```
f = LagrangeFilter(...)
f.kernel = f.sample_kernel
...
```

5.3 Obtaining particle trajectories

As a bit of a sanity check, we could verify that the particles are taking sensible paths through our data. Usually, this is discarded, because it takes up extra memory, and is not actually used in the final result. We can ask for the position to be retained, so that we can examine the advection data:

```
f = LagrangeFilter(...)
f.sample_variables += ["lat", "lon"]
data = f.advection_step(time)
lat, lon = data["lat"][1], data["lon"][1]
```

CHAPTER 6

Examples

Here are some example excerpts for some things you may want to do with the filtering library.

6.1 Load ROMS Data

Suppose we have some ROMS data, with a somewhat complex rotated grid, velocity staggered according to the Arakawa C-grid. Moreover, the velocity and grid data are stored in separate files, and the velocity data is spread over multiple files (they may even have only a single timestep per file)! Due to the C-grid staggering, we want to specify the cell corners (*psi* points) as our lat/lon dimensions. This is consistent with the C-grid interpolation method that OceanParcels will use in this case. Further, because lat/lon are in degrees, we specify this as a *spherical* mesh.

```
filenames = {
    "U": {"lon": "grid.nc", "lat": "grid.nc", "data": "ocean_*.nc"},
    "V": {"lon": "grid.nc", "lat": "grid.nc", "data": "ocean_*.nc"},
}
variables = {"U": "Usur", "V": "Vsur"}
dimensions = {"lon": "lon_psi", "lat": "lat_psi", "time": "ocean_time"}

f = LagrangeFilter(
   "roms_experiment", filenames, variables, dimensions, sample_variables,
   mesh="spherical", c_grid=True,
)
```

Now the filtering library will automatically seed particles at cell corners when performing advection/filtering. Underneath, OceanParcels will interpret the grid correctly, and interpolate the velocity components onto cell corners as well.

CHAPTER 7

Common issues

Unfortunately, the complexity of supporting all sorts of model outputs on different systems can sometimes lead to obscure errors. This page may help to identify a common issue.

7.1 Index errors with a curvilinear grid

Curvilinear grids, i.e. those with 2D lat/lon fields, are somewhat more complex when it comes to the particle advection. If a particle isn't in its correct cell, the underlying index search algorithm in Parcels is linear in the number of cells from its present location to its target location. When we initially seed the particles for filtering, Parcels doesn't know where they should be on the grid, so the index search starts in the corner. For a rectilinear grid this is no problem: we calculate the X and Y indices directly. However, for a curvilinear grid, this could mean thousands of steps for each particle, which quickly becomes intractable.

To prevent Parcels from hopelessly marching along with curvilinear index searches, we have limited the number of steps it can take to 10. This should present no issue once a particle is moving according to the prescribed velocity field, but it could be insufficient for the initial condition. To help things along, we give Parcels a hint about the initial indices for a given parcel, using a KD tree, through the pykdtree library. Unfortunately, if this not installed or otherwise unimportable, the initial index population will silently fail, leading to index search errors during the initial particle sampling. Luckily, the fix is simple enough: install the library (potentially forcing a from-source install if the binary distribution is incompatible with your system).

CHAPTER 8

Contributing

All contributions are welcome, whether they be usage cases, bug reports, or code contributed through pull requests. In the latter case, we ask that you follow the code style, testing, and version control guidelines outlined in this document to make it as seamless as possible to integrate your changes into the repository!

8.1 Code Style

Code itself should be formatted with Black on default settings. This gives a consistent and readable style to all Python files in the codebase. You can run Black manually on changed files, or integrate it into git.

Any public methods and classes that will show up on the *API* page should have a docstring describing its function, arguments, and return values. The docstrings are automatically turned into documentation by the Napoleon sphinx extension, which expects formatting to follow the Google Python Style Guide.

8.2 Testing

All pull requests are tested against a suite of unit and integration tests. For fixes or changes to existing functionality, the test suite should continue passing, unless of course a test was based on the code that was changed. In this case, and with new functionality, tests should be modified and added as required.

