Simons CMAP

Release 0.0.1

Nov 17, 2020

User Guide

Mohammad Dehghani Ashkezari
Norland Raphael Hagen
Mike Denholtz, and
Ginger Armbrust

Simons CMAP is an open-source data service to retrieve, visualize, and analyze oceanic datasets such as in-situ observations, multi-decadal global satellite remote sensing, and model outputs. It enables scientists and the public to dive into the vast, and often underutilized, ocean datasets and to retrieve customized subsets of these massive datasets without going through the time-consuming process of data collection and preparation.

Simons CMAP website is under active development: https://simonscmap.com

This project is supported by the Simons Foundation

An API key is required to run all software packages. To obtain an API key, please register first and then go to https://simonscmap.com/apikeymanagement. Please keep this key for future use. All projects developed by Simons CMAP team are open-source and can be found on Github.

pycmap API	cmap4r API	matcmap API	juliacmap API

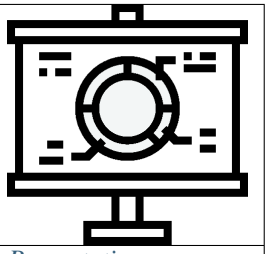
CHAPTER 1

pycmap API

Pycmap is the python package of the Simons CMAP project. It provides a simple and unified interface to the hosted datasets in the Simons CMAP database.

The pycmap documentation is written in the Jupyter Notebook format and available on GitHub. Each notebook contains one or more example codes. You can run the examples on Google cloud using the badge or on Binder using the badge that appears at the top of each tutorial page. Alternatively, you can run the code locally using a Jupyter Notebook.

Please click here to download the pycmap project and associated documentation.

			
Installation	*Data Retrieval Methods*	*Data Visualization Methods*	*Presentations*

1.1 Installation

To use pycmap locally, you will need Python 3+ installed on your computer. We recommend the Anaconda Distribution.

pycmap can be installed using pip:

```
pip install pycmap
```

In order to use pycmap, you will need to obtain an API key from https://simonscmap.com/apikeymanagement.

Note: You may install pycmap on cloud-based Jupyter notebooks (such as Binder, or Colab) by running the following command.

```
!pip install pycmap
```

```
#!pip install pycmap     #uncomment to install pycmap on Colab

import pycmap

pycmap.__version__
```

1.2 Data Retrieval Methods

The methods below can be used to retrieve metadata and to query, subset, and colocalize datasets. To begin, you must create an instance of the pycmap API. This tutorial can be found below.

API	Create an Instance of the API
Query	Generic SQL Query
Catalog	Retrieve Catalog
Search Catalog	Search Catalog by Keyword
Datasets	Retrieve List of Datasets
Metadata	Retrieve Variable Metadata
Dataset Metadata	Retrieve Dataset Metadata
Dataset Columns	Retrieve Column Names
Dataset Head	Retrieve Top Rows of Given Dataset
Variable Long Name	Returns Long Name of Given Variable
Variable Unit	Returns Unit for Given Variable, if applicable
Variable Resolution	Returns Spatial and Temporal Resolution of Given Variable
Variable Coverage	Returns Spatial and Temporal Coverage of Given Variable
Variable Stat	Returns Summary Statistics for Given Variable
If Column Exists	Returns True if Column Exists
Is Gridded Product	Returns True if Variable is Gridded Product
Is Climatology Product	Returns True if Dataset is Climatology Product
List of Cruises	Returns Cruise Expedition Details
Cruise Details by Name	Returns Cruise Specific Details by Name
Cruise Spatio-Temporal Bounds	Returns Cruise Spatio-Temporal Bounds
Cruise Trajectory	Returns Trajectory for Specific Cruise
Cruise Variables	Returns Variables Associated with Specific Cruise
Retrieve Dataset	Returns the Entire Dataset
Data Subset: Generic Space-Time Cut	Returns a Subset of Data Defined by Space and Time
Data Subset: Time Series	Returns a Subset of Data Aggregated by Time
Data Subset: Depth Profile	Returns a Subset of Data Aggregated by Depth
Match (colocalize) Datasets	Colocalizes Variables between Datasets
Match (colocalize) Cruise Track with Datasets	Colocalizes Variables along Cruise Trajectory

1.2.1 API

class API

To retrieve data, create an instance of the system's API and pass the API key. It is not necessary to pass the API key every time you run a code locally, because it will be stored locally. The API class has other optional parameters to adjust its behavior. All parameters can be updated persistently at any point in the code.

```
class pycmap.API(token=<API KEY>, baseURL='https://simonscmap.com',
headers=None, vizEngine='plotly', exportDir='./export/', > exportFormat='.
csv', figureDir='./figure/')|
```

Parameters

token: string, required Access token (API Key) required to make client requests. You may get an API key here: https://simonscmap.com/apikeymanagement

baseURL: string, optional, default: https://simonscmap.com Root endpoint of Simons CMAP API.

headers: dict, optional, default: None Additional headers to add to the client requests.

vizEngine: string, optional, default: 'plotly' Data visualization library used to render the graphs. The other option for visualization library is 'bokeh'. Note some of the graphs (such as correlation matrix) are only supported by plotly. See Data Visualization for more details.

exportDir: string, optional, default: './export/' Path to local directory where the exported data are stored.

exportFormat: string, optional, default: '.csv' File format of the exported files. Currently, only csv format is supported.

figureDir: string, optional, default: './figure/' Path to local directory where the figures are stored. The interactive figure objects are stored in the form of html files. Note: inline graphs (e.g. on Jupyter notebook) are not saved.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>', vizEngine='plotly')
```

1.2.2 Query

query(*query*)

```
This is an 'optional' page. If you are not planning and/or not interested
in SQL coding, simply ignore this page please!
```

Simons CMAP datasets are hosted in a SQL database and pycmap package provides the user with a number of pre-developed methods to extract and retrieve subsets of the data. The rest of this documentation is dedicated to exploring and explaining these methods. In addition to the pre-developed methods, we intend to leave the

database open to custom scan queries for interested users. This method takes a custom SQL query statement and returns the results in form of a Pandas dataframe. The full list of table names and variable names (fields) can be obtained using the *Catalog* method. In fact, one may use this very method to retrieve the table and field names: `query('EXEC uspCatalog')`. A Dataset is stored in a table and each table field represents a variable. All data tables have the following fields:

- [time] [date or datetime] NOT NULL,
- [lat] [float] NOT NULL,
- [lon] [float] NOT NULL,
- [depth] [float] NOT NULL,

Note: Tables which represent a climatological dataset, such as 'tblDarwin_Nutrient_Climatology', will not have a 'time' field. Also, if a table represents a surface dataset, such as satellite products, there would be no 'depth' field. 'depth' is a positive number in meters; it is zero at the surface and increasing towards the ocean floor. 'lat' and 'lon' are in degrees units, ranging from -90° to 90° and -180° to 180°, respectively.

Please keep in mind that some of the datasets are massive in size (10s of TB), avoid queries without WHERE clause (`SELECT * FROM TABLENAME`). Always try to add some constraints on time, lat, lon, and depth fields (see the basic examples below).

Moreover, the database hosts a wide range of predefined stored procedures and functions to streamline nearly all CMAP data services. For instance, retrieving the catalog information is achieved using a single call of this procedure: uspCatalog. These predefined procedures can be called using the pycmap package (see Example 3). Alternatively, one may use any SQL client to execute these procedures to retrieve and visualize data (examples: Azure Data Studio, or Plotly Falcon). Using the predefined procedures, all CMAP data services are centralized at the database layer which dramatically facilitates the process of developing apps with different programming languages (pycmap, web app, cmap4r, . . .). Please note that you can improve the current procedures or add new procedures by contributing at the CMAP database repository. Below is a selected list of stored procedures and functions, their arguments will be described in more detail subsequently:

- uspCatalog
- uspSpaceTime
- uspTimeSeries
- uspDepthProfile
- uspSectionMap
- uspCruises
- uspCruiseByName
- uspCruiseBounds
- uspWeekly
- uspMonthly
- uspQuarterly
- uspAnnual
- uspMatch
- udfDatasetReferences
- udfMetaData_NoRef

Parameters

query: string Generic scan SQL statement. A full list of tables can be found in the Catalog.

Returns Pandas dataframe.

1.2.2.1 Example 1:

A generic query returning dissolved iron (Fe) from Pisces model dataset.

```
# !pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.query(
          '''
          SELECT [time], lat, lon, depth, Fe FROM tblPisces_NRT
          WHERE
          [time] BETWEEN '2017-06-03' AND '2017-06-03' AND
          lat BETWEEN 10 AND 55 AND
          lon BETWEEN -180 AND 100 AND
          depth BETWEEN 0 AND 0.5
          ORDER BY [time], lat, lon, depth
          '''
          )
```

1.2.2.2 Example 2:

A sample query returning the timeseries of sea surface temperature (sst).

```
# !pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.query(
          '''
          SELECT [time], AVG(lat) AS lat, AVG(lon) AS lon, AVG(sst) AS sst FROM tblsst_
↪AVHRR_OI_NRT
          WHERE
          [time] BETWEEN '2016-06-01' AND '2016-10-01' AND
          lat BETWEEN 23 AND 24 AND
          lon BETWEEN -160 AND -158
          GROUP BY [time]
          ORDER BY [time]
          '''
          )
```

1.2.2.3 Example 3:

A sample query calling a predefined stored procedure.

```
# !pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.query('EXEC uspCatalog')
```

1.2.3 Catalog

get_catalog()

Returns a dataframe containing the details of all variables at Simons CMAP database. This method requires no input.

A static version of the Catalog can be found at the Simons CMAP documentation.

Alternatively, the catalog may be explored interactively at Simons CMAP website: https://simonscmap.com

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_catalog()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCatalog
```

1.2.4 Search Catalog

search_catalog(*keywords*)

Returns a dataframe containing a subset of Simons CMAP catalog of variables.

All variables at Simons CMAP catalog are annotated with a collection of semantically related keywords. This method takes the passed keywords and returns all of the variables annotated with similar keywords. The passed keywords should be separated by blank space. The search result is not sensitive to the order of keywords and is not case sensitive. The passed keywords can provide any 'hint' associated with the target variables. Below are a few examples:

- the exact variable name (e.g. NO3), or its linguistic term (nitrate)
- methodology (e.g. model, satellite ...), instrument (e.g. CTD, SeaFlow), or disciplines (e.g. physics, biology ...)
- the official cruise name (e.g. KOK1606), or unofficial cruise name (Falkor)
- the name of data producer (e.g. Penny Chisholm) or institution name (MIT)

If you searched for a variable with semantically related keywords and did not get the correct results, please let us know. We can update the keywords at any point.

Parameters

keywords: string Search keywords separated by a blank space. The search result is not sensitive to the order of keywords and is not case sensitive.

Returns Pandas dataframe.

Example 1:

List of all measurements by University of Hawaii hosted by Simons CMAP.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.search_catalog('University of Hawaii')
```

Example 2:

Returns a list of nitrite measurements during the Falkor cruise, if exists.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.search_catalog('nitrite falkor')
```

Example 3:

Returns a list of optical measurements during the Gradients 1 cruise (KOK1606), if exists.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.search_catalog('optics KOK1606')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspSearchCatalog 'space-separated keywords'
```

Example:

List of satellite chlorophyll products:

```
EXEC uspSearchCatalog 'chl satellite'
```

1.2.5 Datasets

datasets()

Returns a dataframe containing the list of datasets hosted by Simons CMAP database. The returned dataframe includes the dataset names and the table names storing the datasets. This method requires no input.

A static version of the dataset list can be found at the Simons CMAP documentation. Alternatively, the datasets may be explored interactively at Simons CMAP website: https://simonscmap.com/catalog

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.datasets()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspDatasets
```

1.2.6 Metadata

get_metadata (*table*, *variable*)

Returns a dataframe containing the associated metadata of a variable (such as data source, distributor, references, and etc.). The inputs can be string literals (if only one table, and variable is passed) or a list of string literals.

Parameters

table: string or list of string The name of table where the variable is stored. A full list of table names can be found in the Catalog.

variable: string or list of string: Variable short name. A full list of variable short names can be found in the Catalog.

Returns Pandas dataframe.

Example

Retrieve dataset metadata for the satellite SST and Argo salinity datasets.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_metadata(['tblsst_AVHRR_OI_NRT', 'tblArgoMerge_REP'], ['sst', 'argo_merge_
↪salinity_adj'])
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableMetaData 'table', 'variable'
```

Example:

Metadata associated with Argo salinity measurements:

```
EXEC uspVariableMetaData 'tblArgoMerge_REP', 'argo_merge_salinity_adj'
```

1.2.7 Dataset Metadata

get_dataset_metadata(*tableName*)

Returns a dataframe containing the dataset metadata. (such as a list of all variables, data source, distributor, references, and etc..)

Parameters

tableName: string The name of the table where the dataset is stored. A full list of table names can be found in the Catalog.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_dataset_metadata('tblArgoMerge_REP')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspDatasetMetadata 'tableName'
```

Example:

Metadata associated with the Argo dataset:

```
EXEC uspDatasetMetadata 'tblArgoMerge_REP'
```

1.2.8 Dataset Columns

columns(*table*)

Returns the column names of a given dataset.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog or *Datasets* method.

Example

Column names associated with the AMT 13 Cruise dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.columns('tblAMT13_Chisholm')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspColumns 'tableName'
```

Example:

Column names associated with the AMT 13 Cruise dataset.

```
EXEC uspColumns 'tblAMT13_Chisholm'
```

1.2.9 Dataset Head

head(*tableName*, *rows=5*)

Returns top rows of a given dataset.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog or *Datasets* method.

rows: int, default: 5 Number of top rows to be returned (default 5).

Returns Pandas dataframe.

Example

Retrieves the top five rows of the Falkor 2018 cruise dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.head('tblFalkor_2018')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspHead 'tableName', 'rows'
```

Example

Retrieves the top five rows of the Falkor 2018 cruise dataset.

```
EXEC uspHead 'tblFalkor_2018', '5'
```

1.2.10 Variable Long Name

get_var_long_name(*tableName*, *varName*)
Returns the long name of a given variable.

Parameters

tableName: string The name of the table associated with the dataset. A full list of table names can be found in the Catalog.

variable: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns String literal.

Example

Returns the long name of the short name variable, adt, in the Altimetry Reprocessed dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_var_long_name('tblAltimetry_REP', 'adt')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableLongName 'tableName', 'varName'
```

Example

Returns the long name of the short name variable, adt, in the Altimetry Reprocessed dataset.

```
EXEC uspVariableLongName 'tblAltimetry_REP', 'adt'
```

1.2.11 Variable Unit

get_unit (*tableName*, *varName*)
Returns the unit for a given variable, if applicable.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

varName: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns String literal.

Example

Returns the unit of the short name variable, silica_hot, in the HOT Particle Flux dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_unit('tblHOT_ParticleFlux', 'silica_hot')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableUnit 'tableName', 'varName'
```

Example

Returns the unit of the short name variable, silica_hot, in the HOT Particle Flux dataset.

```
EXEC uspVariableUnit 'tblHOT_ParticleFlux', 'silica_hot'
```

1.2.12 Variable Resolution

get_var_resolution(*tableName*, *varName*)

Returns spatial and temporal resolutions of the given variable.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

varName: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns Pandas dataframe.

Example

Returns the spatial and temporal resolution of the short name variable, AOD, in the MODIS Aerosol Optical Depth dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_var_resolution('tblModis_AOD_REP', 'AOD')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableResolution 'tableName', 'varName'
```

Example

Returns the spatial and temporal resolution of the short name variable, AOD, in the MODIS Aerosol Optical Depth dataset.

```
EXEC uspVariableResolution 'tblModis_AOD_REP', 'AOD'
```

1.2.13 Variable Coverage

get_var_coverage(*tableName*, *varName*)

Returns spatial and temporal coverage of the given variable.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

variable: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns Pandas dataframe.

Example

Returns the spatial and temporal coverage of the short name variable, chl, in the Chlorophyll Reprocessed dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_var_coverage('tblCHL_REP', 'chl')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableCoverage 'tableName', 'varName'
```

Example

Returns the spatial and temporal coverage of the short name variable, chl, in the Chlorophyll Reprocessed dataset.

```
EXEC uspVariableCoverage 'tblCHL_REP', 'chl'
```

1.2.14 Variable Stat

get_var_stat (*tableName*, *varName*)

Returns summary statistics involving the min, max, mean, standard deviation, count, and quantiles for the given variable.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

variable: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns Pandas dataframe.

Example

Returns summary statistics for the short name variable, Prochlorococcus, in the HOT LAVA Cruise dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_var_stat('tblHOT_LAVA', 'Prochlorococcus')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspVariableStat 'tableName', 'varName'
```

Example

Returns summary statistics for the short name variable, Prochlorococcus, in the HOT LAVA Cruise dataset.

```
EXEC uspVariableStat 'tblHOT_LAVA', 'Prochlorococcus'
```

1.2.15 If Column Exists

has_field(*tableName*, *varName*)

Returns True if the specified column (field) exists in the table Returns False if either table or variable does not exist (also see *Dataset Columns*).

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

varName: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

Returns Boolean

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap
```

(continues on next page)

(continued from previous page)

```
api = pycmap.API(token='<YOUR_API_KEY>')
api.has_field('tblAltimetry_REP', 'sla')
```

1.2.16 Is Gridded Product

is_grid(*tableName*, *varName*)

Returns True if the specified variable represents a gridded product; otherwise returns False. For instance, model outputs or satellite products in the form of structured arrays are considered gridded products, while underway cruise measurements with irregular spatial or temporal resolutions are considered "sparse" products.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

varName: string or list of string Variable short name. A full list of variable short names can be found in the Catalog.

returns Boolean.

