Rough class specifications for DART

I. obs_sequence (Called obs stream by operational folks)

This is the highest level class on the observations side.

Defines type obs_sequence_type
Has file_id
file_pointer
status is in or out or both
obs and defs or just defs

1. function init_output_obs_sequence(file_name, additional file level meta_data, number of copies of obs associated with each obs_set, meta_data for the individual copies ...) returns an obs_sequence_type that is initialized and ready for output. The global metadata for the output file is written. At this level it must not be model or observation type dependent. the third argument specifies the number of values of actual observations that will be output with the file, with zero indicating it will just be observation definitions. This is useful for setting up observation definition files for synthetic observation type experiments. Need to decide on action if the file_name already exists. For initial implementation, the file level meta_data is just a string describing the file. The meta_data for the individual copies of the obs should also initially be just a string describing the individual observations sets meanings, for instance ensemble member number, prior, posterior, truth, regular obs, or whatever.

2. function init_input_obs_sequence(file_name) returns an obs_sequence_type which is attached to the obs_sequence format file with file_name. Things are initialized so that the obs_sequence_type is ready to return the first obs_set in the file. If the file does not exist or if its initial global metadata is invalid execution should terminate through the error handler.

3. function number_of_data(obs_sequence_type) returns an integer, 0 if no data, otherwise number of copies of obs data (for ensembles, prior, posterior, etc) at each time. This returns the number of observation values associated with each obs_set.

4. function get_global_meta_data(obs_sequence_type) returns the global metadata associated with an obs_sequence. For initial implementation this is just a descriptive string.

5. function get_num_obs_copies(obs_sequence_type) returns the number of copies of observation data associated with each observation set. For just a specification this is 0, for a standard obs_sequence file it is 1, but it might be larger for files that contain truth, noisy obs, prior, posterior, ensemble members, etc.

6. function get_copy_meta_data(obs_sequence_type, index) returns the meta-data associated with the indexth copy of the observations. Error if index exceeds number of copies. For initial implementation meta-data is just a text string describin what this particular copy contains.

7. function end_of_sequence(obs_sequence_type) returns true if the pointer is at the end of the file.

8. function terminate_obs_sequence(obs_sequence_type) terminates output or input, closes files.

9. function init_inout_obs_sequence(file_name, global meta_data, number of copies of obs associated with each obs_set, meta_data for the individual copies) returns an obs_sequence_type but this one can have stuff dynamically appended (required for targeted observations etc.). Not recommended for initial implementation but must be supported in final product.

10. function output_obs_set(obs_sequence_type, obs_set_type, optional index) writes the obs_set to the file associated with this obs_sequence_type. An error occurs if this obs_set is not in proper time order for this sequence; need to determine exactly what action should be taken. If only the obs_set_def is to be output, index should have the value 0. If index is not included, it is assumed that the data (time and observation values) are for the first copy. If an index > 1 is included, it indicates that data is for this copy. For initial implementation, insist that calls to output_obs_set be in index order (1, 2, ...n) and that all calls for a particular obs_set be made before calls for any other obs_set. Should check for compliance, but for initial implementation may just trust to users care.

11. function get_next_obs_set(obs_sequence_type, optional prev_obs_set???, optional index) : Returns the next obs_set from the obs_sequence. The obs_sets are ordered sequentially by the end of the time interval that they span. In the simple case, they are simply ordered by the scalar time at which observations are taken. The obs_sequence module is assumed to have some global state that keeps track of what was the last observation set returned. However, an optional argument that gives an obs_set and asks for the next obs_set might also be supported. If this is implemented, need to document what doing this does to the global state pointer (probably best to reset it to this point?). The optional index allows access to sequences associated with different copies of the observations. An index of 0 returns only the obs_set_def information and no data, no index or a value of 1 returns the data associated with the first copy, an index greater than 1 returns data associated with the corresponding copy (error if number available is exceeded).

12. function get_first_obs_after(obs_sequence_type, time_type) returns obs_sequence with pointer set to the first observation in this obs_sequence whose observation interval begins after the time_type. Not needed for initial implementation.

13. function get_last_obs_before(obs_sequence_type, time_type) returns obs_sequence with pointer set to last observation in this obs_sequence whose observation interval ends before the time_type. Need to clarify action if no such obs_set exists. Not needed for initial implementation.

14. Additional search and query options might be needed eventually.

II. Obs_set

Defines a set of observations that are all associated with the same time interval and the associated observation values. This pushes time interpolation to this rather high level. This should be re-evaluated at design review.

Defines type obs_set_type

composed of
values of observations
flag indicating whether observations are present or if this is just definition
generalized_time representing interval over which observation is taken
obs_set_def_type

1. function get_obs_set_time(obs_set_type :: set): Returns a generalized_time_type that describes the characteristics of this set of observations.