Tests are logically grouped in the `test/` directory. Groups of unit tests should be named accordingly, whereas most large-scale integration tests will probably end up in `test_filtering.py`.

To run the tests locally, install the required dependencies with

```
$ pip install -e '.[build]'
```

or simply install *pytest* manually. From the root directory of the project, running `pytest` will execute the entire test suite.

8.3 Version Control

git is a powerful tool, but it's very easy to make things messy. Commits should be used for a logical, contained change to the code. They should be formatted as follows:

```
Short (72 chars or less) summary of changes

Explanatory text, wrapped to 72 characters. Make sure this is
separated from the summary by a blank line.

More text, if you need it. If your commit is targeting a particular
issue, you can end the message with a line telling GitHub to
automatically close it, like this:

Closes #42
```

The summary line should be written in the imperative form, without a terminating full stop. You should be able to insert your summary in the following sentence: *This commit will <summary>*. Context motivating the change, or additional details are welcomed in commit messages so that they may stand on their own.

Pull requests should have a clean, linear sequence of commits. If you have opened a pull request, you own the history of that branch. This means that you are free to rewrite history as required. This is useful in particular in two cases:

1. Changes are made to `master` upon which your pull request should be based.

   In this case, you should `git rebase` to "replay" your commits onto the base branch, rather than introducing a merge commit.

2. You need to make a minor change to an existing commit on your branch.

   This happens often, if you're trying to appease the code formatter, or a typo/tweak is pointed out to you. Rather than adding a new commit (usually with a message like "fix typo"), feel free to rewrite history to incorporate the change into the existing commit. This can be done with an interactive rebase, or fixup commit that can be squashed in at a later time. See the relevant section of the Git Book for some inspiration.

Whenever you have rewritten history, you are likely to need a force push. This is okay for pull requests and strictly personal branches, but nowhere else!

CHAPTER 9

API

9.1 filtering

The main lagrangian-filtering module.

This module contains the crucial datastructure for lagrangian-filtering, *LagrangeFilter*. See project documentation for examples on how to construct a filtering workflow using this library.

class filtering.filtering.**LagrangeFilter**(*name*, *filenames_or_dataset*, *variables*, *dimensions*, *sample_variables*, *c_grid=False*, *uneven_window=False*, *window_size=None*, *minimum_window=None*, *highpass_frequency=5e-05*, *advection_dt=datetime.timedelta(seconds=300)*, ***kwargs*)

The main class for a Lagrangian filtering workflow.

The workflow is set up using the input files and the filtering parameters. Filtering can be performed all at once, or on individual time levels.

Data must contain horizontal velocity components *U* and *V* to perform the Lagrangian frame transformation. Any variables that should be filtered must be specified in the *sample_variables* list (this includes velocity).

Note: We use the OceanParcels convention for variable names. This means that `U`, `V`, `lon`, `lat`, `time` and `depth` are canonical names for properties required for particle advection. The mapping from the actual variable name in your data files to these canonical names is contained in the *variables* and *dimensions* dictionaries. When specifying *filenames* or *sample_variables*, the canonical names must be used, however any other variables may use whatever name you would like.

Once the *LagrangeFilter* has been constructed, you may call it as a function to perform the filtering workflow. See *filter()* for documentation.

Example

A straightforward filtering workflow:

```
f = LagrangeFilter(
    name, filenames, variables, dimensions, sample_variables,
)
f()
```

Would result in a new file with the given *name* and an appropriate extension containing the filtered data for each of the *sample_variables*.

Parameters

- **name** (*str*) – The name of the workflow
- **filenames_or_dataset** (*Union[Dict[str, str], xarray.Dataset]*) – Either a mapping from data variable names to the files containing the data, or an xarray Dataset containing the input data.

