**VirtualFleet** is a python library dedicated to compute and analyse trajectory simulations of virtual Argo floats.
CHAPTER ONE

DOCUMENTATION

Getting Started
- Installation
- Introduction
- Usage
- Applications
- What is Argo?

Help & reference
- What’s New
- Software development roadmap
- API reference

This software is developed by:

[Logos of LOPS, Ifremer, and ARGO France]
## 1.1 Installation

### 1.1.1 Instructions

Install the last release with pip:

```bash
code
pip install VirtualFleet
code```

you can also try to work with the latest dev. version:

```bash
code
pip install git+http://github.com/euroargodev/VirtualFleet.git@master
code```

### 1.1.2 Required dependencies

- numpy >= 1.18
- pandas >= 1.1
- xarray >= 2022.12
- parcels >= 2.4.0
- zarr >= 2.13.3
- tqdm >= 4.64.1

## 1.2 Introduction

The optimisation of the Argo array is quite complex to determine in specific regions, where the local ocean dynamic shifts away from standard large scale open ocean. These regions are typically the Boundary Currents where turbulence is more significant than anywhere else, and Polar regions where floats can temporarily evolve under sea-ice. Furthermore, with the development of Argo float recovery initiatives, it becomes more crucial than ever to be able to predict float trajectories with mission parameters that can be modified on-demand remotely.

**VirtualFleet** aims to help the Argo program to optimise floats deployment and programming.

To do so, the **VirtualFleet** software provides a user-friendly, high-level, collection of Python modules and functions to make it easy to simulate *Argo floats*.

The general principle is to customise the behavior of particles in a *Lagrangian simulation framework* ([Ocean-Parcels2019](#)) using output from ocean circulation models.

### 1.2.1 Simulated cycle

Although a virtual Argo float is not as complex as a real one, it simulates the typical Argo float cycle with the most important phases:

- descent to a predefined parking depth (phase 0),
- free drifting phase at this predefined parking depth (phase 1),
- descent to a predefined profiling depth followed by an ascent to surface (phase 2-3),
- a free drift at the surface to mimic time for geo-positioning and satellite data transmission (phase 4).

This cycle and the simplified associated mission parameters are illustrated [figure 1.1](#).
1.2.2 Components

In the VirtualFleet software:

- a virtual float is represented by an `ArgoParticle` instance,
- a virtual float cycle is encoded by an `ArgoFloatKernel()` function,

Furthermore:

- a `VirtualFleet` instance represents a fleet of virtual floats, including a deployment plan and the ocean velocity field to transport floats,
- one use a `VirtualFleet` instance to execute a simulation.

The 3 technical requirements to be able to simulate a virtual fleet are the following:

1. a deployment plan,
2. a velocity field,
3. and a virtual float mission configuration.

You will find all details about these requirements, together with the VirtualFleet software helpers to fulfill them in the “Preparation of a simulation” section.

Executing a simulation is explained in the “Running a virtual fleet simulation” section and “Simulation analysis” get you started with the analysis of the results.
1.3 Usage

1.3.1 Preparation of a simulation

In order to create a simulation of virtual Argo floats, you need to provide the following:

- a float deployment plan, as a dictionary with lat/lon/time arrays,
- a velocity field, as a parcels.fieldset.FieldSet instance,
- and a virtual float mission configuration, as a dictionary.

These requirements are explained below, together with VirtualFleet helpers to do it.

But first, let's import the usual suspects:

```python
import numpy as np
from datetime import timedelta
from virtualargofleet import Velocity, FloatConfiguration
```

### Deployment plan

You need to define a deployment plan for your virtual fleet. The VirtualFleet simulator expects a dictionary with arrays for the latitude, longitude and time of virtual floats to deploy. Depth is set by default to the surface, but this can be provided if necessary.

Example:

```python
# Number of floats we want to simulate:
nfloats = 10

# Define space/time locations of deployments:
lat = np.linspace(30, 38, nfloats)
lon = np.full_like(lat, -60)
tim = np.array(['2019-01-01' for i in range(nfloats)], dtype='datetime64')

# Define the deployment plan as a dictionary:
my_plan = {'lat': lat, 'lon': lon, 'time': tim}
```

### Velocity field

Then, you need to define the velocity field to be used by the virtual fleet.

**Note:** The VirtualFleet simulator can take any Parcels parcels.fieldset.FieldSet as input.

However, to make things easier, we provide a convenient utility function `Velocity()` to be used for some standard pre-defined velocity fields. It allows to easily create a `VelocityField` instance that will be used as input to the VirtualFleet simulator.

The 2 main ways to get a `VelocityField` instance with the `Velocity()` function are:

1/ Using a `xarray.Dataset`:
root = "~/data/GLOBAL-ANALYSIS-FORECAST-PHY-001-024"
ds = xr.open_mfdataset(glob.glob("%s/20201210*.nc" % root))
VELfield = Velocity(model='GLOBAL_ANALYSIS_FORECAST_PHY_001_024', src=ds)

2/ Using a custom definition of the required arguments:

root = "~/data/GLOBAL-ANALYSIS-FORECAST-PHY-001-024"
filenames = {'U': root + '/20201210*.nc',
             'V': root + '/20201210*.nc'}
variables = {'U': 'uo', 'V': 'vo'}
dimensions = {'time': 'time', 'depth': 'depth', 'lat': 'latitude', 'lon': 'longitude'}
VELfield = Velocity(model='custom',
                    src=filenames,
                    variables=variables,
                    dimensions=dimensions)

In this later case, the function Velocity() will take care of creating a parcels.fieldset.FieldSet with the appropriate land/sea mask and circular wrapper if the field is global.

Currently, VirtualFleet supports the following values for the model options of Velocity():

- GLORYS12V1, PSY4QV3R1, GLOBAL_ANALYSIS_FORECAST_PHY_001_024
- MEDSEA_ANALYSISFORECAST_PHY_006_013
- ARMOR3D, MULTIOLS_GLO_PHY_TSUV_3D_MYNRT_015_012
- custom if you want to set your own model definition

Argo floats mission parameters

To define the float mission configuration parameters, VirtualFleet takes a simple dictionary with parameters as input. The virtual float cycle and the simplified associated mission parameters are illustrated figure 1.1. The minimal set of parameters to provide is: parking_depth, profile_depth, vertical_speed, cycle_duration and life_expectancy.

