# UMD-LETKF Documentation

**Travis C Sluka**

**Feb 05, 2020**

# Contents:

Universal Multi-Domain Local Ensemble Transform Kalman Filter

# CHAPTER 1

# Overview

The Universal Multi-Domain Local Ensemble Transform Kalman Filter (UMD-LETKF) is a rewrite of the LETKF originally developed by [Hunt2007], coded by [Miyoshi2005], with additional modifications for the ocean by Steve Penny.

It is built with the following design choices in mind:

- **model agnostic library** - A single generic LETKF library is provided that can be compiled once and then used in all domains of a coupled LETKF system. Redundancies in code are eliminated this way. Most specialization for a given domain are done through configuration files, and a generic driver is provided that should handle most use cases. A custom driver can easily be built to interface with the library if model specific code needs to be added.
- **object oriented design** - Several default implementations of classes for *observation* I/O, *model state* I/O, and *localization* are provided. If different functionality is required, the user can create their own derived classes and register them with the LETKF library.
- **multi-model strong coupling** - By being model agnostic, the code should allow for easy transition from weakly coupled to strongly coupled DA. The same LETKF code can be used for multiple independent executables (one for each domain), and cross-domain observations can be assimilated by selecting the appropriate observation I/O and localization classes.

CHAPTER 2

# Installation

## 2.1 Dependencies

The following dependencies are required in order to compile UMD-LETKF. *(These should all be available in package managers for standard Linux installations)*

- CMake
- Fortran Compiler *( gfortran >= 5.0, Intel >= 16.0, tested)*
- NetCDF4 w/ API for Fortran
- MPI *(openmpi and intelmpi tested)*
- BLAS
- LAPACK

The following dependencies are **optional**, depending on the features the user wants enabled for UMD-LETKF at compile time, and can be installed separately, or built as part of UMD-LETKF with the appropriate *CMake options*

- WGRIB2 API from NCEP *(optional)*

## 2.2 Compiling

First, download the source code, which includes external repositories (libyaml and geoKdTree)

```
git clone https://github.com/travissluka/UMD-LETKF.git
cd UMD-LETKF
git submodule update --init
```

Then, create a directory in which UMD-LETKF will be built

```
mkdir build
cd build
```

Configure with cmake, pointing it to the location of the source directory (the parent directory in this example), and build

```
cmake ../
make
```

CMake might complain about certain libraries not being found, such as NetCDF4. If this happens, you might need to specify the path to these libraries with the *-DCMAKE_PREFIX_PATH=* option. (see *CMake options*)

## 2.3 Running Tests

To run the test cases, specify the *-DLETKF_ENABLE_TESTS=ON* when running *cmake*

```
cmake ../ -DLETKF_ENABLE_TESTS=ON
make
ctest
```

This will download test data and the correct reference solutions, and test to make sure the answers are within margin of error of being identical to the reference solutions. Note that the test data and reference solutions are updated every now and then on Dropbox, so old versions of the GitHub are not guaranteed to pass the tests. It is therefore recommended that you are using the latest version of UMD-LETKF before running the test cases.

The above commands will simply tell you whether or not the test cases pass or fail. To see the actual output of the UMD-LETKF, run with the *-VV* flag. You can also run specific subsets of tests. Run with the *-N* flag to see a list of the available tests, and run with *-R <testname>* to run a specific set of tests.

**Warning:** The reference answers for these tests were generated with GCC7 and so may not match Intel compilers that have I have not extensively tested (version 17.0+ ). Hopefully this will be updated in the near future. So, don't be alarmed if your ctests fail if you're using Intel compiler.

## 2.4 CMake options

There are several command line options that can be passed to CMake when configuring the build. The following options are a small subset of the options most relevant to UMD-LETKF

- **-DCMAKE_BUILD_TYPE=Debug**

  compile in debug mode, the code runs slower, but more likely to produce a useful error message if there is a unexpected run-time error. Without this flag UMD-LETKF will by default compile in **Release** mode

- **-DCMAKE_PREFIX_PATH=...**

  used to specify one or more directories in which to search for the required libraries (NetCDF, MPI, WGRIB2, etc...) If more than one directory is specified, they should be separated by a semicolon and surrounded by quotation marks. E.g.

```
cmake ../ -DCMAKE_PREFIX_PATH="$NETCDF_DIR;$WGRIB2_API_DIR"
```

- **-DLETKF_ENABLE_GRIB=ON**

  Builds the optional grib *state* I/O module. If enabled, either the path to the wgrib2 api needs to be specified in *CMAKE_PREFIX_PATH*, or *LETKF_BUILD_GRIB* needs to be enabled.

- **-DLETKF_BUILD_GRIB=ON**

  If *LETKF_ENABLE_GRIB* is on, the wgrib2 api will be downloaded from the NCEP server and built before the rest of the UMD-LETKF library is built

- **-DLETKF_ENABLE_TESTS=ON**

  The test cases and reference solutions are downloaded. After compiling the tests can be run by using *ctest*

# CHAPTER 3

# Configuration

All configuration of UMD-LETKF is done through a single YAML configuration file. The UMD-LETKF, when run, will by default look for a configuration file in the same directory named *letkf.yaml*. Or, a different file can be specified on the command line

```
./letkfdriver <somefile.yaml>
```

An introduction to the YAML format can be found here.