  Filenames can contain globs if the data is spread across multiple files.
- **variables** (*Dict[str, str]*) – A mapping from canonical variable names to the variable names in your data files.
- **dimensions** (*Dict[str, str]*) – A mapping from canonical dimension names to the dimension names in your data files.
- **sample_variables** (*List[str]*) – A list of variable names that should be sampled into the Lagrangian frame of reference and filtered.
- **mesh** (*Optional[str]*) – The OceanParcels mesh type, either "flat" or "spherical". "flat" meshes are expected to have dimensions in metres, and "spherical" meshes in degrees.
- **c_grid** (*Optional[bool]*) – Whether to interpolate velocity components on an Arakawa C grid (defaults to no).
- **indices** (*Optional[Dict[str, List[int]]]*) – An optional dictionary specifying the indices to which a certain dimension should be restricted.
- **uneven_window** (*Optional[bool]*) – Whether to allow different lengths for the forward and backward advection phases.
- **window_size** (*Optional[float]*) – The nominal length of the both the forward and backward advection windows, in seconds. A longer window may better capture the low-frequency signal to be removed.
- **minimum_window** (*Optional[float]*) – If provided, particles will be filtered if they successfully advected for at least this long in total. This can increase the yield of filtered data by salvaging particles that would otherwise be considered dead.
- **highpass_frequency** (*Optional[float]*) – The 3dB cutoff frequency for filtering, below which spectral components will be attenuated. This should be an angular frequency, in [rad/s].
- **advection_dt** (*Optional[datetime.timedelta]*) – The timestep to use for advection. May need to be adjusted depending on the resolution/frequency of your data.
- ****kwargs** (*Optional*) – Additional arguments are passed to the Parcels FieldSet constructor.

advection_step(*time*, *output_time=False*)

Perform forward-backward advection at a single point in time.

This routine is responsible for creating a new ParticleSet at the given time, and performing the forward and backward advection steps in the Lagrangian transformation.

Parameters

- **time** (*float*) – The point in time at which to calculate filtered data.
- **output_time** (*Optional[bool]*) – Whether to include "time" as a numpy array in the output dictionary, for doing manual analysis.

Note: If `output_time` is True, the output object will not be compatible with the default filtering workflow, *filter_step()*!

If `output_dt` has not been set on the filtering object, it will default to the difference between successive time steps in the first grid defined in the parcels FieldSet. This may be a concern if using data which has been sampled at different frequencies in the input data files.

Returns

A dictionary of the advection data, mapping variable names to a pair. The first element is the index of the sampled timestep in the data, and the second element is a lazy dask array concatenating the forward and backward advection data.

Return type Dict[str, Tuple[int, dask.array.Array]]

create_out(*clobber=False*)

Create a netCDF dataset to hold filtered output.

Here we create a new `netCDF4.Dataset` for filtered output. For each sampled variable in the input files, a corresponding variable in created in the output file, with the dimensions of the output grid.

Parameters **clobber** (*Optional[bool]*) – Whether to overwrite any existing output file with the same name as this experiment. Default behaviour will not clobber an existing output file.

Returns

A tuple containing a single dataset that will hold all filtered output and the name of the time dimension in the output file.

Return type Tuple[`netCDF4.Dataset`, str]

filter(**args*, ***kwargs*)

Run the filtering process on this experiment.

Note: Instead of *f.filter(...)*, you can call *f(...)* directly.

This is main method of the filtering workflow. The timesteps to filter may either be specified manually, or determined from the window size and the timesteps within the input files. In this latter case, only timesteps that have the full window size on either side are selected.

Note: If *absolute* is True, the times must be the same datatype as those the input data. For dates with a calendar, this is likely `numpy.datetime64` or `cftime.datetime`. For abstract times, this may simply be a number.

Parameters

- **times** (*Optional[List[float]]*) – A list of timesteps at which to run the filtering. If this is omitted, all timesteps that are fully covered by the filtering window are selected.
- **clobber** (*Optional[bool]*) – Whether to overwrite any existing output file with the same name as this experiment. Default behaviour will not clobber an existing output file.
- **absolute** (*Optional[bool]*) – If *times* is provided, this argument determines whether to interpret them as relative to the first timestep in the input dataset (False, default), or as absolute, following the actual time dimension in the dataset (True).

filter_step(*advection_data*)

Perform filtering of a single step of advection data.

