Developing a new biogeochemical model

Jump to bottom

Jorn Bruggeman edited this page on Jun 29 · 63 revisions

Getting started quickly - examples

- a passive tracer: `src/models/bb/passive/passive.F90`
- a simple nutrient-phytoplankton-zooplankton-detritus (NPZD) ecosystem model: `src/models/gotm/npzd.F90`
- air-sea exchange (of oxygen): `src/models/gotm/ergom.F90` (`do_surface` routine)
- benthic variables, sedimentation, resuspension: `src/models/gotm/ergom.F90` (`do_bottom` routine)
- light, operating over a water column: `src/models/gotm/light.F90`
- a modular NPZD model `src/models/examples/npzd`
- time/space-varying sinking rates: `src/models/akvaplan/antiparasitic.F90` (`get_vertical_movement` routine)
- directly setting (overriding) the value of state variables: `src/models/iow/age/iow_age.F90`
- pelagic variables not affected by water movement (e.g., mussel ropes): `src/models/bb/filter_feeder/filter_feeder.F90` (note the `disable_transport` property)
- suspended particulate matter, mud and sand, interacting size classes: `src/models/iow/spm/spm.F90`
- (near)instantaneous partitioning between dissolved and adsorbed phases: `src/models/akvaplan/antiparasitic.F90`
- accessing time-averaged or vertically-averaged values of environmental variables: `src/models/examples/mean.F90`

An overview of biogeochemical models implemented in FABM is available here.

A few words in advance

The spatial domain

FABM recognizes three spatial domains: the pelagic, the water surface and the bottom. Conceptually, the pelagic is a three-dimensional environment with an explicit depth dimension, whereas the surface and bottom are two-dimensional boundaries (top and bottom) of the pelagic domain. Thus, biogeochemical variables that can appear anywhere in the water column are part of FABM's pelagic domain, while variables attached to its top or bottom interface (e.g., sea ice algae, benthic communities) are part of FABM's surface or bottom domain.

Bottom and surface-attached variables remain fixed in place, but pelagic state variables are transported by water movement. This is typically described by advection-diffusion equations. FABM outsources the transport of biogeochemical variables to the physical host model. Biogeochemical models written for FABM only need to provide sink and source terms for their biogeochemical state variables - they can trust advection and diffusion of their state variables to be handled elsewhere. Similarly, time integration of the sink and source terms (as part of the final advection-diffusion-reaction equation) is handled externally as well; FABM models only provide instantaneous rates of change.

Biogeochemical processes are typically controlled by local conditions: knowing the *local* model state and *local* environment suffices to calculate all *local* biogeochemical process rates. For that reason, FABM does not require models to deal explicitly with spatial dimensions of the model. Instead, it uses abstract statements (for instance, preprocessor macros `_LOOP_BEGIN_` and `_LOOP_END_`) at the beginning and end of every subroutine to loop over the spatial domain (if any). Within the section enclosed by these statements, a single point in space is processed. This abstraction allows FABM to be used in physical host models that use very different spatial domains (e.g., a 0D well-mixed box, 1D water column, 2D depth-averaged basin, full 3D basin).

Object-oriented programming

From a programmer's perspective, a biogeochemical model combines data and procedures. Data include (constant) parameter values, while procedures *do things*, such as update the model state. Traditional Fortran-based models represent data and functionality with global or module-level variables and procedures, which are accessed by external code (e.g., a hydrodynamic model) where appropriate. This is *not* the case in FABM, which relies on the object-oriented features of Fortran 2003 to allow model code to remain isolated and self-contained.

A biogeochemical model in FABM organizes all code in one or more Fortran modules, and then further groups all model data and functionality (parameter values, references to state variable data, subroutines, functions) in a *model object*. The contents of this model object is specified by a Fortran derived type, which is typically the first thing that appears in a biogeochemical module. Grouping data and functionality in this way has several advantages:

- Since the model object contains all information about the model, this is the only thing that needs to be passed to FABM. Adding a new biogeochemical model to FABM therefore requires only the insertion of two lines in the FABM code (a `use` statement to access the module, and an `allocate` statement that creates the model object).
- As all model data is contained in an object, and not in global or module level variables, it is easy to create any number of copies of the model with different configurations (e.g., different parameter values). These copies can run concurrently to create complex coupled ecosystems (e.g., use multiple copies of a phytoplankton type to create a phytoplankton community), or to run ensemble simulations or parameter sensitivity studies.

The main thing to keep in mind is that this means *the model cannot store any information specific to its configuration or state in global or module-level variables*. All data that can vary between simulations reside in the derived type. As a result, the model's module typically only contains the model's derived type and supporting procedures, perhaps along with a few constants declared with the `parameter` attribute.

Floating point precision

Variables that represent real numbers are declared with `real` or `double precision` in Fortran. The numerical precision of the resulting variables is platform and compiler dependent. To reduce these dependencies and to be able to match the floating point precision of its host models, FABM requires floating point variables to be declared with type `real(rk)`. `rk` is an integer constant defined in module `fabm_types`. This module is used by every biogeochemical model, which means `rk` is universally available. Floating point constants can be declared with the same precision by appending `_rk` to their value, e.g., `1.0_rk`, `2.0e-9_rk`.

Preprocessor macros

Biogeochemical model code is independent of the spatial domain that the model will run in. This domain may be 0D (a well-mixed box), 1D (e.g., a water column), 2D (e.g., a depth-averaged model) or 3D. To make this possible without hurting computational performance (e.g., we still want to support vectorization of spatial operations), preprocessor macros are used whenever spatial data is read (e.g., `_GET_`) or written (e.g., `_ADD_SOURCE_`, `_SET_DIAGNOSTIC_`). The use of these macros is documented below and is for the most part intuitive: they can be used without having to know how they are implemented. Two things are worth keeping in mind though:

- Preprocessor macros cannot be broken over multiple lines: everything that you put between the parentheses following the macro name must appear on the same line. This is needed for some Fortran compilers.