**Warning:** In some places the UMD-LETKF does not perform extensive error checking on the configuration file, meaning an improperly defined configuration file might result in cryptic messages and crash. If you come across this, please *let me know* as I am slowly trying to make sure all yaml misconfigurations produce helpful error messages.

Each of the major sections of the configuration file are described below. **Each main section is required**, though within a section specific parameters may be optional.

## 3.1 localization

The localization class determines how spatial and temporal localization is performed, a crucial aspect of how an LETKF operates. This include localization for horizontal, vertical, temporal, and state variable components.

**Warning:** The localization configuration is perhaps the least finished part of UMD-LETKF. Things here will likely change quite a bit as localization methods are added for other domains, and/or a way to generically and flexibly specify localization is added.

#### Parameters

The following parameters are available regardless of which localization class is selected. Additional parameters, described in subsequent sections here, will be required depending on which localization class is selected.

**class** **type:** *string*, `required`

The name of the localization class to use.

Currently, UMD-LETKF has two built-in classes that can be used, additional localization classes can be implemented by the user. The following options are available:

- *loc_novrt* - A generic class that has no localization in the vertical, only in the horizontal.
- *loc_ocean* - Localization specific to the ocean.

---

**Note:** All localization radii defined below are given as a standard deviation of a Gaussian. (Even though they are implemented as a compact Gaspari-Cohn function)

---

### 3.1.1 loc_novrt

The `loc_novrt` localization class implements a basic horizontal-only localization. The bare minimum needed to have a working LETKF. No vertical localization is performed.

#### Parameters

**hzloc** **type:** *hzloc*, `required`

Horizontal localization specification used for all observation types.

#### Example

```
localization:
  class: loc_novrt
  hzloc:
    type: linearinterp_lat
    value:
    - {lat: 0.0,  radius: 500.0e3}
    - {lat: 90.0, radius:  50.0e3}
```

### 3.1.2 loc_ocean

The `loc_ocean` localization class implements a localization strategy specific to the ocean. Namely, satellite and insitu observations can be given a different horizontal localization radius (given the abundance of satellite observations compared to insitu, satellite observations should be given a smaller horizontal localization radius). Also, vertical localization of the satellite observations to just the ocean mixed layer, can be enabled

#### Parameters

**save_diag** **type:** *logical*, **default:** *true*

If true, diagnostic information specific to the ocean localization will be saved. See *loc_ocean Diagnostics* for more information on the fields that are saved.

**diag_file type:** *string*, **default:** *diag.loc_ocean.nc*

The file to which ocean localization diagnostics are saved, if `save_diag` is set to true. See *loc_ocean Diagnostics* for more information

**hzloc_prof type:** *hzloc*, `required`

The horizontal localization specification for insitu profiles.

Insitu profiles are determined to be the observations and platform types that are NOT included in the following `sat_obs` or `sat_plats` parameters.

**hzloc_sat type:** *hzloc*, `required`

The horizontal localization specification for satellite observations.

Satellite observations are determined to be the observations and platform types that are included in the following `sat_obs` or `sat_plats` parameters. For each observation, if its type matches one listed in `sat_obs`, or its platform type matches one listed in `sat_plats`, it is considered a satellite observation (it does not have to match both).

**tloc_prof type:** *float*, **default:** *-1.0*

Temporal localization for insitu profiles (in hours). If < 0, temporal localization is disabled.

**tloc_sat type:** *float*, **default:** *-1.0*

Temporal localization for satellite observations (in hours). If < 0, temporal localization is disabled.

**vtloc_surf type:** *vtloc*, **default:** *type=none*

The vertical localization specification for satellite observations.

Insitu profiles do not have any vertical localization.

**sat_obs type:** *array of strings*, **optional**

An array of observation names that are to be treated as satellite observation for localization purposes. See *Observation and Platform Names*.

**sat_plats type:** *array of strings*, **optional**

An array of platform names that are to be treated as satellite observations for localization purposed. See *Observation and Platform Names*

### `vtloc` Parameters

Specification of the vertical localization.

**type type:** *string*, **default:** *none*

The type of vertical localization to use for the ocean. Currently two options are available:

- **none** - vertical localization is off, observations impact the entire vertical column.
- **bkg_t** - surface observations are localized to the surface mixed layer, as calculated from a change in background temperature criteria.

**bkg_t_delta type:** *float*, `required`

The change in background temperature (Celsius) from the surface to some depth, used for calculating the depth of the ocean mixed layer.

**bkg_t_var** **type:** *string*

The name of the background temperature variable used for calculating the mixed layer depth. This state variable name must be one of those given in *state.statedef*.

#### Example

```
localization:
  class: loc_ocean
  save_diag: true
  hzloc_prof:
    type: linearinterp_lat
    value:
    - {lat: 0.0,  radius: 720.0e3}
    - {lat: 90.0, radius: 200.0e3}
  hzloc_sat:
    type: linearinterp_lat
    value:
    - {lat: 0.0,  radius: 500.0e3}
    - {lat: 90.0, radius: 50.0e3}
  sat_plats:
   - ocn_sat
  vtloc_surf:
    type: bkg_t
    bkg_t_delta: 0.2
    bkg_t_var: ocn_t
```

### 3.1.3 Common Types

#### `hzloc` Parameters

This parameter type is used to specify the characteristics of the horizontal localization.

**type** **type:** *string*, `required`

The type of horizontal localization to use. Currently, the only valid option is `linearinterp_lat`. This type gives a horizontal localization radius that changes with latitude. Several latitudes are specified, along with the desired radius, and linear interpolation is used to calculate the radius for any valid latitude.