VirtualFleet provides the convenient utility class FloatConfiguration to make things easier. It allows to simply load a default configuration and can be passed directly to a VirtualFleet instance.

You can start with a default configuration like this:

cfg = FloatConfiguration('default')

Or you can fetch online a specific float cycle mission (data are retrieved from the Euro-Argo meta-data API):

cfg = FloatConfiguration([6902920, 98])

1.3. Usage
Float configurations can be saved in json files:

cfg.to_json("myconfig.json")

This can be useful for later re-use because you can load a configuration from such a file:

cfg = FloatConfiguration("myconfig.json")

Examples of such json files can be found in this folder.

Once you created a FloatConfiguration instance, you can modify one or more of the parameter values with the update method like this:

cfg.update('parking_depth', 500)

If you want the same mission configuration for all your virtual floats, you can pass this configuration when instantiating a VirtualFleet:

VFleet = VirtualFleet(plan=my_plan, fieldset=VELfield, mission=cfg)

But you can also customized the mission of each float by passing an array of mission configurations to the VirtualFleet instance:

mission = [
    FloatConfiguration('default').update('parking_depth', 100),
    FloatConfiguration('default').update('parking_depth', 200),
    FloatConfiguration('default').update('parking_depth', 500),
    FloatConfiguration('default').update('parking_depth', 1000),
    FloatConfiguration('default').update('parking_depth', 1500),
]

VFleet = VirtualFleet(plan=my_plan, fieldset=VELfield, mission=mission)

### 1.3.2 Running a virtual fleet simulation

If you have all the requirements fulfilled:

- a deployment plan, from a dictionary with lat/lon/time arrays,
- a velocity fieldset, from a parcels.fieldset.FieldSet instance, or possibly from a VelocityField instance,
- and a float mission configuration, from a dictionary, or possibly from a FloatConfiguration instance.

you can move on to run a simulation.

So let’s import the usual suspects:

from datetime import timedelta
from virtualargofleet import VirtualFleet
and create a virtual fleet with all requirements:

```python
V Fleet = VirtualFleet(plan=my_plan, fieldset=VELfield.fieldset, mission=cfg.mission)
```

```
<VirtualFleet>
- 10 floats in the deployment plan
- No simulation performed

Warning: This code assumes you named the deployment plan dictionary my_plan, the velocity field instance VELfield and the float mission configuration instance cfg following the standard “Preparation of a simulation”.

Note: You can also provide the fieldset and mission arguments directly with VirtualFleet objects:

```python
V Fleet = VirtualFleet(plan=my_plan, fieldset=VELfield, mission=cfg)
```

To execute the simulation, we use the `VirtualFleet.simulate()` method by providing at least the total simulation duration time as a timedelta (or number of days):

```python
V Fleet.simulate(duration=timedelta(days=2))
```

```
<VirtualFleet>
- 10 floats in the deployment plan
- Number of simulation(s): 1
- Last simulation meta-data:
  - Duration: 02d 00h 00m 00s
  - Data recording every: 01h 00m
  - Trajectory file: ./v24co0jc.zarr
  - Execution time: 00d 00h 00m 04s
  - Executed on: laptop_guillaume_boulot.lan

By default, virtual floats positions are saved hourly along their trajectories. This is enough to properly resolve profile positions but can be increased using the record argument. See the method documentation here `VirtualFleet.simulate()`.

The simulated floats trajectories will be saved in the current directory as a zarr file. You can control where to save trajectories with the output_folder and output_file options, or set the output option to False to not save results at all.

Note that you can continue the simulation where it was left, using the restart option:

```python
V Fleet.simulate(duration=timedelta(days=3), restart=True)
```

```
<VirtualFleet>
- 10 floats in the deployment plan
- Number of simulation(s): 2
- Last simulation meta-data:
  - Duration: 03d 00h 00m 00s
  - Data recording every: 01h 00m
  - Trajectory file: ./ns6hj1__.zarr
  - Execution time: 00d 00h 00m 06s
  - Executed on: laptop_guillaume_boulot.lan
```

1.3. Usage
In this scenario, a new output file is created and trajectories start from where the previous simulation left virtual floats.

### 1.3.3 Simulation analysis

**Note:** This code assumes you named the `VirtualFleet` instance `VFleet` following the standard “Running a virtual fleet simulation”.

In order to look at the virtual floats trajectories you can read data directly from the output file:

```python
ds = xr.open_zarr(VFleet.output)
```

You can quickly plot the last position of the floats:

```python
VFleet.plot_positions()
```

You can extract a profile index from the trajectory file, after the VFleet simulation:

```python
VFleet.to_index()
```

or create an Argo-like profile index:

```python
VFleet.to_index("simulation_profile_index.txt")
```

or from any trajectory file using the utility function `utilities.simu2index()`:

```python
from virtualargofleet.utilities import simu2index, simu2csv
df = simu2index(xr.open_zarr("trajectory_output.zarr"))
# or to create the index file:
simu2csv("trajectory_output.zarr", index_file="output_ar_index_prof.txt")
```

### 1.4 Applications

#### 1.4.1 Miscellaneous figures

- Gulf Stream, Example 1
- Gulf Stream, Example 2
- Mediterranean Sea
- Real vs virtual floats comparison
- 3D float cycle representation
Gulf Stream, Example 1

10 floats advected (initial positions in yellow dots) for 1 year, dt = 5 minutes.