Example

Checking if the short name variable, argo_merge_salinity_adj, is a gridded product in the Argo dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.is_grid('tblArgoMerge_REP', 'argo_merge_salinity_adj')
```

1.2.17 Is Climatology Product

is_climatology(*tableName*)

Returns True if the specified dataset represents a climatological product; otherwise returns False.

Parameters

tableName: string The name of table associated with the dataset. A full list of table names can be found in the Catalog.

Returns Boolean.

Example

Checking if the table, tblDarwin_Plankton_Climatology, is a climatological product in the Darwin Climatology dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.is_climatology('tblDarwin_Plankton_Climatology')
```

1.2.18 List of Cruises

cruises()

Returns a dataframe containing the details of all cruise expeditions stored at Simons CMAP database. This method requires no input.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.cruises()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCruises
```

1.2.19 Cruise Details by Name

cruise_by_name(*cruiseName*)

Returns a dataframe containing some details about the specified cruise. The details include cruise official name, nickname, ship name, start/end time/location, etc ...

Parameters

cruiseName: string The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruises() method.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.cruise_by_name('KOK1606')

# alternatively, you may pass cruise nickname, if exists.
# below, "meso_scope" is a nickname, the official name is "KM1709"
# api.cruise_by_name('meso_scope')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCruiseByName 'KOK1606'
```

1.2.20 Cruise Spatio-Temporal Bounds

cruise_bounds(*cruiseName*)

Returns a dataframe containing the spatio-temporal bounding box accosiated with the specified cruise. Effectively, this method returns a subset of the outputs returned by the cruise_by_name method.

Parameters **cruiseName: string** The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruise method.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.cruise_bounds('KOK1606')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCruiseByName 'KOK1606'
```

1.2.21 Cruise Trajectory

cruise_trajectory(*cruiseName*)

Returns a dataframe containing the trajectory of the specified cruise.

Parameters

cruiseName: string The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruise method.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.cruise_trajectory('KM1513')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCruiseTrajectoryByName 'Cruise Official Name'
```

Example:

```
EXEC uspCruiseTrajectoryByName 'KM1513'
```

1.2.22 Cruise Variables

cruise_variables(*cruiseName*)

Returns a dataframe containing all registered variables (at Simons CMAP) during the specified cruise.

Parameters

cruiseName: string The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruise method.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.cruise_variables('SCOPE_Falkor1')
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspCruiseVariablesByName 'Cruise Official Name'
```

Example:

```
EXEC uspCruiseVariablesByName 'FK180310-1'
```

1.2.23 Retrieve Dataset

get_dataset (*tableName*)

Returns the entire dataset. It is not recommended to retrieve datasets with more than 1 million rows using this method. For large datasets, please use the *Data Subset: Generic Space-Time Cut* method and retrieve the data in smaller chunks. Note that this method does not return the dataset metadata. Use the *Metadata* method to get the dataset metadata.

Parameters

tableName: string The name of the table associated with the dataset. A full list of table names can be found in the Catalog or *Datasets* method.

Returns Pandas dataframe.

Example

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.get_dataset('tblMGL1704_Gradients2_IFCB_Abundance')
```

1.2.24 Data Subset: Generic Space-Time Cut

space_time(*table*, *variable*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*)

Returns a subset of data according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). The results are ordered by time, lat, lon, and depth (if exists), respectively.

Parameters

table: string Table name (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: string Variable short name which directly corresponds to a field name in the table. A subset of this variable is returned by this method according to the spatio-temporal cut parameters (below). Pass * wild card to retrieve all fields in a table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitue ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

Returns Pandas dataframe.

Example 1:

This example retrieves a subset of in-situ salinity measurements by Argo floats.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap
```

(continues on next page)

(continued from previous page)

```
api = pycmap.API(token='<YOUR_API_KEY>')
api.space_time(
              table='tblArgoMerge_REP',
              variable='argo_merge_salinity_adj',
              dt1='2015-05-01',
              dt2='2015-05-30',
              lat1=28,
              lat2=38,
              lon1=-71,
              lon2=-50,
              depth1=0,
              depth2=100
              )
```

Example 2:

Query all variables within the global Mesoscale Eddy dataset (see the wild card at line 10). The last few lines of code (lines 21-27) makes a couple of simple visualizations showing the eddy radius distribution.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import pycmap

api = pycmap.API(token='YOUR_API_KEY>')
df = api.space_time(
              table='tblMesoscale_Eddy',
              variable='*',
              dt1='2018-01-01',
              dt2='2018-01-01',
              lat1=-90,
              lat2=90,
              lon1=-180,
              lon2=180,
              depth1=0,
              depth2=0
              )

fig, axes = plt.subplots(nrows=1, ncols=2)
ax1 = df['eddy_radius'].plot.hist(ax=axes[0], bins=50)
_ = ax1.set_xlabel('Eddy Radius (km)')
ax2 = df.plot(kind='scatter', x='lat', y='eddy_radius', ax=axes[1], alpha=0.3)
ax2.yaxis.tick_right()
ax2.yaxis.set_label_position('right')
_ = ax2.set_ylabel('Eddy Radius (km)')
```

Example 3:

This example retrieves a subset of sea surface temperature measured by satellite. Notice, depth1 and depth2 values are automatically ignored because this is a surface dataset. A simple plot is made to visualize the retrieved data.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

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

(continues on next page)

(continued from previous page)

```
import pycmap



def plot(df):
    lat = df.lat.unique()
    lon = df.lon.unique()
    shape = (len(lat), len(lon))
    data = df.sst.values.reshape(shape)
    plt.imshow(data, extent=[np.min(lon), np.max(lon), np.min(lat), np.max(lat)],␣
↪cmap='coolwarm', origin='bottom', vmin=0, vmax=30)
    plt.title('Sea Surface Temperature')
    plt.colorbar()
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.show()


api = pycmap.API(token='<YOUR_API_KEY>')
df = api.space_time(
                    table='tblsst_AVHRR_OI_NRT',
                    variable='sst',
                    dt1='2016-04-30',
                    dt2='2016-04-30',
                    lat1=10,
                    lat2=70,
                    lon1=-180,
                    lon2=-80,
                    depth1=0,
                    depth2=0
                    )
plot(df)
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspSpaceTime 'tableName', 'variable', 'dt1', 'dt2', 'lat1', 'lat2', 'lon1', 'lon2
↪', 'depth1', 'depth2'
```

Example:

```
EXEC uspSpaceTime 'tblsst_AVHRR_OI_NRT', 'sst', '2016-04-30', '2016-04-30', '10', '70
↪', '-180', '80', '0', '0'
```

1.2.25 Data Subset: Time Series

time_series(*table*, *variable*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *interval=None*)

Returns a subset of data according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). The returned data subset is aggregated by time: at each time interval, the mean and standard deviation of the variable values within the space-time constraints are computed. The sequence of these values construct the timeseries. The timeseries data can be binned weekly, monthly, quarterly, or annually, if the interval parameter is set (this feature is not applicable to climatological datasets). The resulted timeseries is returned in the form of a Pandas dataframe ordered by time.

Parameters

table: string Table name (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: string Variable short name which directly corresponds to a field name in the table. A subset of this variable is returned by this method according to the spatio-temporal cut parameters (below). A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

interval: None or string, default: None The timeseries bin size. If None, the native dataset time resolution is used as the bin size. Below is a list of interval values for other binning options: - **'w'** or **'week'** for weekly timeseries. - **'m'** or **'month'** for monthly timeseries. - **'s'** or **'q'** for seasonal/quarterly timeseries. - **'a'** or **'y'** for annual/yearly timeseries.

Returns Pandas dataframe.

Example 1:

This example retrieves the timeseries of SiO4 measurements conducted by HOT team at University of Hawaii, spanning from 1988 to 2016.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap
```

(continues on next page)

(continued from previous page)

```
api = pycmap.API(token='<YOUR_API_KEY>')
api.time_series(
                table='tblHOT_Bottle',
                variable='SiO4_bottle_hot',
                dt1='1988-12-01',
                dt2='2016-10-15',
                lat1=22,
                lat2=23,
                lon1=-159,
                lon2=-157,
                depth1=0,
                depth2=0
                )
```

Example 2:

This example retrieves a 24-year long timeseries of absolute dynamic topography (closely related to sea surface height) measured by satellite. Note depth1 and depth2 values are automatically ignored because this is a surface dataset. The 'interval' parameter (line 24) has 'y' indicating yearly binning (inter-annual timeseres). This example takes a few moments to run as the altimetry dataset is very large (multi-decade daily-global remote sensing). The last few lines of code (lines 28-32) make a simple plot to visualize the retrieved data.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pycmap



api = pycmap.API(token='<YOUR_API_KEY>')
table, variable = 'tblAltimetry_REP', 'adt'
df = api.time_series(
                    table=table,
                    variable=variable,
                    dt1='1994-01-01',
                    dt2='2017-12-31',
                    lat1=30,
                    lat2=32,
                    lon1=-160,
                    lon2=-158,
                    depth1=0,
                    depth2=0,
                    interval='y'
                    )


plt.errorbar(df['year'], df['adt'], yerr=df['adt_std'], fmt='ob', capsize=3, alpha=0.
↪4)
plt.fill_between(df['year'], df['adt']-df['adt_std'], df['adt']+df['adt_std'], color=
↪'gray', alpha=0.2)
plt.xlabel('Year')
plt.ylabel(api.get_var_long_name(table, variable) + api.get_unit(table, variable))
plt.show()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspTimeSeries 'tableName', 'variable', 'dt1', 'dt2', 'lat1', 'lat2', 'lon1',
↪'lon2', 'depth1', 'depth2'
```

Examples (different intervals):

```
EXEC uspTimeSeries 'tblsst_AVHRR_OI_NRT', 'sst', '2016-01-01', '2016-12-31', '20', '25
↪', '-160', '-158', '0', '0'
```

```
EXEC uspWeekly 'tblsst_AVHRR_OI_NRT', 'sst', '2016-01-01', '2016-12-31', '20', '25',
↪'-160', '-158', '0', '0'
```

```
EXEC uspMonthly 'tblsst_AVHRR_OI_NRT', 'sst', '2016-01-01', '2016-12-31', '20', '25',
↪'-160', '-158', '0', '0'
```

```
EXEC uspQuarterly 'tblsst_AVHRR_OI_NRT', 'sst', '2016-01-01', '2016-12-31', '20', '25
↪', '-160', '-158', '0', '0'
```

```
EXEC uspAnnual 'tblsst_AVHRR_OI_NRT', 'sst', '2015-01-01', '2018-12-31', '20', '25',
↪'-160', '-158', '0', '0'
```

1.2.26 Data Subset: Depth Profile

depth_profile(*table*, *variable*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*)

Returns a subset of data according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). The returned data subset is aggregated by depth: at each depth level the mean and standard deviation of the variable values within the space-time constraints are computed. The sequence of these values construct the depth profile. The resulting depth profile is returned in the form of a Pandas dataframe ordered by depth.

Parameters

table: string Table name (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: string Variable short name which directly corresponds to a field name in the table. A subset of this variable is returned by this method according to the spatio-temporal cut parameters (below). A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

Returns Pandas dataframe.

Example 1:

This example retrieves a depth profile of in-situ chlorophyll concentration measurements by Argo Floats.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
table, variable = 'tblArgoMerge_REP', 'argo_merge_chl_adj'
df = api.depth_profile(
                       table=table,
                       variable=variable,
                       dt1='2016-04-30',
                       dt2='2016-04-30',
                       lat1=20,
                       lat2=24,
                       lon1=-170,
                       lon2=-150,
                       depth1=0,
                       depth2=1500
                       )
```

Example 2:

This example retrieves a depth profile of modeled chlorophyll concentration estimated by Pisces, a weekly 0.5° resolution BioGeoChemical model. The last few lines of code (lines 22-25) create a simple plot showing the chlorophyll depth profile. The deep chlorophyll maximum (DCM) is approximately at 100 m, closely matching the in-situ observations by ARGO Floats (see Example 1).

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import pycmap
```

(continues on next page)

(continued from previous page)

```
api = pycmap.API(token='<YOUR_API_KEY>')
table, variable = 'tblPisces_NRT', 'CHL'
df = api.depth_profile(
                       table=table,
                       variable=variable,
                       dt1='2016-04-30',
                       dt2='2016-04-30',
                       lat1=20,
                       lat2=24,
                       lon1=-170,
                       lon2=-150,
                       depth1=0,
                       depth2=1500
                       )

plt.plot(df['depth'], df[variable])
plt.xlabel('Depth [m]')
plt.ylabel(api.get_var_long_name(table, variable) + api.get_unit(table, variable))
plt.show()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspDepthProfile 'tableName', 'variable', 'dt1', 'dt2', 'lat1', 'lat2', 'lon1',
↪'lon2', 'depth1', 'depth2'
```

Example:

```
EXEC uspDepthProfile 'tblPisces_NRT', 'CHL', '2016-04-30', '2016-04-30', '20', '24',
↪'-170', '-150', '0', '1500'
```

1.2.27 Match (colocalize) Datasets

match(*sourceTable*, *sourceVar*, *targetTables*, *targetVars*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *temporalTolerance*, *latTolerance*, *lonTolerance*, *depthTolerance*)

Colocalizes the source variable (from source table) with the target variable (from target table). The matching results rely on the tolerance parameters because they set the matching boundaries between the source and target datasets. Notice the source has to be a single non-climatological variable. You may pass empty string ('') as source variable if you only want to get the time and location info from the source table. Please note that the number of matching entries between each target variable and the source variable might vary depending on the temporal and spatial resolutions of the target variable. In principle, if the source dataset is fully covered by the target variable's spatio-temporal range, there should always be matching results if the tolerance parameters are larger than half of their corresponding spatial/temporal resolutions. Please explore the Catalog to find appropriate target variables.

This method returns a dataframe containing the source variable joined with the target variable(s).

Note: Currently, the 'match' method is not optimized for matching very large subsets of massive datasets such as models and satellites. It would be best to use this method to colocalize in-situ measurements such as station-based or underway cruise datasets (which are typically 'small') with any other datasets (models, satellites, or other observations).

Stay tuned!

Parameters

sourceTable: string Table name of the source dataset. A full list of table names can be found in the Catalog.

sourceVar: string The source variable short name. The target variables are matched (colocalized) with this variable. A full list of variable short names can be found in the Catalog.

targetTables: list of string Table names of the target datasets to be matched with the source data. Note source dataset can be matched with multiple target datasets. A full list of table names can be found in the Catalog.

targetVars: list of string Variable short names to be matched with the source variable. A full list of variable short names can be found in the Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

temporalTolerance: list of int Temporal tolerance values between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single integer value is given, that would be applied to all target datasets. This parameter is in days except when the target variable represents monthly climatology data in which case it is in months. Notice fractional values are not supported in the current version.

latTolerance: list of float or int Spatial tolerance values in meridional direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target

datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

lonTolerance: list of float or int Spatial tolerance values in zonal direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

depthTolerance: list of float or int Spatial tolerance values in vertical direction [m] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets.

Returns Pandas dataframe.

1.2.27.1 Example 1:

In this example, the abundance of a prochlorococcus strain (MIT9313PCR, see lines 5-6) measured by Chisholm lab during the AMT13 cruise (Atlantic Meridional Transect Cruise 13) is colocalized with 3 target variables (lines 7-8):

- 'MIT9312PCR_Chisholm' from the same source dataset
- 'phosphate*WOA*clim' from World Ocean Atlas monthly climatology dataset
- 'chl' (chlorophyll) from weekly averaged satellite dataset

Tip1:

The space-time cut parameters (lines 9-16) have been set in such a way to encompass the entire source dataset 'tblAMT13_Chisholm' (see the dataset page for more details). Note that the last data point at the source dataset has been measured at '2003-10-12 12:44:00'. For simplicity dt2 has been set to '2003-10-13', but you could also use the exact date-time '2003-10-12 12:44:00'.

Tip2:

AMT13 cruise trajectory is already in Simons CMAP database. Therefore, another way to find reasonable values for the space-time cut parameters (lines 9-16) is to use the outputs of the following command: api.cruise_bounds('AMT13').

Tip3:

The temporalTolerance parameter is set to [0, 0, 1] (line 17). This means:

- $\pm$0 day temporal tolerance when matching with 'MIT9312PCR_Chisholm' (exact date-time matching)
- $\pm$0 month temporal tolerance when matching with 'phosphate*WOA*clim' (this is a monthly climatology dataset)
- $\pm$4 day temporal tolerance when matching with 'chl' (this is a weekly averaged dataset)

Tip4:

The latTolerance and lonTolerance parameters are set to [0, 0.5, 0.25] (lines 18-19). This means:

- ±0 day temporal tolerance when matching with 'MIT9312PCR_Chisholm' (exact date-time matching)
- ±0 month temporal tolerance when matching with 'phosphate*WOA*clim' (this is a monthly climatology dataset)
- ±4 day temporal tolerance when matching with 'chl' (this is a weekly averaged dataset)

Tip5:

The depthTolerance parameter is set to [0, 5, 0] (line 20). This means:

- ±0 meters vertical tolerances when matching with 'MIT9312PCR_Chisholm' (exact depth matching).
- ±5 meters vertical tolerances when matching with 'phosphate*WOA*clim' (note that this dataset, similar to model outputs, does not have uniform depth levels).

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.match(
         sourceTable='tblAMT13_Chisholm',
         sourceVar='MIT9313PCR_Chisholm',
         targetTables=['tblAMT13_Chisholm', 'tblWOA_Climatology', 'tblChl_REP'],
         targetVars=['MIT9312PCR_Chisholm', 'phosphate_WOA_clim', 'chl'],
         dt1='2003-09-14',
         dt2='2003-10-13',
         lat1=-48,
         lat2=48,
         lon1=-52,
         lon2=-11,
         depth1=0,
         depth2=240,
         temporalTolerance=[0, 0, 1],
         latTolerance=[0, 0.5, 0.25],
         lonTolerance=[0, 0.5, 0.25],
         depthTolerance=[0, 5, 0]
         )
```

1.2.27.2 Example 2:

The source variable in this example is particulate pseudo cobalamin ('Me_PseudoCobalamin_Particulate_pM' see lines 5-6) measured by Ingalls lab during the KM1315 cruise. This variable is colocalized with one target variabele, 'picoprokaryote' concentration, from Darwin model (lines 7-8). The colocalized data, then is visualized. please review the above Example 1, since the mentioned tips apply to this example too.

Tip1:

The employed Darwin model outputs are a 3-day averaged dataset, and therefore ±2 day temporal tolerance is used (line 17).

Tip2:

The employed Darwin model outputs have has a 0.5 degree spatial resolution in zonal and meridional directions, and so ±0.25 degree spatial tolerance is used (line 18-19).

Tip3:

Darwin model first depth level is at 5 m (not 0), and so ±5 meter vertical tolerance should cover all surface measurements (line 20).