**value** **type:** *array of lat/radius values*

An array of `lat` / `radius` pairs. See the example below for clarification.

- **lat:** absolute value of latitude in degrees
- **radius:** horizontal localization radius, meters. Given as the standard deviation of a Gaussian.

Note that all latitude values are positive. Currently, different values cannot be given for southern/northern hemisphere. If 0.0 and 90.0 are not included in the list of latitudes, they are implicitly added using the radius of the nearest given latitude.

#### Example

Note that in this example a latitudes between 0.0 degrees and 5.0 degrees have a localization radius of 500 km, all latitudes above 50.0 degrees have a radius of 100 km. In between they are appropriately linearly interpolated.

```
hzloc:
  type: linearinterp_lat
  value:
    - {lat:  5.0, radius: 500.0e3}
    - {lat: 10.0, radius: 300.0e3}
    - {lat: 50.0, radius: 100.0e3}
```

## 3.2 mpi

These parameters directly affect how UMD-LETKF scatters the model state across the processors and how it optimizes file I/O.

**Warning:** In an attempt to improve memory usage with the MPI calls, I over-optimized, resulting in overly slow performance in the MPI scatter/gather calls if a large domain or high number of ensemble members are used. This will be fixed in the future.

### Parameters

**ens_size** **type:** *integer*, `required`

The number of ensemble members.

**ppn** **type:** *integer*, **default:** *1*

The number of processors per node.

Optional, but may help improve I/O performance by evenly distributing across nodes the the PEs which perform simultaneous I/O.

### Example

```
mpi:
  ens_size: 20
  ppn: 10
```

## 3.3 observation

Parameters for specifying how the observations are read in, or how synthetic test observations are generated, are controlled under this section. The exact contents of this configuration section depends on which I/O `class` is used.

### Parameters

The following parameters are available regardless of which observation reader class is selected. Additional parameters, described in subsequent sections here, will be required depending on which localization class is selected.

**class** **type:** *string*, `required`

The name of the observation I/O class to use.

Currently, the UMD-LETKF has three built-in classes. Additional classes may be implemented by the user. The following options are available, and their specific configuration requirements are described in the following sections.

- *obsio_ioda* - Provides the ability to read observation files in the JEDI IODA format from the JEDI hofx application.
- *obsio_nc* - A generic NetCDF4 file reader.
- *obsio_test* - Generates synthetic observations from a specified increment value.

**Note:** If you're hooking up your model to the UMD-LETKF for the first time, you're best bet is to use the *obsio_test* reader first (to make sure everything else is hooked up correctly), before trying the *obsio_nc* or *obsio_ioda* readers with real observations.

#### Filename Ensemble Placeholder

Some of the classes below require filenames for the per-ensemble member observation input. In these cases the `#ENSX#` placeholder can be used within the string of the filename. It is replaced with the ensemble member number (starting at 1), padded with zeros to ensure the number is `X` digits long. For example `sst_obs.#ENS4#.nc` will be substituted as `sst_obs.0001.nc`, `sst_obs.0002.nc`, ...

#### Observation and Platform Names

The observation I/O classes require that names are given for different observations and platforms. These can be set to whatever the user wants, and their use can be considered optional. The exact name is not important, but may be referenced by other sections of the configuration (such as *localization*). As a general reference, the observation type should reflect which variable is observed (e.g. ocn_sst, ocn_t) and the platform type can reflect either specific platforms (e.g viirs, avhrr) or a general satellite vs. insitu. Hopefully this will make more sense when seen how it is used in the *localization* section.

**Note:** The observation and platform names should be short, with a 10 character max.

The following documentation describes the observation reader classes that are available for use.

### 3.3.1 obsio_ioda

This observation I/O class can be used to read observation operator output files from the Joint Effort for Data Assimilation Integration (JEDI) based applications. Files are in the IODA NetCDF format. (More of an explanation about this will likely be added once the JEDI repositories are made public.) Currently only the *hofx* or *hofx3d* applications are supported, not the *enshofx*. (Odds are you will be wanting to use the *hofx3d* application only anyway).

**Warning:** Efficient distribution of the read operations across PEs has not been implemented for this class. Large operational size datasets might be a little slow until this is fixed.

#### Parameters

**ioda_files** **type:** *ioda_file* list, `required`

This section contains a list of files that should be loaded, each with the following parameters:

#### `ioda_file` Parameters

Each set of ioda files to be read requires the following parameters:

**file** **type:** *string*, `required`

The base name of the file to read.

The notation of the *Filename Ensemble Placeholder* should be used since there should be separate files for each individual ensemble member. Also, JEDI applications currently produce output files for each PE of the application, so the filename given will automatically try appending the appropriate `_0001.nc`, `_0002.nc`, `...` suffixes.

**vars** **type:** *list of array(s)*, `required`

For each desired variable in the input file, an array is given with three values that have the following meaning:

1. observation name - see *Observation and Platform Names*
2. platform name - see *Observation and Platform Names*
3. variable name as given in the IODA observation file

#### Example

```
observation:
  class: obsio_ioda
  ioda_files:
  - file: mem#ENS1#/sst.out
    vars:
    - [ocn_sst, sst_viirs, sea_surface_temperature]
  - file: mem#ENS1#/insitu.out
    vars:
    - [ocn_s, insitu, sea_water_salinity]
    - [ocn_t, insitu, sea_water_temperature]
```

### 3.3.2 obsio_nc

The NetCDF reader will read in two types of files. The first is the main observation file given by the `filename_obs` parameter below and the format of which is described by *Observation File Format*. This file provides each observation type, location, and value. The second set of files are the per-ensemble member observation operator files, given by the `filename_obshx` parameter below and the format of which is described by *Observation H(x) File Format*.