2. function contains_data(obs_set_type) returns logical, true if this set has observations and false if it only has a definition.

3. function get_obs_values(obs_set_type :: set) : Returns a vector of 'real' values that came from the instrument(s). Some value might be returned if the data is missing for certain elements, but this is not defined. If integer or other types of obs were happening might have to extend or overwrite this, but can't think of a relevant case.

4. subroutine set_obs_set_time(obs_set_type, generalized_time_type) sets the time interval associated with a particular observation.

5. subroutine set_obs_set_values(obs_set_type, vector of values) : Sets the observation values for an obs_set. Possibly used for setting up assimilation system simulation experiments as an example.

6. subroutine set_obs_set_missing(obs_set_type, vector of logicals): The observations corresponding to any true elements are set to missing.

7. function copy_obs_set(obs_set_type) returns a copy of the obs_set, overloaded to =.

7. function obs_value_missing(obs_set_type) returns a logical array with false for all entries where an observation exists and true if observation is missing.

8. function read_obs_set(file pointer), reads an obs set from file with this pointer (can just be an index for now)

9. function write_obs_set(obs_set, file_pointer), writes an obs_set to file with this pointer.

10. function read_obs_set_data(file pointer), reads only the data values for an obs set.

11. function write_obs_set(obs_set, file_pointer), writes only the data values for an obs set the the file.

ALSO INHERITS the calls from obs_set_def which should act directly on the obs_set_def in the obs_def structure.

III. obs_set_def

Defines a set of associated observations. A definition only includes information about the spatial aspects of the set of observations (the values and the time interval are added by obs_set).

Defines type obs_set_def_type

contains information on the
error_covariance / error correlation
list of obs_set_defs
list of obs_set sizes
---
list of obs_defs
num of obs_defs

Should have a unique key for assimilation algorithm caching

As currently envisioned, obs_set_defs are viewed as being recursive. Each obs_set_def is composed of zero or more obs_set_defs (called obs_subsets in description) plus zero or more individual scalar observations. The abstraction is also associated with an ordered set of unitary observations obtained by recursively descending the subsets.

1. function diag_obs_err_cov(obs_set_def :: set) : Returns true if error covariance for this obs set is diagonal, false otherwise. Initial implementation can support only diagonal covariances with great simplification.

2. function get_diag_obs_err_cov(obs_set_def :: set) : Returns the diagonal of the observation error covariance matrix for this set in a one dimensional array.

3. function get_obs_err_cov(obs_set_def :: set) : Returns a two dimensional array containing the observational error covariance for this set. Not needed in initial implementation.

4. function get_expected_obs(obs_set_def :: set, model_state_vector or extended model state vector:: state) : Returns a one dimensional array containing the expected values for this observation set given the model state. Need to give more thought to how the extended state is passed around if it is required and whether it is of fixed or variable composition (probably variable for big models). For initial implementation need only implement raw state part.

5. function get_unique_id(obs_set_def) returns the unique integer key associated with this set. May want interfaces to access obs_set_defs by key from an obs_sequence at some later point.

6. function get_total_num_obs(obs_set_def :: set): Returns the total number of observations in a set, including the observations included recursively in all subsets. This is easily computed as sets are built up recursively.

7. function get_obs_def(obs_set_def, index) returns handle to the the index-th observation (this may be a problem for efficiency with obs sets being built up recursively) in the set. The order is established by the recursive search of the set. Need error action if index exceeds total number of obs in set.

8. function get_number_single_obs(obs_set_def), returns the number of observation definitions in the set that are not included in other subsets; not clear that this is necessary as a public interface.

9. function get_number_obs_subsets(obs_set_def), returns the number of observations sets that are contained in the current set; not clear that this is necessary as a public inteface.

10. function get_single_obs(obs_set_def, index), returns the index-th observation from the single observations in this set. May not be necessary. Need an error action if index exceeds number of single obs in set.

11. function get_obs_subset(obs_set_def, index), returns the index-th obs subset handle; not clear if this needs to be a public interface. Need an error return if index exceeds the number of subsets.

12. function init_obs_set_def(num_subsets, num_obs) returns a handle to a structure for an obs_set_def. Specifying the number of subsets and num_obs for the single obs is annoying but precludes the need for dynamic data structures at this point. Might want to make them optional at a much later time.

13. subroutine add_obs_subset(obs_set_def, obs_set_def (subset)) adds the second set as a subset of the first. Need error returns if the first set is already full.

14. subroutine add_obs(obs_set_def, obs_def) adds the obs_def to this obs_set as a single observation. Need error return if the number of single obs would be exceeded by this insertion.