```
# !pip install pycmap -q     # uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
df = api.match(
               sourceTable='tblKM1314_Cobalmins',
               sourceVar='Me_PseudoCobalamin_Particulate_pM',
               targetTables=['tblDarwin_Phytoplankton'],
               targetVars=['picoprokaryote'],
               dt1='2013-08-11',
               dt2='2013-09-05',
               lat1=22.5,
               lat2=50,
               lon1=-159,
               lon2=-128,
               depth1=0,
               depth2=300,
               temporalTolerance=[2],
               latTolerance=[0.25],
               lonTolerance=[0.25],
               depthTolerance=[5]
               )


plt.plot(df['picoprokaryote'], df['Me_PseudoCobalamin_Particulate_pM'], '.')
plt.xlabel('picoprokaryote' + api.get_unit('tblDarwin_Phytoplankton', 'picoprokaryote
↪'))
plt.ylabel('Me_PseudoCobalamin_Particulate_pM' + api.get_unit('tblKM1314_Cobalmins',
↪'Me_PseudoCobalamin_Particulate_pM'))
plt.show()
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspMatch
 'sourceTable',
 'sourceVariable',
 'targetTable',
 'targetVariable',
 'dt1',
 'dt2',
 'lat1',
```

(continues on next page)

(continued from previous page)

```
 'lat2',
 'lon1',
 'lon2',
 'depth1',
 'depth2',
 'timeTolerance',
 'latTolerance',
 'lonTolerance',
 'depthTolerance'
```

Example:

```
EXEC uspMatch
 'tblKM1314_Cobalmins',
 'Me_PseudoCobalamin_Particulate_pM',
 'tblDarwin_Phytoplankton',
 'picoprokaryote',
 '2013-08-09 00:00:00',
 '2013-09-07 00:00:00',
 '22.25',
 '50.25',
 '-159.25',
 '-127.75',
 '-5',
 '305',
 '2',
 '0.25',
 '0.25',
 '5'
```

1.2.28 Match (colocalize) Cruise Track with Datasets

along_track(*cruise*, *targetTables*, *targetVars*, *depth1*, *depth2*, *temporalTolerance*, *latTolerance*, *lonTolerance*, *depthTolerance*)

This method colocalizes a cruise trajectory with the specified target variables. The matching results rely on the tolerance parameters because these parameters set the matching boundaries between the cruise trajectory and target datasets. Please note that the number of matching entries for each target variable might vary depending on the temporal and spatial resolutions of the target variable. In principle, if the cruise trajectory is fully covered by the target variable's spatio-temporal range, there should always be matching results if the tolerance parameters are larger than half of their corresponding spatial/temporal resolutions. Please explore the Catalog to find appropriate target variables to colocalize with the desired cruise.

This method returns a dataframe containing the cruise trajectory joined with the target variable(s).

Parameters

cruise: string The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using Cruises() method.

targetTables: list of string Table names of the target datasets to be matched with the source data. Notice source dataset can be matched with multiple target datasets. A full list of table names can be found in the Catalog.

targetVars: list of string Variable short names to be matched with the source variable. A full list of variable short names can be found in the Catalog.

depth1: float Start depth [m]. This parameter sets the lower bound of the depth cut on the target datasets. 'depth1' and 'depth2' allow matching a cruise trajectory (which is at the surface, hopefully!) with target variables at lower depth. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the depth cut on the target datasets. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

temporalTolerance: list of int Temporal tolerance values between the cruise trajectory and target datasets. The size and order of values in this list should match those of targetTables. If only a single integer value is given, that would be applied to all target datasets. This parameter is in days except when the target variable represents monthly climatology data in which case it is in months. Notice fractional values are not supported in the current version.

latTolerance: list of float or int Spatial tolerance values in meridional direction [deg] between the cruise trajectory and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

lonTolerance: list of float or int Spatial tolerance values in zonal direction [deg] between the cruise trajectory and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

depthTolerance: list of float or int Spatial tolerance values in vertical direction [m] between the cruise trajectory and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets.

Returns Pandas dataframe.

1.2.28.1 Example 1:

This example demonstrates how to colocalize the "Diel" cruise (official name: KM1513) with 2 target variables (lines 8-9):

- 'synecho_abundance' from underway seaflow dataset
- 'NO3' from Pisces model

The last few lines of code (lines 21-24), plot the colocalized synecho_abundance versus NO3 concentration. |

Tip1:

The official name of this cruise is 'KM1513'. One can use the unofficial name, 'diel', instead (line 7).

Tip2:

A full list of cruise expeditions can be retrieved using the Cruises() method.

Tip3:

The temporalTolerance parameter is set to [0, 4] (line 12). This means:

- ±0 day temporal tolerance when matching with 'synecho_abundance' (exact date-time matching)
- ±4 day temporal tolerance when matching with 'NO3' (Pisces is a weekly averaged dataset)

Tip4:

The latTolerance and lonTolerance parameters are set to [0, 0.25] (lines 13-14). This means:

- ±0 degree spatial tolerances (in meridional and zonal directions) when matching with 'synecho_abundance' (exact lat/lon matching).
- ±0.25 degrees spatial tolerances (in meridional and zonal directions) when matching with 'NO3'. This dataset has 0.25 degree spatial resolution which means one may reduce the spatial tolerance for this target dataset down to 0.25/2 = 0.125 degrees.

Tip5:

The depthTolerance parameter is set to [5, 5] (line 20). This means: - ±5 meters vertical tolerances when matching with 'synecho_abundance' - ±5 meters vertical tolerances when matching with 'NO3'.

Note that the Pisces dataset has several depth levels between surface and 5 m. The spacing between the depth levels is not have uniform.

```
#!pip install pycmap -q     # uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
df = api.along_track(
                    cruise='diel',
                    targetTables=['tblSeaFlow', 'tblPisces_NRT'],
                    targetVars=['synecho_abundance', 'NO3'],
                    depth1=0,
                    depth2=5,
                    temporalTolerance=[0, 4],
                    latTolerance=[0, 0.25],
                    lonTolerance=[0, 0.25],
                    depthTolerance=[5, 5]
                    )


plt.plot(df['NO3'], df['synecho_abundance'], '.')
plt.ylabel('synecho_abundance' + api.get_unit('tblSeaFlow', 'synecho_abundance'))
plt.xlabel('NO3' + api.get_unit('tblPisces_NRT', 'NO3'))
plt.show()
```

1.2.28.2 Example 2:

Imagine you would like to colocalize a 'large' number of variables along the track of multiple cruises. Hard-coding the variable names, table names, and tolerance parameters (as shown in Example 1) is an error-prone process. This example show an alternative approach to implement multi-variable colocalization.

Here, we colocalize two open-ocean North-Pacific transect cruises ('KOK1606' [gradient1], 'MGL1704' [gradient2]) with 14 variables from satellite datasets, model outputs, underway cruise measurements, and World-Ocean-Atlas climatology dataset. A full list of variables can be retrieved using the get_catalog() command (COMMENT: Make get_catalog() a hyperlink). Also, please review the tips mentioned Example 1 since they are generally relevant to

this example. It takes a few minutes to run this script since we are colocalizing two long cruises with multiple target variables. Reduce the number of cruises (line 12), and/or number of target variables (lines 19-36) to save time.

As a simple show case, the colocalized synechococcus abundance is plotted against latitude and is compared with phosphaste concentration from World Ocean Atlas monthly climatology dataset (line 91). The full colocalized dataset is stored in a csv file on local machine.

Tip:

Once the colocalization is finished, you may add new "calculated" columns to the final dataframe: `df['NO3_divided_Fe'] = df['NO3'] / df['Fe']`

```
import pandas as pd
from collections import namedtuple
import pycmap



def open_ocean_cruises():
    return ['MGL1704', 'KOK1606']


def match_params():
    Param = namedtuple('Param', ['table', 'variable', 'temporalTolerance',
↪'latTolerance', 'lonTolerance', 'depthTolerance'])
    params = []
    ######## ship data (not calibrated)
    params.append(Param('tblCruise_Salinity', 'salinity', 0, 0.1, 0.1, 5))
    params.append(Param('tblCruise_Temperature', 'temperature', 0, 0.1, 0.1, 5))
    ######## underway seaflow
    params.append(Param('tblSeaFlow', 'prochloro_abundance', 0, 0.1, 0.1, 5))
    params.append(Param('tblSeaFlow', 'synecho_abundance', 0, 0.1, 0.1, 5))
    ######## satellite
    params.append(Param('tblCHL_REP', 'chl', 4, 0.25, 0.25, 5))
    params.append(Param('tblModis_AOD_REP', 'AOD', 15, 1, 1, 5))
    params.append(Param('tblAltimetry_REP', 'sla', 1, 0.25, 0.25, 5))
    ####### model
    params.append(Param('tblPisces_NRT', 'Fe', 4, 0.5, 0.5, 5))
    params.append(Param('tblPisces_NRT', 'NO3', 4, 0.5, 0.5, 5))
    params.append(Param('tblPisces_NRT', 'PO4', 4, 0.5, 0.5, 5))
    params.append(Param('tblDarwin_Nutrient_Climatology', 'NH4_darwin_clim', 0, 0.5,␣
↪0.5, 5))
    params.append(Param('tblDarwin_Nutrient_Climatology', 'SiO2_darwin_clim', 0, 0.5,␣
↪0.5, 5))
    ####### WOA
    params.append(Param('tblWOA_Climatology', 'density_WOA_clim', 0, .75, .75, 5))
    params.append(Param('tblWOA_Climatology', 'phosphate_WOA_clim', 0, 0.75, 0.75, 5))

    tables, variables, temporalTolerance, latTolerance, lonTolerance, depthTolerance␣
↪= [], [], [], [], [], []
    for i in range(len(params)):
        tables.append(params[i].table)
        variables.append(params[i].variable)
        temporalTolerance.append(params[i].temporalTolerance)
```

(continues on next page)

(continued from previous page)

```
        latTolerance.append(params[i].latTolerance)
        lonTolerance.append(params[i].lonTolerance)
        depthTolerance.append(params[i].depthTolerance)
    return tables, variables, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance



def plot(api, df):
    tbl1, tbl2 = 'tblSeaFlow', 'tblWOA_Climatology'
    var1, var2 = 'prochloro_abundance', 'phosphate_WOA_clim'
    fig, ax1 = plt.subplots()
    ax2 = ax1.twinx()
    ax1.plot(df['lat'], df[var1], 'g.', alpha=0.4)
    ax2.plot(df['lat'], df[var2], 'b.', alpha=0.4)
    ax1.set_xlabel('latitude [deg]')
    ax1.set_ylabel(var1 + api.get_unit(tbl1, var1), color='g')
    ax2.set_ylabel(var2 + api.get_unit(tbl2, var2), color='b')
    plt.show()
    return



def main():
    api = pycmap.API(token='<YOUR_API_KEY>')
    cruises = open_ocean_cruises()
    tables, variables, temporalTolerance, latTolerance, lonTolerance, depthTolerance␣
↪= match_params()
    df = pd.DataFrame({})
    for cruise in cruises:
        print('\n*******************************')
        print('Colocalizing %s cruise...' % cruise)
        print('*******************************\n')
        data = api.along_track(
                               cruise=cruise,
                               targetTables=tables,
                               targetVars=variables,
                               temporalTolerance=temporalTolerance,
                               latTolerance=latTolerance,
                               lonTolerance=lonTolerance,
                               depthTolerance=depthTolerance,
                               depth1=0,
                               depth2=5
                               )
        if len(df) < 1:
            df = data
        else:
            df = pd.concat([df, data], ignore_index=True)
        data.to_csv('%s.csv' % cruise, index=False)
    df.to_csv('sfMatch.csv', index=False)
    plot(api, df)
    return df



if __name__ == '__main__':
    df = main()
```

(continues on next page)

(continued from previous page)

```
# the results are stored in csv files at the current working address
# if you are running this script on colab, run the following command to list the␣
↪generated csv files:
#!ls
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspMatch
  'sourceTable',
  'sourceVariable',
  'targetTables',
  'targetVariables',
  'dt1',
  'dt2',
  'lat1',
  'lat2',
  'lon1',
  'lon2',
  'depth1',
  'depth2',
  'timeTolerance',
  'latTolerance',
  'lonTolerance',
  'depthTolerance'
```

Example:

```
EXEC uspMatch
  'tblKM1314_Cobalmins',
  'Me_PseudoCobalamin_Particulate_pM',
  'tblDarwin_Phytoplankton',
  'picoprokaryote',
  '2013-08-09 00:00:00',
  '2013-09-07 00:00:00',
  '22.25',
  '50.25',
  '-159.25',
  '-127.75',
  '-5',
  '305',
  '2',
  '0.25',
  '0.25',
  '5'
```

1.3 Data Visualization Methods

The main focus of pycmap (and the broader Simons CMAP project) is to provide the user with a unifying API to extract subsets of datasets with different dimensions and different spatio-temporal resolutions. This will drastically reduce the time-consuming data collection and preparation processes, especially considering the massive size of the archived multi-decadal global oceanic datasets generated by satellites or numerical models.

The simple and unified interface of the pycmap API enables the general public and scientists to dive into the vast, and often under underutilized, ocean datasets without being concerned about the underlying dataset file structure. Visualizing the retrieved data is often a good entry point to explore and study these datasets and pycmap offers a number of built-in methods for simple interactive data visualizations. The pycmap visualization engine can be set to either of these interactive data visualization libraries: plotly, or bokeh. Plotly has a larger variation of graphs while bokeh is faster to render the graph object.

The default visualization library is plotly; the code block below shows how to change the visualization engine (see API page for more details).

Note: Data visualization requires a valid API key which can be obtained from https://simonscmap.com/apikeymanagement. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Example

```
import pycmap
pycmap.API(vizEngine='bokeh')  # vizEngine options: 'plotly' ,'bokeh'
```

1.3.1 Data Visualization Methods

Histogram Plot	Histogram Plot
Time Series Plot	Time Series Plot
Regional Map, Contour Plot, 3D Surface Plot	Regional Map, Contour Plot, 3D Surface Plot
Section Map, Section Contour	Section Map, Section Contour
Depth Profile	Depth Profile
Cruise Track Plot	Cruise Track Plot
Correlation Matrix	Correlation Matrix
Correlation Matrix Along Cruise Track	Correlation Matrix Along Cruise Track
Climatology	Climatology

1.3.1.1 Histogram Plot

plot_hist(*tables*, *variables*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *exportDataFlag=False*, *show=True*)

Creates a histogram graph for each variable according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). Change the APIs vizEngine parameter if you wish to use a different visualization library. Returns the generated graph objects in form of a python list. One may use the returned objects to modify the graph properties.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

tables: list of string Table names (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: list of string Variable short name which directly corresponds to a field name in the table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

Returns

A list of graph objects. Below are the graph's properties and methods.

Properties

data: dataframe Graph data points to be visualized.

bins: int, default: 50 Number of histogram bins.

pdf: boolean, default: False If True the histogram shows a probability density function, otherwise absolute counts are displayed.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

xlabel: str Graph's x-axis label.

ylabel: str Graphs's y-axis label.

title: str Graphs's title.

Methods

render() Displays the plot according to the set properties.

Example 1:

This example creates three histogram graphs comparing dissolved oxygen measured during the Falkor_2018 cruise, estimated by Darwin climatology model, and World Ocean Atlas. The graphs are made using the default visualization library (plotly) which may be changed by: `pycmap.API(vizEngine='bokeh')`

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine, previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')


from pycmap.viz import plot_hist

go = plot_hist(
              tables=['tblFalkor_2018', 'tblDarwin_Nutrient_Climatology', 'tblWOA_Climatology'],
              variables=['CTD_Oxygen', 'O2_darwin_clim', 'oxygen_WOA_clim'],
              dt1='2018-03-01',
              dt2='2018-04-30',
              lat1=21,
              lat2=25,
              lon1=-161,
              lon2=155,
              depth1=0,
              depth2=100,
              exportDataFlag=False,
              show=True
              )
```

```
# here is how to modify a graph:

go[0].bins = 20
go[0].pdf = False
go[0].height = 600
go[0].width = 600
go[0].xlabel = "new xlabel"
go[0].title= "graph's title"
go[0].render()
```

1.3.1.2 Time Series Plot

plot_timeseries(*tables*, *variables*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *exportDataFlag=False*, *show=True*, *interval=None*)

Creates a timeseries graph for each variable according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). By definition, timeseries data points are aggregated by time: at each time interval the mean and standard deviation of the variable values within the space-time constraints are computed. The sequence of these values construct the timeseries. If the **interval** parameter is set, timeseries can be binned weekly, monthly, quarterly, or annually, (this feature is not applicable to climatological datasets).

Change the **'APIs vizEngine'_** parameter if you wish to use a different visualization library. Returns the generated graph objects in form of a python list. One may use the returned objects to modify the graph properties.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

tables: list of string Table names (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: list of string Variable short name which directly corresponds to a field name in the table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note latitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note latitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

interval: None or string, default: None The timeseries bin size. If None, the native dataset time resolution is used as the bin size. Below is a list of interval values for other binning options:

- **'w'** or **'week'** for weekly timeseries.
- **'m'** or **'month'** for monthly timeseries.
- **'s'** or **'q'** for seasonal/quarterly timeseries.
- **'a'** or **'y'** for annual/yearly timeseries.

Returns

list of graph objects A list of graph objects. Below are the graph's properties and methods.

Properties

data: dataframe Graph data points to be visualized.

line: boolean, default: True If True, line plot is superimposed on markers.

color: string Line and marker colors (e.g. 'red', 'green', ..) or rgb hex '#FF0023'.

msize: int Marker size.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

xlabel: str The graphs's x-axis label.

ylabel: str The graphs's y-axis label.

title: str The graphs's title.

