Model Space Fortran Interfaces

Geometry

The model space is defined by a certain geometry, usually a grid definition but it can be more abstract, for example for spectral models. Given the wide variety of possible geometries, it is not possible to try to define them all at the abstract level. Instead, this is delegated to each model implementation. The abstract layer will only construct and destruct these objects and pass them to lower level methods that require them.

The data structure inside the geometry might be complex as it will contain the definition of the grid but also information related to its distribution across processors. The interface visible by the JEDI layer is very simple.

Geometry

```
type :: geometry
contains
  procedure :: create  ! Constructor
  procedure :: delete  ! Destructor
  procedure :: print   ! Print human readable info
end type geometry

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

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

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

Fields

The JEDI layer defines distinct classes for manipulating states and increments. For most models and data assimilation systems, states and increments will be implemented on the same grid and rely on the same data structures, but the methods associated to the two classes are different. In general, the most practical way to implement this will be to use a lower level class to represent model fields which will in turn be used by the state and increment classes. Note that there is no need for inheritance and that the state and increment each contain an fields object. At the Fortran level, those fields are implemented as derived types that contain the actual data and perform actual operations on them. The states and increments are not visible as such.

The second copy constructor (copy_interpolate) construction new fields with the geometry passed by argument and copies the fields from the other argument into them. In most cases this only involves changing the resolution. It is possible to implement distinct field representations for states and increments. In that case the second copy constructor must include an additional conversion step between the two geometries.

Fields

```
type :: fields
contains
  procedure :: create            ! Constructor (calls read from file)
  procedure :: copy              ! Copy constructor
  procedure :: copy_interpolate  ! Copy and change geometry (resolution)
  procedure :: delete            ! Destructor
  procedure :: interpolate       ! Interpolate (eg to obs locations)
  procedure :: read              ! Read (usually from file)
  procedure :: save              ! Write (usually to file)
  procedure :: add               ! Add fields contents
  procedure :: sub               ! Subtract fields contents
  procedure :: mult              ! Multiply fields contents by scalar
  procedure :: dot_product       ! Dot product of fields contents
  procedure :: norm              ! Compute norm (for tests mostly)
  procedure :: print             ! Print human readable info
end type fields

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

subroutine copy(self, other)
  type(fields), intent(inout) :: self
  type(fields), intent(in)    :: other
end subroutine copy

subroutine copy_interpolate(self, geom, other)
  type(fields), intent(inout) :: self
  type(geometry), intent(in) :: geom
  type(fields), intent(in)    :: other
end subroutine copy_interpolate

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

subroutine interp(self, locs, gom)
  type(fields), intent(in)                :: self
  type(locations), intent(in)             :: locs
  type(fields_at_locations), intent(inout) :: gom
end subroutine interp

subroutine read(self, config)
  type(fields), intent(inout) :: self
  type(config), intent(in)   :: config
end subroutine read

subroutine save(self, config)
  type(fields), intent(in)  :: self
  type(config), intent(in) :: config
end subroutine save

subroutine norm(self)
  type(fields), intent(in) :: self
end subroutine norm

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

The fields derived type handles fields for the model state (and other states, for example the background in data assimilation) and increments.

Other classes

Classes for handling auxiliary model space quantities (for example for parameter estimation or model error estimation) will be added.