Joint Effort for Data assimilation Integration

IODA subsystem

Joint Center for Satellite Data Assimilation (JCSDA)

JEDI Academy – November 14, 2018

Requirements and Goals

From Yannick's presentation:

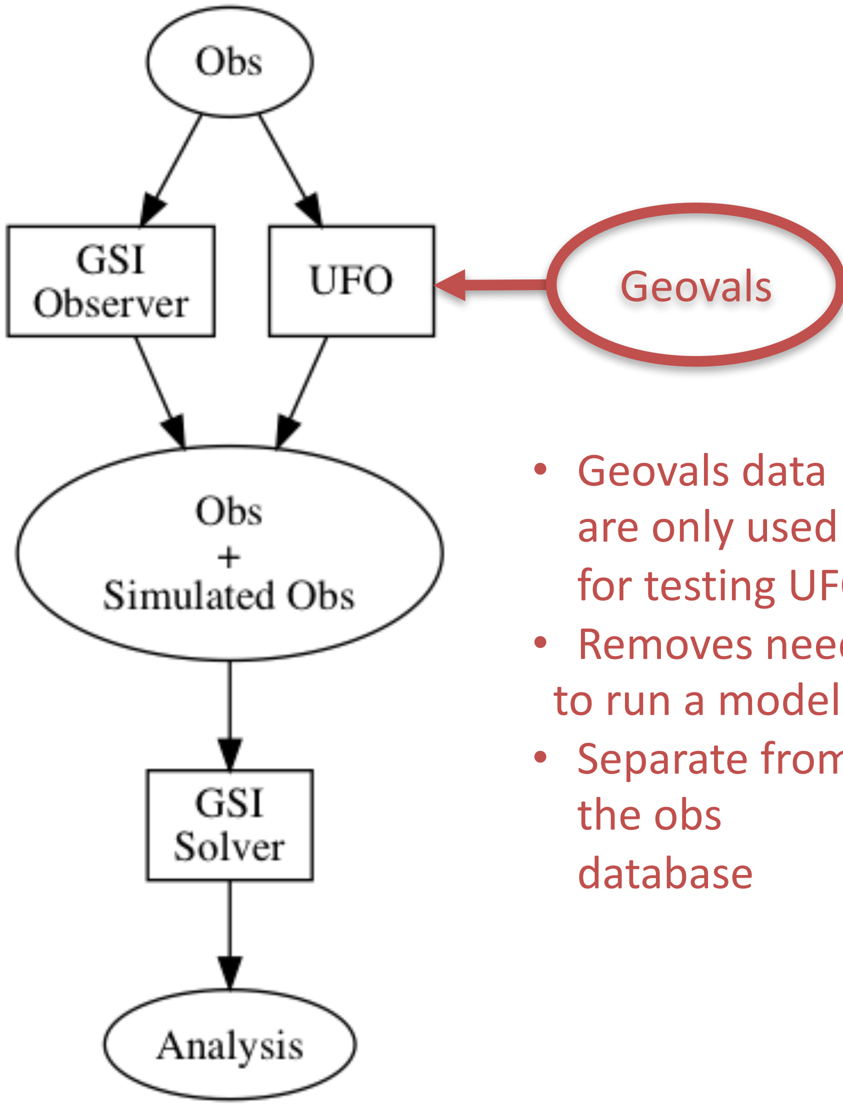
- **Interface to isolate science code from data storage**

- Three levels:
  - Long term storage (historic database)
  - Files on disk (one DA cycle)
  - In memory handling of observations (hardware specific?)

- Two environments:
  - Plotting, analyzing, verifying on workstation
  - DA and other HPC applications (MPI, threads, GPUs…)

- Goal: one interface, possibly several implementations?

Observation Data Flow



- Geovals data are only used for testing UFO
- Removes need to run a model
- Separate from the obs database

- For development purposes, we have a flow that places UFO in parallel with the GSI Observer.
  - UFO can be checked by running the same data through the GSI Observer
- Observation operators in UFO (or the GSI Solver) create simulated observations from model fields.
- The original observations plus the simulated observations are passed onto the GSI Solver to complete the generation of the analysis state.