Methods

render() Displays the plot according to the set properties.

Example:

This example generates two timeseries graphs showing remotly sensed sea level anomaly, and sea surface salinity over a weekly-binned one-year period. The graphs are made using the default visualization library (plotly) which may be changed by: `pycmap.API(vizEngine='bokeh')`

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')


from pycmap.viz import plot_timeseries

go = plot_timeseries(
                    tables=['tblAltimetry_REP', 'tblSSS_NRT'],
                    variables=['sla', 'sss'],
                    dt1='2016-04-30',
                    dt2='2017-04-30',
                    lat1=30,
                    lat2=32,
                    lon1=-160,
                    lon2=-158,
```

(continues on next page)

(continued from previous page)

```
                    depth1=0,
                    depth2=0,
                    exportDataFlag=False,
                    show=True,
                    interval='w'
                    )
```

```
# here is how to modify a graph:

go[0].pdf = False
go[0].bins = 20
go[0].xlabel = "new xlabel"
go[0].title= "graph's title"
go[0].width = 600
go[0].height = 600
go[0].render()
```

1.3.1.3 Regional Map, Contour Plot, 3D Surface Plot

plot_map(*tables*, *variables*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *exportDataFlag=False*, *show=True*, *levels=0*, *surface3D=False*)

Creates map graphs for each variable according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). If the specified space-time domain involves multiple dates and/or depth levels, individual maps are made per date and depth level. To create contour plots, set the contour **levels** parameter to a positive integer number. Also, setting the **surface3D** parameter to True will generate maps in 3D mode. Note that contour and 3D sufrace maps are only supported by plotly visualization library. In the case of sparse dataset, the retrieved data is superimposed on a geospatial map.

Change the APIs vizEngine parameter if you wish to use a different visualization library. Returns the generated graph objects in form of a python list. One may use the returned objects to modify the graph properties.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

tables: list of string Table names (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: list of string Variable short name which directly corresponds to a field name in the table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note latitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note latitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

levels: int, default: 0 Number of contour levels. If 0, regional maps are generated (no contour lines). Currently, contour plots are only supported by plotly visualization library.

surface3D: boolean, default: False If True, maps are rendered in 3D mode. Currently, 3D map plots are only supported by plotly visualization library.

Returns

list of graph objects A list of graph objects. Below are the graph's properties and methods.

Properties

data: dataframe Graph data points to be visualized.

level: int, default: 0 Number of contour levels. Only applicable to plotly.

surface3D: boolean, default: False If True, maps are rendered in 3D mode. Only applicable to plotly.

cmap: str or cmocean colormap Colormap name. Any matplotlib (e.g. 'viridis', ..) or cmocean (e.g. cmocean.cm.thermal, ..) colormaps can be passed to this property. A full list of matplotlib and cmocean color palettes can be found at the following links: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html https://matplotlib.org/cmocean/

vmin: float This parameter defines the lower bound of the colorbar.

vmax: float This parameter defines the upper bound of the colorbar.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

xlabel: str The graphs's x-axis label.

ylabel: str The graphs's y-axis label.

title: str The graphs's title.

Methods

render() Displays the plot according to the set properties.

Example 1: Gridded Maps

This example makes two regional maps showing the phosphate climatology and dissolved iron, respectively. The graphs are made using the default visualization library (plotly) which may be changed by: `pycmap.API(vizEngine='bokeh')`

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')

from pycmap.viz import plot_map

go = plot_map(
             tables=['tblWOA_Climatology', 'tblPisces_NRT'],
             variables=['phosphate_WOA_clim', 'Fe'],
             dt1='2016-04-30',
             dt2='2016-04-30',
             lat1=10,
             lat2=70,
             lon1=-180,
             lon2=-80,
             depth1=0,
             depth2=0.5,
             exportDataFlag=False,
             show=True
             )
```

```
# here is how to modify a graph:

go[1].cmap = 'PRGn'
go[1].vmin = 0
go[1].vmax = 5e-5
go[1].width = 900
go[1].height = 700
go[1].render()
```

Example 2: Sparse Maps

This example visualizes an example of sparse data: synechococcus abundance from Global Pikophytoplankton dataset.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')

from pycmap.viz import plot_map

plot_map(
        tables=['tblGlobal_PicoPhytoPlankton'],
        variables=['synechococcus_abundance'],
        dt1='1990-01-30',
        dt2='1995-12-30',
```

(continues on next page)

(continued from previous page)

```
        lat1=10,
        lat2=70,
        lon1=-180,
        lon2=80,
        depth1=0,
        depth2=100,
        exportDataFlag=False,
        show=True
        )
```

Example 3: Contour Plot

This example creates a contour plot using the satellite Sea Surface Temperature (SST). Notice the **levels** parameter sets the number of contour levels. Currently, contour plots are only supported by the plotly library.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')

from pycmap.viz import plot_map

go = plot_map(
             tables=['tblsst_AVHRR_OI_NRT'],
             variables=['sst'],
             dt1='2016-04-30',
             dt2='2016-04-30',
             lat1=10,
             lat2=70,
             lon1=-180,
             lon2=-80,
             depth1=0,
             depth2=0,
             exportDataFlag=False,
             show=True,
             level
```

Example 4: 3D Surface

This example creates a 3D map using model estimates of dissolved nitrate (NO3). Notice the **surface3D** parameter is set to True. Currently, 3D map plots are only supported by the plotly library.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')
```

(continues on next page)

(continued from previous page)

```
from pycmap.viz import plot_map

go = plot_map(
              tables=['tblPisces_NRT'],
              variables=['NO3'],
              dt1='2016-04-30',
              dt2='2016-04-30',
              lat1=-90,
              lat2=90,
              lon1=-180,
              lon2=180,
              depth1=0,
              depth2=0.5,
              exportDataFlag=False,
              show=True,
              surface3D=True
              )
```

1.3.1.4 Section Map, Section Contour

plot_section(*tables*, *variables*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *exportDataFlag=False*, *show=True*, *levels=0*)

Creates section maps for each variable according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). If the specified space-time domain involves multiple dates, individual maps are made per date. Prior to creating the section graph, the retrieved data is averaged along the meridional direction if longitude range is larger than latitude range ((lon2-lon1) > (lat2-lat1)); otherwise data is averaged along the zonal axis. To create contour plots, set the contour **levels** parameter to a positive integer number. Note that contour plot is only supported by plotly visualization library. Also, `plot_section()` function only applies to gridded datasets with depth component (e.g. model outputs) and does not apply to sparse datasets.

Change the APIs vizEngine parameter if you wish to use a different visualization library. Returns the generated graph objects in form of a python list. One may use the returned objects to modify the graph properties.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

tables: list of string Table names (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: list of string Variable short name which directly corresponds to a field name in the table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut.

Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

levels: int, default: 0 Number of contour levels. If 0, regional maps are generated (no contour lines). Currently, contour plots are only supported by plotly visualization library.

Returns

list of graph objects

A list of graph objects. Below are the graph's properties and methods.

Properties

data: dataframe Graph data points to be visualized.

level: int, default: 0 Number of contour levels. Only applicable to plotly.

cmap: str or cmocean colormap Colormap name. Any matplotlib (e.g. 'viridis', ..) or cmocean (e.g. cmocean.cm.thermal, ..) colormaps can be passed to this property. A full list of matplotlib and cmocean color palettes can be found at the following links: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

https://matplotlib.org/cmocean/

vmin: float This parameter defines the lower bound of the colorbar.

vmax: float This parameter defines the upper bound of the colorbar.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

xlabel: str Graph's x-axis label.

ylabel: str Graph's y-axis label.

title: str Graphs's title.

Methods

render() Displays the plot according to the set properties.

Example 1: Section Map

This example makes a meridional section map showing the dissolved nitrate. The retrieved data is averaged along the zonal direction because the selected region is elongated along the meridional direction: (lat2-lat1) > (lon2-lon1). The graphs are made using the default visualization library (plotly) which may be changed by: `pycmap.API(vizEngine='bokeh')`

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')

from pycmap.viz import plot_section

go = plot_section(
                  tables=['tblPisces_NRT'],
                  variables=['NO3'],
                  dt1='2016-04-30',
                  dt2='2016-04-30',
                  lat1=10,
                  lat2=60,
                  lon1=-160,
                  lon2=-158,
                  depth1=0,
                  depth2=5000,
                  exportDataFlag=False,
                  show=True
                  )
```

```
# here is how to modify a graph:

import cmocean

go[0].cmap = cmocean.cm.balance
go[0].vmin = 0
go[0].vmax = 60
go[0].width = 700
go[0].height = 800
go[0].render()
```

Example 2: Section Contour

This example makes a cross-basins section map showing estimates of SIO2 concentration calculated by Darwin model.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')

from pycmap.viz import plot_section
```

(continues on next page)

(continued from previous page)

```
plot_section(
            tables=['tblDarwin_Nutrient'],
            variables=['SIO2'],
            dt1='2008-01-05',
            dt2='2008-01-05',
            lat1=-50,
            lat2=-46,
            lon1=-180,
            lon2=180,
            depth1=0,
            depth2=2000,
            exportDataFlag=False,
            show=True,
            levels=10
            )
```

1.3.1.5 Depth Profile

plot_depth_profile(*tables*, *variables*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *exportDataFlag=False*, *show=True*)

Creates a depth profile graph for each variable according to the specified space-time constraints (dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2). By definition, depth profile data points are aggregated by depth: at each depth level the mean and standard deviation of the variable values within the space-time constraints are computed. The sequence of these values construct the depth profile.

Change the APIs vizEngine parameter if you wish to use a different visualization library. Returns the generated graph objects in form of a python list. One may use the returned objects to modify the graph properties.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

tables: list of string Table names (each dataset is stored in a table). A full list of table names can be found in Catalog.

variable: list of string Variable short name which directly corresponds to a field name in the table. A full list of variable short names can be found in Catalog.

dt1: string Start date or datetime. This parameter sets the lower bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. This parameter sets the upper bound of the temporal cut. Example values: '2016-05-25' or '2017-12-10 17:25:00'.

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at the surface and increases towards the ocean floor).

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

Returns

list of graph objects

A list of graph objects. Below are the graph's properties and methods.

Properties

data: dataframe Graph data points to be visualized.

line: boolean, default: True If True, line plot is superimposed on markers.

color: string Line and marker colors (e.g. 'red', 'green', ..) or rgb hex '#FF0023'.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

xlabel: str Graphs's x-axis label.

ylabel: str Graphs's y-axis label.

title: str Graphs's title.

Methods

render() Displays the plot according to the set properties.

Example:

This example compares the depth profile of chlorophyll concentration retrieved from Argo Floats observations, Pisces model estimations, and Darwin model calculations. The depth profiles from the two models demonstrate close consistency with the Argo measurements. The graphs are made using the default visualization library (plotly) which may be changed by: `pycmap.API(vizEngine='bokeh')`

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>')


from pycmap.viz import plot_depth_profile

go = plot_depth_profile(
                        tables=['tblArgoMerge_REP', 'tblPisces_NRT', 'tblDarwin_
↪Ecosystem'],
                        variables=['argo_merge_chl_adj', 'CHL', 'CHL'],
                        dt1='2014-04-25',
                        dt2='2014-04-30',
                        lat1=20,
                        lat2=24,
                        lon1=-170,
                        lon2=-150,
                        depth1=0,
                        depth2=1500,
                        exportDataFlag=False,
                        show=True
                        )
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspDepthProfile 'tableName', 'variable', 'dt1', 'dt2', 'lat1', 'lat2', 'lon1',
↪'lon2', 'depth1', 'depth2'
```

Example:

```
EXEC uspDepthProfile 'tblPisces_NRT', 'CHL', '2016-04-30', '2016-04-30', '20', '24',
↪'-170', '-150', '0', '1500'
```

1.3.1.6 Cruise Track Plot

plot_cruise_track(*cruise*)

Plots cruise trajectory(s) on a geospatial map.

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

cruiseName: list of string A list of official cruise names. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruises() method.

Returns This method has no returns.

Example:

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='<YOUR_API_KEY>')


from pycmap.viz import plot_cruise_track
plot_cruise_track(['KM1712', 'gradients_1'])
```

1.3.1.7 Correlation Matrix

plot_corr_map(*sourceTable*, *sourceVar*, *targetTables*, *targetVars*, *dt1*, *dt2*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*, *temporalTolerance*, *latTolerance*, *lonTolerance*, *depthTolerance*, *method='spearman'*, *exportDataFlag=False*, *show=True*)

This function computes and plots the pair-correlation coefficient between the source and target variables. The results are visualized in form of a correlation matrix. To compute the correlations, the source and target variables have to be colocalized first (see *Match (colocalize) Datasets*). The colocalization procedure relies on the tolerance parameters because they set the matching boundaries between the source and target datasets. Note the source has to be a single non-climatological variable. In principle, if the source dataset is fully covered by the target variable's spatio-temporal range, there should always be matching results if the tolerance parameters are larger than half of their corresponding spatial/temporal resolutions. Please explore the Catalog to find appropriate target variables. Currently, this visualization is only supported by plotly visualization library.

Returns the generated correlation graph object. One may modify the graph properties (see example below).

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

Parameters

sourceTable: string Table name of the source dataset. A full list of table names can be found in Catalog.

sourceVar: string The source variable short name. The target variables are matched (colocalized) with this variable. A full list of variable short names can be found in Catalog.

targetTables: list of string Table names of the target datasets to be matched with the source data. Note source dataset can be matched with multiple target datasets. A full list of table names can be found in Catalog.

dt1: string Start date or datetime. Both source and target datasets are filtered before matching. This parameter sets the lower bound of the temporal cut.

Example values: '2016-05-25' or '2017-12-10 17:25:00'.

dt2: string End date or datetime. Both source and target datasets are filtered before matching. This parameter sets the upper bound of the temporal cut.

lat1: float Start latitude [degree N]. Both source and target datasets are filtered before matching. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90 to 90 degrees.

lat2: float End latitude [degree N]. Both source and target datasets are filtered before matching. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90 to 90 degrees.

lon1: float Start longitude [degree E]. Both source and target datasets are filtered before matching. This parameter sets the lower bound of the zonal cut. Note longitude ranges from -180 to 180 degrees.

lon2: float End longitude [degree E]. Both source and target datasets are filtered before matching. This parameter sets the upper bound of the zonal cut. Note longitude ranges from -180 to 180 degrees.

depth1: float Start depth [m]. Both source and target datasets are filtered before matching. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. Both source and target datasets are filtered before matching. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

temporalTolerance: list of int Temporal tolerance values between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single integer value is given, that would be applied to all target datasets. This parameter is in day units except when the target variable represents monthly climatology data in which case it is in month units. Note fractional values are not supported in the current version.

latTolerance: list of float or int Spatial tolerance values in meridional direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

lonTolerance: list of float or int Spatial tolerance values in zonal direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets.

A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

depthTolerance: list of float or int Spatial tolerance values in vertical direction [m] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets.

method: str, default: 'spearman' Correlation algorithm. 'spearman' is a rank correlation algorithm and is a metric for monotonic relationships. Other options involve **'pearson'** and **'kendall'**. *'pearson'* is the standard correlation coefficient, more favorable for linear correlations. *'kendall'* evaluates Kendall Tau correlation coefficient.

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

Returns

the graph object

Below are the graph's properties and methods.

Properties

x: list of string Correlation matrix column titles (covariate names).

y: list of string Correlation matrix row titles (covariate names).

z: numpy.ndarray Computed pairwise correlation coefficients.

cmap: str or cmocean colormap Colormap name. Any matplotlib (e.g. 'viridis', ..) or cmocean (e.g. cmocean.cm.thermal, ..) colormaps can be passed to this property. A full list of matplotlib and cmocean color palettes can be found at the following links: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

https://matplotlib.org/cmocean/

vmin: float This parameter defines the lower bound of the colorbar.

vmax: float This parameter defines the upper bound of the colorbar.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

title: str The graphs's title.

Methods

render() Displays the plot according to the set properties.

Example

In this example the abundance of a prochlorococcus strain (MIT9313PCR, see lines 37-38) measured by Chisholm lab during the AMT13 cruise (Atlantic Meridional Transect Cruise 13) is colocalized with 7 target variables (lines 7-8):

- 'MIT9312PCR*Chisholm', 'MED4PCR*Chisholm', and 'sbact_Chisholm' from the same source dataset
- 'phosphate*WOA*clim', and 'nitrate*WOA*clim' from World Ocean Atlas monthly climatology dataset
- 'chl' from weekly averaged satellite chlorophyll dataset
- 'picoprokaryote' from 3-day averaged Darwin model. Colocalizing this variable will take longer time than others as the 3-day averaged Darwin dataset is massive (multi-decadal global 3D dataset)!

Tip: The space-time cut parameters (lines 41-48) have been set in such a way to encompass the entire source dataset 'tblAMT13_Chisholm' (see the dataset page for more details). Notice that the last data point at the source dataset has been measured at '2003-10-12 12:44:00'. For simplicity dt2 has been set to '2003-10-13', but you could also use the exact date-time '2003-10-12 12:44:00'.

Please review the **Example 1** at Match (colocalize) Datasets page since all of the mentioned tips directly apply to this example too.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>', vizEngine='plotly')

from collections import namedtuple
from pycmap.viz import plot_corr_map



def match_params():
    Param = namedtuple('Param', ['table', 'variable', 'temporalTolerance',
↪'latTolerance', 'lonTolerance', 'depthTolerance'])
    params = []
    ######## self-matching: colocalizing with some other variables in the tblAMT13_
↪Chisholm dataset
    params.append(Param('tblAMT13_Chisholm', 'MIT9312PCR_Chisholm', 0, 0, 0, 0))
    params.append(Param('tblAMT13_Chisholm', 'MED4PCR_Chisholm', 0, 0, 0, 0))
    params.append(Param('tblAMT13_Chisholm', 'sbact_Chisholm', 0, 0, 0, 0))
    ####### WOA: World Ocean Atlas Monthly Climatology
    params.append(Param('tblWOA_Climatology', 'nitrate_WOA_clim', 0, .5, .5, 5))
    params.append(Param('tblWOA_Climatology', 'phosphate_WOA_clim', 0, 0.5, 0.5, 5))
    ####### Satellite
    params.append(Param('tblCHL_REP', 'chl', 4, 0.25, 0.25, 0))
    ####### Darwin Model
    params.append(Param('tblDarwin_Phytoplankton', 'picoprokaryote', 2, 0.25, 0.25,␣
↪5))


    tables, variables, temporalTolerance, latTolerance, lonTolerance, depthTolerance␣
↪= [], [], [], [], [], []
    for i in range(len(params)):
        tables.append(params[i].table)
        variables.append(params[i].variable)
        temporalTolerance.append(params[i].temporalTolerance)
        latTolerance.append(params[i].latTolerance)
        lonTolerance.append(params[i].lonTolerance)
        depthTolerance.append(params[i].depthTolerance)
    return tables, variables, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance
```

(continues on next page)

(continued from previous page)

```
targetTables, targetVars, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance = match_params()
go = plot_corr_map(
                   sourceTable='tblAMT13_Chisholm',
                   sourceVar='MIT9313PCR_Chisholm',
                   targetTables=targetTables,
                   targetVars=targetVars,
                   dt1='2003-09-14',
                   dt2='2003-10-13',
                   lat1=-48,
                   lat2=48,
                   lon1=-52,
                   lon2=-11,
                   depth1=0,
                   depth2=240,
                   temporalTolerance=temporalTolerance,
                   latTolerance=latTolerance,
                   lonTolerance=lonTolerance,
                   depthTolerance=depthTolerance
                   )
```

```
# here is how to modify the graph:
import numpy as np

# print correlation values
# print(go.z)
# print(go.x)
# print(go.y)
go.z = np.abs(go.z)
go.cmap = 'Greys'
go.width = 1000
go.height = 1000
go.render()
```

1.3.1.8 Correlation Matrix Along Cruise Track

plot_cruise_corr_map(*cruise*, *targetTables*, *targetVars*, *depth1*, *depth2*, *temporalTolerance*, *latTolerance*, *lonTolerance*, *depthTolerance*, *method='spearman'*, *exportDataFlag=False*, *show=True*)

This function colocalizes the target variables along track of a cruise (see Match (colocalize) Cruise Track with Datasets). It then computes and visualizes a correlation matrix showing the pairwise correlation coefficients between the colocalized variables. The colocalization procedure relies on the tolerance parameters because they set the matching boundaries between the cruise track and target datasets. Currently, this graph is only supported by plotly library.