15. subroutine set_err_cov(obs_set_def, obs_subset_index, obs_subset_index, obs_index1, obs_inces2, cov) sets error covariance between two obs subsets/obs. Only two of the subset / obs indices can be set in the call. The covariance is a two dimensional matrix with the covariance values. Error is required if the indices are outside the range for this set or if the covariance matrix is not the correct size for the pair. Not needed in initial implementation.

16. function read_obs_set_def(file_pointer), returns an obs_set_type with contents read in from a standard file format (this may be relatively sophisticated as it will have to allocate storage, etc. Should make use of calls defined above.

17. function write_obs_set_def(obs_set_def, file_pointer) outputs an obs_set_def in standard file format.

Need to push up obs_def functions to this level

18. function get_obs_locations(obs_set_def) returns ordered set of locations for the observations in the set.

19. function get_num_close_states(obs_set_def, radius) returns array of number of close states for each of the observations in the set.

20. subroutine get_close_states(obs_set_def, radius, number, indices) returns array of numbers plus 2D array of indices for state variables close to each ob in the set.

21. function copy_obs_set_def(obs_set_def) copies the obs_set_def (this will be involved). Overloaded to =.

IV. obs_def

Defines a single scalar observations spatial characteristics and kind.

Defines type obs_def_type

contains information on
error variance
params associated with observation
location (a location type)
‘kind’ (says something about forward operator)

unique key for caching

1. function get_expected_obs(obs_def, model_state_vector / extended state vector), returnsexpected value given state or extended state vector. For initial implementation only need to deal with raw state.

2. function get_err_var(obs_def), returns observational error variance of observation definition.

3. function get_obs_location(obs_def), returns a location_type for this observation. For some more complex types of observations the definition of location may involve additional parameters, but let’s avoid that complexity for this stage.

4. function get_obs_kind(obs_def), returns the kind of this observation, this probably needs to itself be a class (obs_kind class) although it could just be some fixed text string representation for now.

5. function init_obs_def(obs_kind, obs_location, error_variance) returns an obs_def with this kind, location and error variance.

6. function read_obs_def(file_pointer) reads an obs_def from standard format off the file. Will need error returns if read fails.

7. function write_obs_def(file_pointer) writes an obs_def in standard format to the file.

8. function get_obs_def_id(obs_def) returns integer key to this definition.

9. function get_num_close_state(obs_def, radius) this assumes that the model class data is global (only one type of model in use). At some point it would be nice to generalize this, but not now.

10. subroutine get_close_states(obs_def, radius, number, close_state_list) See above.

11. subroutine set_obs_def_location(obs_def, location)

12. subroutine set_err_var(obs_def, err_var)

13. subroutine set_obs_kind(obs_def, obs_kind)

14. function copy_obs_def(obs_def) returns obs_def overloaded to =.

V. location

This is the level at which the details of the spatial representation come into play. There will be different location data representations for different spaces (sort of like the one and twod loc and dist mods but more general).

Defines type location_type

1. function get_dist(loc1, loc2), returns some metric of distance between two locations.

2. function set_loc(space model specific arguments, for instance, lat, lon, height) returns a location type with these values.

3. subroutine get_loc(location_type, space model specific arguments) returns values of the location in the arguments; inverse of set_loc.

4. function write_location(file_pointer) writes the location to the file in a standard format.

5. function read_location(file_pointer) reads the location from the file in a standard format. Need error return if read fails.

VI. generalized_time

For early implementations, let's just assume all times are discrete and use the standard time manager package to do this. However, need to keep possibility of a weighted observation time interval in mind. Need to be able to output and input a time (does this exist in the current time manager specification?)

VII. obs_kind

This is specific to a particular model and some notion of externally defined observation type. For a model, need to be able to get a forward operator to generate this quantity in conjunction with spatial (and possibly temporal, currently pushed to highest level) interpolation. For now, the obs_kind is just a string.

1. function get_obs_kind(obs_kind) returns string specifiying the obs_kind.

2. function set_obs_kind(string) returns and obs_kind with this string as its definition.

Can this whole level just be dropped for initial implementation?

VIII. assim_model

This is the interface needed to a model by the assimilation algorithms and observation networks. This is model specific and tries to abstract away the model details. Because of F90 limitations, seems appropriate to think of only being able to work with one model type (class) at a time, i.e. one couldn't work on a separate atmosphere and ocean model at once with these interfaces. Assim_model is a class for a particular kind of model and the instances are the state (including some indexed time). Initialization and end calls are for the class data (for instance setting up transform stuff for a spectral model). Smart choices about timestepping can be pushed into calls at this level. Extended state should not be viewed as part of the class data, but is instead the return from some functions operating on the state. May want a concept of static extended state (things that can be obtained by operating on the state variables without time integration) and extended state that requires integration. For initial implementation, assume that model timestep is fixed at a particular delta_t and that all observations will be specified as falling exactly on one of these timesteps. The general timestepping computations can probably be pushed into an auxiliary module since they will be used for all models.