The Lagrangian-transformed data from *advection_step()* is high-pass filtered in time, leaving only the signal at the origin point (i.e. the filtered forward and backward advection data is discarded).

Note: If an inertial filter object hasn't been attached before this function is called, one will automatically be created.

Parameters **advection_data** (*Dict[str, Tuple[int, dask.array.Array]]*) – A dictionary of particle advection data from a single timestep, returned from *advection_step()*.

Returns

A dictionary mapping sampled variable names to a 1D dask array containing the filtered data at the specified time. This data is not lazy, as it has already been computed out of the temporary advection data.

Return type Dict[str, dask.array.Array]

make_meridionally_periodic(*width=None*)

Mark the domain as meridionally periodic.

This will add a halo to the northern and southern edges of the domain, so that they may cross over during advection without being marked out of bounds. If a particle ends up within the halo after advection, it is reset to the valid portion of the domain.

If the domain has already been marked as meridionally periodic, nothing happens.

Due to the method of resetting particles that end up in the halo, this is incompatible with curvilinear grids.

Parameters **width** (*Optional[int]*) – The width of the halo, defaults to 5 (per parcels). This needs to be less than half the number of points in the grid in the y direction. This may need to be adjusted for small domains, or if particles are still escaping the halo.

Note: This causes the kernel to be recompiled to add another stage which resets particles that end up in the halo to the main domain.

If the kernel has already been recompiled for zonal periodicity, it is again reset to include periodicity in both directions.

---

make_zonally_periodic(*width=None*)

Mark the domain as zonally periodic.

This will add a halo to the eastern and western edges of the domain, so that they may cross over during advection without being marked out of bounds. If a particle ends up within the halo after advection, it is reset to the valid portion of the domain.

If the domain has already been marked as zonally periodic, nothing happens.

Due to the method of resetting particles that end up in the halo, this is incompatible with curvilinear grids.

Parameters **width** (*Optional[int]*) – The width of the halo, defaults to 5 (per parcels). This needs to be less than half the number of points in the grid in the x direction. This may need to be adjusted for small domains, or if particles are still escaping the halo.

Note: This causes the kernel to be recompiled to add another stage which resets particles that end up in the halo to the main domain.

If the kernel has already been recompiled for meridional periodicity, it is again reset to include periodicity in both directions.

---

particleset (*time*)
Create a ParticleSet initialised at the given time.

Parameters **time** (*float*) – The origin time for forward and backward advection on this ParticleSet.

Returns

A new ParticleSet containing a single particle at every gridpoint, initialised at the specified time.

Return type parcels.particlesets.particlesetsoa.ParticleSetSOA

seed_lat
The 2D grid of seed particle latitudes.

Note: This is determined by *set_particle_grid()* and *seed_subdomain()*.

Returns The seed particle latitudes.

Return type numpy.ndarray

seed_lon
The 2D grid of seed particle longitudes.

Note: This is determined by *set_particle_grid()* and *seed_subdomain()*.

Returns The seed particle longitudes.

Return type numpy.ndarray

seed_subdomain (*min_lon=None*, *max_lon=None*, *min_lat=None*, *max_lat=None*, *skip=None*)
Restrict particle seeding to a subdomain.

This uses the full set of available data for advection, but restricts the particle seeding, and therefore data filtering, to specified latitude/longitude.

Points in the output dataset that fall outside this seeding range will not be written, and will thus have a missing value.

Parameters

- **min_lon** (*Optional[float]*) – The lower bound on longitude for which to seed particles. If not specifed, seed from the western edge of the domain.
- **max_lon** (*Optional[float]*) – The upper bound on longitude for which to seed particles. If not specifed, seed from the easter edge of the domain.
- **min_lat** (*Optional[float]*) – The lower bound on latitude for which to seed particles. If not specifed, seed from the southern edge of the domain.
- **max_lat** (*Optional[float]*) – The upper bound on latitude for which to seed particles. If not specifed, seed from the northern edge of the domain.
- **skip** (*Optional[int]*) – The number of gridpoints to skip from the edge of the domain.

set_output_compression(*complevel=None*)

Enable compression on variables in the output NetCDF file.