---

**Note:** Although the configuration here allows for observation data that is common across all ensemble members to be specified in a separate `filename_obs` file, they do not have to be. All observation data could be in the per-ensemble member `filename_obshx` files. In this case, observation files should contain all the data required by both the *Observation H(x) File Format* and *Observation File Format* specs, and the `filename_obs` should simply point to one of the ensemble files.

---

### Parameters

**filename_obs type:** *string*, `required`

The name of the observation file to read in.

The expected contents of this NetCDF file are specified by *Observation File Format*.

**filename_obshx type:** *string*, `required`

The name of the per-ensemble observation operator file.

The *Filename Ensemble Placeholder* should be used to read in each individual ensemble member file. The expected contents of this file are specified by *Observation H(x) File Format*.

**obsdef type:** list of *obsplat_def*, `required`

Provides a mapping from the integer values of the observation type in the NetCDF file with a human readable name. See also *Observation and Platform Names*.

**platdef type:** list of *obsplat_def*, `required`

Provides a mapping from the integer values of the platform type in the NetCDF file with a human readable name. See also *Observation and Platform Names*.

**read_inc type:** *boolean*, `required`

If true, the values given in the per-ensemble member files are given as increments, $y^o - h(x)$, otherwise they are taken as the direct output of an observation operator, $h(x)$.

### `obsplat_def` Parameters

These parameters are required for the `obsdef` and `platdef` sections of *obsio_nc* and are used to associate a human readable `name` with the integer `id` that is stored in the NetCDF file

**name type:** *string*

Name of the observation or platform. Note the advice of *Observation and Platform Names*

**id type:** *integer*

The integer value in the NetCDF file.

**description type:** *string*

Optional description of the observation or platform type. Not needed by UMD-LETKF other than for the sanity of the user.

### Example

```
observation:
  class: obsio_nc
  obsdef:
  - name: ocn_t
    id:   2210
    description: "ocean insitu temperature (C)"
  - name: ocn_s
    id:   2220
    description: "ocean salinity (PSU)"
  platdef:
  - name: ocn_prf
```

(continues on next page)

(continued from previous page)

```
    id:   1
    description: "all insitu obs"
  - name: ocn_sat
    id:   1000
    description: "all satellite based obs"
  filename_obs:   obs.nc
  filename_obshx: "obs.#ENS4#.nc"
  read_inc: false
```

### Observation File Format

The NetCDF file containing observation data needs to contain the following dimensions and variables of the same name. An example file can be found in the test data for UMD-LETKF.

---

**Note:** I realize the variable "depth" is required and that that "height" is not a valid option. Since UMD-LETKF was started for ocean DA, this will be addressed once non-ocean localization classes are implemented.

---

### dimensions

**obs** Number of observations in the file

### variables

All variables here are of size `obs`

**depth type:** *float*

The depth of the observation in meters.

**err type:** *float*

The standard deviation of the observation error

**hr type:** *float*

The time offset (in hours) from the analysis time. Only actually used if temporal localization is used.

**lat type:** *float*

Latitude in degrees

**lon type:** *float*

Longitude in degrees

**obid type:** *integer*

The observation id. See *obsplat_def*

**plat type:** *integer*

The platform id. See *obsplat_def*

**qc type:** *integer*

Quality control flag. Observation is used by UMD-LETKF only if `qc` is zero.

**val type:** *float*

The value of the observation.

#### Observation H(x) File Format

The NetCDF file containing per-ensemble member observation operator data needs to contain the following dimensions and variables of the same name. An example file can be found in the test data for UMD-LETKF.

#### dimensions

**obs** Number of observations in the file.

#### variables

All variables here are of size `obs`

**hx type:** *float*

The value of the observation operator from a single ensemble member background. This can either contain the value ($h(x)$), or the observation increment ($y^o - h(x)$), depending on the value of `read_inc` in *obsio_nc*

### 3.3.3 obsio_test

This observation I/O class can be used to generate synthetic observations from the state background mean using a specified increment. This method can be useful when wanting to perform a quick single-obs test, bypassing the need to generate observation files. Test observations can only be generated directly from the state background (i.e. the identity observation operator is used.)

#### Parameters

**synthetic_obs** This section contains an array of arrays (see the example below if that doesn't make sense). Each observation specification contains an array of nine values, in the following order

1. **observation_id** - A string reflecting the type of observation (see *Observation and Platform Names*).
2. **platform_id** - A string reflecting the type of platform (see *Observation and Platform Names*).
3. **state_variable** - The state variable that this observation is generated from. The value given must be one of the name of one of the state variables given in the *state.statedef* section.
4. **latitude** - in degrees
5. **longitude** - in degrees
6. **depth/height** - The value in the vertical coordinate. If this observation is being generated from a 2D surface state field then the depth/height here is ignored.
7. **time** - the time offset (in hours) from the analysis time. This value is only used if temporal localization is enabled.
8. **increment** - The value of this observation will be generated as the increment plus the background
9. **error** - standard deviation of the observation error

### Example

This example generates two observations from the background temperature, both with an observation increment of 1 degree and observation error of 0.2 degree.

```
observations:
  class: obsio_test
  synthetic_obs:
  - [ocn_sst, satellite, ocn_t, 20.0, -140.0,  0.0, 0.0, 1.0, 0.2]
  - [ocn_t,   insitu,    ocn_t, 25.0, -162.0, 10.0, 0.0, 1.0, 0.2]
```

## 3.4 solver

Parameters for the core LETKF solver, diagnostic output, and covariance inflation.