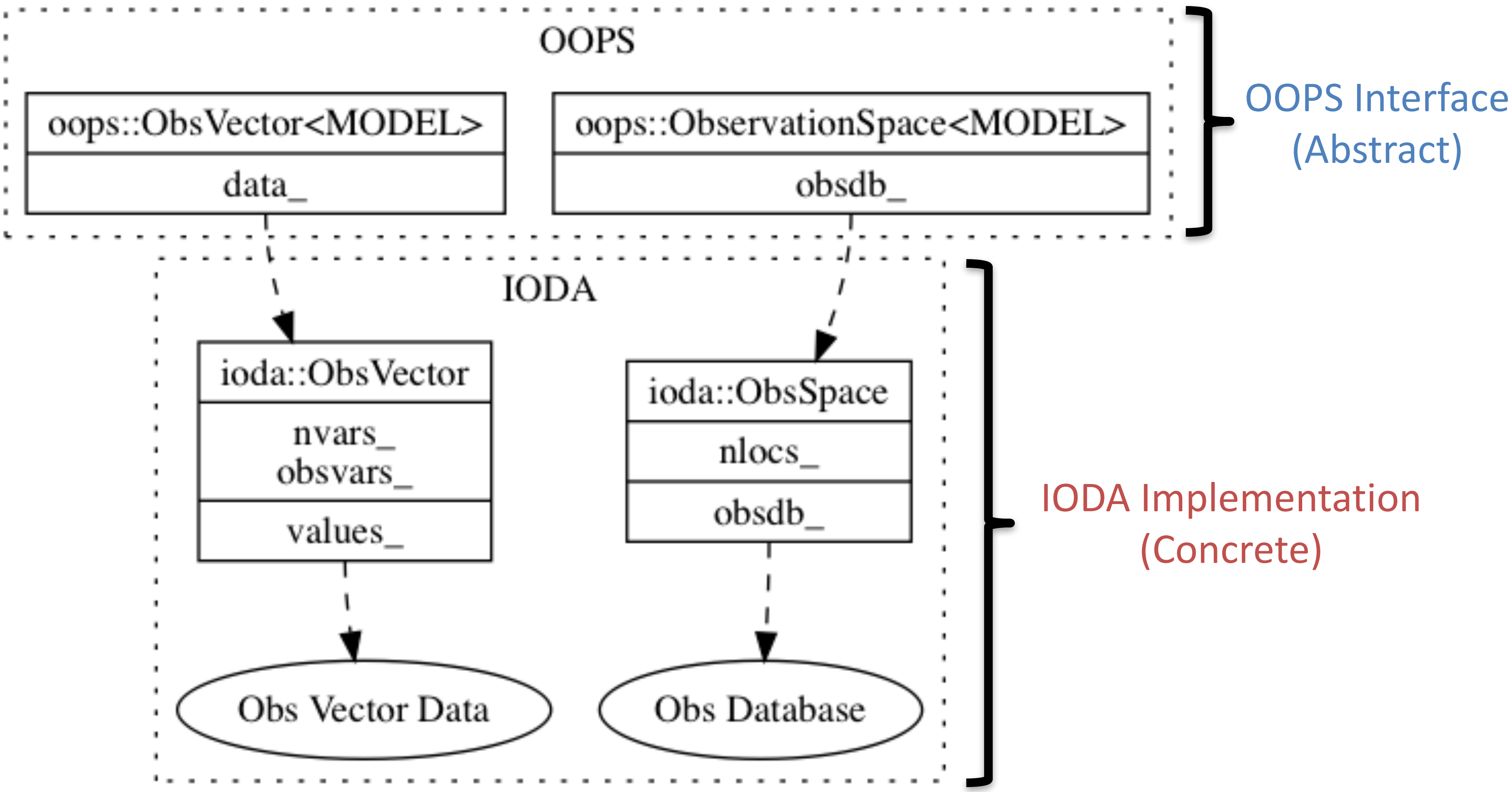
Handling Observation Data

- Currently, we have a prototype observation data store
  - Netcdf, ODB API
  - C++ and Fortran interfaces provided
  - Enables us to have access to small amounts of observation data and therefore continue development of the other JEDI subsystems
- We require a full-fledged database implementation which can handle large amounts of data and operate in an HPC environment (MPI)
  - SQL-like interface
- Now that we have the prototype implementation, we can switch our focus to the long-term database implementation

