Framework for Aquatic Biogeochemical Models (FABM)

Jorn Bruggeman, Bolding & Burchard ApS.

January 2010 – January 2011

Table of contents

Introduction

The Framework for Aquatic Biogeochemical Models is a general framework that provides the "glue" between an arbitrary physical host model (usually a spatially explicit hydrodynamic model), and any number of arbitrary biogeochemical models. The host maintains abiotic variables such as temperature, light and salinity, and handles space-explicit operations such as advection and diffusion (if applicable). Biogeochemical models are unaware of the spatial context: they operate based on spatially and temporally local conditions only. FABM also provides functionality to *couple* different biogeochemical models running side by side: the user can configure models at runtime to share variables, allowing, for instance, for one model to provide the prey density to another model's predator, or for onemodel to change the concentration of dissolved inorganic carbon used by another model to calculate CO_2 dissolution and pH.

FABM comes with drivers for several physical host models. Currently, the one-dimensional General Ocean Turbulence Model (GOTM – http://www.gotm.net) and the three-dimensional Modular Ocean Model version 4 (MOM4 – http://www.gfdl.noaa.gov/fms) are supported. Also, FABM has experimentally been coupled to the one-dimensional Dynamic Lake Model (DLM, based on the DYnamic REservoir Simulation Model DYRESM http://www.cwr.uwa.edu.au/software1/models1.php?mdid=2). Efforts to couple FABM to the three-dimensional General Estuarine Transport Model (GETM – http://www.getm.eu) are underway.

FABM also provides a library of biogeochemical models of small to moderate complexity. Currently, these include a simple Nutrient-Phytoplankton-Zooplankton-Detritus (NPZD) model

(taken from GOTM), a model of the dissolved inorganic carbon system (Blackford, Plymouth Marine Laboratory, UK), and a model of the invasive comb jelly species *Mnemiopsis* (Salihoglu, Institute of Marine Sciences, Middle East Technical University, Turkey).

This document describes the design philosophy of FABM, and how it may be coupled to other physical hosts and new or existing biogeochemical models.

Design considerations

Programming language

To maximize portability across platforms, FABM is written in pure Fortran. Thus, it does not make use of components written in other languages (e.g., C), and does not require pre-processing with scripting languages (e.g., shell scripts, Perl, Python). Moreover, FABM only uses language elements supported by Fortran 95 or earlier; it does not demand compiler support for features from the newer Fortran 2003 and 2008 standards, as current compilers only support (often small) subsets of their functionality [1].

Although FABM dictates that only Fortran 95 may be used, an FPP-compatible preprocessor required to compile FABM. This is necessary to:

1. Achieve compatibility with spatial environments with any number of dimensions (e.g., a non-spatial 0D setting, a 1D water column, or a full 3D basin). This means that the rank of spatially explicit arrays can vary, which is something that Fortran does not support in safe manner[2]. Biogeochemical models use preprocessor definitions to index spatially-explicit arrays and to define spatial locations; in turn, the physical host model provides the correct preprocessor definitions that correspond to its spatial dimensions.

2. Support optional compilation of FABM (and its contained biogeochemical models) as one-dimensional vectorized version that operates on a full spatial dimension, rather than the standard zero-dimensional version operating on a local point in space. By compiling as a 1D version, the number of subroutine calls is reduced, and the compiler

[1] Although FABM does not require compiler support for Fortran 2003/2008, it does optionally use specific features when preprocessor-based fallback to (less-efficient and/or more error prone) Fortran 95 constructs can be provided. Specifically, FABM uses derived types that contain member variables with the `allocatable` attribute (ISO technical report TR-15581: Enhanced Data Type Facilities, incorporated in FORTRAN 2003). For compilers that not support this, these member are declared with the `pointer` attribute instead (and `associated` is used rather than `allocated`, and the variables are then initialized to `null()`).To obtain this behaviour, this is implemented with the `_ALLOCATABLE_` preprocessor definition, which defaults to `allocatable` if ISO TR-15581 support is present, and to `pointer` otherwise.

[2] Fortran does allow for arrays of variable rank in the form of the "assumed-size array", which is declared with an asterisk denoting the upper bound of the last dimension. Use of this construct is not recommended, first because it means the compiler does not know the size of the array in memory (causing problems if the data need to be copied for any reason, e.g., on distributed platforms), and second because it requires arrays to be contiguous in memory. This requirement may not be met by all physical host models that FABM is coupled to.

has an easier job vectorizing the code (translating loops into single vector operations). This benefits performance.

3. Use certain post-Fortran-95 constructs that benefit performance, but need to be omitted or replaced if the compiler does not support them.

The use of the preprocessor is limited to the definition of preprocessor variables and function-like macros through the use of `#define`, to conditional compilation through the use of `#if`, `#else`, `#endif`, and to inclusion of other files through `#include`.