Returns the generated correlation graph object. One may modify the graph properties (see example below).

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API

properties are stored locally after being called the first time.

Parameters

cruise: string The official cruise name. If applicable, you may also use cruise "nickname" ('Diel', 'Gradients_1' ...). A full list of cruise names can be retrieved using cruise method.

targetTables: list of string Table names of the target datasets to be matched with the source data. Note source dataset can be matched with multiple target datasets. A full list of table names can be found in Catalog.

targetVars: list of string Variable short names to be matched with the source variable. A full list of variable short names can be found in Catalog.

depth1: float Start depth [m]. This parameter sets the lower bound of the depth cut on the target datasets. 'depth1' and 'depth2' allow matching a cruise trajectory (which is at the surface, hopefully!) with target varaiables at lower depth. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the depth cut on the target datasets. Note depth is a positive number (depth is 0 at the surface and increases towards the ocean floor).

temporalTolerance: list of int Temporal tolerance values between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single integer value is given, that would be applied to all target datasets. This parameter is in day units except when the target variable represents monthly climatology data in which case it is in month units. Note fractional values are not supported in the current version.

latTolerance: list of float or int Spatial tolerance values in meridional direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

lonTolerance: list of float or int Spatial tolerance values in zonal direction [deg] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets. A "safe" value for this parameter can be slightly larger than the half of the target variable's spatial resolution.

depthTolerance: list of float or int Spatial tolerance values in vertical direction [m] between pairs of source and target datasets. The size and order of values in this list should match those of targetTables. If only a single float value is given, that would be applied to all target datasets.

method: str, default: 'spearman' Correlation algorithm. 'spearman' is a rank correlation algorithm and is a metric for monotonic relationships. Other options involve **'pearson'** and **'kendall'**. *'pearson'* is the standard correlation coefficient, more favorable for linear correlations. *'kendall'* evaluates Kendall Tau correlation coefficient.

exportDataFlag: boolean, default: False If True, the graph data points are stored on the local machine. The export path and file format are set by the APIs parameters.

show: boolean, default: True If True, the graph object is returned and is displayed. The graph file is saved on the local machine at the figureDir directory. If False, the graph object is returned but not displayed.

Returns

the graph object

Below are the graph's properties and methods.

Properties

x: list of string Correlation matrix column titles (covariate names).

y: list of string Correlation matrix row titles (covariate names).

z: numpy.ndarray Computed pairwise correlation coefficients.

cmap: str or cmocean colormap Colormap name. Any matplotlib (e.g. 'viridis', ..) or cmocean (e.g. cmocean.cm.thermal, ..) colormaps can be passed to this property. A full list of matplotlib and cmocean color palettes can be found at the following links: https://matplotlib.org/3.1.0/tutorials/colors/colormaps.html

https://matplotlib.org/cmocean/

vmin: float This parameter defines the lower bound of the colorbar.

vmax: float This parameter defines the upper bound of the colorbar.

height: int Graph's height in pixels.

width: int Graph's width in pixels.

title: str Graphs's title.

Methods

render() Displays the plot according to the set properties.

Example

This example colocalizes the Gradient 2 cruise (MGL1704) with 12 variables, including underway measurements of prochlorococcus, synechococcus, and picoeukaryote abundances by SeaFlow dataset, satellite products (adt, chl, sst), and model estimates (see the `match_params()` function below for more details). Please explore the catalog to find more appropriate target variables.

Returns the generated correlation graph object. One may modify the graph properties (see example below).

Review Match (colocalize) Cruise Track with Datasets, and Match (colocalize) Datasets pages for more details and tips!

Note: This method requires a valid API key. It is not necessary to set the API key every time because the API properties are stored locally after being called the first time.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary
# uncomment the lines below if the API key has not been registered on your machine,␣
↪previously.
# import pycmap
# pycmap.API(token='YOUR_API_KEY>', vizEngine='plotly')

from collections import namedtuple
from pycmap.viz import plot_cruise_corr_map


def match_params():
    Param = namedtuple('Param', ['table', 'variable', 'temporalTolerance',
↪'latTolerance', 'lonTolerance', 'depthTolerance'])
    params = []
    ######## seaflow
    params.append(Param('tblSeaFlow', 'prochloro_abundance', 0, 0.1, 0.1, 5))
    params.append(Param('tblSeaFlow', 'synecho_abundance', 0, 0.1, 0.1, 5))
    params.append(Param('tblSeaFlow', 'picoeuk_abundance', 0, 0.1, 0.1, 5))
    ####### WOA: World Ocean Atlas Monthly Climatology
    params.append(Param('tblWOA_Climatology', 'silicate_WOA_clim', 0, .5, .5, 5))
    params.append(Param('tblWOA_Climatology', 'oxygen_WOA_clim', 0, 0.5, 0.5, 5))
    ####### Satellite
    params.append(Param('tblSST_AVHRR_OI_NRT', 'sst', 1, 0.25, 0.25, 5))
    params.append(Param('tblSSS_NRT', 'sss', 1, 0.25, 0.25, 5))
    params.append(Param('tblAltimetry_REP', 'adt', 1, 0.25, 0.25, 5))
    params.append(Param('tblCHL_REP', 'chl', 4, 0.25, 0.25, 0))
    ####### Models
    params.append(Param('tblPisces_NRT', 'NO3', 4, 0.5, 0.5, 5))
    params.append(Param('tblDarwin_Plankton_Climatology', 'prokaryote_c01_darwin_clim
↪', 0, 0.5, 0.5, 5))
    params.append(Param('tblDarwin_Plankton_Climatology', 'prokaryote_c02_darwin_clim
↪', 0, 0.5, 0.5, 5))

    tables, variables, temporalTolerance, latTolerance, lonTolerance, depthTolerance␣
↪= [], [], [], [], [], []
    for i in range(len(params)):
        tables.append(params[i].table)
        variables.append(params[i].variable)
        temporalTolerance.append(params[i].temporalTolerance)
        latTolerance.append(params[i].latTolerance)
        lonTolerance.append(params[i].lonTolerance)
        depthTolerance.append(params[i].depthTolerance)
    return tables, variables, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance



targetTables, targetVars, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance = match_params()
go = plot_cruise_corr_map(
                        cruise='MGL1704', # Gradients_2
                        targetTables=targetTables,
                        targetVars=targetVars,
                        depth1=0,
                        depth2=5,
                        temporalTolerance=temporalTolerance,
                        latTolerance=latTolerance,
```

(continues on next page)

(continued from previous page)

```
                        lonTolerance=lonTolerance,
                        depthTolerance=depthTolerance
                        )
```

```
# here is how to modify the graph:
import numpy as np

# print correlation values
# print(go.z)
# print(go.x)
# print(go.y)
go.z = np.abs(go.z)
go.cmap = 'Greys'
go.width = 1000
go.height = 1000
go.render()
```

1.3.1.9 Climatology

climatology(*table*, *variable*, *period*, *periodVal*, *lat1*, *lat2*, *lon1*, *lon2*, *depth1*, *depth2*)

Computes the climatology of a gridded dataset over a spatial domain delimited by (lat1, lat2, lon1, lon2, depth1, depth2). Note this method does not apply to sparse datasets. The parameter period specifies the climatology interval (e.g weekly, monthly...) and periodVal sets the interval value. For example, to compute the climatology of a variable for the month of October, period is set to 'month' and periodVal is set to 10. Please avoid using periods that are finner than the temporal resolution of the underlying dataset. For instance, if the dataset is a weekly-averaged product, do not set the period to 'dayofyear'.

The output of this method is a Pandas DataFrame ordered by time, lat, lon, and depth (if exists), respectively.

Parameters

table: string Table name (each dataset is stored in a table). A full list of table names can be found in catalog.

variable: string Variable short name which directly corresponds to a field name in the table. A subset of this variable's climatology is returned by this method according to the spatio cut parameters (below). A full list of variable short names can be found in catalog.

period: string Specifies the aggregation period and can be either one of the following values: 'd' or 'dayofyear' for day-of-year climatology 'w' or 'week' or 'weekly' for weekly climatology 'm' or 'month' or 'monthly' for monthly climatology 'a' or 'y' or 'yearly' for interannual averaging

periodVal: string Sets the value of the climatology interval. For instance, to retrieve the chlorophyll climatology for the month of March period is set to 'month' and periodVal is set to 3 (see example below).

lat1: float Start latitude [degree N]. This parameter sets the lower bound of the meridional cut. Note latitude ranges from -90° to 90°.

lat2: float End latitude [degree N]. This parameter sets the upper bound of the meridional cut. Note latitude ranges from -90° to 90°.

lon1: float Start longitude [degree E]. This parameter sets the lower bound of the zonal cut. Note longitue ranges from -180° to 180°.

lon2: float End longitude [degree E]. This parameter sets the upper bound of the zonal cut. Note longitue ranges from -180° to 180°.

depth1: float Start depth [m]. This parameter sets the lower bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

depth2: float End depth [m]. This parameter sets the upper bound of the vertical cut. Note depth is a positive number (it is 0 at surface and grows towards ocean floor).

Returns Pandas dataframe.

Example 1:

This example computes and retrieves a subset of sea surface salinity weekly climatology for the week of 46.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

import pycmap

api = pycmap.API(token='<YOUR_API_KEY>')
api.climatology(
                table='tblSSS_NRT',
                variable='sss',
                period='week',
                periodVal=46,
                lat1=20,
                lat2=50,
                lon1=-170,
                lon2=-120,
                depth1=0,
                depth2=0
                )
```

Example 2:

This example computes and retrieves a subset of chlorophyll climatology for the month of March. Notice, depth1 and depth2 values are automatically ignored because this is a surface dataset. A simple plot is made to visualize the retrieved data.

```
#!pip install pycmap -q     #uncomment to install pycmap, if necessary

%matplotlib inline
import matplotlib.pyplot as plt
import numpy as np
import pycmap



def plot(df):
    lat = df.lat.unique()
    lon = df.lon.unique()
    shape = (len(lat), len(lon))
    data = df.chl.values.reshape(shape)
    plt.imshow(data, extent=[np.min(lon), np.max(lon), np.min(lat), np.max(lat)],␣
↪cmap='viridis', origin='lower', vmin=0, vmax=0.5)
    plt.title('Sea Surface Chlorophyll [mg/m^3]')
    plt.colorbar()
    plt.xlabel('Longitude')
    plt.ylabel('Latitude')
    plt.show()


api = pycmap.API(token='<YOUR_API_KEY>')
df = api.climatology(
                     table='tblCHL_REP',
                     variable='chl',
                     period='month',
                     periodVal=3,
                     lat1=10,
                     lat2=70,
                     lon1=-180,
                     lon2=-100,
                     depth1=0,
                     depth2=0
                     )
plot(df)
```

SQL Statement

Here is how to achieve the same results using a direct SQL statement. Please refer to *Query* for more information.

```
EXEC uspAggregate 'tableName', 'variable', 'period', 'periodVal', 'lat1', 'lat2',
↪'lon1', 'lon2', 'depth1', 'depth2'
```

Example:

```
EXEC uspAggregate 'tblCHL_REP', 'chl', 'month', '3', '10', '70', '-180', '100', '0',
↪'0'
```

1.4 Presentations

Below are past presentations and tutorials

Presentation Name	Presenter	DOI	Colab	Binder
SCOPE2019	Mohammad Dehghani Ashkezari	DOI badge (incoming)		

CHAPTER 2

cmap4r API

cmap4r is a package developed for the R programming language to interact with the Simons CMAP database. It shares some of the structure and functionality of the pycmap package. More details can be found on the cmap4r github page or in the cmap4r online documentation. Both are listed below:

	
Github	Docs

CHAPTER 3

matcmap API

matcmap is the MATLAB client for Simons CMAP project. It provides access to the Simons CMAP database where various ocean data sets are hosted. These data sets include multi-decadal remote sensing observations (satellite), numerical model estimates, and field expeditions.

This package is adopted from pycmap which is the python client of Simons CMAP ecosystem.

Clone or download the repository. The source code is in the src directory. The CMAP.m file abstracts the Simons CMAP API and provides the user with several methods to query the database and extract subsets of data. In order to make API requests, you need to get an API key from Simons CMAP website. Once you got your API key, run the following command (in the MATLAB Command Window) to store the API key on your local machine:

```
CMAP.set_api_key('your api key');
```

In the MATLAB Command Window run the following command to see the docs:

```
doc CMAP
```

1. Get the list of data sets:

```
CMAP.datasets()
```

2. Retrieve a subset of sea surface temperature measured by satellite. A simple plot is made to visualize the retrieved data.

```
tbl = CMAP.space_time(...
                  'tblsst_AVHRR_OI_NRT',... % table
                  'sst',... % variable
                  '2016-04-30',... % dt1
                  '2016-04-30',... % dt2
                  10,... % lat1
                  70,... % lat2
                  -180,... % lon1
                  -80,... % lon2
                  0,... % depth1
```

(continues on next page)

(continued from previous page)

```
                    0);    % depth2

lat = unique(tbl.lat);
lon = unique(tbl.lon);
sst = reshape(tbl.sst, length(lon), length(lat));
imagesc(lon, lat, sst');
axis xy;
title('Sea Surface Temperature');
colorbar();
```

CHAPTER 4

juliacmap API

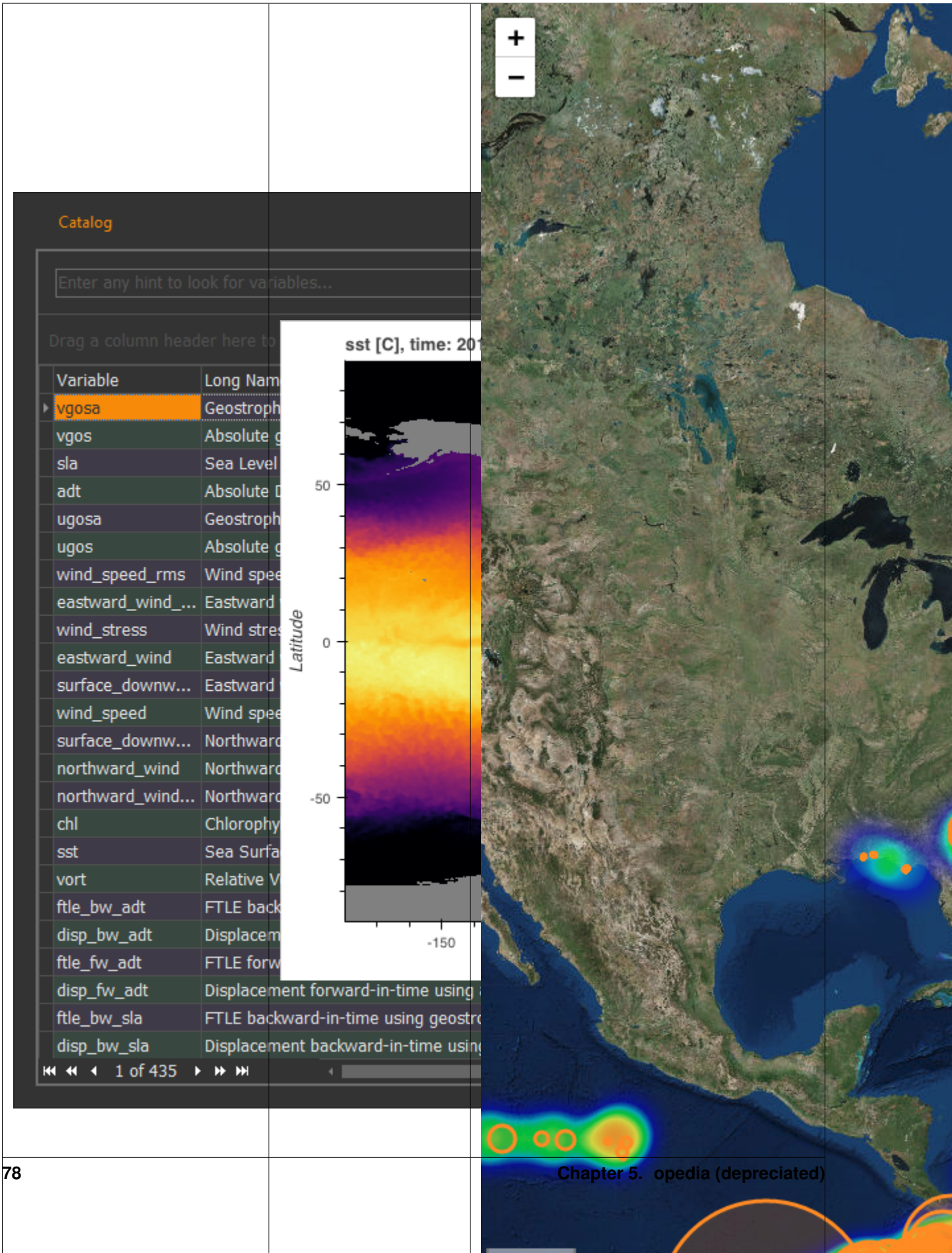
The Julia CMAP API is currently in development.