- Preprocessor macros cannot appear as the statement in a "logical if". For instance, do not use

  ```
  if (CONDITION) _ADD_SOURCE_(...)
  ```

  If you want such conditional assignment, use a "block if" instead:

  ```
  if (CONDITION) then
     _ADD_SOURCE_(...)
  end if
  ```

  This is needed because preprocessor macros may translate into multiple statements at compile time. In this case, a logical if would apply to the first statement only; subsequent statements would be executed independent of `CONDITION`.

Create an institute directory for your model

In FABM, `src/models` contains subdirectories for each institute or individual contributing one or more biogeochemical models. The first time you add a model, you will have to create this "institute directory". To do so, first choose a short, descriptive identifier that describes your institute, or alternatively, yourself. This identifier can consist of lower-case letters only (no underscores). For an institute, the chosen name is usually its accepted acronym. For individuals, the convention is to use the initials followed by the last name (lower case letters only, no periods). In this document, the chosen institute (or individual) name will be referred to by `INSTITUTE`.

Adding necessary files

The institute directory at minimum needs to contain two files. First, it needs to contain a file with the name `INSTITUTE_model_library.F90`. This file is the starting point for FABM when it wants to create a new model. Initially, it should contain the following:

```fortran
module INSTITUTE_model_library

   use fabm_types, only: type_base_model_factory, type_base_model

   ! Add use statements for new models here

   implicit none

   private

   type, extends(type_base_model_factory) :: type_factory
   contains
      procedure :: create
   end type

   type (type_factory), save, target, public :: INSTITUTE_model_factory

contains

   subroutine create(self, name, model)

      class (type_factory), intent(in) :: self
      character(*),         intent(in) :: name
      class (type_base_model), pointer :: model

      select case (name)
         ! Add case statements for new models here
      end select

   end subroutine create

end module
```

The module and factory name must appear exactly as shown above, that is, they must equal `INSTITUTE_model_library` and `INSTITUTE_model_factory`, respectively. The build system depends on this! Note that `INSTITUTE` needs to be replaced with your chosen institute name.

Second, your institute directory needs to contain a file with the name `CMakeLists.txt`. This file contains compilation instructions for CMake, which include a list of all source files to compile. It should initially contain the following:

```cmake
add_library(fabm_models_INSTITUTE OBJECT
            INSTITUTE_model_library.F90
           )

add_dependencies(fabm_models_INSTITUTE fabm_base)
```

As before, `INSTITUTE` needs to be replaced by your chosen institute name.

Making FABM aware of the new directory

FABM uses a build system based on CMake, which uses a master configuration file `src/CMakeLists.txt`. In this file, each institute must be referenced in the variable `DEFAULT_INSTITUTES`, near the top of the file:

```
set(DEFAULT_INSTITUTES
    aed          # Aquatic Eco Dynamics, University of Western Australia
    au           # University of Aarhus
    bb           # Bolding & Burchard
    examples     # Examples supplied with FABM itself
    gotm         # Models ported from original GOTM/BIO library
    hzg          # Helmholtz-Zentrum Geesthacht
    iow          # Leibniz Institute for Baltic Sea Research
    klimacampus  # KlimaCampus Hamburg
    metu         # Middle East Technical University
    msi          # Marine Systems Institute, Tallinn University of Technology
    niva         # Norsk Institutt for Vannforskning
    pml          # Plymouth Marine Laboratory
   )
```

Your institute needs to be added to this list in order for CMake to use your institute-specific CMakeLists.txt. Please add the corresponding line while preserving alphabetical order.

By adding your institute here, FABM will *automatically* reference your model library (specifically, it will auto-generate `fabm_library.F90` with the necessary references). This *requires* that your module and factory are named as described above (`INSTITUTE_model_library` and `INSTITUTE_model_factory`).

Creating an empty model

Now that you have an institute directory, you can create a Fortran source file for your model. This requires choosing a short, descriptive name for the model, consisting of lower-case letters and underscores only; in this document, the name will be referred to by `MODELNAME`.

You can organize the code in your institute directory any way you like. The simplest scenario is described here: a new model is added directly below the institute directory in a file called `MODELNAME.F90`. Thus, its full path in the FABM source tree is `src/models/INSTITUTE/MODELNAME.F90`. To get started, you can add the following code to this file to create an empty model:

```
#include "fabm_driver.h"

module INSTITUTE_MODELNAME

   use fabm_types

   implicit none

   private

   type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME
      ! Add variable identifiers and parameters here.
   contains
      procedure :: initialize
      ! Reference model procedures here.
   end type

contains

   subroutine initialize(self, configunit)
      class (type_INSTITUTE_MODELNAME), intent(inout), target :: self
      integer,                          intent(in)            :: configunit

      ! Register model parameters and variables here.
   end subroutine initialize
   ! Add model subroutines here.

end module
```

This model will compile but does nothing. You can now:

- Add parameters and variables: declare them as part of your model's derived type, register them from the `initialize` subroutine.
- Add subroutines to provide sink and source terms, surface fluxes, bottom fluxes, vertical movement, etc.: add each subroutine to the module body below the `initialize` subroutine, then make it part of your model's derived type with a `procedure` statement as is done for `initialize` .