This enables zlib compression on the output file, which can significantly improve filesize at a small expense to computation time.

Parameters **complevel** (*Optional[int]*) – If specified as a value from 1-9, this overrides the default compression level (4 for the netCDF4 library).

set_particle_grid(*field*)

Set the grid for the sampling particles by choosing a field.

By default, particles are seeded on the gridpoints of the first field in the Parcels FieldSet (usually U velocity). To use another grid, such as a tracer grid, pass the relevant field name to this function. This field name should be in Parcels-space, i.e. the keys in the `variables` dictionary.

Note: Because we get the particle grid from the Parcels gridset, and changing halos alters the underlying grids, this needs to be called before *make_zonally_periodic()* or *make_meridionally_periodic()*.

Parameters **field** (*str*) – The name of the field whose grid to use for particles.

filtering.filtering.**ParticleFactory**(*variables*, *name='SamplingParticle'*, *BaseClass=<class 'parcels.particle.JITParticle'>*)

Create a Particle class that samples the specified variables.

The variables that should be sampled will be prepended by `var_` as class attributes, in case there are any namespace clashes with existing variables on the base class.

Parameters

- **variables** (*List[str]*) – A list of variable names which should be sampled.
- **name** (*str*) – The name of the generated particle class.
- **BaseClass** (Type[parcels.particle._Particle]) – The base particles class upon which to append the required variables.

Returns The new particle class

Return type Type[parcels.particle._Particle]

9.2 filter

Inertial filter objects.

This definition allows for the definition of inertial filters. These may be as simple as constant frequency, or may vary depending on latitude or even arbitrary conditions like vorticity.

class filtering.filter.**Filter**(*frequency*, *fs*, ***kwargs*)

The base class for inertial filters.

This holds the filter state, and provides an interface for applying the filter to advected particle data.

Parameters

- **frequency** (*Union[float, Tuple[float, float]]*) – The low-pass or high-pass cutoff frequency of the filter in [/s], or a pair denoting the band to pass.
- **fs** (*float*) – The sampling frequency of the data over which the filter is applied in [s].
- ****kwargs** (*Optional*) – Additional arguments are passed to the *create_filter()* method.

apply_filter(*data*, *time_index*, *min_window=None*)

Apply the filter to an array of data.

Parameters

- **data** (*dask.array.Array*) – An array of (time x particle) of advected particle data. This can be a dask array of lazily-loaded temporary data.
- **time_index** (*int*) – The index along the time dimension corresponding to the central point, to extract after filtering.
- **min_window** (*Optional[int]*) – A minimum window size for considering particles valid for filtering.

Returns An array of (particle) of the filtered particle data, restricted to the specified time index.

Return type dask.array.Array

static create_filter(*frequency*, *fs*, *order=4*, *filter_type='highpass'*)

Create a filter.

This creates an analogue Butterworth filter with the given frequency and sampling parameters.

Parameters

- **frequency** (*float*) – The high-pass angular cutoff frequency of the filter in [/s].
- **fs** (*float*) – The sampling frequency of the data in [s].
- **order** (*Optional[int]*) – The filter order, default 4.
- **filter_type** (*Optional[str]*) – The type of filter, one of ("highpass", "bandpass", "lowpass"), defaults to "highpass".

static pad_window(*x*, *centre_index*, *min_window*)

Perform minimum window padding of an array.

Note: This performs in-place modification of `x`.

Parameters

- **x** (*numpy.ndarray*) – An array of (time x particle) of particle dat.a
- **centre_index** (*int*) – The index of the seeding time of the particles, to identify the forward and backward advection data.
- **min_window** (*int*) – The minimum window size; particles with at least this many non-NaN datapoints are padded with the last valid value in each direction.

class filtering.filter.**FrequencySpaceFilter**(*frequency*, *fs*)
A filter defined and applied in frequency space.

This may be used, for example, to implement a sharp cutoff filter, without the possible imprecision of representing the cutoff as a time-domain sinc function.