**Dataset**: Gulf Stream subset of the Operational Mercator daily ocean analysis and forecast system at 1/12 degree.

**Run**: 2 cores in use - 36 Gb of memory in use - Runtime = 00:05:30

---

Gulf Stream, Example 2

100 floats advected for 1 year, dt = 5 minutes

**Dataset**: Gulf Stream subset of the Operational Mercator daily ocean analysis and forecast system at 1/12 degree.

**Run**: 12 cores in use - 38 Gb of memory in use - Runtime = 00:05:42
Mediterranean Sea

10 floats advected for 1 year, \( dt = 5 \) minutes

**Dataset** : Daily Mediterranean MFS - EAS4 of CMCC, at 1/24 degree.

**Run** : - 3 cores in use - 186 Gb of memory in use - Runtime = 00:41:29
Real vs virtual floats comparison

Simulation of floats deployed in 2019 near Bermuda (plain lines are the virtual floats, dashed lines the real ones). Using the Virtual Fleet software, we do not expect every single virtual float trajectories to be similar to the real ones, but we rather expect a long term regional sampling consistent with reality. Sampling metrics are being developed to assess that automatically with the software.
VirtualFleet was created by the Euro-Argo RISE project in order to provide information for an improved design of the Argo array in the boundary regions of interest to the partners represented in figure 1.2 below:

All results were described in the supplement for D2.3. For the most recent version of the document, please check the project list of deliverables.

This supplement document includes recommendations for the boundary current of interest based on trajectories of virtual argo floats performed with the VirtualFleet software.
**Fig. 1.2: Boundary current of interest in EA-RISE WP2.3**

**Gulf Stream Extension**

All results are published on this public repository: [https://github.com/euroargodev/VirtualFleet_GulfStream](https://github.com/euroargodev/VirtualFleet_GulfStream)

VirtualFleet was used to:

- reproduce the observed North Atlantic Argo sampling over the 2008-2018 period (this was done to validate the velocity field),
- simulate the 2008-2018 period sampling in the case where floats entering the Gulf Stream Extension region saw their mission parameters temporarily modified until they would exit the region.

We then compared the 11 years simulation difference in profile density computed on a 1x1 degree grid.

Results are shown figure 1.3 for experiments where the cycling frequency was increased to 5 days and drifting depths changed to: 500, 1000, and 1500 db:

---

**5 days cyc. / 500db drift**

Experiment (#1112758) vs Control (#6113257), 3006 profiles

Too shallow: floats drifting too fast out of the box

**5 days cyc. / 1000db drift**

Experiment (#1014795) vs Control (#6113257), 3646 profiles

Too deep: floats taken in the under current, drifting upstream the GSE

**5 days cyc. / 1500db drift**

Experiment (#1009923) vs Control (#6113257), 3528 profiles

Standard drifting depth: Homogeneous profiles density increase

---

Fig. 1.3: Synthesis of Gulf Stream VirtualFleet experiments

We see that if floats drift at 500db, they are taken by the GS out of the box, downstream/eastward, too fast. The result is that the upstream region is now less sampled, to the benefit of the eastern part of the box and outside of it. This is...
not the expected outcome. On the other hand, if floats drift at 1500db, they are taken by the southward flowing under current. The result is a better sampling of the GS along the U.S. east coast but a rather in-homogeneous increase over the GSE box. Keeping the drifting depth to 1000db seems the best solution in the case where the cycling frequency is increased to 5 days. This set-up leads to an homogeneous increase of the profiles density in the GSE box and a smaller impact on the downstream/eastward sampling decrease.

**Nordic Seas**

All results are published on this *private* repository: https://github.com/euroargodev/VirtualFleet_NordicSeas

**VirtualFleet** was used to:

- determine trajectories of Argo floats deployed along the IO-PAN Arex (Poland) Cruises and IMR cruises (Norway) for many possible mission parameters, with the goal of optimizing a trajectory score based on trajectory length, residence time and covered latitude range in the area of interest. This was determined using the an optimal approach developed at IPMA.

- simulate a large number of virtual floats using velocity fields from different years (2010-2019) and different combinations of cycle length (3, 5, 7 and 10 days) and parking positions (250, 350, 500, 750, 1000 and 1250 db).

**Gulf of Cadiz**

All results are published on this *private* repository: https://github.com/euroargodev/VirtualFleet_Optimization

**VirtualFleet** was used to:

- to simulate and find the best Argo float configuration to maximise the resident time of each float deployed in the Gulf of Cadiz (GoC) region.

**Western Mediterranean and Ligurian Sea**

All results are published on this *private* repository: https://github.com/euroargodev/VirtualFleet_WMed

**VirtualFleet** was used to:

- determine trajectories of Argo floats deployed in small regions inside or outside the BCs for many possible mission parameters, with the goal of determining the best deployment location and parameters to sample the area of interest (using profile density maps).

**References**

1.4.3 **VirtualFleet Recovery**

**VirtualFleet - Recovery** is a CLI and webAPI to make predictions of Argo float positions. It makes Argo floats trajectory predictions easy, in order to facilitate recovery. The library automatically download ocean velocity forecasts and produces a prediction patch using current float mission parameters or modified ones.

All details can be found at: https://github.com/euroargodev/VirtualFleet_recovery
1.5 What is Argo?

Argo is a real-time global ocean in situ observing system.

The ocean is a key component of the Earth climate system. It thus needs a continuous real-time monitoring to help scientists better understand its dynamic and predict its evolution. All around the world, oceanographers have managed to join their efforts and set up a Global Ocean Observing System among which Argo is a key component.

Argo is a global network of nearly 4000 autonomous probes measuring pressure, temperature and salinity from the surface to 2000m depth every 10 days. The localisation of these probes is nearly random between the 60th parallels (see live coverage here). All probes data are collected by satellite in real-time, processed by several data centers and finally merged in a single dataset (collecting more than 2 millions of vertical profiles data) made freely available to anyone through a ftp server or monthly zip snapshots.

The Argo international observation array was initiated in 1999 and soon revolutionized our perspective on the large scale structure and variability of the ocean by providing seasonally and regionally unbiased in situ temperature/salinity measurements of the ocean interior, key information that satellites can’t provide (Riser et al, 2016).

The Argo array reached its full global coverage (of 1 profile per month and per 3x3 degree horizontal area) in 2007, and continuously pursues its evolution to fulfill new scientific requirements (Roemmich et al, 2019). It now extents to higher latitudes and some of the floats are able to profile down to 4000m and 6000m. New floats are also equipped with biogeochemical sensors, measuring oxygen and chlorophyll for instance. Argo is thus providing a deluge of in situ data: more than 400 profiles per day.