As FABM uses Fortran 95 exclusively, the set of programming paradigms that can be used is somewhat restricted. Specifically, Fortran 95 does not allow for true object-oriented programming features such as inheritance or polymorphism; those would have been useful to implement different biogeochemical models all implementing a specific set of abstract interfaces. Instead, such functionality is now obtained by defining a generic model that implements functionality for all different biogeochemical models (derived type `type_model` in `src/fabm.F90`). As a result, any new biogeochemical model needs to be added to this generic type in order to be usable in FABM.

Handling in-memory data efficiently

FABM avoids copying of in-memory data whenever possible. This applies in particular to spatially explicit data, which can be extremely expensive to copy in larger (3D) models. Thus, the transfer of variable values between the physical host model and the biogeochemical models in principle does not use any temporary storage, or any arrays to support the FABM "glue layer": biogeochemical models operate directly on data contained by the physical host. To achieve this, FABM uses "assumed-shape arrays" to represent array arguments in subroutines. By using this type of array, data will be passed between subroutines in the form of an array descriptor: a compact compiler-generated structure (35-107 bytes in the Intel Fortran Compiler version 11) that can represent any arbitrary slice of an array. The resulting array slice need not be contiguous in memory, as long as all values of a single variable can be addressed with a single array slice. This allows maximum flexibility to the physical host model to store data in any form.

Local conditions are passed to biogeochemical models in the form of a set of spatially explicit arrays (one per variable, represented in memory by compiler-generated array descriptors), and a set of none (0D host), one (1D host), or more (e.g., 3 for 3D hosts) integer indices for each spatial dimension. Passing the full spatially explicit structures in combination with variable indices takes away the need for the compiler to generate a new in-memory array descriptor for each variable at each location, while looping over the spatial domain. Additionally, by accepting integer indices to specify location (and by allowing the driver in between the physical host and FABM to control these indices while enumerating over space), the biogeochemical models can also loop over data that are present in the same array, but cannot be addressed in a single array slice. This applies, for instance, to 3D models that use z coordinates for depth and have a depth-varying domain – in that case, the bottom layer (or the surface) cannot be addressed with a single slice, as the appropriate depth index will vary in horizontal space.

Vectorization

By default, FABM lets biogeochemical models operate on (0D) local conditions only. This minimizes code complexity. However, as evaluation of biogeochemistry in FABM happens

through subroutines and functions, it implies that evaluation across a large spatial domain requires many subroutine calls – one per function, per grid point. This can be expensive and cannot be vectorized by most compilers (that is, they cannot replace the loop with single vector operations supported by most modern processors), as the loops over the spatial domain and the code executed per grid cell are spread over different files (namely, in the FABM driver for the physical host, the FABM core files, and the specific biogeochemical models). This could be a particular problem on systems with vector processors, whose performance is directly determined by the compiler's ability to vectorize as many operations as possible.

In many cases, the loss of performance due to the subroutine call per grid point will be dependent on model complexity: for larger biogeochemical models, the overhead of the subroutine calls will be negligible. This applies in particular if the models internally use subroutines as well. Performance losses can be small even for simple models – tests in GOTM and MOM4 have shown that the subroutine overhead of the 4-variable NPZD model cannot be detected in practice.

Nevertheless, by defining a single additional preprocessor variable, FABM can be compiled as a vectorized version, with each of its subroutines operating directly on all values across a single spatial dimension [currently the spatial dimension must be depth], rather than on individual points in space. This has been shown to improve performance of biogeochemical calculations with up to 25 % for a simple NPZD model; the overall performance gain may be less for more complex models and in scenarios where calculations of physical processes take up a larger fraction of all computational time.

Run-time configuration and input data

FABM itself comes with no configuration settings other than the need to specify one or more identifiers of the biogeochemical models that should be run. It is expected that the physical host model has methods in place to handle run time configuration. Selection of biogeochemical models for FABM should be integrated with these existing methods (in the driver between FABM and the physical host). For instance, GOTM reads all configuration settings from Fortran namelists. Hence, the driver that couples FABM to GOTM reads the biogeochemical model selection, as well as a few driver specific settings, from a namelist that should be contained in a separate file `fabm.nml`.

The biogeochemical models themselves generally require considerable runtime configuration, often in the form of values for all model parameters. Biogeochemical models interfacing to FABM are encouraged to allow runtime configuration of all settings a user might want to change. It is strongly recommended that biogeochemical models obtain their settings from namelists, as defined in the Fortran 90 standard. However, FABM simply specifies that they must read their settings (if any) from an open text file provided by FABM, and that the models do not use hard-coded paths to any other configuration files. Thus, model could read data in arbitrary format from the provided open file, and they could also open additional configuration files (as long as they obtain the paths to those additional files from the file provided by FABM). It is expected that most biogeochemical models will simply read their settings in the form of one or more namelists from the provided open file.