Parameters

- **frequency** (*float*) – The high-pass cutoff frequency of the filter in [/s].
- **fs** (*float*) – The sampling frequency of the daat over which the filter is applied in [s].

apply_filter(*data*, *time_index*, *min_window=None*)
Apply the filter to an array of data.

class filtering.filter.**SpatialFilter**(*frequencies*, *fs*, ***kwargs*)
A filter with a programmable, variable frequency.

Example

A Coriolis parameter-dependent filter:

```
Ω = 7.2921e-5
f = 2*Ω*np.sin(np.deg2rad(ff.seed_lat))
filt = SpatialFilter(f, 1.0 / ff.output_dt)
```

Parameters

- **frequencies** (*numpy.ndarray*) – An array with the same number of elements as seeded particles, containing the cutoff frequency to be used for each particle, in [/s].
- **fs** (*float*) – The sampling frequency of the data over which the filter is applied in [s].
- ****kwargs** (*Optional*) – Additional arguments are passed to the *create_filter()* method.

apply_filter(*data*, *time_index*, *min_window=None*)
Apply the filter to an array of data.

static create_filter(*frequencies*, *fs*, *order=4*, *filter_type='highpass'*)
Create a series of filters.

This creates an analogue Butterworth filter with the given array of frequencies and sampling parameters.

Parameters

- **frequencies** (*numpy.ndarray*) – The high-pass cutoff frequencies of the filters in [/s].
- **fs** (*float*) – The sampling frequency of the data in [s].
- **order** (*Optional[int]*) – The filter order, default 4.

- **filter_type** (*Optional[str]*) – The type of filter, one of ("highpass", "lowpass"), defaults to "highpass". Note that bandpass spatial filters aren't supported.

9.3 file

ParticleFile implementations for receiving particle advection data from OceanParcels.

Parcels defines the *ParticleFile* class, which has a `write` method to write a *ParticleSet* to disk. The frequency at which this is called is determined by the `outputdt` property.

class filtering.file.**LagrangeParticleArray**(*particleset*, *outputdt=inf*, *variables=None*)
A ParticleFile for in-memory caching of advected data.

For smaller spatial extents, or sufficient memory, it is easier to work with in-memory arrays to cache advection data.

Important: This requires a bit more management than *LagrangeParticleFile*: after forward advection, reverse the data with *reverse_data()* then skip the first output of backward advection (the sampling of initial particle positions) with *set_skip()*.

Parameters

- **particleset** (*parcels.particlesets.particlesetsoa.ParticleSetSOA*) – The particle set for which advection data will be cached. We use this to get the names and types of sampled variables.
- **outputdt** (*Optional[float]*) – The frequency at which advection data should be saved. If not specified, or infinite, the data will be saved at the first timestep only.
- **variables** (*Optional[List[parcels.particle.Variable]]*) – An explicit subset of variables to output. If not specified, all variables belonging to the particleset's particletype that are `to_write` are written.

reverse_data()
Reverse all cached advection data.

This can be used before and after a backward advection to make sure the data is correctly ordered.

set_skip(*n*)
Skip a number of subsequent output steps.

This is particularly useful to ignore the first advection output, which is the values of particles before the kernel is called, and often contains junk unless an explicit zero-time sampling kernel is used.

Parameters **n** (*int*) – The number of output steps to skip.

write(*particleset*, *time*, *deleted_only=False*)
Write data from a particle set.

This is intended to be called from a particle execution kernel. The frequency of writes is determined by the `outputdt` attribute on the class.

Particles which have been deleted (due to becoming out of bounds, for example) are masked with NaN.

Parameters

- **particleset** (*parcels.particlesets.particlesetsoa.ParticleSetSOA*) – Particle set with data to write to disk.

- **time** (*float*) – Timestamp into particle execution at which this write was called.
- **deleted_only** (*Optional[bool]*) – Whether to only write deleted particles (does not do anything, only present for compatibility with the call signature on the parcels version of the class).

class filtering.file.**LagrangeParticleFile**(*particleset*, *outputdt=inf*, *variables=None*, *output_dir='.'*)

A ParticleFile for on-disk caching in a temporary HDF5 file.