Each Argo probe is an autonomous, free drifting, profiling float, i.e. a probe that can’t control its trajectory but is able to control its buoyancy and thus to move up and down the water column as it wishes. Argo floats continuously operate the same program, or cycle, illustrated in the figure below. After 9 to 10 days of free drift at a parking depth of about 1000m, a typical Argo float dives down to 2000m and then shoals back to the surface while measuring pressure, temperature and salinity. Once it reaches the surface, the float sends by satellite its measurements to a data center where they are processed in real time and made freely available on the web in less than 24h00.

Typical 10 days program, cycle, of an Argo float:
1.6 What’s New

1.6.1 v0.3.1 (22 Nov. 2023)

Note: This is the last version compatible with Parcels versions < 3.0.0

New features

- Mission parameters can now be set for each floats of the deployment plan. This is useful to limit the number of simulations to explore a set of configuration parameters. (#22) by K. Balem.

For instance:

```python
# Number of floats
nfloats = 5

# Define space/time locations of deployments:
lat = np.linspace(40, 41, nfloats)
lon = np.full_like(lat, 5)
tim = np.array(["2020-01-16" for i in range(nfloats)], dtype='datetime64')
my_plan = {"lat": lat, "lon": lon, "time": tim}

mission = [
    FloatConfiguration('default').update('parking_depth', 100),
]  # (continues on next page)
floatConfiguration('default').update('parking_depth', 200),
floatConfiguration('default').update('parking_depth', 500),
floatConfiguration('default').update('parking_depth', 1000),
floatConfiguration('default').update('parking_depth', 1500)
]

VFleet = VirtualFleet(plan=my_plan, fieldset=VELfield, mission=mission)

1.6.2 v0.3.0 (25 Jan. 2023)

By G. Maze and K. Balem.

This last release is a major one. It introduces new features and breaking changes in the API.

New features

• New Argo float configuration manager. It was designed to make easier the access, management and backup of the virtual floats mission configuration parameters. All details are available on the API page `FloatConfiguration` and the documentation page “Argo floats mission parameters”.

```python
cfg = FloatConfiguration('default')  # Internally defined
cfg = FloatConfiguration('cfg_file.json')  # From json file
cfg.update('parking_depth', 500)  # Update one parameter value

cfg.params  # Return the list of parameters

cfg.mission  # Return the configuration as a dictionary, to be pass on a VirtualFleet instance

cfg.to_json("cfg_file.json")  # Save to file for later re-use
```

• New Argo virtual float type: this new float type can change their mission parameters when they enter a specific geographic area (a rectangular domain). In order to use these floats, you can load a `FloatConfiguration` instance with the local-change name, like this:

```python
cfg = FloatConfiguration('local-change')
cfg.update('area_cycle_duration', 120)  # Update default parameters for your own experiment
```

where you will note the added properties area_*:

```xml
<FloatConfiguration><local-change>
  - area_cycle_duration (Maximum length of float complete cycle in AREA): 120.0 [hours]
  - area_parking_depth (Drifting depth in AREA): 1000.0 [m]
  - area_xmax (AREA Eastern bound): -48.0 [deg_longitude]
  - area_xmin (AREA Western bound): -75.0 [deg_longitude]
  - area_ymax (AREA Northern bound): 45.5 [deg_latitude]
  - area_ymin (AREA Southern bound): 33.0 [deg_latitude]
  - cycle_duration (Maximum length of float complete cycle): 240.0 [hours]
  - life_expectancy (Maximum number of completed cycle): 200 [cycle]
  - parking_depth (Drifting depth): 1000.0 [m]
  - profile_depth (Maximum profile depth): 2000.0 [m]
  - vertical_speed (Vertical profiling speed): 0.09 [m/s]
</FloatConfiguration>
```
Passing this specific `FloatConfiguration` instance to a `VirtualFleet` will automatically select the appropriate Argo float parcel kernels (`app_parcels.ArgoFloatKernel_exp`). This new float type was developed for the EAREISE WP2.3 Gulf-Stream experiment.

- All Argo float types (default and local-change) now come with a proper cycle number property. This makes much easier the tracking of the float profiles.

Utilities:

- `utilities.simu2index, utilities.simu2csv`: An Argo profile index extractor from the simulation netcdf output. It is not trivial to extract the position of virtual float profiles from the trajectory file of the simulation output. We made this easier with these functions. It also comes bundled with the `VirtualFleet.to_index` method.

- `utilities.set_WMO`: A function to identify virtual floats with their real WMO from the deployment plan. This could be handful if the deployment plan is actually based on real floats with WMO.

- `utilities.get_float_config`: A function to retrieve Argo float cycle configuration using the Euro-Argo meta-data API:

Breaking changes

- Huge internal refactoring, with proper submodule assignment!

- The former `VelocityField` function to work with known velocity fields is now `Velocity()`. The new `VelocityField` refers to the class used to manage a velocity field.

- Options in `VirtualFleet`:
  - instantiation argument `vfield` has been replaced by `fieldset` and now must take a `parcels.fieldset.FieldSet` or a `VelocityField` instance.
  - the `VirtualFleet.simulate()` method has been refactored to use more explicit arguments and now takes `datetime.timedelta` as values, instead of mixed/confusing integer units.

1.6.3 v0.2.0 (30 Aug. 2021)

By K. Balem

```python
# Mission parameters
parking_depth = 1000. # in m
profile_depth = 2000.
vertical_speed = 0.09 # in m/s
cycle_duration = 10. # in days

mission = {'parking_depth':parking_depth, 'profile_depth':profile_depth, 'vertical_speed':vertical_speed, 'cycle_duration':cycle_duration}

