Observation Operator Fortran Interfaces

The general approach is that observation operators will be implemented as a hierarchy of classes all inheriting from an abstract base class. During the execution of an application, it is intended that one instance of each class will be created per observation type being used in that run, grouping all observations of that type in this object. This grouping has two aims: lower level data can be organized as arrays for greater efficiency (better memory access patterns and threading) and it will be easier to implement methods accounting for correlations within each observation type.

The abstract interfaces defined by the base types are the ones that will be used by the JEDI abstract layer. All data should be passed by interface (no global variables) or be private to a given observation type and contained in the object. The use of common interfaces will ensure easy exchanges of observation operators between partners and easy addition of new observation types without any modification in data assimilation algorithms.

The abstract base class at the Fortran level is not in principle needed by a system like JEDI which relies on its own mechanism for handling observation types. However, it is convenient for two reasons: it defines an interface that guaranties compatibility for higher level code and it avoids duplication of common code between types.

Unit tests will be developed and implemented with each method.

The interfaces below are the minimum requirement for the JEDI abstract layer. Each implementation could can define its own additional methods, although their use should be limited to internal purposes.

Observation Operator

```
type, abstract :: obs_operator
contains
  procedure(op_create), deferred :: create           ! Constructor
  procedure(op_delete), deferred :: delete           ! Destructor
  procedure(op_h_oper), deferred :: h_oper           ! Observation operator
  procedure(op_inputs), deferred :: input_variables  ! input variables required by observation operator
  procedure(op_print),  deferred :: print            ! Prints human readable info
end type obs_operator

abstract interface
  subroutine op_create(self, config)
    class(obs_operator), intent(inout) :: self
    type(config), intent(in) :: config
  end subroutine op_create
end interface

abstract interface
  subroutine op_delete(self)
    class(obs_operator), intent(inout) :: self
  end subroutine op_delete
end interface

abstract interface
  subroutine op_h_oper(self, gom, hofx, bias)
    class(obs_operator), intent(in) :: self
    type(state_at_locations), intent(in)         :: gom    ! Model values after interpolations
    type(obs_data), intent(inout)                :: hofx   ! H(x) output values
    type(obs_aux_variable), optional, intent(in) :: bias   ! Bias correction predictors
  end subroutine op_h_oper
end interface

abstract interface
  function op_inputs(self)
    class(variables), pointer    :: op_inputs
    class(obs_error), intent(in) :: self
  end function op_inputs
end interface

abstract interface
  subroutine op_print(self)
    class(obs_operator), intent(in) :: self
  end subroutine op_print
end interface
```

The linear observation operators are kept in a separate class. They can be implemented as tangent linear and adjoint code or explicit Jacobians, the interface stays the same.

Linear Observation Operators

```
type, abstract :: linear_obs_operator
contains
  procedure(op_create), deferred :: create    ! Constructor (compute/store trajectory/Jacobians)
  procedure(op_delete), deferred :: delete    ! Destructor
  procedure(op_h_optl), deferred :: h_oper_tl ! Linearized observation operator
  procedure(op_h_opad), deferred :: h_oper_ad ! Adjoint observation operator
  procedure(op_print),  deferred :: print     ! Prints human readable info
end type linear_obs_operator

abstract interface
  subroutine op_create(self, traj, bias, config)
    class(linear_obs_operator), intent(inout)    :: self
    class(obs_operator), intent(in)              :: oper  ! Nonlinear observation operator
    type(state_at_locations), intent(in)         :: traj  ! Nonlinear trajectory "state"
    type(obs_aux_variable), optional, intent(in) :: bias  ! Nonlinear trajectory bias predictors
    type(config), intent(in) :: config
  end subroutine op_create
end interface

abstract interface
  subroutine op_delete(self)
    class(linear_obs_operator), intent(inout) :: self
  end subroutine op_delete
end interface

abstract interface
  subroutine op_h_optl(self, dgom, dhofx, dbias)
    class(linear_obs_operator), intent(in)   :: self
    type(state_at_locations), intent(in)     :: dgom    ! Increment values after interpolations
    type(obs_vector), intent(inout)          :: dhofx   ! H.dx output values
    type(obs_aux_incr), optional, intent(in) :: dbias   ! Bias correction predictors increment
  end subroutine op_h_optl

abstract interface
  subroutine op_h_opad(self, dgom, dhofx, dbias)
    class(linear_obs_operator), intent(in)      :: self
    type(state_at_locations), intent(inout)     :: dgom    ! H^T.dy output values (at obs locations)
    type(obs_vector), intent(in)                :: dhofx   ! dy input
    type(obs_aux_incr), optional, intent(inout) :: dbias   ! Bias correction predictors gradient
  end subroutine op_h_opad
end interface

abstract interface
  subroutine op_print(self)
    class(linear_obs_operator), intent(in) :: self
  end subroutine op_print
end interface
```