A temporary HDF5 file is used to store advection data. Data is stored in 2D, with all particles contiguous as they appear in their particle set. The *time* dimension is extendable, and is appended for each write operation. This means that the number of writes does not need to be known ahead-of-time.

Important: Before calling the particle kernel, a group must be created in the file by calling *set_group()*.

Example

Create an instance of the class, and run forward advection on a *ParticleSet*:

```
f = LagrangeParticleFile(ps, output_dt)
f.set_group("forward")
ps.execute(kernel, dt=advection_dt, output_file=f)
```

Note: Advection data is stored to a *NamedTemporaryFile* that is scoped with the same lifetime as this *ParticleFile*. This should ensure that upon successful completion, the temporary files are cleaned up, yet they will remain if an error occurs that causes an exception.

Parameters

- **particleset** (*parcels.particlesets.particlesetsoa.ParticleSetSOA*) – The particle set for which this file should cache advection data on disk. It's assumed the number of particles contained within the set does not change after initialisation.
- **outputdt** (*Optional[float]*) – The frequency at which advection data should be saved. If not specified, or infinite, the data will be saved at the first timestep only.
- **variables** (*Optional[List[parcels.particle.Variable]]*) – An explicit list subset of particletype variables to output. If not specified, all variables belonging to the particletype that are `to_write` are written.
- **output_dir** (*Optional[str]*) – The directory in which to place the temporary output file.

data (*group*)

Return a group from the HDF5 object.

Each variable saved from particle advection is available as a *Dataset* within the group, as well the `time` attribute.

Parameters **group** (*str*) – The name of the group to retrieve.

Returns

The group from the underlying HDF5 file. If the group hasn't been initialised with *set_group()*, it will be empty.

Return type h5py.Group

set_group(*group*)
Set the group for subsequent write operations.

This will create the group, and datasets for all variables that will be written by this object, if they do not already exist. Otherwise, the group will simply be selected without overwriting existing data.

Parameters **group** (*str*) – The name of the group.

write(*particleset*, *time*, *deleted_only=False*)
Write data from a particle set.

This is intended to be called from a particle execution kernel. The frequency of writes is determined by the `outputdt` attribute on the class.

Particles which have been deleted (due to becoming out of bounds, for example) are masked with NaN.

Parameters

- **particleset** (*parcels.particlesets.particlesetsoa.ParticleSetSOA*) – Particle set with data to write to disk.
- **time** (*float*) – Timestamp into particle execution at which this write was called.
- **deleted_only** (*Optional[bool]*) – Whether to only write deleted particles (does not do anything, only present for compatibility with the call signature on the parcels version of the class).

9.4 analysis

Analysis routines for Lagrangian particle data.

These routines take an already-configured 'LagrangeFilter' and produce diagnostic output.

filtering.analysis.**power_spectrum**(*filter*, *time*)
Compute the mean power spectrum over all particles at a given time.

This routine gives the power spectrum (power spectral density) for each of the sampled variables within *filter*, as a mean over all particles. It will run a single advection step at the specified time. The resulting dictionary contains a *freq* item, with the FFT frequency bins for the output spectra.

Parameters

- **filter** (filtering.LagrangeFilter) – The pre-configured filter object to use for running the analysis.
- **time** (*float*) – The time at which to perform the analysis.

Returns

A dictionary of power spectra for each of the sampled variables on the filter.

Return type Dict[str, numpy.ndarray]

CHAPTER 10

Citing

The description and analysis of the Lagrangian filtering method has been published in JAMES, and can be cited as:

> Shakespeare, C. J., Gibson, A. H., Hogg, A. M., Bachman, S. D., Keating, S. R., & Velzeboer, N. (2021). A new open source implementation of Lagrangian filtering: A method to identify internal waves in high-resolution simulations. *Journal of Advances in Modeling Earth Systems*, 13, e2021MS002616. https://doi.org/10.1029/2021MS002616

CHAPTER 11

Indices and tables

- genindex
- modindex
- search

Python Module Index

f

Index