VFleet = vaf.virtualfleet(lat=lat, lon=lon, depth=dpt, time=tim, vfield=VELfield, ...
      mission=mission)
```
1.6.4 v0.1.0 (29 Jun. 2020)

By K. Balem

This is the first release of Virtual Fleet with a single kernel (type of virtual Argo float) available and all its parameters are set internally.

1.7 Software development roadmap

- Core library for specific Argo kernel on Parcels
- Tutorial notebooks to get started
- Extraction of a real plan for a given region (using argopy), not automatic yet, but in notebooks
- Initial public release in June 2020
- Allow users to control mission parameters (eg: cycle time, parking depth, and custom kernels)
- Module to compute Virtual Fleet KPI and statistics (eg. sampling, trajectories)
- Improve documentation with regard to preparing the velocity field for the Virtual Fleet
- Add ISA option
- Clean software packaging (pypi install and dependencies)
- Unit testing ?

Post issues on Github to request features or if you find bugs.

You can also use the Gitter chat to quickly get in touch with the developers.

1.8 API reference

This page provides an auto-generated summary of VirtualFleet’s API. For more details and examples, refer to the relevant chapters in the main part of the documentation.

1.8.1 Top-levels classes

VirtualFleet

```
VirtualFleet(plan, fieldset, mission[, isglobal])  Argo Virtual Fleet simulator.
```
virtualargofleet.VirtualFleet

class VirtualFleet(plan: dict, fieldset: FieldSet | VelocityField, mission: dict | FloatConfiguration | Iterable[dict] | Iterable[FloatConfiguration], isglobal: bool = False, **kwargs)

Argo Virtual Fleet simulator.

This class makes it easy to process and analyse a simulation.

__init__(plan: dict, fieldset: FieldSet | VelocityField, mission: dict | FloatConfiguration | Iterable[dict] | Iterable[FloatConfiguration], isglobal: bool = False, **kwargs)

Create an Argo Virtual Fleet simulator

Parameters

• plan (dict) – A dictionary with the deployment plan coordinates as keys: lat, lon, time, [depth] Each value are Numpy arrays describing where Argo floats are deployed. Depth is optional, if not provided it will be set to 1m.

• fieldset (parcels.fieldset.FieldSet or VelocityField) – A velocity field

• mission (dict or FloatConfiguration or an iterable of those) – A dictionary with the following Argo float mission parameters: parking_depth, profile_depth, vertical_speed and cycle_duration. A FloatConfiguration instance can also be passed.

An iterable of dictionaries or FloatConfiguration can be passed to specified mission parameters for each virtual floats. In this case, the length of the iterable must match the length of the deployment plan.

• isglobal (bool, optional, default=False) – A boolean indicating weather the velocity field is global or not

Methods

__init__(plan, fieldset, mission[, isglobal]) Create an Argo Virtual Fleet simulator
plot_positions() Plot the last position of virtual Argo Floats
simulate(duration[, step, record, output, ...]) Execute a Virtual Fleet simulation
to_index([file_name]) Return last simulated profile index dataframe

Attributes

ParticleSet Return ParticleSet
fieldset Return FieldSet
output Return absolute path to the last simulation trajectory output file

Methods

VirtualFleet.simulate(duration[, step, ...]) Execute a Virtual Fleet simulation
VirtualFleet.to_index([file_name]) Return last simulated profile index dataframe
VirtualFleet.plot_positions() Plot the last position of virtual Argo Floats
virtualargofleet.VirtualFleet.simulate

VirtualFleet.simulate(duration, step=datetime.timedelta(seconds=300),
record=datetime.timedelta(seconds=3600), output=True, verbose_progress=True,
restart=False, **kwargs)

Execute a Virtual Fleet simulation

Parameters

• duration (datetime.timedelta,) – Length of the simulation
• step (datetime.timedelta, default=5 minutes) – Time step for the computation
• record (datetime.timedelta, default=1 hours) – Time step for writing the output
• output (bool, default=False) – Should the simulation trajectories be saved on file or not
• output_file (str) – Name of the zarr file where to store simulation results
• output_folder (str) – Name of folder where to store the ‘output_file’ zarr archive

Return type

self

virtualargofleet.VirtualFleet.to_index

VirtualFleet.to_index(file_name=None)

Return last simulated profile index dataframe

Return a pandas.DataFrame index of profiles. If the file_name option is provided, an Argo profile index csv file is written.

Parameters

file_name (str, default: None) – Name of the index file to write

virtualargofleet.VirtualFleet.plot_positions

VirtualFleet.plot_positions()

Plot the last position of virtual Argo Floats

Use parcels.particleset.baseparticleset.BaseParticleSet.show()

Attributes

<table>
<thead>
<tr>
<th>VirtualFleet.ParticleSet</th>
<th>Return ParticleSet</th>
</tr>
</thead>
<tbody>
<tr>
<td>VirtualFleet.fieldset</td>
<td>Return FieldSet</td>
</tr>
<tr>
<td>VirtualFleet.output</td>
<td>Return absolute path to the last simulation trajectory output file</td>
</tr>
</tbody>
</table>
virtualargofleet.VirtualFleet.ParticleSet

**property** VirtualFleet.ParticleSet
   Return ParticleSet
   **Return type**
   parcels.particleset.particlesetsoa.ParticleSetSOA

virtualargofleet.VirtualFleet.fieldset

**property** VirtualFleet.fieldset
   Return FieldSet
   **Return type**
   parcels.fieldset.FieldSet

virtualargofleet.VirtualFleet.output

**property** VirtualFleet.output
   Return absolute path to the last simulation trajectory output file

**FloatConfiguration**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FloatConfiguration(name)</td>
<td>Float mission configuration manager</td>
</tr>
<tr>
<td>FloatConfiguration.to_json(file_name)</td>
<td>Return or save json dump of configuration</td>
</tr>
<tr>
<td>FloatConfiguration.update(key, new_value)</td>
<td>Update value to an existing parameter</td>
</tr>
<tr>
<td>FloatConfiguration.mission</td>
<td>Return the float configuration as a dictionary to be used by a VirtualFleet</td>
</tr>
<tr>
<td>FloatConfiguration.tech</td>
<td>Float configuration as a dictionary using Argo technical keys</td>
</tr>
<tr>
<td>FloatConfiguration.params</td>
<td>List of parameter keys</td>
</tr>
</tbody>
</table>

virtualargofleet.FloatConfiguration

**class** FloatConfiguration(name: str | list = 'default', *args, **kwargs)
   Float mission configuration manager
   Create a default configuration and then possibly update parameter values
   Can be used to create a virtual fleet, to save or load float configurations
Examples