Making the model available

To make FABM aware of your new model, two lines must be added to your library, `src/models/INSTITUTE/model_library.F90` :

```
module INSTITUTE_model_library

   use fabm_types, only: type_base_model_factory, type_base_model

   use INSTITUTE_MODELNAME
   ! Add use statements for new models here

   ...

contains

   subroutine create(self, name, model)

      ...

      select case (name)
         case ('MODELNAME'); allocate(type_INSTITUTE_MODELNAME::model)
         ! Add new case statements for new models here
      end select

   end subroutine create

end module
```

Note the two newly added lines above, both of which mention INSTITUTE_MODELNAME. The first line references the Fortran module that contains your new model with "use INSTITUTE_MODELNAME", the second line links your model name to its associated derived type: "case ('MODELNAME'); allocate(type_INSTITUTE_MODELNAME::model)".

Including the model during compilation

Your own CMakeLists.txt file, `src/models/INSTITUTE/CMakeLists.txt` , specifies the Fortran source files that need to be compiled. Your model source file(s) must be added as part of the add_library command, for instance:

```
add_library(fabm_models_INSTITUTE OBJECT
            INSTITUTE_model_library.F90
            MODELNAME.F90
           )
```

The model source file is added here with the line "MODELNAME.F90". Note that paths to source files must be given relative to the institute directory.

Testing the empty model

At this point, FABM should compile with the new (empty) model (see Building and installing). The new model may be activated in a FABM simulation by creating a `fabm.yaml` file that references your new model, e.g.,

```
instances:
  mymodel:
    model: INSTITUTE/MODELNAME
```

Here, `mymodel` is an arbitrary identifier for your model that you can choose yourself. It will be used as prefix for your model's variable names.

Adding variables

FABM distinguishes three types of model variables, each allowing different types of access:

- state variables: you can read the variable value and provide its local rate of change (source minus sink terms)
- diagnostic variables: you can set the variable value (but not read it!)
- dependencies: you can read the variable value

All variables must be registered in the model's `initialize` subroutine. Registration is done by calling a *variable registration subroutine*, which sets an *identifier* that may be used later to access the variable value. Variable identifiers must be stored as member of the model's derived type, `type_INSTITUTE_MODELNAME`.

In general, adding a variable thus involves:

- adding the variable identifier to the model's derived type, `type_INSTITUTE_MODELNAME`.
- adding a call to a variable registration subroutine in the `initialize` subroutine.

State variables

State variables are quantities for which the model specifies the rate of change. They are initialized at the start of the simulation and time-integrated by the host model using your instantaneous rates of change. Rates of change include local source and sink terms, but *not* transport (advection, diffusion, sinking or floating, etc.), which is handled by the host model.

FABM uses a *state variable identifier* to retrieve or modify the state variable's values. The data type of this identifier defines the variable's spatial domain:

- `type_state_variable_id` is used for pelagic variables. These will be transported by water movement.
- `type_bottom_state_variable_id` is used for bottom-associated variables (e.g., benthic fauna, sediment). These will remain fixed in place.
- `type_surface_state_variable_id` is used for surface-associated variables (e.g., sea ice algae, surface microlayer constituents). These will remain fixed in place.

A model needs to store the variable identifier in its derived type to be able to access the state variable during simulation. By convention, such identifiers are named `id_VARNAME`, with `VARNAME` being chosen by the model author. Thus, the derived type should be modified as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

   type (type_state_variable_id) :: id_VARNAME

contains

   ...

end type
```

Note that for bottom-bound and surface-bound state variables, `type_state_variable_id` would be replaced with `type_bottom_state_variable_id` and `type_surface_state_variable_id`, respectively.

To tell FABM about a state variable, call subroutine `self%register_state_variable` from `initialize`:

```
subroutine initialize(self,configunit)

   ...

   call self%register_state_variable(self%id_VARNAME, 'NAME', 'UNITS', 'LONG_NAME')