### Parameters

**save_diag** **type:** *boolean*, **default:** *true*

Whether a diagnostic file is saved at the end of the UMD-LETKF run.

This file contains diagnostic information such as the number of observations used per grid cell, maximum horizontal localization radius, etc. See *LETKF Solver Diagnostics*

**diag_file** **type:** *string*, **default:** *"diag.solver.nc"*

The file name of the diagnostic output.

Only used if `save_diag` is set to true.

### Example

```
solver:
  save_diag: true
  diag_file: "diag.solver.nc"
```

### 3.4.1 inflation

Covariance inflation methods for increasing the ensemble spread. This section is optional. If not provided the default of "no inflation" will be used.

**Note:** The *RTPS* and *RTPP* methods cannot be enabled simultaneously.

### Parameters

**mul** **type:** *float*, **default:** *1.0*

The amount of *Multiplicative* inflation to apply.

Valid parameters are greater than or equal to 1.0. Value of 1.0 indicates multiplicative inflation is off.

**rtpp** **type:** *float*, **default:** *0.0*

The percentage of *Relaxation to Prior Perturbations (RTPP)* to apply.

Valid parameters are between 0.0 and 1.0. Value of 0.0 indicates RTPS is off, 1.0 indicates the analysis ensemble perturbations are relaxed 100% back toward the background perturbation. Unless you know what you are doing, you are better off using *Relaxation to Prior Spread (RTPS)*. Cannot be used if RTPS is enabled

**rtps** **type:** *float*, **default:** *0.0*

The percentage of *Relaxation to Prior Spread (RTPS)* to apply.

Valid parameters are between 0.0 and 1.0. A value of 0.0 indicates RTPS is off, 1.0 indicates analysis spread is relaxed 100% back toward the background spread. Values between 0.5 and 0.8 are often good choices. You could in theory use values greater than 1.0 to result in analysis spread that is larger than background spread, but I have no idea why you would want to do this. Cannot be used if RTPP is enabled.

### Example

```
solver:
  inflation:
    rtps: 0.6
    rtpp: 0.0
    mul:  1.0
```

## 3.5 state

This section defines what the model state looks like, both in terms of the state variables (*statedef* section) and the horizontal/vertical grid (*hzgrid* / *vtgrid* sections). The specifics of what is in each subsections may include additional parameters, depending on the stateio class being used (but everything is currently identical for the two builtin state I/O classes provided by default.)

### File Specification Format

Wherever the `filevar` type is used in the sections below, the following format is used to define which file and variable name the data is pulled from.:

```
{file: file_name, variable: variable_name}
```

### Parameters

The following parameters are available regardless of which state I/O class is selected. Additional parameters, described in subsequent sections here, will be required depending on which state I/O class is selected.

**class** **type:** *string*, `required`

The name of the I/O class used to handle the state.

Currently, UMD-LETKF has two built-in classes that can be used, additional stateio can be implemented by the user. The following options are available:

- **stateio_grib** - Handles state I/O through grib2 formatted files. Word of caution: this has not been well tested yet! But it should work, I think. Only available if built with the *-DLETKF_ENABLE_GRIB=ON* option.
- **stateio_nc** - Handles state I/O through NetCDF formatted files.

**hzgrid type:** *hzgrid* `required`

The definitions of the horizontal grid(s).

**statedef type:** *statedef* `required`

The definitions of the state variables.

**verbose type:** *boolean*, **default:** *false*

Sets if diagnostic information is printed to the console.

If true, diagnostic information indicating which processor is responsible for reading or writing which file will be displayed.

**vtgrid type:** *vtgrid* `required`

The definitions of the vertical grid(s).

#### Example

```
state:
  class: stateio_nc
  verbose: true
```

### 3.5.1 hzgrid

One or more horizontal grid specification(s) are given by defining how to read the latitude, longitude, and optional mask. Each grid will have the following parameters.

---

**Note:** Currently only **one** horizontal grid can be specified. Although models often produce some variables on a staggered grid, you can still use the lat/lon of the grid center, and assuming the localization radius is not too small, this limitation should make very little difference.

---

#### Parameters

In the following parameters for the horizontal grid specification, at least one (or both) of the `1d` and `2d` set of parameters needs to be defined for latitude and longitude.

**name type:** *string*, `required`

A unique name for the horizontal grid.

The exact name doesn't really matter, but it is referenced in the subsequent *statedef* sections for assigning a horizontal grid to each state variable.

**lat2d type:** *filevar*

Specifies the 2D latitude grid, in degrees.

**lat1d** **type:** *filevar*

Specifies the 1D latitude grid, in degrees.

The latitude for each row of the grid will be identical. If `lat2d` is specified as well, `lat1d` will only be used as the nominal latitude for the output files. It will not be used to determine lat/lon for each grid-point in the LETKF algorithm.