Defines type assim_model_type
time type with associated global base time that defines model time
state vector
meta data describing state vectors

1. function get_model_size() returns integer size of model state vector; need to be very careful to understand multiple time-level state interactions. Note that this is class data.

2. function write_state_restart(state vector, file_name) writes out the state vector and the time in a form with machine precision so that time integration can be resumed without evidence of interruption.

3. function read_state_restart(state vector, file_name) reads a machine precision restart from file_name.

4a. function init_diag_output(file_name, global_meta_data, copies_of_fields_per_time, meta_data_per_copy) returns file_id for an output diagnostic file that will output a given number of copies (say posterior, prior, truth, ensembles, etc) of fields at each time. For now the global meta-data is a text string describing the file contents and the meta_data_per_copy is an array of text strings describing the contents of each copy of data.

4. function output_diagnostics(file_id, state vector, time, copy_index optional) outputs diagnostic state information for this time for the copy_index copy. If copy_index is not present it is assumed to be 1. For now, the copies at a given time must be called sequentially and no other calls for output for this file_id may be made until all copies are output at this time.

5. function initialize_assim_model() initializes the class data for the model, for instance transform data for spherical harmonics. Returns the size of the models state vector.

6. function end_assim_model() shuts down and cleans up class data for model.

7. function get_max_dt()

8. function get_min_dt()

9. function get_other_dt_stuff()

10. function get_last_state_time_before(time)

10. function get_first_state_time_after(time)

10. function get_closest_state_time_to(time)

10. function get_next_state_time(time)

10. function get_prev_state_time(time)

11. function get_initial_condition() returns an assim_model_type to start from. This might come from a file or from something else, controlled by model runtime paramerters??? This needs to be given some help through some runtime interface I think. One option is that a 'restart' file is available in some standard place, this is read in and the corresponding state returned (watch out for multi-level timestepping stuff here). Another option is just some basic spin-up state. The assim_model_type returned has a time associated with it. For now, this time needs to just come in with whatever is read from file or generated and is probably controlled by a runtime input. Coordinating the time between the model and observation set areas should be automated eventually but can be pushed to runtime for now.

12. function advance_state(assim_model_type, target_time, extended state requests optional) returns state and optionally extended state vector advanced in time to the target_time. If the model has flexibility to do so, it will be advanced to exactly target_time. If it has a fixed time_step, target_time should be within some small tolerance (for floating point stuff) of a time to which the model can go or an error should be returned. The philosophy here is that the requisite time computations should be done first and then the model should be advanced. For initial implementation, assume that model has single dt and that all observations will fall exactly on a dt interval.

12. function get_static_extended_state(assim_model_type, optional field specifier) returns static extended state variables computed from model state. Not needed in initial implementation.

13. function interpolate(state vector / extended state vector, location, field_id) returns value for this field interpolated to the location. Need to be very clear here. What is a field_id and how is it obtained. What about 2D fields in 3d models, etc. For now, only implement for state vector and add extended state at later date. Field_id needs to be associated with some sort of string i.d. that is part of the model metadata.

14. subroutine get_state_meta_data(index, location, optional kind) returns metadata for the indexed state variable as a location plus a kind (if the model has more than one kind of variable).

15. subroutine get_extended_state_meta_data(index, location, optional kind) returns metadata for the indexed extended state variable. May need to further refine this for large models with a large number of possible extended state types.

16. subroutine get_close_states(location_type, radius, number, indices) returns list of state indices within radius of the given location and the number of such state variables. May want to make kind an optional input. Need error handling if there is a storage issue. Should probably implement with a clean extensible list. Note that these are class functions that don't depend on a particular state instantiation.

17. function get_num_close_states(location_type, radius) returns the number of state variables within radius distance of this location.

18. function get_model_time(assim_model_type) returns time type that is the time for this state.

19. subroutine set_model_time(assim_model_type, time_type)

20. function get_model_state_vector(assim_model_type, plus additional arguments to specify a portion?)

21. subroutine set_model_state_vector(assim_model_type, state_vector, plus additional arguments to specify a portion?

22. function copy_assim_model(assim_model_type) returns assim_model_type overloaded to =.

IX. obs_model class

This is the level at which classes become model dependent on the observation taking side of things. This class needs to know considerable detail about how the model data is laid out, how to interpolate, how to do forward operators for different types of observations, etc. Probably the biggest chunks of code someone will have to write in order to get things going? Maybe there should be one more layer that just does interpolation and knows about the model?

1. function take_obs(model state vector or extended model state vector, location, obs_kind_id) there may be a variety of different algorithms. Making use of interpolation if needed from assim_model, this computes the observation given the state and returns the value.