end subroutine initialize
```

`register_state_variable` takes four required arguments:

1. `id` : the variable identifier, as added to the model's derived type in the previous step.
2. `name` : the variable name (a string consisting of letters and underscores only)
3. `units` : the variable unit (a string)
4. `long_name` : a descriptive name for the variable (a string)

In addition, `register_state_variable` takes the following optional arguments:

Argument name	Description	Data type	Default value
`initial_value`	A default initial value for the variable, applied across the full spatial domain. Host models generally allow the user to override this with a spatially varying field.	`real(rk)`	`0.0_rk`
`minimum`	Minimum value that this variable is allowed to take. This is a hard constraint which the physical model will try to enforce (e.g., by limiters in numerical schemes). If the variable value drops below this minimum, the host may artifically reset it to the boundary (clipping), or mask it in model output.	`real(rk)`	`-1e20_rk`
`maximum`	Maximum value that this variable is allowed to take. This is a hard constraint which the physical model will try to enforce (e.g., by limiters in numerical schemes). If the variable value exceeds this maximum, the host may artifically reset it to the boundary (clipping), or mask it in model output.	`real(rk)`	`1e20_rk`
`missing_value`	Value that may be used to indicate missing data. It should lie outside the range bounded by the `minimum` and `maximum` arguments.	`real(rk)`	`-2e20_rk`
`vertical_movement`	A space and time independent value for the rate of vertical movement (e.g., sinking, floating) of the variable. Unit: m s^{-1}. Positive values indicate upward movement (floating/rising), negative values indicate downward movement (sinking/settling). N.B. if you require *time and/or space varying* vertical	`real(rk)`	`0.0_rk`

Argument name	Description	Data type	Default value
	movement, you should implement `get_vertical_movement` .		
`specific_light_extinction`	A space and time independent value for the variable-specific light extinction coefficient. Unit: m^{-1} *VARIABLEUNIT*$^{-1}$. N.B. light extinction formulations that cannot be captured with constant specific light extinction coefficients can be implemented by creating a diagnostic for the desired light attenuation in m^{-1}, and having it contribute to total attenuation.	`real(rk)`	`0.0_rk`
`no_precipitation_dilution`	Flag indicating that the value of this variable is by default not affected by surface freshwater fluxes (precipitation or evaporation). This may be the case for dissolved gases, for instance, which are also present in rain. Some host models may allow the user to override this setting by providing time and/or space varying concentrations in rain water.	`logical`	`.false.`
`no_river_dilution`	Flag indicating that the value of this variable is by default not affected by river runoff. Some host models may allow the user to override this setting by providing time and/or space varying concentrations in river water.	`logical`	`.false.`
`output`	How to include this variable in model output. Set it to `output_none` to exclude the variable from output. This settings specifies a *default* only - many host models allow the user to select variables for output; this includes variables that have been registered with `output_none` .	`integer`	`output_instantaneous`
`standard_variable`	An object that describes a "standard variable". If you provide this argument, you promise that this state variable has a well-defined meaning and units. Other models that request the same standard variable as dependency will then *automatically* be coupled to your state variable. Predefined standard variables are available as members of the `standard_variables` object, provided by the `fabm_types` module.	`type_standard_variable`	

Arguments that are of type `reak(rk)` must be exactly of the right type. If they are specified as numeric constant, this must be a floating point number with trailing `_rk`. For instance, `1.0_rk` and `3.e6_rk` are fine, but `1.0` and `3.e6` are not. And neither are `1` and `1_rk`, as they are integers rather than reals!

Registered state variables can be accessed at any time except during initialization (i.e., from `initialize` itself) with `_GET_(self%id_VARNAME, VARIABLE)` for pelagic state variables, and `_GET_HORIZONTAL_(self%id_VARNAME, VARIABLE)` for bottom-bound and surface-bound state variables. Here, `VARIABLE` is a local variable of type `real(rk)` that should receive the current value of the state variable (local in time and space). Note that this access is read-only: it allows you to retrieve the state variable value, but not to change it. The state variable value is changed by specifying a rate of change (`do`, `do_surface`, `do_bottom` subroutines for pelagic, surface-attached, and bottom-attached state variables, respectively).

Diagnostic variables

Diagnostic variables are quantities that the model can calculate at any given time from the value of the model state variables and/or environmental variables. Models commonly define such variables to diagnose model behavior. Models could, for instance, declare diagnostic variables for the rates of particular biogeochemical processes. By doing so, these rates will be available in model output.

FABM uses a *diagnostic variable identifier* to set the diagnostic variable's values. The data type of this identifier defines the variable's spatial domain:

- `type_diagnostic_variable_id` is used for pelagic variables
- `type_surface_diagnostic_variable_id` is used for variables associated with the water surface, such as the air-sea CO_2 flux.
- `type_bottom_diagnostic_variable_id` is used for variables associated with the bottom, e.g., deposition at the bed.

A model needs to store the variable identifier in its derived type to be able to provide values for the diagnostic variable during simulation. By convention, such identifiers are named `id_VARNAME`, with `VARNAME` being chosen by the model author. Thus, the derived type should be modified as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

   type (type_diagnostic_variable_id) :: id_VARNAME

contains

   ...

end type
```

Note that for diagnostic variables associated with surface or bottom, `type_diagnostic_variable_id` would be replaced with `type_surface_diagnostic_variable_id` or `type_bottom_diagnostic_variable_id`, respectively.

To tell FABM about a diagnostic variable, call subroutine `self%register_diagnostic_variable` from `initialize`:

```
subroutine initialize(self, configunit)

   ...

   call self%register_diagnostic_variable(self%id_VARNAME, 'NAME', 'UNITS', 'LONG_NAME')

end subroutine initialize
```

`register_diagnostic_variable` takes four required arguments:

1. `id` : the variable identifier, as added to the model's derived type in the previous step.
2. `name` : the variable name (a string, letters and underscores only)
3. `units` : the variable unit (a string)
4. `long_name` : a descriptive name for the variable (a string)

In addition, `register_diagnostic_variable` takes the following optional arguments:

Argument name	Description	Data type	Default value
`source`	This tell FABM which model subroutine will set the value of the diagnostic. For interior diagnstics, for instance, it defaults to `source_do`, which implies that the `do` subroutine will set the value.	`integer`	`source_do` for interior diagnostics, `source_do_surface` for surface diagnostics, `source_do_bottom` for bottom diagnostics
`missing_value`	Value that may be used to indicate missing data.	`real(rk)`	`-2e20_rk`
`output`	How to include this variable in model output. Set it to `output_none` to exclude the variable from output. This settings specifies a *default* only - many host models allow the user to select variables for output; this includes variables that have been registered with `output_none`.	`integer`	`output_instantaneous`
`standard_variable`	An object that describes a "standard variable". If you provide this argument, you promise that this diagnostic has a well-defined meaning and units. Other models that request the same standard variable as dependency will then *automatically* be coupled to your diagnostic. Predefined standard variables are available as members of the `standard_variables` object, provided by the `fabm_types` module.	`type_standard_variable`	

The value of diagnostic variables can be set at any time except during initialization (i.e., from `initialize` itself) with `_SET_DIAGNOSTIC_(self%id_VARNAME, VALUE)` for pelagic variables, `_SET_SURFACE_DIAGNOSTIC_(self%id_VARNAME, VALUE)` for variables associated with the water surface, and `_SET_BOTTOM_DIAGNOSTIC_(self%id_VARNAME, VALUE)` for variables associated with the bottom. `VALUE` is the value that you want the diagnostic variable to take, of type `real(rk)`.

Note that access to diagnostic variables is write-only: you can set the variable value, but not read it. If you also want to query the value of the diagnostic variable, this can be achieved by additionally registering it as a dependency (with the name of the diagnostic variable and dependency being equal).