**lon2d** **type:** *filevar*

Specifies the 2D longitude grid, in degrees.

**lon1d** **type:** *filevar*

Specifies the 1D longitude grid, in degrees.

The longitude for each column of the grid will be identical. If `lon2d` is specified as well, `lon1d` will only be used as the nominal longitude for the output files. It will not be used to determine lat/lon for each grid-point in the LETKF algorithm.

**mask** **type:** *filevar*, **(optional)**

Specifies the optional mask.

The mask is optional, but can increase the UMD-LETKF speed in domains such as the ocean where land points should skipped over. For the input data, grid-points with values of 0.0 are masked out and not used.

#### Example

In the following example, a single horizontal grid named `hz1` is specified, the latitude, longitude, and mask of the grid are obtained from the appropriate variables of the `grid/ocean.hgrid.nc` file.

```
state:
  hzgrid:
  - name: hz1
    lat2d: {file: grid/ocean.hgrid.nc, variable: geolat}
    lon2d: {file: grid/ocean.hgrid.nc, variable: geolon}
    lat1d: {file: grid/ocean.hgrid.nc, variable: lath}
    lon1d: {file: grid/ocean.hgrid.nc, variable: lonh}
    mask:  {file: grid/ocean.hgrid.nc, variable: wet}
```

### 3.5.2 vtgrid

Definitions for depth/height information of the vertical grid(s) are specified here. One or more sets of vertical grids can be defined.

#### Parameters

**name** **type:** *string*, `required`

A unique name for the vertical grid.

The exact name doesn't really matter, but it is referenced in the subsequent *statedef* sections for assigning a vertical grid to each state variable.

**vert0d** `not yet implemented`

**vert1d** **type:** *filevar*

Vertical coordinates for a column that don't vary in the horizontal direction.

**vart2d** `not yet implemented`

**vert3d** `not yet implemented`

---

**Note:** `vert1d` can also use a constant value specification for now, see the following example. This is needed for surface fields, and is a temporary work around until the `vert0d` parameter is implemented.

---

#### Example

In the following example one vertical grid named `vt1` is specified for the 3D variables, and another `vt_surf` is specified with a constant value (surface) for the surface only variables

```
state:
  vtgrid:
  - name: vt1
    vert1d: {file: Vertical_coordinate.nc, variable: Layer}
  - name: vt_surf
    vert1d: {constant: 0.0}
```

### 3.5.3 statedef

This section defines one or more state variables. It defines what the state variables are that should be read and written by UMD-LETKF, which grid specification they use, and if there are any optional bounds checking on the final state value or the analysis increment that is applied to the background.

#### Filename String Placeholders

The `input` and `output` parameters below can use special placeholders in the filename string that get replaced at run-time.

- `#ENSX#` This placeholder is replaced with the ensemble number (starting at 1), padded with zeros to ensure the number is of length `X`. This can also be replaced with `mean` or `sprd` for the ensemble mean and spread output files.
- `#TYPE#` This placeholder is replaced with either `ana` or `bkg` if the output file is for the analysis or background.

As an example, the specification string `ocn.#TYPE#.#ENS4#.nc` will be used to generate the following files `ocn.bkg.mean.nc`, `ocn.bkg.sprd.nc`, `ocn.ana.mean.nc`, `ocn.ana.sprd.nc`, `ocn.ana.0001.nc`, `ocn.ana.0002.nc`, `ocn.ana.0003.nc`, `...`

#### Parameters

**name** **type:** *string*, `required`

A unique name for the state variable.

The exact name doesn't really matter, but it may be referenced in other sections of the configuration (such as localization)

**hzgrid type:** *string*, `required`

The name of the horizontal grid to use from the *hzgrid* section of the configuration file.

The x/y dimensions of the input data given below must match the dimensions given for the specified horizontal grid.

**vtgrid type:** *string*, `required`

The name of the vertical grid to use from the *vtgrid* section of the configuration file.

The z dimension of the input data given below must match the dimensions given for the specified vertical grid.

**input type:** *filevar*, `required`

The file and variable name of per-ensemble background data.

Each ensemble member is assumed to be in a separate file, and so the input filename should use the #ENSX# placeholder. (See *Filename String Placeholders*)

**output type:** *filevar*, `required`

The file and variable name of the per-ensemble, and mean/spread data.

Each ensemble member is assumed to be in a separate file, and so the output filename should used the #ENSX# and #TYPE# placeholders (see *Filename String Placeholders*). In addition to the analysis per-ensemble output, this handles the mean and spread output files for the analysis and background.

**ana_bounds type:** *float[2]*, **(optional)**

The bounds to which the final analysis should be clamped.

**ana_inc_max type:** *float*, **(optional)**

The maximum absolute value allowed for the analysis increment.

Any increment with an absolute value greater than this will be clamped (respecting the original sign of the increment).