CHAPTER 5

opedia (depreciated)

Note: Opedia is the now depreciated python package for the Simons CMAP project. It has since been replaced by pycmap.

The tutorials include both GUI and script approaches. The GUI tutorials may include a video demonstration and the script approach include example code.


Catalog
Enter any hint to look for variables...
Drag a column header here to
Variable
Long Nam
vgosa
Geostroph
vgos
Absolute g
sla
Sea Level
adt
Absolute D
ugosa
Geostroph
ugos
Absolute g
wind_speed_rms
Wind spee
eastward_wind_...
Eastward
wind_stress
Wind stres
eastward_wind
Eastward
surface_downw...
Eastward
wind_speed
Wind spee
surface_downw...
Northward
northward_wind
Northward
northward_wind...
Northward
chl
Chlorophy
sst
Sea Surfa
vort
Relative V
ftle_bw_adt
FTLE back
disp_bw_adt
Displacem
ftle_fw_adt
FTLE forw
disp_fw_adt
Displacement forward-in-time using
ftle_bw_sla
FTLE backward-in-time using geostr
disp_bw_sla
Displacement backward-in-time usin
1 of 435
sst [C], time: 201
Latitude
50
0
-50
-150
+
−

5.1 Retrieve Data Catalog

Dataset descriptions and variables names can be accessed through the Catalog or through the getCatalog.py script.

This short Python script will pull all of the variables in the opedia database and write them to a .csv file in ./data/catalog.csv.

```
# Pull catalog from opedia database:
In [1]: from opedia import getCatalog
```

5.2 Map Gridded Data

Gridded data from CMAP can be mapped using **plotRegional** function. This functionally is best for mapping gridded datasets such as satellite or model products. Multiple datasets and variables can be queried in one request.

5.2.1 Recommended Data Sets for Gridded Regional Mapping:

SST	Chlorophyll_REP	Darwin_3day	Pisces

5.2.2 Code Tutorial

Jupyter Notebook

```
from opedia import plotRegional as REG


tables = ['tblsst_AVHRR_OI_NRT', 'tblPisces_NRT']     # see catalog.csv  for the␣
↪complete list of tables and variable names
variables = ['sst', 'Fe']                             # see catalog.csv  for the␣
↪complete list of tables and variable names
startDate = '2016-04-30'
endDate = '2016-04-30'
lat1, lat2 = 10, 70
lon1, lon2 = -180, -80
depth1, depth2 = 0, 0.5
fname = 'regional'
exportDataFlag = False        # True if you you want to download data

REG.regionalMap(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2, depth1,
↪ depth2, fname, exportDataFlag)
```

5.3 Map Sparse Data

Non gridded data from CMAP can now be mapped mapped using **plotRegional** function. This functionally is best for exploring cruise and float data. CMAP will visualize sparse data values as circular markers with the circle size corresponding to the variables value. A data density heatmap will also be overlaid on the map. In addition to this web map, CMAP will also create a 'dashboard' of the selected sparse data over time, latitude, longitude and depth.

Note: The mapping library used has limitations on the number of points it can render. If your query is greater than ~10k points, you may consider using the *Retrieve Data* functionality of CMAP and plotting locally.

5.3.1 Recommended Data Sets for Sparse Regional Mapping:

Flombaum	Argo	Chisholm_AMT13	SeaFlow

5.3.2 Code Tutorial

```
from opedia import plotRegional as REG


tables = ['tblFlombaum']    # see catalog.csv  for the complete list of tables and␣
↪variable names
variables = ['prochlorococcus_abundance_flombaum']                                  # see␣
↪catalog.csv  for the complete list of tables and variable names
startDate = '1987-09-17'
endDate = '2008-11-10'
lat1, lat2 = -90, 90
lon1, lon2 = -180, 180
depth1, depth2 = 0, 5
fname = 'regional'
exportDataFlag = False       # True if you you want to download data

REG.regionalMap(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2, depth1,
↪ depth2, fname, exportDataFlag)
```

5.3.3 Sparse Regional Map Dashboard

5.4 Section Plot

Create a section plot of modeled colored dissolved organic matter

Notes:

- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2° .

5.4.1 Code Tutorial

Jupyter Notebook

```
from opedia import plotSection as SEC

tables = ['tblDarwin_Nutrient_Climatology']    # see catalog.csv  for the complete␣
↪list of tables and variable names
variabels = ['CDOM_darwin_clim']                              # see catalog.csv  for␣
↪the complete list of tables and variable names



dt1 = '2016-04-30'   # PISCES is a weekly model, and here we are using monthly␣
↪climatology of Darwin model
dt2 = '2016-04-30'
lat1, lat2 = 23, 55
lon1, lon2 = -159, -157
depth1, depth2 = 0, 3597
fname = 'sectional'
exportDataFlag = False       # True if you you want to download data

SEC.sectionMap(tables, variabels, dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2,␣
↪fname, exportDataFlag)
```

5.5 Plot Time Series

Create a Time Series plot using satellite and modeled data

Note:

- Pisces model is a weekly-averaged global model with spatial resolution 1/2° X 1/2° (data is available only at one-week intervals).
- Satellite wind data set is a 6-hourly global product with spatial resolution 1/4° X 1/4° .
- Satellite Altimetry data set is a daily-global product with spatial resolution 1/4° X 1/4° .

5.5.1 Code Tutorial

Jupyter Notebook

```
from opedia import plotTS as TS

tables = ['tblSST_AVHRR_OI_NRT', 'tblAltimetry_REP', 'tblPisces_NRT']     # see␣
↪catalog.csv  for the complete list of tables and variable names
variables = ['sst', 'sla', 'NO3']                                          # see␣
↪catalog.csv  for the complete list of tables and variable names
startDate = '2016-03-29'
endDate = '2016-05-29'
lat1, lat2 = 25, 30
lon1, lon2 = -160, -155
depth1, depth2 = 0, 5
fname = 'TS'
exportDataFlag = False       # True if you you want to download data

TS.plotTS(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2, depth1,␣
↪depth2, fname, exportDataFlag)
```

5.6 Plot Depth Profile

Create depth profile plots using model and BGC-Argo float profiles.

Notes:

- Pisces model is a weekly-averaged global model with spatial resolution 1/2° X 1/2° (data is available only at one-week intervals).
- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2° .
- Argo float data set has irregular temporal and spatial resolution.

5.6.1 Code Tutorial

Visualize Argo in-situ data vs modeled data in a mid-depth to surface depth profile

Jupyter Notebook 1

```
from opedia import plotDepthProfile as DEP

tables = ['tblArgoMerge_REP', 'tblPisces_NRT', 'tblDarwin_Chl_Climatology']     # see␣
↪catalog.csv  for the complete list of tables and variable names
variables = ['argo_merge_chl_adj', 'CHL', 'chl01_darwin_clim']
startDate = '2016-04-30'   # PISCES is a weekly model, and here we are using monthly␣
↪climatology of Darwin model
endDate = '2016-04-30'
lat1, lat2 = 20, 24
lon1, lon2 = -170, -150
depth1, depth2 = 0, 1500
fname = 'DEP'
exportDataFlag = False      # True if you you want to download data

DEP.plotDepthProfile(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2,␣
↪depth1, depth2, fname, exportDataFlag)
```

Create depth profile plots using model and in-situ measurements at station ALOHA

Notes:

- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 12×12 .
- Pisces model is a weekly-averaged global model with spatial resolution 12×12 (data is available only at one-week intervals).
- Station ALOHA is a fixed loaction at north of Hawaii: 22 45 N 158 W.

Jupyter Notebook 2

```
from opedia import plotDepthProfile as DEP

tables = ['tblHOT_PP', 'tblDarwin_Chl_Climatology', 'tblPisces_NRT']     # see␣
↪catalog.csv  for the complete list of tables and variable names
```

(continues on next page)

(continued from previous page)

```
variables = ['chl_hot', 'chl01_darwin_clim', 'CHL']                    # see catalog.
↪csv  for the complete list of tables and variable names
startDate = '2015-04-30'   # PISCES is a weekly model, and here we are using monthly␣
↪climatology of Darwin model
endDate = '2016-04-30'
lat1, lat2 = 20, 24
lon1, lon2 = -159, -157
depth1, depth2 = 0, 170
fname = 'DEP'
exportDataFlag = False      # True if you you want to download data

DEP.plotDepthProfile(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2,␣
↪depth1, depth2, fname, exportDataFlag)
```

5.7 Plot XY

Create an XY plot of two or more variables

Note:

- Satellite SST data set is a daily-global product with spatial resolution 1/4° X 1/4° .
- Satellite Altimetry data set is a daily-global product with spatial resolution 1/4° X 1/4° .

5.7.1 Code Tutorial

Jupyter Notebook

```
from opedia import plotXY as XY

tables = ['tblSST_AVHRR_OI_NRT', 'tblAltimetry_REP']
variables = ['sst', 'sla']                                          # see catalog.csv ␣
↪for the complete list of tables and variable names
startDate = '2016-03-29'
endDate = '2016-05-29'
lat1, lat2 = 25, 30
lon1, lon2 = -160, -155
depth1, depth2 = 0, 5
fname = 'XY'
exportDataFlag = False      # True if you you want to download data

XY.plotXY(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2, depth1,␣
↪depth2, fname, exportDataFlag)
```

5.8 Plot Histogram

Plot Distribution (Satellite, Core Argo Floats) Create histograms of sea surface temperature (satellite), and temperature / salinity measurements by Argo floats.

Note:

- Satellite SST data set is a daily-global product with spatial resolution 14×14 .

- Argo float data set has irregular temporal and spatial resolution.

5.8.1 Code Tutorial

Jupyter Notebook

```
from opedia import plotDist as DIS

tables = ['tblSST_AVHRR_OI_NRT', 'tblArgoMerge_REP', 'tblArgoMerge_REP']           #␣
↪see catalog.csv  for the complete list of tables and variable names
variables = ['sst', 'argo_merge_temperature_adj', 'argo_merge_salinity_adj']       #␣
↪see catalog.csv  for the complete list of tables and variable names
startDate = '2016-04-30'
endDate = '2016-04-30'
lat1, lat2 = 20, 24
lon1, lon2 = -170, 150
depth1, depth2 = 0, 1500
fname = 'DEP'
exportDataFlag = False      # True if you you want to download data

DIS.plotDist(tables, variables, startDate, endDate, lat1, lat2, lon1, lon2, depth1,␣
↪depth2, fname, exportDataFlag)
```

5.9 Exact Amplicon Sequence Variants

Query by taxonomy level, clustering thereshold, and size fraction

The example below retrieves the "topN" number of most abundant sequenced organisms along track of the cruise. One can aggregate and visualize the relative abundance of the organisms according to their taxonomy, clustering levels, and size fractions. The cruise, 'ANT28-5', is an Atlantic latitudinal transect.

Thanks to Jed Fuhrman and Jesse McNichol (USC) for the beautiful dataset!

5.9.1 Code Tutorial

Jupyter Notebook

```
from opedia import esv

############## set parameters ###############
# only plot the top_N number of most abundant organisms
topN = 5
# aggregate organisims by their taxa level
tax = ['domain', 'kingdom', 'phylum', 'class', 'order', 'genus', 'species'][5]
depth1 = 20
depth2 = depth1
cruise_name = 'ANT28-5'
cluster_level = [89, 92, 96, 97, 98, 99, 100][0]        # minimum similarity␣
↪precentage to be clustred
size_frac_lower = [0.2, 3, 8][0]                        # size in micro-meter
size_frac_upper = [None, 3, 8][1]                       # size in micro-meter
#############################################

esv.plotESVs(topN, tax, depth1, depth2, cruise_name, cluster_level, size_frac_lower,␣
↪size_frac_upper)
```

(continues on next page)

(continued from previous page)

5.10 Colocalize Cruise

5.10.1 Example 1: Field Expedition

Colocalize Darwin model and satellite data with cruise. Compare the underway (in-situ) picoeukaryote abundance measurements performed during the "Gradient1.0" (aka SCOPE_16) with satellite chlorophyll data and picoeukaryote climatological estimates provided by Darwin model.

Notes:

- In-Situ picoeukaryote abundance measurements are results of the SeaFlow data set with 3-minute temporal resolution and irregular spatial resolution
- Satellite Chlorophyll data used in this example is a daily-global reprocessed and optimally interpolated data set with 4 km×4 km spatial resolution.
- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 12×12 .

5.10.1.1 Code Tutorial

Jupyter Notebook 1

```
from opedia import plotCruise as CRS

DB_Cruise = True                  # < True > if cruise trajectory already exists in DB.
↪ < False > if arbiturary cruise file (e.g. virtual)
source = 'tblSeaFlow'             # cruise table name or path to csv trajectory file
cruise = 'TN292'           # cruise name, or file name of the csv trajectory file
resampTau = '6H'                  # resample the cruise trajectory making trajectory␣
↪time-space resolution coarser: e.g. '6H' (6 hourly), '3T' (3 minutes), ... '0'␣
↪(ignore)
fname = 'alongTrack'              # figure filename
tables = ['tblSeaFlow', 'tblDarwin_Plankton_Climatology', 'tblCHL_OI_REP']    # list␣
↪of varaible table names
variables = ['picoeuk_abundance', 'picoeukaryote_c03_darwin_clim', 'chl']          ␣
↪  # list of variable names
spatialTolerance = 0.3            # colocalizer spatial tolerance (+/- degrees)
exportDataFlag = False            # export the cruise trajectory and colocalized data␣
↪on disk
depth1 = 0                        # depth range start (m)
depth2 = 5                        # depth range end (m)


df = CRS.getCruiseTrack(DB_Cruise, source, cruise)
df = CRS.resample(df, resampTau)
loadedTrack = CRS.plotAlongTrack(tables, variables, cruise, resampTau, df,␣
↪spatialTolerance, depth1, depth2, fname, exportDataFlag, marker='-', msize=30, clr=
↪'darkturquoise')
```

5.10.2 Example 2: Virtual Cruise

Colocalize Darwin model, satellite data with a virtual cruise. Colocalize a virtual cruise with satellite chlorophyll data and picoeukaryote climatological estimates provided by Darwin model. The trajectory of the virtual cruise is stored in a .csv file.

Notes:

- Satellite sea surface temperatue data used in this example is a daily-global near-real-time and optimally interpolated data set with 4km × 4km spatial resolution 14×14 .
- Satellite Chlorophyll data used in this example is a daily-global reprocessed and optimally interpolated data set with 4 km×4 km spatial resolution.
- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2° .

5.10.2.1 Code Tutorial

Jupyter Notebook 2

```
from opedia import plotCruise as CRS
import os

DB_Cruise = False                                   # < True > if cruise trajectory␣
↪already exists in DB. < False > if arbiturary cruise file (e.g. virtual)
source = './virtual_parity_scope_2.csv'             # cruise table name or path to csv␣
↪trajectory file
cruise = os.path.splitext(source)[0]                # cruise name, or file name of the␣
↪csv trajectory file
resampTau = '6H'                                    # resample the cruise trajectory␣
↪making trajectory time-space resolution coarser: e.g. '6H' (6 hourly), '3T' (3␣
↪minutes), ... '0' (ignore)
fname = 'alongTrack'                                # figure filename
tables = ['tblSST_AVHRR_OI_NRT', 'tblCHL_OI_REP', 'tblDarwin_Plankton_Climatology']  ␣
↪ # list of varaible table names
variables = ['sst', 'chl', 'picoeukaryote_c03_darwin_clim']                          ␣
↪ # list of variable names
spatialTolerance = 0.3                              # colocalizer spatial tolerance (+/
↪- degrees)
depth1 = 0.3                                        # depth range start (m)
depth2 = 5                                          # depth range end (m)
exportDataFlag = False                              # export the cruise trajectory and␣
↪colocalized data on disk



df = CRS.getCruiseTrack(DB_Cruise, source, cruise)
df = CRS.resample(df, resampTau)
loadedTrack = CRS.plotAlongTrack(tables, variables, cruise, resampTau, df,␣
↪spatialTolerance, depth1, depth2, fname, exportDataFlag, marker='-', msize=30, clr=
↪'darkturquoise')
```

5.11 Lagrangian Sampling

5.11.1 Code Tutorial

Colocalize a Lagrangian path with model and satellite data

As far as oceanic observation methodology is concerned, one can consider two major class of observations:

- Eulerian
- Lagrangian

Eulerian measurements are made at a fixed frame of reference which means the sensor is fixed at a given location and fluid is passing by. Lagrangian measurements are made from the perspective of water parcels which means the sensor propagates with the flow. This example demonstrates how to advect a passive tracer (massless particle) with the flow and construct a Lagrangian path. We then colocalize the Lagrangian path with other data sets (SST satellite and model-generated diazotroph concentration) to simulate a Lagrangian measurement.