External dependencies

Biogeochemical processes often depend on the physical environment, i.e., on light, temperature, pressure, salinity, etc. These variables are not part of the biogeochemical model - they are contained in the physical host. Additionally, a model may depend on quantities (state variables, diagnostics) that are defined and managed by other biogeochemical models that run concurrently.

FABM offer two means to define dependencies:

- by a known "standard variable" with well-defined meaning. For instance: temperature, salinity, pressure, photosynthetically active radiation, wind speed, bottom stress.
- by custom variable name, units, and long name. At run-time, the user must link this dependency to a variable provided by the physical host model or by other biogeochemical model. This is specified in `fabm.yaml`, in the coupling section of the biogeochemical model that has registered the dependency.

It is recommended to use references to standard variables whenever possible.

FABM uses a *dependency identifier* to retrieve the variable's values. The data type of this identifier defines the variable's spatial domain:

- `type_dependency_id` is used for pelagic variables
- `type_horizontal_dependency_id` is used for variables that lack a vertical dimension (surface or bottom fields such as wind speed, pCO_2, bottom stress, but also longitude and latitude)
- `type_global_dependency_id` is used for space-independent variables (e.g., the day of the year).

The model must store the identifier as member of its derived type to be able to access the variable's value during simulation. By convention, such identifiers are named `id_VARNAME`, with `VARNAME` being chosen by the model author. Thus, the derived type should be modified as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

   type (type_dependency_id) :: id_VARNAME

contains

   ...

end type
```

For dependencies that lack a vertical dimension (e.g., surface or bottom fields), `type_dependency_id` would be replaced with `type_horizontal_dependency_id`. For dependencies that are space-independent, `type_dependency_id` would be replaced with `type_global_dependency_id`.

To tell FABM about a dependency, call subroutine `self%register_dependency` from `initialize`. Two forms exist for this routine, as shown here:

```
subroutine initialize(self,configunit)

   ...

   call self%register_dependency(self%id_VARNAME, STANDARD_VARIABLE)
   call self%register_dependency(self%id_VARNAME, 'NAME', 'UNITS', 'LONG_NAME')

end subroutine initialize
```

The first form of `register_dependency` defines its dependency by a "standard variable identity". This enables implicit coupling: if another biogeochemical model or the physical host registers a state variable or diagnostic with the same identity, your dependency will be automatically coupled to it. This form takes two required arguments:

1. `id`: the variable identifier added to the model's derived type in the previous step.
2. `standard_variable`: this is an object that describes a "standard variable". A large number of objects representing standard variables are available as members of the `standard_variables` object, provided by the `fabm_types` module. Thus, a standard variable can be specified as `standard_variables%temperature`, `standard_variables%practical_salinity` and similar. If your desired variable is not present in this list, you can define a custom standard variable by providing `type_interior_standard_variable(name='<NAME>',units='<UNITS>')`, `type_surface_standard_variable(name='<NAME>',units='<UNITS>')`, or `type_bottom_standard_variable(name='<NAME>',units='<UNITS>')` for the standard variable argument.

The second form of `register_dependency` defines its dependency by its name, units and long name. This allows manual coupling: the user can couple the dependency at run time to a variable from another biogeochemical model. This form takes four required arguments:

1. `id`: the variable identifier added to the model's derived type in the previous step.
2. `name`: a short variable name (a string consisting of letters and underscores only).

3. `units` : the variable unit (a string)
4. `long_name` : a descriptive name for the variable (a string)

Registered dependencies can be accessed at any time except during initialization (i.e., from `initialize` itself) with the following:

- `_GET_(self%id_VARNAME, VARIABLE)` for variables in the pelagic domain, such as temperature, salinity, and light intensity.
- `_GET_HORIZONTAL_(self%id_VARNAME, VARIABLE)` for variables without vertical dimension, such as surface wind speed.
- `_GET_GLOBAL_(self%id_VARNAME, VARIABLE)` for space-independent variables, such as the day of the year.

In all cases, `VARIABLE` is a local variable of type `real(rk)` that should receive the current value of the variable (local in time and space). Note that this access is read-only: it allows you to retrieve the variable value, but not to change it.

Contributing to aggregate quantities

While a model in FABM is in principle a stand-alone, isolated bit of code, it can let its variables contribute to globally known quantities, such as total chlorophyll, total carbon, or total light attenuation. This may done for any of the following reasons:

- to register that the variable must be considered in mass balance checks
- to affect the light field by contributing to light attenuation, absorption and scattering.
- to allow other models to retrieve such totals for their own purpose (e.g., empirical parameterizations dependent on total chlorophyll or carbon)

To specify a contribution to aggregate quantities, call `self%add_to_aggregate_variable` from the model's `initialize` subroutine, using the following syntax:

```
call self%add_to_aggregate_variable(STANDARD_VARIABLE, self%id_VARNAME)
```

This routine takes two required arguments:

1. `standard_variable` is an object that describes a "standard variable". Typically, this is a member of the `standard_variables` object provided by the `fabm_types` module, for instance, `standard_variables%total_carbon` . You can also create your own custom aggregate quantity, by providing `type_interior_standard_variable(name='<NAME>', units='<UNITS>', aggregate_variable=.true.)` for the standard variable argument. For surface and bottom associated variables, substitute `type_surface_standard_variable` and `type_bottom_standard_variable` , respectively. For quantities that are defined both in the interior and at the interfaces, such as total carbon, use `type_universal_standard_variable` .
2. `id` is the identifier of the contributing variable. Typically, this is a state variable identifier, but it may also be a diagnostic variable identifier.

The subroutine additionally takes an optional `scale_factor` argument, which specifies the value that the contributing variable must be multiplied with to arrive at the units of the aggregate quantity. For instance, if model variable with identifier `self%id_c` describes biomass carbon in mmol m^{-3}, while the biomass also has a fixed molar nitrogen to carbon ratio of 0.15, you would use the following:

```
call self%add_to_aggregate_variable(standard_variables%total_carbon, self%id_c)
call self%add_to_aggregate_variable(standard_variables%total_nitrogen, self%id_c, scale_factor=0.15_rk)
```

Similarly, if model variable with identifier `self%id_c` has units mg carbon m^{-3}, you could contribute to total carbon in mmol m^{-3} with the following

```
call self%add_to_aggregate_variable(standard_variables%total_carbon, self%id_c,scale_factor=1._rk/12._rk)
```

If you would like a state variable or diagnostic variable with identifier `self%id_c` to contribute to light attenuation using a specific attenuation of 0.001 (m^{-1} per variable units), you could use the following

```
call self%add_to_aggregate_variable(standard_variables%attenuation_coefficient_of_photosynthetic_radiative_flux, self%:
```

Adding parameters

Model parameters are space- and time-independent variables that are set once, at the start of a simulation. They then remain unchanged for the duration of the simulation. Biogeochemical models in FABM store the value of their parameters in the model's derived type, `type_INSTITUTE_MODELNAME` . The values are set once in the subroutine `initialize` , which typically retrieves parameter values from file.

Adding a new parameter `NAME` first requires adding it to the model's derived type:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME
   ...
   real(rk) :: NAME
contains
   ...
end type
```

In this case the data type of the parameter is `real(rk)` : a floating point number with precision specified by FABM-provided constant `rk` . In addition to `real(rk)` , the following data types may be used as well: `integer` , `logical` , `character` .

Subsequently, the parameter value must be set by calling `get_parameter` from the `initialize` subroutine:

```
subroutine initialize(self,configunit)
   ...
   call self%get_parameter(self%NAME, 'NAME', 'UNITS', 'LONG_NAME', DEFAULT_VALUE)
end subroutine initialize
```

Subroutine `get_parameter` takes the following arguments:

argument	interpretation	required
`value`	the variable that should receive the parameter value. Typically this is a member of the model's derived type, and therefore prefixed with `self%` . The variable can be of the following types: `real(rk)` , `integer` , `logical` , `character` .	yes
`name`	a short name for the parameter (a string consisting of letters and underscores only)	yes
`units`	a unit specifier (a string).	no
`long_name`	a descriptive name for the parameter (a string, white-space allowed).	no
`default`	a default value for the parameter (type equal to that of the `value` argument).	no
`scale_factor`	scale factor to apply to the retrieved parameter value. This argument is only supported if `value` is of type `real(rk)` . The product of `scale_factor` and the parameter value set in `fabm.yaml` will be stored in argument `value` . Note that this implies that the `units` argument must apply to the parameter value *before* it is multiplied with `scale_factor` . A common use of `scale_factor` is to allow the user to specify rates in d^{-1} in `fabm.yaml` , while the model works internally in s^{-1}. In that case, `scale_factor` is set to `1._rk/86400` and `units` is set to "d-1".	no

Although arguments `units` and `long_name` are optional, it is *strongly* recommended that you provide them, as some programs and utilities present this information to the user.

Providing process rates and diagnostics

In the pelagic

A model that features pelagic state variables must specify their rate of change in a subroutine with the name `do`. This is also the place to calculate the value of any diagnostic variables that apply to the pelagic. The `do` subroutine is implemented by adding the following template to the module body, after the `initialize` subroutine:

```
subroutine do(self, _ARGUMENTS_DO_)
   class (type_INSTITUTE_MODELNAME), intent(in) :: self
   _DECLARE_ARGUMENTS_DO_

   _LOOP_BEGIN_
   _LOOP_END_
end subroutine do
```

`_ARGUMENTS_DO_`, `_DECLARE_ARGUMENTS_DO_`, `_LOOP_BEGIN_`, and `_LOOP_END_` are all preprocessor macros. They are replaced at compile time by domain-specific Fortran code, tailored to the host model. For instance, `_ARGUMENTS_DO_` and `_DECLARE_ARGUMENTS_DO_` are replaced by code that includes indices for all spatial dimensions, while `_LOOP_BEGIN_` and `_LOOP_END_` are replaced by a `do` / `end do` loop in host models with a spatial dimension (in non-spatial models, they are replaced by empty strings).

To make FABM aware of the new `do` subroutine, it must be referenced in the model derived type as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

contains

   ...

   procedure :: do

end type
```

To make the `do` subroutine actually *do* something, such as compute rates of change, logic must be added between `_LOOP_BEGIN_` and `_LOOP_END_`. Typically, this section contains the following:

1. `_GET_` statements that retrieve the current value of the model's variables. The full syntax for variable value retrieval is `_GET_(VARIABLEID, TARGETVARIABLE)`. Here, `VARIABLEID` is an identifier, either of type `type_state_variable_id` or `type_dependency_id`. `TARGETVARIABLE` is the name of the local variable of type `real(rk)` that should receive the value. It is also possible to retrieve the value of horizontal variables, e.g., relating to geographic location, the water surface or the bottom. The syntax for this is `_GET_HORIZONTAL_(VARIABLEID, TARGETVARIABLE)`. In this case, `VARIABLEID` must be an identifier of type `type_surface_state_variable_id`, `type_bottom_state_variable_id`, or `type_horizontal_dependency_id`.
2. Expressions that calculate the rate of change of state variables and the value of diagnostic variables. Model parameters can be accessed as `self%PARAMETERNAME`.
3. For each pelagic state variable, a `_ADD_SOURCE_(VARIABLEID, RATE)` statement that transfers the calculated rate of change to FABM. Here, `VARIABLEID` is an identifier of type `type_state_variable_id`. `RATE` is the rate of change in state variable units **per second**.
4. For each diagnostic variable, a `_SET_DIAGNOSTIC_(VARIABLEID, VALUE)` statement that transfers the calculated diagnostic value to FABM. Here, `VARIABLEID` is an identifier of type `type_diagnostic_variable_id`. `VALUE` is the value that you want the diagnostic to take; it must be of type `real(rk)`.

This section generally uses local variables to hold values retrieved with `_GET_` and the intermediate result of computations. These variables must be declared between `_DECLARE_ARGUMENTS_DO_` and `_LOOP_BEGIN_`, by adding statements such as `real(rk) :: VARIABLE`.

For a simplistic nutrient-phytoplankton model, the `do` subroutine could look like this:

```
subroutine do(self, _ARGUMENTS_DO_)
   class (type_examples_np), intent(in) :: self
   _DECLARE_ARGUMENTS_DO_

   real(rk) :: n, p
   real(rk) :: mu

   _LOOP_BEGIN_

      ! Obtain concentration of nutrient and phytoplankton.
      _GET_(self%id_n,n)
      _GET_(self%id_n,p)

      ! Compute phytoplankton growth.
      mu = self%mu_max*p*n/(n+self%K)

      ! Send rates of change to FABM.
      _ADD_SOURCE_(self%id_n,-mu)
      _ADD_SOURCE_(self%id_p,mu)

      ! Send the value of diagnostic variables to FABM.
      _SET_DIAGNOSTIC_(self%id_mu,mu)

   _LOOP_END_
end subroutine do
```

Note that state variable identifiers `id_n` and `id_p`, diagnostic variable identifier `id_mu` and parameters `mu_max` and `K` must have declared in the model's derived type and registered in subroutine `initialize`, as described in the sections above. Also, the growth rate `mu` is here stored as diagnostic variable, which means it will be included explicitly in model output in addition to the concentrations of nutrient and phytoplankton.

At the surface

If a model features surface-attached state variables (e.g., sea ice algae, surface microlayer constituents), or pelagic variables that are exchanged across the water surface (e.g., dissolved gases such as O_2 and CO_2), it must provide their rate of change and their surface flux, respectively, in a subroutine with the name `do_surface`. This is also the place to calculate the value of any diagnostic variables that apply to the water surface. The `do_surface` subroutine can be implemented by adding the following template to the module body:

```
subroutine do_surface(self, _ARGUMENTS_DO_SURFACE_)
   class (type_INSTITUTE_MODELNAME), intent(in) :: self
   _DECLARE_ARGUMENTS_DO_SURFACE_

   _SURFACE_LOOP_BEGIN_
   _SURFACE_LOOP_END_
end subroutine do_surface
```

To make FABM aware of the new `do_surface` subroutine, it must be referenced in the model derived type as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

contains

   ...
   procedure :: do_surface

end type
```

To make the `do_surface` subroutine actually *do* something, such as compute exchange across the water surface, logic must be added between `_SURFACE_LOOP_BEGIN_` and `_SURFACE_LOOP_END_`. Typically, this section contains the following:

1. Statements that retrieve the current value of the model's variables. For pelagic variables (e.g., temperature, salinity, and the model's own pelagic state variables), the near-surface value is obtained with `_GET_(VARIABLEID, TARGETVARIABLE)`. Here, `VARIABLEID` is an identifier of type `type_state_variable_id` or `type_dependency_id`. `TARGETVARIABLE` is the name of a local variable with data type `real(rk)` that should receive the value. For the model own surface-attached state variables and surface-related dependencies (e.g., wind speed, air temperature, air pressure), the value is obtained with `_GET_HORIZONTAL_(VARIABLEID, TARGETVARIABLE)`. In this case, `VARIABLEID` must be an identifier of type `type_surface_state_variable_id` or `type_horizontal_dependency_id`.
2. Expressions that calculate the rate of change of surface-attached state variables, the surface flux of pelagic state variables, and the value of surface-related diagnostic variables (if any). Model parameters can be accessed as `self%PARAMETERNAME`.
3. For each surface-attached state variable, a `_ADD_SURFACE_SOURCE_(VARIABLEID, RATE)` statement that sends its rate of change to FABM. Here, `VARIABLEID` is the identifier of type `type_surface_state_variable_id`. `RATE` is the source term in state variable units **per second**.
4. For each pelagic state variable that is subject to surface exchange (which includes uptake or release by surface-attached state variables!), a `_ADD_SURFACE_FLUX_(VARIABLEID, RATE)` statement that sends the flux over the water surface to FABM. Here, `VARIABLEID` is an identifier of type `type_state_variable_id`. `RATE` is the flux across the water surface in state variable units **times meter, per second** (e.g., for O_2 in mmol m^{-3}, the flux should be given in in mmol m^{-2} s^{-1}). A positive flux increases the near-surface value of the state variable; a negative flux decreases it.
5. For each surface-related diagnostic variable, a `_SET_SURFACE_DIAGNOSTIC_(VARIABLEID, VALUE)` statement that transfers the calculated diagnostic value to FABM. Here, `VARIABLEID` is an identifier of type `type_surface_diagnostic_variable`. `VALUE` is the floating point value that the diagnostic variable should take, of type `real(rk)`.

This section generally uses local variables to hold values retrieved with `_GET_` / `_GET_HORIZONTAL_` and the intermediate result of computations. These variables must be declared between `_DECLARE_ARGUMENTS_DO_SURFACE_` and `_SURFACE_LOOP_BEGIN_`, by adding statements such as `real(rk) :: VARIABLE`.