### Example

```
state:
  statedef:
  - name: ocn_s
    hzgrid: hz1
    vtgrid: vt1
    ana_bounds: [0, 50.0]
    ana_inc_max: 2
    input:  {variable: salt, file: "ocn.bkg.#ENS4#.nc"}
    output: {variable: salt, file: "ocn.#TYPE#.#ENS4#.nc"}
  - name: ocn_ssh
    hzgrid: hz1
    vtgrid: vt_surf
    input:  {variable: ssh, file: "ocn.bkg.#ENS4#.nc"}
    output: {variable: ssh, file: "ocn.#TYPE#.#ENS4#.nc"}
```

## 3.6 Example Configuration

The following is a complete example yaml configuration file. This exact file is used by the `ocean.vtloc` ctest to test the vertical localization method of the ocean data assimilation.

```
---
mpi:
  ens_size: 10
  ppn: 1


solver:
  inflation:
    rtps: 0.0
    rtpp: 0.0
    mul: 1.0


state:
  class: stateio_nc
  verbose: false
  compression: 0

  hzgrid:
  - name: hz1
    lat2d: {file: grid/ocean.hgrid.nc, variable: geolat}
    lon2d: {file: grid/ocean.hgrid.nc, variable: geolon}
    lat1d: {file: grid/ocean.hgrid.nc, variable: lath}
    lon1d: {file: grid/ocean.hgrid.nc, variable: lonh}
    mask:  {file: grid/ocean.hgrid.nc, variable: wet}

  vtgrid:
  - name: vt1
    vert1d: {file: grid/ocean.vgrid.nc, variable: Layer}

  statedef:
  - name: ocn_t
    hzgrid: hz1
    vtgrid: vt1
    input:  {file: "bkg/bkg.#ENS4#.nc", variable: Temp}
    output: {file: "#TYPE#.#ENS4#.nc",  variable: Temp}

  - name: ocn_s
    hzgrid: hz1
    vtgrid: vt1
    input:  {file: "bkg/bkg.#ENS4#.nc", variable: Salt}
    output: {file: "#TYPE#.#ENS4#.nc",  variable: Salt}

  - name: ocn_u
    hzgrid: hz1
    vtgrid: vt1
    input:  {file: "bkg/bkg.#ENS4#.nc", variable: u}
    output: {file: "#TYPE#.#ENS4#.nc",  variable: u}

  - name: ocn_v
    hzgrid: hz1
    vtgrid: vt1
```

(continues on next page)

(continued from previous page)

```
    input:  {file: "bkg/bkg.#ENS4#.nc", variable: v}
    output: {file: "#TYPE#.#ENS4#.nc",  variable: v}


localization:
  class: loc_ocean
  save_diag: true

  hzloc_prof:
    type: linearinterp_lat
    value:
    - {lat: 0.0,  radius: 720.0e3}
    - {lat: 90.0, radius: 200.0e3}

  hzloc_sat:
    type: linearinterp_lat
    value:
    - {lat: 0.0,  radius: 500.0e3}
    - {lat: 90.0, radius: 50.0e3}

  sat_obs:
  sat_plats:
  - ocn_sat

  vtloc_surf:
    type: bkg_t
    bkg_t_delta: 0.2
    bkg_t_var: ocn_t


observation:
  class: obsio_test

  synthetic_obs:
  - [ocn_t, ocn_sat, ocn_t, 20.0, -140.0, 5.0, 0.0, -1.0, 0.1]
  - [ocn_t, ocn_prf, ocn_t,  0.0, -140.0, 5.0, 0.0, -1.0, 1.0]
  - [ocn_t, ocn_prf, ocn_t,  0.0, -150.0, 5.0, 0.0, -1.0, 1.0]
  - [ocn_t, ocn_prf, ocn_t,  0.0, -160.0, 5.0, 0.0, -1.0, 1.0]
  - [ocn_t, ocn_prf, ocn_s,  0.0, -160.0, 5.0, 0.0,  0.5, 1.0]
```

# CHAPTER 4

# Diagnostic Output

## 4.1 LETKF Solver Diagnostics

The core LETKF solver will output several fields of diagnostic information at the end of the program if enabled in the yaml configuration file. See *solver.save_diag* and *solver.diag_file* configuration sections.

These help get a sense of how many observations are being used by each grid-point.

- **col_maxhz** - the maximum horizontal search radius for the grid column
- **col_obscount** - the number of observations, for each grid column, that were found within the given col_maxhz radius
- **lg_obscount** - the subset of observations that were allowed to be used for each localization group
- **lg_obsloc** - the sum of the localization values for observations used by each localization group. A single observation can have a localization value between *0.0* and *1.0*. This gives a general sense of the amount of impact observations have.

## 4.2 loc_ocean Diagnostics

If the *loc_ocean* is used, additional fields will be saved to diagnose how vertical localization is performed.

- **vtloc_surf_lvl** - the depth, in number of levels, of the surface localization group
- **vtloc_surf_depth** - The depth, in meters, of the surface localization group

# CHAPTER 5

# Reference

## 5.1 LETKF Algorithm

The following is a brief conceptual overview from [Sluka2016] of how the LETKF algorithm operates, for a complete description see [Hunt2007].

The local ensemble transform Kalman filter (LETKF) is a type of ensemble Kalman filter (EnKF) which uses an ensemble of forecasts $\left\{\mathbf{x}^{b(i)} : i = 1, 2, ..., k\right\}$ to determine the statistics of the background error covariance. This information is combined with new observations, $\mathbf{y}^o$, to generate an analysis mean, $\bar{\mathbf{x}}^a$, and a set of new ensemble members, $\mathbf{x}^{a(i)}$. First, the model state is mapped to observation space by applying a nonlinear observation operator $H$ to each background ensemble member

$$\mathbf{y}^{b(i)} = H\mathbf{x}^{b(i)}$$

note, that the application of the observation operator is applied *outside* this UMD-LETKF library.