Notes:

- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2°
- Satellite SST data set is a daily-global product with spatial resolution 1/4° X 1/4° .

Jupyter

```
from opedia import Lagrangian as LAG
import pandas as pd

fmt='%Y-%m-%d'
dt = 24*3600                                                    # propagation␣
↪time step (day seconds)
direction = 1                                                   # propagation␣
↪direction (forward/backward in time  <1> / <-1>)
startDate = '2016-01-01'                                        # propagation␣
↪start date
endDate = '2016-02-01'                                          # propagation␣
↪end date
lat0 = 36                                                       # starting poin␣
↪latitude
lon0 = -73                                                      # starting poin␣
↪longitude
fname = 'Tracer'                                                # figure␣
↪filename (and/or shape filename)
tables = ['tblSST_AVHRR_OI_NRT', 'tblDarwin_Plankton_Climatology']    # list of␣
↪varaible table names
variables = ['sst', 'diazotroph_c10_darwin_clim']               # list of␣
↪variable names
spatialTolerance = 0.3                                          # colocalizer␣
↪spatial tolerance (+/- degrees)
exportDataFlag = False                                          # export the␣
↪cruise trajectory and colocalized data on disk
depth1 = 0                                                      # depth range␣
↪start (m)
depth2 = 5                                                      # depth range␣
↪end (m)


df = pd.DataFrame()
```

(continues on next page)

(continued from previous page)

```
df['time'], df['lat'], df['lon'] = LAG.propagate(direction, startDate, endDate, lat0,␣
↪lon0, fmt, dt)
LAG.plotAlongTrack(dt, fmt, tables, variables, df, spatialTolerance, depth1, depth2,␣
↪exportDataFlag, fname, marker='-', msize=30, clr='darkturquoise')
```

5.12 Eddy Sampling

Colocalize an Eddy path with model and satellite data

As far as oceanic observation methodology is concerned, one can consider two major class of observations:

- Eulerian
- Lagrangian

Eulerian measurements are made at a fixed frame of reference which means the sensor is fixed at a given location and fluid is passing by. Lagrangian measurements are made from the perspective of water parcels which means the sensor propagates with the flow. This example demonstrates how to advect a passive tracer (massless particle) with the flow and construct a Lagrangian path. We then colocalize the Lagrangian path with other data sets (SST satellite and model-generated diazotroph concentration) to simulate a Lagrangian measurement.

Notes:

- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2°
- Satellite SST data set is a daily-global product with spatial resolution 1/4° X 1/4° .

5.12.1 Code Tutorial

Jupyter

```
from opedia import eddy as EDD

eddyTable = 'tblChelton'                                                    # eddy table␣
↪name
startDate = '2014-04-01'                                                    # eddy cores␣
↪within the delimited space-time (start date)
endDate = '2014-05-01'                                                      # eddy cores␣
↪within the delimited space-time (end date)
lat1 = 23                                                                   # eddy cores␣
↪within the delimited space-time (start lat)
lat2 = 29                                                                   # eddy cores␣
↪within the delimited space-time (end lat)
lon1 = -161                                                                 # eddy cores␣
↪within the delimited space-time (start lon)
lon2 = -151                                                                 # eddy cores␣
↪within the delimited space-time (end lon)
fname = 'eddy'                                                              # figure␣
↪filename (and/or shape filename)
tables = ['tblSST_AVHRR_OI_NRT', 'tblDarwin_Nutrient_Climatology']          # list of␣
↪varaible table names
variables = ['sst', 'DON_darwin_clim']                                  # list of variable␣
↪names
spatialTolerance = 0.3                                                      # colocalizer␣
↪spatial tolerance (+/- degrees)
```

(continues on next page)

(continued from previous page)

```
exportDataFlag = False                                                    # export the␣
↪cruise trajectory and colocalized data on disk
depth1 = 0                                                                # depth range␣
↪start (m)
depth2 = 5                                                                # depth range␣
↪end (m)


cores = EDD.getEddies(eddyTable, startDate, endDate, lat1, lat2, lon1, lon2)
EDD.colocalize(tables, variables, cores, spatialTolerance, depth1, depth2,␣
↪exportDataFlag, fname, marker='-')
```

5.13 Front Sampling

Colocalize an LCS path with model and satellite data

As far as oceanic observation methodology is concerned, one can consider two major class of observations:

- Eulerian
- Lagrangian

Eulerian measurements are made at a fixed frame of reference which means the sensor is fixed at a given location and fluid is passing by. Lagrangian measurements are made from the perspective of water parcels which means the sensor propagates with the flow. This example demonstrates how to advect a passive tracer (massless particle) with the flow and construct a Lagrangian path. We then colocalize the Lagrangian path with other data sets (SST satellite and model-generated diazotroph concentration) to simulate a Lagrangian measurement.

Notes:

- Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° X 1/2°
- Satellite SST data set is a daily-global product with spatial resolution 1/4° X 1/4° .

5.13.1 Code Tutorial

Jupyter

```
from opedia import ftle as FTLE

ftleTable = 'tblLCS_REP'                                                  # eddy table␣
↪name
startDate = '2017-04-30'                                                  # eddy cores␣
↪within the delimited space-time (start date)
endDate = '2017-04-30'                                                    # eddy cores␣
↪within the delimited space-time (end date)
lat1 = 23                                                                 # eddy cores␣
↪within the delimited space-time (start lat)
lat2 = 29                                                                 # eddy cores␣
↪within the delimited space-time (end lat)
lon1 = -161                                                               # eddy cores␣
↪within the delimited space-time (start lon)
lon2 = -151                                                               # eddy cores␣
↪within the delimited space-time (end lon)
fname = 'front'                                                           # figure␣
↪filename (and/or shape filename)
```

(continues on next page)

(continued from previous page)

```
tables = ['tblSST_AVHRR_OI_NRT', 'tblDarwin_Nutrient_Climatology']    # list of␣
↪varaible table names
variables = ['sst', 'DON_darwin_clim']                                # list of variable␣
↪names
spatialTolerance = 0.3                                                # colocalizer␣
↪spatial tolerance (+/- degrees)
exportDataFlag = False                                                # export the␣
↪cruise trajectory and colocalized data on disk
depth1 = 0                                                            # depth range␣
↪start (m)
depth2 = 5                                                            # depth range␣
↪end (m)


cores = FTLE.getEddies(eddyTable, startDate, endDate, lat1, lat2, lon1, lon2)
EDD.colocalize(tables, variables, cores, spatialTolerance, depth1, depth2,␣
↪exportDataFlag, fname, marker='-')
```

5.14 Colocalize Custom Dataset

Colocalize a virtual cruise with satellite chlorophyll data and picoeukaryote climatological estimates provided by Darwin model. The trajectory of the virtual cruise is stored in a .csv file.

Notes:

Satellite sea surface temperatue data used in this example is a daily-global near-real-time and optimally interpolated data set with 4km × 4km spatial resolution 1/4° X 1/4° .

Satellite Chlorophyll data used in this example is a daily-global reprocessed and optimally interpolated data set with 4km × 4km spatial resolution.

Darwin_Climatology is a monthly climatology version of the Darwin model with spatial resolution 1/2° ×1 /2° .

5.14.1 Code Tutorial

Jupyter_Notebook

```
from opedia import colocalize as COL

DB = False                              # < True > if source data exists in the␣
↪database. < 0 > if the source data set is a spreadsheet file on disk.
source = './KM1314_ParticulateCobalamins_2018_06_12_vPublished.xlsx'            # the␣
↪source table name (or full filename)
temporalTolerance = 1                  # colocalizer temporal tolerance (+/- days)
latTolerance = 0.3                     # colocalizer meridional tolerance (+/- degrees)
lonTolerance = 0.3                     # colocalizer zonal tolerance (+/- degrees)
depthTolerance = 5                     # colocalizer depth tolerance (+/- meters)
tables = ['tblSST_AVHRR_OI_NRT', 'tblPisces_NRT', 'tblDarwin_Plankton_Climatology']  ␣
↪ # list of varaible table names
variables = ['sst', 'Fe', 'picoeukaryote_c03_darwin_clim']
↪# list of variable names
exportPath = './loaded.csv'          # path to save the colocalized data set

COL.matchSource(DB, source, temporalTolerance, latTolerance, lonTolerance,␣
↪depthTolerance, tables, variables, exportPath)
```

(continues on next page)

(continued from previous page)

5.15 Retrieve Data

5.15.1 Space-Time

This tutorial shows how to retrieve a generic distribution of a variable within a predefined space-time domain. You need to know the variable and table names, both which can be found in the catalog. Data is retrieved in form of a dataframe with time, space, and variable columns.

```
from opedia import subset

############## set parameters ###############
table = 'tblsst_AVHRR_OI_NRT'
variable = 'sst'
dt1 = '2016-06-01'
dt2 = '2016-06-05'
lat1, lat2, lon1, lon2 = 23, 24, -160, -158
depth1, depth2 = 0, 0
#############################################

df = subset.spaceTime(table, variable, dt1, dt2, lat1, lat2, lon1, lon2, depth1,␣
↪depth2)    # retrieves a DataFrame
#df.to_csv('data.csv', index=False)      # save the retrieved data into a csv file
```

5.15.2 Time-Series

This tutorial shows how to retrieve time series of a variable within a predefined space-time domain. You need to know the variable and table names, both which can be found in the catalog. The timeSeries function computes the mean and standard deviation of the variable per time period. Data is retrieved in form of a dataframe with time, space, and variable columns.

```
from opedia import subset

############## set parameters ###############
table = 'tblsst_AVHRR_OI_NRT'
variable = 'sst'
dt1 = '2016-06-01'
dt2 = '2016-07-01'
lat1, lat2, lon1, lon2 = 23, 24, -160, -158
depth1, depth2 = 0, 0
#############################################

df = subset.timeSeries(table, variable, dt1, dt2, lat1, lat2, lon1, lon2, depth1,␣
↪depth2)    # retrieves a DataFrame
#df.to_csv('data.csv', index=False)      # save the retrieved data into a csv file
```

5.15.3 Depth Profile

This tutorial shows how to retrieve depth profile of a variable within a predefined space-time domain. You need to know the variable and table names, both which can be found in the catalog. The depthProfile function computes the mean and standard deviation of the variable per depth period. Data is retrieved in form of a dataframe.

```
from opedia import subset

############## set parameters ###############
table = 'tblPisces_NRT'
variable = 'CHL'
dt1 = '2016-04-30'
dt2 = '2016-04-30'
lat1, lat2, lon1, lon2 = 23, 24, -160, -158
depth1, depth2 = 0, 6000
#############################################

df = subset.depthProfile(table, variable, dt1, dt2, lat1, lat2, lon1, lon2, depth1,␣
↪depth2)    # retrieves a DataFrame
#df.to_csv('data.csv', index=False)      # save the retrieved data into a csv file
```

5.15.4 Section Subset

This tutorial shows how to retrieve section profile of a variable within a predefined space-time domain. You need to know the variable and table names, both which can be found in the catalog. Data is retrieved in form of a dataframe with time, space, and variable columns.

```
from opedia import subset

############## set parameters ###############
table = 'tblPisces_NRT'
variable = 'Fe'
dt1 = '2016-04-30'
dt2 = '2016-04-30'
lat1, lat2, lon1, lon2 = 22, 50, -160, -158
depth1, depth2 = 0, 6000
#############################################

subset.section(table, variable, dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2)
↪# retrieves a DataFrame
#df.to_csv('data.csv', index=False)      # save the retrieved data into a csv file
```

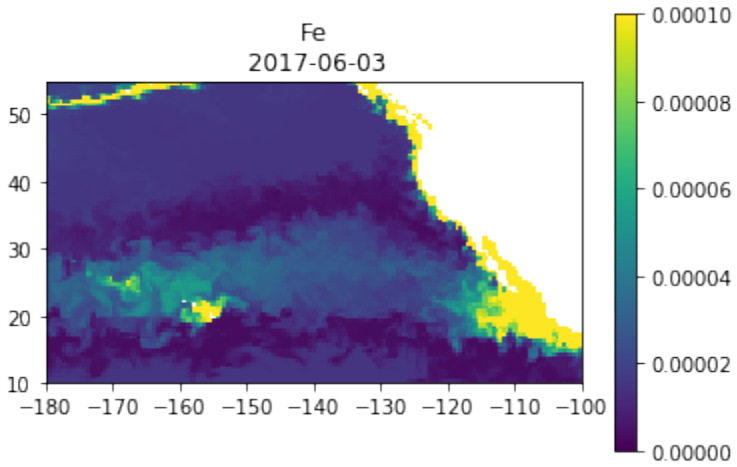
5.16 SQL Queries

5.16.1 Regional Map SQL

If you are familiar with SQL or T-SQL language, you can use "dbFetch()" function to execute any generic query and retrieve data. Below is a simple example showing how to retrieve a snapshot and plot a basic map.

```
"""Import the Libraries"""
from opedia import db
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline

"""Define the plot"""
def plot(dt, lat, lon, data):
    plt.imshow(data, extent=[np.min(lon), np.max(lon), np.min(lat), np.max(lat)],␣
↪origin='bottom', vmin=0, vmax=1e-4)
    plt.title(field + '\n ' + dt1)
    plt.colorbar()
    plt.show()

"""Construct the query statement"""
def prepareQuery(args):
    query = "SELECT [time], lat, lon, depth, %s FROM %s WHERE "
    query = query + "[time] BETWEEN'%s' AND '%s' AND "
    query = query + "lat BETWEEN %f AND %f AND "
    query = query + "lon BETWEEN %f AND %f AND "
    query = query + "depth BETWEEN %f AND %f "
    query = query + "ORDER BY [time], lat, lon, depth "
    query = query % args
    return query

"""Retrieve regional data and plot"""
############## set parameters ###############
table = 'tblPisces_NRT'
field = 'Fe'        # Mole concentration of dissolved Iron
dt1 = '2017-06-03'
dt2 = '2017-06-03'
lat1, lat2, lon1, lon2 = 10, 55, -180, -100
depth1 = 0
depth2 = 1
#############################################
args = (field, table, dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2)
query = prepareQuery(args)
df = db.dbFetch(query)
lat = df.lat.unique()
lon = df.lon.unique()
shape = (len(lat), len(lon))
data = df[field].values.reshape(shape)
#df.to_csv(field+'.csv', index=False)    # export data
plot(dt1, lat, lon, data)
```

Output



5.16.2 Time Series SQL

If you are familiar with SQL or T-SQL language, you can use "dbFetch()" function to execute any generic query and retrieve data. Below is a simple example showing how retrieve time series and plot.

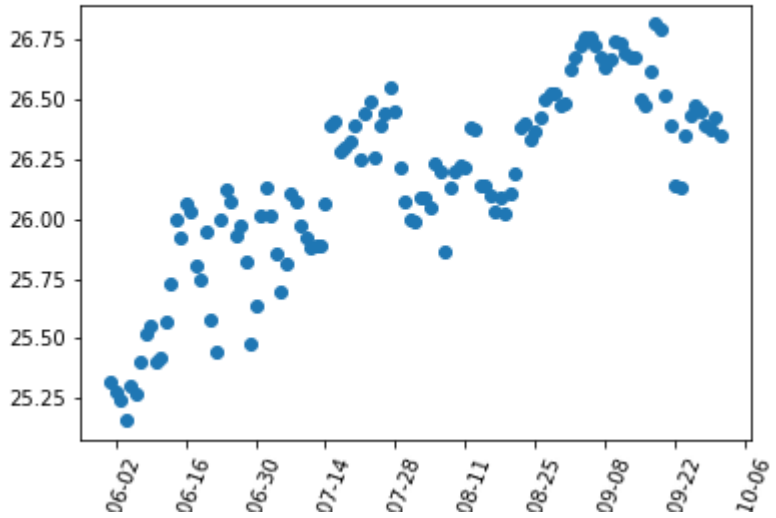
```
"""Import the libraries"""
from opedia import db
import matplotlib.pyplot as plt
%matplotlib inline

"""Define the plot"""
def plot(t, y):
    plt.plot(t, y, 'o')
    plt.xlabel('time')
    plt.show()

"""Construct the query statement"""
def prepareQuery(args):
    query = "SELECT [time], AVG(lat) AS lat, AVG(lon) AS lon, AVG(%s) AS %s FROM %s␣
↪WHERE "
    query = query + "[time] BETWEEN'%s' AND '%s' AND "
    query = query + "lat BETWEEN %f AND %f AND "
    query = query + "lon BETWEEN %f AND %f "
    query = query + "GROUP BY [time] "
    query = query + "ORDER BY [time] "
    query = query % args
    return query

"""Retrieve time-series datcd a and plot"""
############## set parameters ###############
table = 'tblsst_AVHRR_OI_NRT'
variable = 'sst'
dt1 = '2016-06-01'
dt2 = '2016-10-01'
lat1, lat2, lon1, lon2 = 23, 24, -160, -158
#############################################
args = (variable, variable, table, dt1, dt2, lat1, lat2, lon1, lon2)
query = prepareQuery(args)
df = db.dbFetch(query)
```

(continues on next page)

(continued from previous page)

```
#df.to_csv(variable+'.csv', index=False)     # export data
plot(df['time'], df[variable])
```

Output



5.16.3 Depth Profile SQL

If you are familiar with SQL or T-SQL language, you can use "dbFetch()" function to execute any generic query and retrieve data. Below is a simple example showing how to retrieve a depth profile and plot.