```python
>>> cfg = FloatConfiguration('default')  # Internally defined
>>> cfg = FloatConfiguration('local-change')  # Internally defined
>>> cfg = FloatConfiguration('cfg_file.json')  # From any json file
>>> cfg = FloatConfiguration([6902919, 132])  # From Euro-Argo Fleet API
>>> cfg.update('parking_depth', 500)  # Update one parameter value
>>> cfg.params  # Return the list of parameters
>>> cfg.mission  # Return the configuration as a dictionary
>>> cfg.tech  # Return the configuration as a dictionary using Argo technical keys
>>> cfg.to_json("cfg_file.json")  # Save to file for later re-use
>>> cfg
```

__init__(name: str | list = 'default', *args, **kwargs)

Parameters

**name** (str) – Name of the configuration to load

Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>([name])</td>
<td>param name Name of the configuration to load</td>
</tr>
<tr>
<td>to_json([file_name])</td>
<td>Return or save json dump of configuration</td>
</tr>
<tr>
<td>update(key, new_value)</td>
<td>Update value to an existing parameter</td>
</tr>
</tbody>
</table>

Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mission</td>
<td>Return the float configuration as a dictionary to be used by a VirtualFleet</td>
</tr>
<tr>
<td>params</td>
<td>List of parameter keys</td>
</tr>
<tr>
<td>tech</td>
<td>Float configuration as a dictionary using Argo technical keys</td>
</tr>
</tbody>
</table>

virtualargofleet.FloatConfiguration.to_json

FloatConfiguration.to_json(file_name: str | None = None)

Return or save json dump of configuration

If no file name is provided, just return the configuration as a json structure

Parameters

**file_name** (str, default:None, optional) – Name of the json file to write configuration to.

Return type

Nothing or json string
virtualargofleet.FloatConfiguration.update

FloatConfiguration.update(key: str, new_value)

Update value to an existing parameter

**Parameters**

- **key** (str) – Name of the parameter to update
- **new_value** – New value to attribute to this parameter

virtualargofleet.FloatConfiguration.mission

**property** FloatConfiguration.mission

Return the float configuration as a dictionary to be used by a VirtualFleet

virtualargofleet.FloatConfiguration.tech

**property** FloatConfiguration.tech

Float configuration as a dictionary using Argo technical keys

virtualargofleet.FloatConfiguration.params

**property** FloatConfiguration.params

List of parameter keys

### Velocity/Field

<table>
<thead>
<tr>
<th><strong>Velocity([model])</strong></th>
<th>Function to return a VelocityField instance for known products</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>VelocityField()</strong></td>
<td>Class prototype to manage a Virtual Fleet velocity field</td>
</tr>
<tr>
<td><strong>VelocityField.add_mask()</strong></td>
<td>Create mask for grounding management</td>
</tr>
<tr>
<td><strong>VelocityField.set_global()</strong></td>
<td>Ensure a global fieldset</td>
</tr>
<tr>
<td><strong>VelocityField.plot()</strong></td>
<td>Quick plot of the ParticleSet</td>
</tr>
<tr>
<td><strong>VelocityField.fieldset</strong></td>
<td>Instance of parcels.fieldset.FieldSet created using the field attribute</td>
</tr>
</tbody>
</table>

virtualargofleet.Velocity

**Velocity(model: str = 'GLOBAL_ANALYSIS_FORECAST_PHY_001_024', *args, **kwargs)**

Function to return a VelocityField instance for known products

Note that you can provide a VelocityField or VelocityField.fieldset to a VirtualFleet instance.

**Parameters**

- **model** (str) – Indicate which model to use by its string definition. Possible values are:
  - custom if you want to set your own model definition
• GLORYS12V1, PSY4QV3R1, GLOBAL_ANALYSIS_FORECAST_PHY_001_024
• MEDSEA_ANALYSIS_FORECAST_PHY_006_013
• ARMOR3D, MULTI_OBS_GLO_PHY_TSUV_3D_MYNRT_015_012

**Return type**

VelocityField

**Examples**

Import the module and define the root folder to data:

```python
>>> from virtualargofleet import Velocity
>>> root = '/home/datawork-lops-oh/somovar/WP1/data/GLOBAL-ANALYSIS-FORECAST-PHY-001-024'
```

And then define a velocity field with one of the following 3 methods:

1/ with a custom product:

```python
>>> filenames = {'U': root + '/20201210*.nc',
               'V': root + '/20201210*.nc'}
>>> variables = {'U':'uo', 'V': 'vo'}
>>> dimensions = {'time': 'time', 'depth': 'depth', 'lat': 'latitude', 'lon': 'longitude'}
>>> VELfield = Velocity(model='custom', src=filenames, variables=variables,
                      dimensions=dimensions)
```

2/ with a `xarray.Dataset`:

```python
>>> ds = xr.open_mfdataset(glob.glob('%s/20201210*.nc' % root))
>>> VELfield = Velocity(model='GLOBAL_ANALYSIS_FORECAST_PHY_001_024', src=ds)
```

3/ with a file path pattern:

```python
>>> VELfield = Velocity(model='GLORYS12V1', src='%s/20201210*.nc' % root)
```

**virtualargofleet.VelocityField**

**class VelocityField**

Class prototype to manage a Virtual Fleet velocity field

This prototype provides useful methods to prepare a `parcels.fieldset.FieldSet` for a VirtualFleet simulation. A `VelocityField` instance can be passed directly to a `VirtualFleet` instance.

You can use the `Velocity()` function to instantiate such a class for known products.