At the bottom

If a model features bottom-bound (benthic) state variables, or pelagic variables that are exchanged across the bottom interface, it must provide their rate of change and their bottom flux, respectively, in a subroutine with the name `do_bottom`. This is also the place to calculate the value of any diagnostic variables that apply to the bottom. The `do_bottom` subroutine is implemented by adding the following template to the module body:

```
subroutine do_bottom(self, _ARGUMENTS_DO_BOTTOM_)
   class (type_INSTITUTE_MODELNAME),intent(in) :: self
   _DECLARE_ARGUMENTS_DO_BOTTOM_

   _BOTTOM_LOOP_BEGIN_
   _BOTTOM_LOOP_END_
end subroutine do_bottom
```

To make FABM aware of the new `do_bottom` subroutine, it must be referenced in the model's derived type as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

contains

   ...

   procedure :: do_bottom
```

```
end type
```

To make the `do_bottom` subroutine actually *do* something, such as compute the change in benthic variables, logic must be added between `_BOTTOM_LOOP_BEGIN_` and `_BOTTOM_LOOP_END_`. Typically, this section contains the following:

1. Statements that retrieve the current value of the model's variables. For pelagic variables (e.g., temperature, salinity, and the model's own pelagic state variables), the near-bottom value is obtained with `_GET_(VARIABLEID, TARGETVARIABLE)`. Here, `VARIABLEID` is the identifier of type `type_state_variable_id` or `type_dependency_id`. `TARGETVARIABLE` is the name of a local variable with data type `real(rk)` that should receive the value. For the model own benthic state variables and bottom-related dependencies (e.g., bottom stress), the value is obtained with `_GET_HORIZONTAL_(VARIABLEID, TARGETVARIABLE)`. In this case, `VARIABLEID` must be of type `type_bottom_state_variable_id` or `type_horizontal_dependency_id`.
2. Expressions that calculate the rate of change of benthic state variables, the rate of bottom exchange of pelagic state variables, and the value of bottom-bound diagnostic variables (if any). Model parameters can be accessed as `self%PARAMETERNAME`.
3. For each benthic state variable, a `_ADD_BOTTOM_SOURCE_(VARIABLEID, RATE)` statement that sends its rate of change to FABM. Here, `VARIABLEID` is the identifier of type `type_bottom_state_variable_id`. `RATE` is the source term in state variable units **per second**.
4. For each pelagic state variable that is subject to bottom exchange (which includes uptake or release by benthic state variables!), a `_ADD_BOTTOM_FLUX_(VARIABLEID, RATE)` statement that sends the flux over the bottom interface to FABM. Here, `VARIABLEID` is an identifier of type `type_state_variable_id`. `RATE` is the flux across the bottom interface in state variable units **times meter, per second** (e.g., for nitrate in mmol m^{-3}, the flux should be given in in mmol m^{-2} s^{-1}). A positive flux increases the near-bottom value of the state variable; a negative flux decreases it.
5. For each bottom-related diagnostic variable, a `_SET_BOTTOM_DIAGNOSTIC_(VARIABLEID, VALUE)` statement that transfers the calculated diagnostic value to FABM. Here, `VARIABLEID` is an identifier of type `type_bottom_diagnostic_variable`. `VALUE` is the floating point value that the diagnostic variable should take, of type `real(rk)`.

This section generally uses local variables to hold values retrieved with `_GET_` / `_GET_HORIZONTAL_` and the intermediate result of computations. These variables must be declared between `_DECLARE_ARGUMENTS_DO_BOTTOM_` and `_BOTTOM_LOOP_BEGIN_`, by adding statements such as `real(rk) :: VARIABLE`.

Advanced features

Time and/or space-varying vertical movement

Models can provide a constant (i.e., time- and space-independent) rate of sinking or floating when they register pelagic state variables. This is done by providing the `vertical_movement` argument in the call to `register_state_variable`. However, if the rate or direction of vertical movement of state variables changes in response to state or environment, the model must implement the `get_vertical_movement` subroutine. This has template

```
subroutine get_vertical_movement(self, _ARGUMENTS_GET_VERTICAL_MOVEMENT_)
   class (type_INSTITUTE_MODELNAME), intent(in) :: self
   _DECLARE_ARGUMENTS_GET_VERTICAL_MOVEMENT_

   _LOOP_BEGIN_
   _LOOP_END_
end subroutine get_vertical_movement
```

To make FABM aware of this new subroutine, it must be included in the model derived type as follows:

```
type, extends(type_base_model), public :: type_INSTITUTE_MODELNAME

   ...

contains
```

```
    ...

    procedure :: get_vertical_movement

 end type
```

As in the `do` subroutine, all logic that calculates and sets the rate of movement must be inserted between `_LOOP_BEGIN_` and `_LOOP_END_`. Typically, this involves:

1. `_GET_` statements that retrieve the value of state variables and/or environmental dependencies
2. Expressions that calculate the rate of movement
3. For each pelagic state variable that has a time- and/or space varying rate of vertical movement, a `_ADD_VERTICAL_VELOCITY_(VARIABLEID, VELOCITY)` statement that provides the rate of movement. The target state variable is identified by its identifier `VARIABLEID`, which must be of type `type_state_variable_id`. The rate of movement, `VELOCITY`, must be given in m s^{-1}, with negative values indicating movement towards the bottom (e.g., sinking), and positive values indicating movement towards the surface (e.g., floating).

Any local variables that are used need to be declared between `_DECLARE_ARGUMENTS_GET_VERTICAL_MOVEMENT_` and `_LOOP_BEGIN_`.

Build and test passing DOI 10.5281/zenodo.7737951

Clone this wiki locally

```
https://github.com/fabm-model/fabm.wiki.git
```