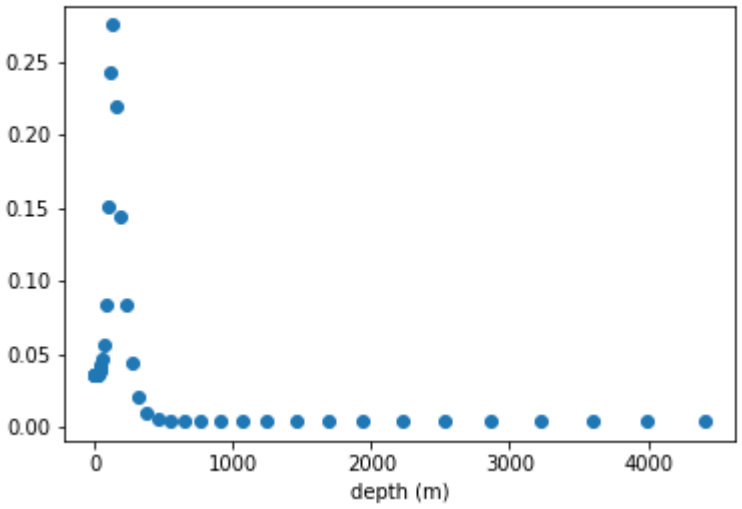
```
"""Import the libraries"""
from opedia import db
import matplotlib.pyplot as plt
%matplotlib inline

"""Define the plot"""
def plot(t, y):
    plt.plot(t, y, 'o')
    plt.xlabel('depth (m)')
    plt.show()

"""Construct the query statement"""
def prepareQuery(args):
    query = "SELECT AVG(lat) AS lat, AVG(lon) AS lon, depth, AVG(%s) AS %s FROM %s␣
↪WHERE "
    query = query + "[time] BETWEEN '%s' AND '%s' AND "
    query = query + "lat BETWEEN %f AND %f AND "
    query = query + "lon BETWEEN %f AND %f AND "
    query = query + "depth BETWEEN %f AND %f "
    query = query + "GROUP BY depth "
    query = query + "ORDER BY depth "
    query = query % args
    return query

"""Retrieve regional data and plot"""
```

(continues on next page)

(continued from previous page)

```
############## set parameters ###############
table = 'tblPisces_NRT'
field = 'CHL'
dt1 = '2016-06-01'
dt2 = '2016-10-01'
lat1, lat2, lon1, lon2 = 23, 24, -160, -158
depth1, depth2 = 0, 5000
#############################################
args = (field, field, table, dt1, dt2, lat1, lat2, lon1, lon2, depth1, depth2)
query = prepareQuery(args)
df = db.dbFetch(query)
#df.to_csv(field+'.csv', index=False)    # export data
plot(df['depth'], df[field])
```

Output



CHAPTER 6

Data Submission

If you wish to suggest a dataset be added to the database or have data that you would like added, please email us (cmap-data-submission@uw.edu).

The current data submission process is:

1. Contact us about adding a dataset to CMAP.
2. Format your data and metadata in the CMAP format.
3. Submit your dataset via email for feedback and review.
4. Once OK'ed, register a DOI for your dataset.
5. Submit finalized DOI via email.

6.1 Data Template

Download Data Template

6.2 Dataset Examples

- Download Gradients 1 Cobalamin Example

- Download AMT01 Extracted Chlorophyll Example
- Download Gradients 1 Influx Flow Cytometry Example
- Download SCOPE HOT Metagenomics Example

CHAPTER 7

File Structure

7.1 Table of Contents

- cruise_names

4. Variable Metadata
    - var_short_name
    - var_long_name
    - var_sensor
    - var_unit
    - var_spatial_res
    - var_temporal_res
    - var_discipline
    - visualize
    - var_keywords
    - var_comment
5. Bibliography
    - *References*

7.2 Introduction

The CMAP data template consists of three sheets: data, dataset metadata, and variable metadata. Data is stored in the first sheet called "data". Metadata that describes the dataset is entered in the second sheet called "dataset_meta_data". Metadata associated with the variables in the dataset are entered in the third sheet called "vars_meta_data". Information must be provided for all columns except those specifcally noted as optional. For those datasets that use the Excel template, a web-based tool is available through Simons CMAP to validate and modify a given dataset to ensure it conforms to structure requirements for dataset submission. Below are a few example datasets that have been prepared using the specifications described in this document:

- Gradients 1 Cobalamin
- AMT01 Extracted Chlorophyll

7.3 Data Sheet

All data points are stored in the "Data" sheet. Each data point must have time and location information. The exact name and order of the time and location columns are shown in the table above. If a dataset does not have depth values (e.g., sea surface measurements), you may remove the depth column. If your dataset represents results of a `Laboratory` study (see dataset_make) fill these fields with the time of study and the location of your laboratory. The columns var1 … varn represent the dataset variables (measurements). Please rename var1 … varn to names appropriate to your data. The format of "time", "lat", "lon", and "depth" columns are described in the following sections. Please review the example datasets listed in the introduction for more detailed information.

- **time*** This column holds datetime values with the following format: %Y-%m-%dT%H:%M:%S The date and time sections are separated by a "T" character. Example: 2010-02-09T18:15:00
    - Year (%Y) is a four-digit value: example 2010

  - Month (%m) is a two-digit value: example 02 (for Feburary)
  - Day (%d) is a two-digit value: example 09
  - Hour (%H) is a two-digit value from 00 to 23: example 18
  - Minute (%M) is a two-digit value from 00 to 59: example 15
  - Second (%S) is a two-digit value from 00 to 59: example 00
  - Time zone: UTC
- **lat*** This column holds the latitude values with the following characteristics:
  - Type: Numeric values from -90 to 90
  - Format: Decimal (not military grid system)
  - Unit: degree North
- **lon*** This column holds the longitude values with the following characteristics:
  - Type: Numeric values from -180 to 180
  - Format: Decimal (not military grid system)
  - Unit: degree East
- **depth** This column holds the depth values with the following characteristics:
  - Type: Positive numeric values. It is 0 at surface with increased values with depth.
  - Format: Decimal
  - Unit: meter
- **var1 ... varn** These columns represent the dataset variables (measurements). Please rename them to names appropriate for your data. Note that these names should be identical to the names defined as var_short_name in the Variable Metadata sheet. Please do not include units in these columns; units are recorded in the Variable Metadata sheet. Leave a given cell empty for those instances when data was not taken and a value is missing. Do not replace the missing data with arbitrary values such as "99999", "0", "UNKNOWN", etc. Please review the example datasets listed in the introduction for more information.

7.4 Dataset Metadata

- **dataset_short_name*** This name is meant to be used in programming codes and scripts. It should only contain a combination of letters, numbers, and underscores (the first character can not be a number). Do not use space, dash, or special characters such as <, +, %, etc. The name must be shorter than 50 characters and is a required field.
  - Required: Yes
  - Constraint: Less than 50 characters
- **dataset_long_name*** A descriptive and human-readable name for the dataset. This name will identify your dataset in the CMAP catalog (Fig.1) and visualization search dialog (Fig.2). Any Unicode character can be used here, but please avoid names longer than 200 characters as they may get trimmed when displayed on graphical interfaces. A full textual description of your dataset, with no length limits, is entered in "dataset_description" . If your dataset is associated with a cruise, we recommend including the official cruise and and the cruise nickname in the dataset_long_name. For example: Underway CTD Gradients 3 KM1906.
  - Required: Yes

  - Constraint: Less than 200 characters

- **dataset_version*** Please assign a version number or an identifier to your dataset such as "1.0.0" or "Final". Version identifiers will help track the evolution of a dataset over time.
  - Required: Yes
  - Constraint: Less than 50 characters
  - Example: 1.0
- **dataset_release_date*** Indicates the release date of the dataset. If your dataset has been previously published or released publicly, please specify that date. Otherwise, use the date the dataset was submitted to CMAP.
  - Required: Yes
  - Constraint: Less than 50 characters
  - Example: 2020-06-22
- **dataset_make*** This is a required field that provides a broad category description of how a dataset was produced (also referred to as `dataset make`). Each dataset requires a single descriptor from a fixed set of options (observation, model, assimilation, laboratory), which are described below. This field will help in discovery of data in CMAP by categorizing datasets according to their `Make` class. Please contact us if you believe your dataset Make is not consistent with any of the categories below:
  - **Observation**: refers to any in-situ or remote sensing measurements such as measurements made during a cruise expedition, data from an in-situ sensor, or satellite observations. Observations made as part of laboratory experiments have their own distinct category and do not fall in this category.
  - **Model**: refers to the outputs of numerical simulations.
  - **Assimilation**: refers to products that are a blend of observations and numerical models.
  - **Laboratory**: refers to the observations made in a laboratory setting such as culture experiment results.
- **dataset_source*** Specifies the group and/or the institute name of the data owner(s). It can also include any link (such as a website) to the data producers. This information will be visible in the CMAP catalog as shown in Fig.3. Also, dataset_source will be annotated to any visualization made using the dataset (Fig. 4). This is a required field and its length must be less than 100 characters.
  - Required: Yes
  - Constraint: Less than 100 characters
  - Example: Armbrust Lab, University of Washington
- **dataset_distributor** If your dataset has already been published by a data distributor provide a link to the data distributor. Otherwise, leave this field empty. This is not a required field.
  - Required: No (optional)
  - Constraint: Less than 100 characters
  - Example: http://marine.copernicus.eu/
- **dataset_acknowledgement** Specify how your dataset should be acknowleged. You may mention your funding agency, grant number, or you may ask those that use your data to acknowledge your dataset with a particular statement. Dataset acknowlegment will be visible in the catalog page (Fig. 5). This is not a required field.
  - Required: No (optional)
  - Constraint: No length limits
- **dataset_history** Use this field if your dataset has evolved over time and you wish to add notes about the history of your dataset. Otherwise, leave this field empty. This is not a required field.

- **dataset_description*** Include any description that you think will help a reader better understand your dataset. This description can include information about data acquisition, processing methods, figures, and links to external content. This field serves as the dataset documentation that is visible in the Simons CMAP catalog (Fig. 6). This field is required.
  - Required: Yes
  - Constraint: No length limits
- **dataset_references** List any publications or documentation that one may cite in reference to the dataset. If there are more than one reference, please put them in separate cells under the dataset_reference column. Leave this field empty if there are no publications associated with this dataset. This is not a required field.
- **climatology** This is a flag indicating whether the dataset represents a climatological product. If your dataset is a climatological product fill this field with "1". Otherwise, leave this field blank. This is not a required field.
- **cruise_names** If your dataset represents measurements made during a cruise expedition (or expeditions), provide a list of cruise official names here. If your dataset is associated with more than one cruise, please put them in separate cells under the cruise_names column. If the cruises have any nicknames, please include them in the same cell as the official cruise name separated by a comma(s). Leave this field blank if your dataset is not associated with a cruise expedition. This is not a required field.
  - Required: No (optional)
  - Constraint: No length limits
  - Example: KOK1606, Gradients 1

7.5 Variable Metadata

- **var_short_name*** This name is meant to be used in programming codes and scripts. It should only contain a combination of letters, numbers, and underscores (the first character can not be a number). Do not use space, dash, or special characters such as <, +, %, etc. Finally, there must be a one-to-one match between the short_names listed here and the variable column names in the "Data" sheet (see vars). `var_short_name` will be seen in the CMAP catalog (Fig. 7), and will appear as the title of the generated figures (Fig. 8). This a required field and must be shorter than 50 characters.
  - Required: Yes
  - Constraint: Less than 50 characters
- **var_long_name*** A descriptive and human-readable label for the variable in accordance with the CF and COARDS conventions [1, 2, 3]. This name will present your variable in the CMAP catalog (Fig. 9) and visualization search dialog (Fig. 10). `var_long_name` can contain any unicode character, but please avoid names longer than 200 characters as they may get trimmed while displayed on graphical interfaces. Please use var_comment if you would like to add a full textual description (with no length limits) for your variable.
  - Required: Yes
  - Constraint: Less than 200 characters
- **var_sensor*** This is a required field that refers to the instrument used to produce the measurements such as `CTD`, `fluorometer`, `flow cytometer`, `sediment trap`, etc. If your dataset is the result of a field expedition but you are not sure about the name of the instrument used for the measurements, use the term "in-situ" to fill out this field. If your dataset is the output of a numerical model or a combination of model and observation, use the term "simulation" and "blend", respectively. This field will significantly help to find and categorize data generated using a similar class of instruments. `var_sensor` will be visible in the Simons CMAP catalog.

- **var_unit** Specifies variable units, if applicable. Leave this field blank if your variable is unitless (e.g. "station numbers" or "quality flags"). Units may contain unicode characters such as subscripts and superscripts. `var_unit` will be visible in the Simons CMAP catalog (see Fig. 9) and in the generated visualizations (see Fig. 8). This field is not required.
  - Required: No (optional)
  - Constraint: Less than 50 characters
  - Example: ug L-1
- **var_spatial_res*** Specifies the spatial resolution of the variable. Typically, gridded products have uniform spatial spacing (such as 0.25° X 0.25°) while field expeditions do not have a regular spatial resolution. If your variable does not have a regular spatial resolution, use the term "irregular" to fill out this field. Note that if samples are taken at a series of distinct but spatially-non-uniform stations, the spatial resolution is considered irregular. `var_spatial_res` may contain unicode characters such as degree symbol (°) and will be visible in the Simons CMAP catalog (see Fig. 9). This field is required.
  - Required: Yes
  - Constraint: Less than 50 characters
  - Example: irregular
- **var_temporal_res*** Specifies the temporal resolution of measurements (such as daily, hourly, 3-minutes, etc). If the measurements do not have a regular temporal spacing, use the term "irregular" to fill out this field. `var_temporal_res` will be visible in the Simons CMAP catalog (see Fig. 9). This field is required.
  - Required: Yes
  - Constraint: Less than 50 characters
  - Example: irregular
- **var_discipline*** Indicates in which disciplines (such as Physics, Biology . . .) this variable is commonly studied. You can specify more than one discipline. If you list multiple disciplines per variable, please separate them by comma. `var_discipline` will be visible in the Simons CMAP catalog (referred to as "Study Domain" in Fig. 9). This field is required.
  - Required: Yes
  - Constraint: Less than 100 characters
  - Example: Physics, BioGeoChemistry
- **visualize** This is a flag field and can only be 0 or 1. Fill this field with 1, if you think this variable can be visualized on a graph by Simons CMAP. In principle, any variable with numeric values can be visualized while variables with string values, station numbers, or quality flags may not be the best candidates for visualization in CMAP. Please consult with the data curation team if you have any questions. This is not a required field.
- **var_keywords*** Every variable in CMAP is annotated with a range of semantically related keywords to ensure a variable can be easily discovered. For example, use of keywords allows you to search using the term "PO4" and retrieve a list of all phosphate data even if "PO4" was not used as the var_long_name for a given dataset. Similarly, if one searches for "MIT", CMAP returns all variables generated by MIT groups, or if one looks for "model", only model outputs are returned. These "semantic" searches are made possible using the keywords that are added to each variable. We would like to have keywords to cover the following areas described below (where applicable). Please note that there is no limit to the number of keywords used for a variable. The keywords are case-insensitive and you may add/remove them at any point (even after data ingestion). This is a required field.
  - Alternative names: other official, unofficial, abbreviation, technical (or jargon) names or notations associated with the variable. Examples: Nitrate, NO3, NO_3

  - Method and Instrument: Keywords related to the method and instruments used for the variable measurements. Examples: observation, in-situ, model, satellite, remote sensing, cruise, CTD, cytometry, ... Note these keywords are not mutually exclusive. For example, a CTD temperature measurement made during a cruise can have all of the following keywords: observation, in-situ, cruise, CTD
  - Data Producers: Keywords associated with the lead scientist/lab name/institute name. Examples: UW, University of Washington, Virginia Armbrust, Ginger
  - Cruise: The official/unoffical name of the cruise(s) during which the variable was measured, if applicable. Examples: KOK1606, Gradients_1, diel
  - Project name: If your data are in the context of a project, include the project name. Examples: HOT, Darwin, SeaFlow

- **var_comment** Use this field to communicate any detailed information about this particular variable with the users. This could include, for example, description of method(s) used to process the raw measurements. `var_comment` is visible in the Simons CMAP catalog (Fig. 11). This field is not required.
  - Required: No (optional)
  - Constraint: No length limits

7.6

7.7 Bibliography

7.7.1 References

1. NetCDF Climate and Forecast (CF) Metadata Conventions
2. Conventions for the standardization of NetCDF files
3. COARDS NetCDF Conventions

CHAPTER 8

FAQ

8.1 What do the and badges do?

Google Colab and Binder are projects that allow you to run code on the cloud without downloading or installing anything on your computer. Simons CMAP has Colab and Binder badges, which allow you to run any of the pycmap examples simply by clicking on the badges.

Google Colab seems to start faster, but currently requires you to uncomment (ie. remove the '#') to install pycmap. Binder takes a little longer to start, but may be easier to use. Feel free to try one or both!

Note: Binder and Colab are great for learning and testing, but if you have a longer term project using pycmap we strongly suggest installing pycmap locally. Please save any work you wish to keep locally, as Binder and Colab are temporary instances and will not save any of your work.

8.2 What is an API Key and how do I get one?

An API key is a unique identifier required to run all software packages. To obtain an API key, please register first and then go to https://simonscmap.com/apikeymanagement. Please keep this key for future use. All projects developed by Simons CMAP team are open-source and can be found on Github.

Note: Please do not share API keys.

8.3 What is a DOI and how do I get one for my dataset?

A DOI or Digital Object Identifier is a digital identifier for a dataset. See the tutorial video below to learn how to obtain a DOI for your dataset using Zenodo.

We require a DOI to be registered for your dataset in the submission process. It allows for users to cite the use of a dataset and properly acknowledge the dataset creators.

Some DOI providers, such as Zenodo, allow for dataset version controlling.

Below is a list of entities that may issue and link your dataset to a unique DOI:

- Zenodo
- Dryad (only accepts published datsets)
- FigShare
- Open_Science_Framework
- Harvard_Dataverse
- NCEI
- ORNL_DAAC
- EDI
- PANGAEA
- SEANOE
- NASA_Goddard
- NERC

8.4 How do I cite data found in CMAP?

DOI or relevant citations for specific datatsets can be found in the Catalog or in the retrieved metadata.

CHAPTER 9

Contact and Contribute

9.1 Who to Contact

If you have questions about using Simons CMAP feel free to contact us at: cmap-data-submission@uw.edu

or preferably join our Slack channel

9.2 How to Contribute

You can contribute to pycmap or cmap4r github pages or make suggestions via email or Slack.

Index