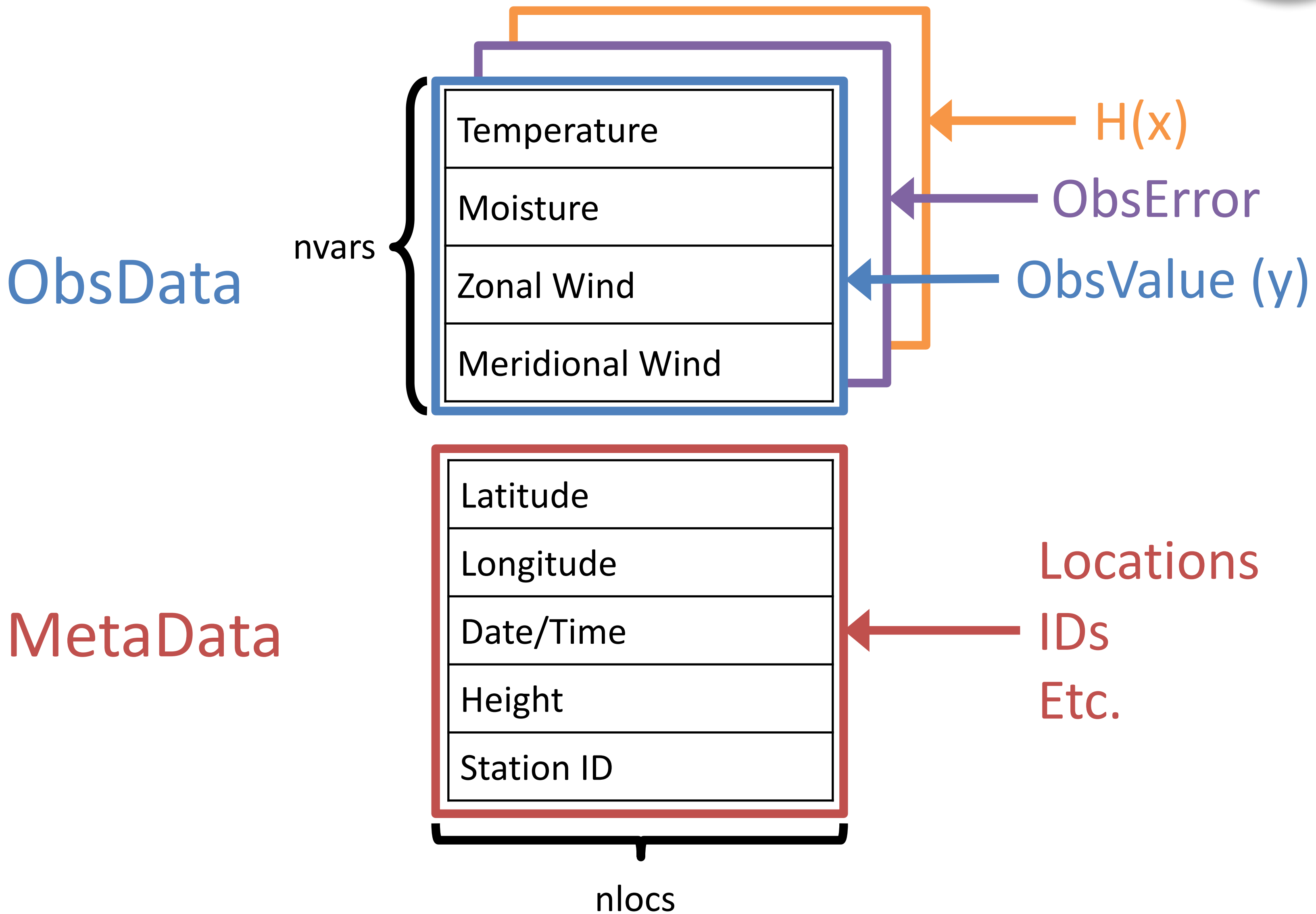
Relationship to OOPS

- IODA is the implementation of the following abstract interfaces in OOPS
- ObsVector
  - ObsVector represents the observation terms H(x) and y in the J(x) cost function
- ObservationSpace
  - Set of ObsVectors
  - Additional meta-data about the particular obs type
    - Radiosonde station ID’s, satellite scan angle and scan position, etc.

Class Structure



Observation database schematic



IODA interface with OOPS

- C++
- Access is through the ObsVector class
  - Corresponds to ObsData tables shown in he database schematic
    - ObsValue, ObsError, H(x)
- ObsVector methods

```
void read(const std::string &);
void save(const std::string &) const;
```

  - Argument is name of the ObsData table (ObsValue, H(x), etc.)
  - ObsVector implementation manages what variables constitute a vector underneath the hood.

IODA interface with UFO

- Fortran
- Access is through the ObsSpace class
  - Corresponds to individual rows in the database schematic
- ObsSpace methods

```
integer function obsspace_get_nlocs(obss)
subroutine obsspace_get_db(obss, group, vname, vect)
subroutine obsspace_put_db(obss, group, vname, vect)
```

  - obss argument is a C pointer to an ObsSpace object
  - group argument is a Fortran string holding the database table name
    - Eg., “ObsValue”, “ObsError”, “MetaData”
  - vname argument is a Fortran string holding the variable (row) name
    - Eg., “air_temperature”, “latitude”
  - vect argument is a Fortran 1D array (vector) of doubles

IODA-UFO Fortran interface example

- It is the client's responsibility to allocate memory for the vector data
- Rows of the tables are nlocs in length
- Radiance example:

```
real(kind_real), allocatable :: Sensor_Zenith_Angle(:)
real(kind_real), allocatable :: Scan_Angle(:)
integer :: nlocs

 nlocs = obsspace_get_nlocs(obss)        ! All table rows are nlocs in length
 allocate(Sensor_Zenith_Angle(nlocs))
 allocate(Scan_Position(nlocs))

 call obsspace_get_db(obss, "MetaData", "Sat_Zenith_Angle", Sensor_Zenith_Angle)
 call obsspace_get_db(obss, "MetaData", "Scan_Angle", Scan_Angle)

 deallocate(Sensor_Zenith_Angle)
 deallocate(Scan_Angle)
```