```python
__init__(*args, **kwargs)
```
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>init</strong>(*)args, *<em>kwargs</em>)</td>
<td>Create mask for grounding management</td>
</tr>
<tr>
<td>add_mask()</td>
<td>Create mask for grounding management</td>
</tr>
<tr>
<td>plot()</td>
<td>Quick plot of the ParticleSet</td>
</tr>
<tr>
<td>set_global()</td>
<td>Ensure a global fieldset</td>
</tr>
</tbody>
</table>

## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>dim</td>
<td>Dimensions dictionary mapping of time, depth, lat and lon on netcdf velocity variable names</td>
</tr>
<tr>
<td>field</td>
<td>Internal definition of the velocity fields; it can be a <code>xarray.Dataset</code> or a dictionary with U and V as keys and list of corresponding files as values</td>
</tr>
<tr>
<td>fieldset</td>
<td>Instance of <code>parcels.fieldset.FieldSet</code> created using the field attribute</td>
</tr>
<tr>
<td>isglobal</td>
<td>Boolean indicating weather the velocity field is global or not, used to add <code>halo_*</code> constants on the fieldset attribute</td>
</tr>
<tr>
<td>name</td>
<td>Shortname ID for this velocity field</td>
</tr>
<tr>
<td>var</td>
<td>Variable dictionary mapping of U and V on netcdf velocity variable names</td>
</tr>
</tbody>
</table>

### virtualargofleet.VelocityField.add_mask

`VelocityField.add_mask()`
Create mask for grounding management

**Requires:**
- `self.field` with U and V keys
- `self.dim` with lon, lat, depth and time keys
- `self.var` with U and V keys

### virtualargofleet.VelocityField.set_global

`VelocityField.set_global()`
Ensure a global fieldset
virtualargofleet.VelocityField.plot

VelocityField.plot()
Quick plot of the ParticleSet

virtualargofleet.VelocityField.fieldset

VelocityField.fieldset = None
Instance of parcels.fieldset.FieldSet created using the field attribute

1.8.2 Utilities

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>utilities.simu2index(ds, N)</td>
<td>Convert a trajectory simulation xarray.Dataset to an Argo index of profiles</td>
</tr>
<tr>
<td>utilities.simu2csv(simu_file, index_file, df)</td>
<td>Save simulation results profile index to file, as Argo index</td>
</tr>
<tr>
<td>utilities.set_WMO(ds, argo_index)</td>
<td>Identify virtual floats with their real WMO</td>
</tr>
<tr>
<td>utilities.get_float_config(wmo, cyc)</td>
<td>Download float configuration using the Euro-Argo metadata API</td>
</tr>
</tbody>
</table>

virtualargofleet.utilities.simu2index

simu2index(ds: Dataset, N: int = 1)
Convert a trajectory simulation xarray.Dataset to an Argo index of profiles
Profiles are identified using the cycle_number dataset variable. A profile is identified if the last observation of a cycle_number sequence is in cycle_phase 3 or 4.
This function remains compatible with older versions of trajectory netcdf files without the cycle_number variable. In this case, a profile is identified if the last observation of a cycle_phase==3 sequence is separated by N days from the next sequence.

Parameters
- ds (xarray.Dataset) – The simulation trajectories dataset
- N (int, optional) – The minimal time lag between cycle_phase sequences to be identified as a new profile. This will be removed in the future when we’ll drop support for old netcdf outputs.

Returns
df – The profiles index

Return type
pandas.DataFrame
virtualargofleet.utilities.simu2csv

**simu2csv**(*simu_file: str, index_file: str | None = None, df: DataFrame | None = None*)

Save simulation results profile index to file, as Argo index

Argo profile index can be loaded with argopy.

**Parameters**

- **simu_file** (*str*) – Path to netcdf file of simulation results, to load profiles from
- **index_file** (*str, optional*) – Path to csv file to write index to. By default, it is set using the simu_file value.
- **df** (*pandas.Dataframe, optional*) – If provided, will be used as the profile index, otherwise, compute index from simu_file

**Returns**

- **index_file** – Path to the Argo profile index created

**Return type**

- **str**

virtualargofleet.utilities.set_WMO

**set_WMO**(*ds: Dataset, argo_index: DataFrame*)

Identify virtual floats with their real WMO

This function will try to identify WMO from argo_index in the ds trajectories.

The Argo index must have at least the longitude and latitude variables. It’s assumed to be the deployment plan.

Real WMO numbers are identified as the closest floats from argo_index to the initial positions of virtual floats from ds.

**Parameters**

- **ds** (*xarray.Dataset*) – The simulation trajectories dataset
- **argo_index** (*pandas.DataFrame*) – The deployment plan profiles index

**Returns**

- **ds** – The simulation trajectories dataset with a new variable wmo

**Return type**

- **xarray.Dataset**

virtualargofleet.utilities.get_float_config

**get_float_config**(*wmo: int, cyc: int | None = None*)

Download float configuration using the Euro-Argo meta-data API

**Parameters**

- **wmo** (*int*) – The float WMO number
- **cyc** (*int, default: None*) – The specific cycle number to retrieve data from. If set to None, all cycles meta-data are fetched.
Returns
A dataframe with relevant float configuration parameters for 1 or more cycle numbers.

Return type
pandas.DataFrame

1.8.3 Parcels Particles and kernels

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>app_parcels.ArgoFloatKernel</code></td>
<td>Default kernel to simulate an Argo float</td>
</tr>
<tr>
<td><code>app_parcels.ArgoParticle</code></td>
<td>Default class to represent an Argo float</td>
</tr>
<tr>
<td><code>app_parcels.ArgoFloatKernel_exp</code></td>
<td>Argo float kernel to simulate an Argo float cycle with change of mission parameters in a specific geographical area</td>
</tr>
<tr>
<td><code>app_parcels.ArgoParticle_exp</code></td>
<td>Class used to represent an Argo float that can temporarily change its mission parameters</td>
</tr>
</tbody>
</table>

**virtualargofleet.app_parcels.ArgoFloatKernel**

`ArgoFloatKernel(particle, fieldset, time)`
Default kernel to simulate an Argo float

It only takes (particle, fieldset, time) as arguments.
Virtual float missions parameters are passed as Variables to the particles.
This function will be compiled at run time.

**Parameters**

- `particle` (*ArgoParticle*) – An instance of virtual Argo float. This instance must also have the following attributes: `parking_depth`, `profile_depth`, `vertical_speed`, `cycle_duration`, `life_expectancy`
