Errors and Error Covariances Fortran Interfaces

Background Errors

Although several implementations of background error formulation and associated covariance matrix might be available for a given model, it is unlikely an inheritance structure between them will bring much benefit. The interface is described without inheritance but like for any of the Fortran entities, the inheritance structure is not visible from the abstract (non-Fortran) layer and can be used if it proves useful.

The background error can be nonlinear. The linearize method is responsible for generating the covariance matrix object associated to the current error.

Background Errors

```
type :: bkg_error
contains
  procedure :: create      ! Constructor
  procedure :: delete      ! Destructor
  procedure :: linearize   ! Generate covariance matrix
  procedure :: print       ! Prints human readable info
end type bkg_error


subroutine create(self, config, geom, bg)
  class(bkg_error), intent(inout) :: self
  type(config), intent(in)        :: config
  type(geometry), intent(in)      :: geom
  type(fields), intent(in)        :: bg     ! Background state
end subroutine create

subroutine delete(self)
  class(bkg_error), intent(inout) :: self
end subroutine delete

function linearize(self, config, fg)
  class(b_error_covariance), pointer :: linearize
  class(bkg_error), intent(in)       :: self
  type(config), intent(in)           :: config
  type(fields), intent(in)           :: fg     ! First-guess state
end function linearize


subroutine print(self)
  class(bkg_error), intent(in) :: self
end subroutine print
```

Once the Observation Errors have been defined, the Observation Error Covariance matrix is easier to define.

Background Error Covariance

```
type :: b_error_covariance
contains
  procedure :: create    ! Constructor
  procedure :: delete    ! Destructor
  procedure :: multiply  ! Multiply by B
  procedure :: inv_mult  ! Multiply by B^{-1}
  procedure :: randomize ! Returns random vector scaled by B
  procedure :: print     ! Prints human readable info
end type b_error_covariance

subroutine create(self, geom, config)
  type(b_error_covariance), intent(inout) :: self
  type(geometry), intent(in) :: geom
  type(config), intent(in)   :: config
end subroutine create

subroutine delete(self)
  type(b_error_covariance), intent(inout) :: self
end subroutine delete

subroutine multiply(self, dx, dz)
  type(b_error_covariance), intent(in) :: self
  type(fields), intent(in)    :: dx    ! input values
  type(fields), intent(inout) :: dz    ! output values, dz = B * dx
end subroutine multiply

subroutine inv_mult(self, dx, dz)
  type(b_error_covariance), intent(in) :: self
  type(fields), intent(in)    :: dx    ! input values
  type(fields), intent(inout) :: dz    ! output values, dz = B^{-1} * dx
end subroutine inv_mult

subroutine random(self, dx)
  type(b_error_covariance), intent(in) :: self
  type(fields), intent(inout) :: dx    ! scaled random output values
end subroutine random

subroutine print(self)
  type(b_error_covariance), intent(in) :: self
end subroutine print
```

The same interface can also be used to represent any model space error covariance, in particular the model error covariance matrix.

Observation Errors

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.

In principle, the same observation error covariance matrix can be used for several observation types. For example, a diagonal **R** can be written once and used several times. For that reason, the inheritance structure for **R** is distinct from that of the observation operators. Handling of observations errors is split in two classes, one for QC and other nonlinear functions, one for the observation error covariance matrix itself.

Observation Error

```
type, abstract :: obs_error
contains
  procedure(obserr_create), deferred :: create          ! Constructor
  procedure(obserr_delete), deferred :: delete          ! Destructor
  procedure(obserr_qc),     deferred :: quality_control ! Apply QC
  procedure(obserr_covar),  deferred :: linearize       ! Generate covariance matrix
  procedure(obserr_print),  deferred :: print           ! Prints human readable info
end type obs_error

abstract interface
  subroutine obserr_create(self, config, obsdb, yobs)
    class(obs_error), intent(inout) :: self
    type(config), intent(in)        :: config
    type(obs_space), intent(in)     :: obsdb
    type(obs_data), intent(in)      :: yobs   ! Observation values
  end subroutine obserr_create
end interface

abstract interface
  subroutine obserr_delete(self)
    class(obs_error), intent(inout) :: self
  end subroutine obserr_delete
end interface

abstract interface
  subroutine obserr_qc(self, ???, ???)  ! What does QC need?
    class(obs_error), intent(in)     :: self
  end subroutine obserr_qc
end interface

abstract interface
  function obserr_linearize(self, config)
    class(obs_error_covariance), pointer :: obserr_linearize
    class(obs_error), intent(in)         :: self
    type(config), intent(in)             :: config
    type(obs_data), intent(in)           :: yobs   ! Equivalent obs values at first guess
  end function obserr_linearize
end interface

abstract interface
  subroutine obserr_print(self)
    class(obs_error), intent(in) :: self
  end subroutine obserr_print
end interface
```

The interface to the quality control method still needs to be defined. What inputs are needed?

Once the Observation Errors have been defined, the Observation Error Covariance matrix is easier to define.

Note: the observation error covariance matrix interfaces exactly correspond to the background error covariance matrix interfaces where the geometry has been replaced by the obs_space and the fields by obs_vectors.

Observation Error Covariance

```
type :: obs_error_covariance
contains
  procedure :: create    ! Constructor
  procedure :: delete    ! Destructor
  procedure :: multiply  ! Multiply by R
  procedure :: inv_mult  ! Multiply by R^{-1}
  procedure :: randomize ! Returns random vector scaled by R
  procedure :: print     ! Prints human readable info
end type obs_error_covariance

subroutine create(self, obsdb, config)
  type(obs_error_covariance), intent(inout) :: self
  type(obs_space), intent(in) :: obsdb
  type(config), intent(in)    :: config
end subroutine create

subroutine delete(self)
  type(obs_error_covariance), intent(inout) :: self
end subroutine delete

subroutine multiply(self, dy, dz)
  type(obs_error_covariance), intent(in) :: self
  type(obs_vector), intent(in)           :: dy    ! input values
  type(obs_vector), intent(inout)        :: dz    ! output values, dz = R * dy
end subroutine multiply

subroutine inv_mult(self, dy, dz)
  type(obs_error_covariance), intent(in) :: self
  type(obs_vector), intent(in)           :: dy    ! input values
  type(obs_vector), intent(inout)        :: dz    ! output values, dz = R^{-1} * dy
end subroutine inv_mult

subroutine random(self, dy)
  type(obs_error_covariance), intent(in) :: self
  type(obs_vector), intent(inout)        :: dy    ! scaled random output values
end subroutine random

subroutine print(self)
  type(obs_error_covariance), intent(in) :: self
end subroutine print
```