A set of intermediate weights, $\bar{\mathbf{w}}^a$ are calculated to find the analysis mean $\bar{\mathbf{x}}^a$

$$\tilde{\mathbf{P}}^a = \left[(k-1)\,\mathbf{I} + \left(\mathbf{Y}^b\right)^T \mathbf{R}^{-1}\mathbf{Y}^b\right]^{-1}$$

$$\bar{\mathbf{w}}^a = \tilde{\mathbf{P}}^a \left(\mathbf{Y}^b\right)^T \mathbf{R}^{-1} \left(\mathbf{y}^o - \bar{\mathbf{y}}^b\right)$$

$$\bar{\mathbf{x}}^a = \bar{\mathbf{x}}^b + \mathbf{X}^b\bar{\mathbf{w}}^a$$

where $\bar{\mathbf{x}}^b$ and $\bar{\mathbf{y}}^b$ are the ensemble mean of the background in model space and observation space, respectively. $\mathbf{X}^b$ and $\mathbf{Y}^b$ are the matrices whose columns represent the ensemble perturbations from those means, and $\mathbf{R}$ is the observation error covariance matrix.

Last, the set of intermediate weights, $\mathbf{W}^a$ are calculated to find the perturbations in model space for the analysis ensemble by

$$\mathbf{W}^a = \left[(k-1)\,\tilde{\mathbf{P}}^a\right]^{1/2}$$

$$\mathbf{X}^a = \mathbf{X}^b\mathbf{W}^a$$

the final analysis ensemble members, $\mathbf{x}^{a(i)}$, are the result of adding each column of $\mathbf{X}^a$ to $\bar{\mathbf{x}}^a$

## 5.2 Inflation Schemes

In order to account for an underdispersive ensemble, several multiplicative inflation schemes have been implemented in UMD-LETKF (with hopefully more to be implemented eventually). If you're not sure which one to pick, it is usually safest to choose *Relaxation to Prior Spread (RTPS)* with a value between 0.0 and 1.0.

### 5.2.1 Multiplicative

The inflation factor $\alpha$, which is greater than or equal to 1.0, increases the magnitude of the analysis perturbations.

$$\mathbf{x}_i^{'a} \leftarrow \alpha \mathbf{x}_i^{'a}$$

This method works sufficiently for domain that are regularly sampled by observations. (e.g. the atmosphere). If a domain is **not** sufficiently sampled (such as the deep ocean), this method may result in the ensemble spread growing far too rapidly and the filter ultimately diverging.

### 5.2.2 Relaxation to Prior Perturbations (RTPP)

The perturbations of the analysis, $\mathbf{x}_i^{'a}$ are relaxed a percentage, $\alpha$, back to the background perturbations, $\mathbf{x}_i^{'b}$ [Zhang2004]. This has the benefit of effectively being a combination of both multiplicative, and additive inflation.

$$\mathbf{x}_i^{'a} \leftarrow (1-\alpha)\,\mathbf{x}_i^{'a} + \alpha \mathbf{x}_i^{'b}$$

### 5.2.3 Relaxation to Prior Spread (RTPS)

The *spread* of the analysis, $\sigma^a$, is relaxed a percentage of the way, $\alpha$, back to the spread of the background, $\sigma^b$ [Whitaker2012].

$$\mathbf{x}_i^{'a} \leftarrow \mathbf{x}_i^{'a} \left( \alpha \frac{\sigma^b - \sigma^a}{\sigma_b} + 1 \right)$$

## 5.3 Custom Plugins

In addition to the built-in classes that are provided for *localization*, *observation*, and *state*, an API is provided allowing a user to code their own classes without have to make any changes to the core LETKF library code.

**Todo:** Obviously this needs documentation, something I will add once the need arises.

## 5.4 Citations

CHAPTER 6

# Support

To stay up to date with latest version of UMD-LETKF, watch this repository on GitHub. For questions on how to use, please contact Travis Sluka at tsluka@umd.edu. For any bugs or feature requests, create an issue on the GitHub Issues page for the repository.

# Bibliography

[Hunt2007] Hunt, B. R., Kostelich, E. J., & Szunyogh, I. (2007). Efficient data assimilation for spatiotemporal chaos: A local ensemble transform Kalman filter. Physica D: Nonlinear Phenomena, 230(1-2), 112–126. http://doi.org/10.1016/j.physd.2006.11.008

[Miyoshi2005] Miyoshi, T. (2005). Ensemble Kalman Filter Experiments with a Primitive-Equation Global Model. University of Maryland. Retrieved from http://hdl.handle.net/1903/3046

[Sluka2016] Sluka, T. C., Penny, S. G., Kalnay, E., & Miyoshi, T. (2016). Assimilating atmospheric observations into the ocean using strongly coupled ensemble data assimilation. Geophysical Research Letters, 43(2), 752–759. https://doi.org/10.1002/2015GL067238

[Whitaker2012] Whitaker, J. S., & Hamill, T. M. (2012). Evaluating Methods to Account for System Errors in Ensemble Data Assimilation. Monthly Weather Review, 140(9),3078–3089. https://doi.org/10.1175/MWR-D-11-00276.1

[Zhang2004] Zhang, F., Snyder, C., & Sun, J. (2004). Impacts of Initial Estimate and Observation Availability on Convective-Scale Data Assimilation with an Ensemble Kalman Filter. Monthly Weather Review, 132(5), 1238–1253. https://doi.org/10.1175/1520-0493(2004)132<1238:IOIEAO>2.0.CO;2