- `fieldset` (parcels.fieldset.FieldSet) – A FieldSet class instance that holds hydrodynamic data needed to transport virtual floats. This instance must also have the following attributes: `-verbose_events, mask`
- `time` –

**virtualargofleet.app_parcels.ArgoParticle**

`class ArgoParticle(*args, **kwargs)`
Default class to represent an Argo float

`ArgoParticle` inherits from `parcels.particle.JITParticle`.

A `VirtualFleet` will create and work with a `parcels.particleset.particlesetsoa.ParticleSetSOA` of this class.

**Return type**
`parcels.particle.JITParticle`

`__init__(*args, **kwargs)`
## Methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__init__(args, **kwargs)</code></td>
<td>Initialize function</td>
</tr>
<tr>
<td><code>getInitialValue(ptype, name)</code></td>
<td>Get initial value</td>
</tr>
<tr>
<td><code>getPType()</code></td>
<td>Get property type</td>
</tr>
<tr>
<td><code>setLastID(offset)</code></td>
<td>Set last ID</td>
</tr>
<tr>
<td><code>set_lonlatdepth_dtype(dtype)</code></td>
<td>Set longitude, latitude, and depth data type</td>
</tr>
</tbody>
</table>

1.8. API reference
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cycle_age</code></td>
<td>Elapsed time since the beginning of the current cycle</td>
</tr>
<tr>
<td><code>cycle_duration</code></td>
<td>Float mission parameter cycle duration in hours</td>
</tr>
<tr>
<td><code>cycle_number</code></td>
<td>Cycle number (starts at 1)</td>
</tr>
<tr>
<td><code>cycle_phase</code></td>
<td>Cycle phase (init_descend = 0, drift = 1, profile_descend = 2, profile_ascend = 3, transmit = 4)</td>
</tr>
<tr>
<td><code>depth</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>depth_nextloop</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>drift_age</code></td>
<td>Elapsed time since the beginning of the drifting phase</td>
</tr>
<tr>
<td><code>dt</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>id</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>in_water</code></td>
<td>Boolean indicating if the virtual float is in land (0) or water (1), used to detect grounding, based on field-set.mask</td>
</tr>
<tr>
<td><code>lastID</code></td>
<td></td>
</tr>
<tr>
<td><code>lat</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>lat_nextloop</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>life_expectancy</code></td>
<td>Float mission parameter life expectancy in cycle</td>
</tr>
<tr>
<td><code>lon</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>lon_nextloop</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>obs_written</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>parking_depth</code></td>
<td>Float mission parameter parking depth in m</td>
</tr>
<tr>
<td><code>profile_depth</code></td>
<td>Float mission parameter profile depth in m</td>
</tr>
<tr>
<td><code>state</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>time</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>time_nextloop</code></td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td><code>vertical_speed</code></td>
<td>Float mission parameter vertical speed in m/s</td>
</tr>
</tbody>
</table>
virtualargofleet.app_parcels.ArgoFloatKernel_exp

ArgoFloatKernel_exp(\textit{particle, fieldset, time})
Argo float kernel to simulate an Argo float cycle with change of mission parameters in a specific geographical area

\textbf{Parameters}

- \textbf{particle (ArgoParticle\_exp)} – A virtual Argo float of ‘local-change’ type
- \textbf{fieldset (parcels.fieldset.FieldSet)} – A FieldSet class instance that holds hydrodynamic data needed to transport virtual floats. This instance must also have the following attributes:
  - \textit{parking\_depth, profile\_depth, vertical\_speed, cycle\_duration, life\_expectancy, mask}
  - \textit{area\_xmin, area\_xmax, area\_ymin, area\_ymax, area\_cycle\_duration, area\_parking\_depth}
- \textbf{time –}

virtualargofleet.app_parcels.ArgoParticle_exp

class ArgoParticle\_exp(*args, **kwargs)
Class used to represent an Argo float that can temporarily change its mission parameters
This class extends \texttt{ArgoParticle}.
To be used by the \texttt{ArgoFloatKernel\_exp} kernel.

\textbf{Return type}
parcels.particle.JITParticle

\texttt{\_\_\_init\_\_}(\*args, **kwargs)

\textbf{Methods}

\begin{tabular}{|l|}
\hline
\texttt{\_\_\_init\_\_}(\*args, **kwargs) \\
getInitialValue(ptype, name) \\
getPType() \\
setLastID(offset) \\
set_lonlatdepth Dtype(dtype) \\
\hline
\end{tabular}
## Attributes

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cycle_age</td>
<td>Elapsed time since the beginning of the current cycle</td>
</tr>
<tr>
<td>cycle_duration</td>
<td>Float mission parameter cycle duration in hours</td>
</tr>
<tr>
<td>cycle_number</td>
<td>Cycle number (starts at 1)</td>
</tr>
<tr>
<td>cycle_phase</td>
<td>Cycle phase (init_descend = 0, drift = 1, profile_descend = 2, profile_ascend = 3, transmit = 4)</td>
</tr>
<tr>
<td>depth</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>depth_nextloop</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>drift_age</td>
<td>Elapsed time since the beginning of the drifting phase</td>
</tr>
<tr>
<td>dt</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>id</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>in_area</td>
<td>Boolean indicating if the virtual float in the experiment area (1) or not (0)</td>
</tr>
<tr>
<td>in_water</td>
<td>Boolean indicating if the virtual float is in land (0) or water (1), used to detect grounding, based on field-set.mask</td>
</tr>
<tr>
<td>lastID</td>
<td></td>
</tr>
<tr>
<td>lat</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>lat_nextloop</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>life_expectancy</td>
<td>Float mission parameter life expectancy in cycle</td>
</tr>
<tr>
<td>lon</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>lon_nextloop</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>obs_written</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>parking_depth</td>
<td>Float mission parameter parking depth in m</td>
</tr>
<tr>
<td>profile_depth</td>
<td>Float mission parameter profile depth in m</td>
</tr>
<tr>
<td>state</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>time</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>time_nextloop</td>
<td>Descriptor class that delegates data access to particle data.</td>
</tr>
<tr>
<td>vertical_speed</td>
<td>Float mission parameter vertical speed in m/s</td>
</tr>
</tbody>
</table>