Biogeochemical model can specify dependencies on external variables – that is, variables that are not contained in the biogeochemical model itself, but perhaps in the physical host or in

another biogeochemical model running in parallel. For instance, many biogeochemical models specify a dependence on temperature, which is provided by all presently supported physical host models. In some cases, however, biogeochemical models may specify dependencies on variables that are not provided by a particular physical host or by any other biogeochemical model. In those cases, the requested values must be read in from an additional data source. (Drivers for) physical host models coupled to FABM must allow for this. It is expected that all physical host models provide functionality for reading temporally-varying spatially-explicit data (both fully spatially explicit data, and horizontal-only data for surface and bottom) from files. The driver between FABM and physical host, therefore, must enumerate the variables in FABM that are requested by biogeochemical models, but cannot be provided directly. It should then obtain data for these variables, either from file using functionality provided by the host (e.g., routines for reading data from text or NetCDF files), or as a single user-configured value applicable across the full spatial domain. [JB: not yet implemented for current GOTM and MOM4 drivers] This may require addition runtime configuration at the side of the driver between FABM and physical host (e.g., the setting of paths to input files).

Coupling different biogeochemical modules

Currently, biogeochemical models must request coupling to external variables during their initialization phase. This includes external variables they aim to modify by adding to their temporal derivatives. The requesting biogeochemical model is, thus, aware of the coupling (the model that is coupled to need not be, though). It is envisioned that FABM may in the future also accept an additional list of coupling links that is provided independently of the biogeochemical model configuration. That would allow for coupling of variables without biogeochemical models being aware of this, and without these models having been designed with coupling in mind. [JB this additional coupling layer just needs to read in links between variables – perhaps simple namelists specifying source and target variable(s)] –and provide the means to specify a variable transformation as well, in the form of an offset and scaling factor. The latter would allow for different units used in the coupled models. Externally defined coupling would require a separation between derivatives returned by the biogeochemical model and the derivatives sent to the physical host, though. Currently, these are one and the same, which benefits performance.]

Obtaining and compiling FABM

Availability

FABM is currently available from a SubVersion (SVN) repository at SourceForge: http://sourceforge.net/projects/rmbm/develop. This location will change in the future, as the present location reflects the earlier name of FABM ("rmbm"), and the SourceForge SVN system does not offer sufficiently fine-grained control over user permissions in the long term.

At present, code and test cases can be obtained on UNIX/Linux/Mac OS X systems by executing:

```
svn co https://rmbm.svn.sourceforge.net/svnroot/rmbm rmbm
```

On Windows, a graphical SVN client may be used, such as TortoiseSVN, http://tortoisesvn.tigris.org.

Build process

FABM comes with make files that perform compilation of all FABM code into a separate code library, which can then be linked against from the physical host. Compilation requires use of the GNU version of the "make" utility. During compilation, the make utility must be provided with configuration settings. This is achieved by setting environment variables. Specifically, the following must be set:

variable	meaning	default
`FABMDIR`	Path to the root of the FABM code tree (containing subdirectories `include`, `src`, `compilers`, etc.). No trailing slash should be added.	`~/FABM/fabm-svn`
`FABMHOST`	Name of the physical host for which the library must be built. This must correspond to the name of a directory below `src/drivers` in the FABM code tree.	`gotm`
`FORTRAN_COMPILER`	Identifier of the Fortran compiler to be used. This must correspond to the extension of one of the `compiler.*` files in the compilers directory in the FABM code tree.	none (must be provided)

After these environment variables are set, compilation works simply by executing `make` in the `src` directory. This creates the FABM library `lib/$(FABMHOST)/$(FORTRAN_COMPILER)/libfabm_prod.a`, with `$(FABMHOST)` and `$(FORTRAN_COMPILER)` corresponding to the values of the environment variables described above as set during compilation. For instance, the GOTM version of the FABM library, compiled with the Intel Fortran Compiler would be located at `lib/gotm/IFORT/libfabm_prod.a`. This library can be linked against during compilation of the physical host model, by providing the path to the library as additional argument during the linking stage. Moreover, during the FABM compilation process, Fortran 90 module files are created at `modules/$(FABMHOST)/$(FORTRAN_COMPILER)/*.mod`. These are needed when interfacing to the FABM library from Fortran code; their location is typically communicated to the Fortran compiler as an additional include directory.

In some cases, the FABM driver for a particular host may depend on one or more routines of modules provided by the host. In that case, FABM cannot be compiled independently of the host. Instead, it must be integrated in the build process of the physical host model, thus bypassing the FABM make files.

Interfacing to biogeochemical models

Concepts

A biogeochemical model is defined as an entity that exports [calculates] the value of one or more (spatially) local variables, when provided with the local abiotic environment (local

temperature, salinity, light intensity, pressure, density, etc.). It is specifically unaware of non-local features of the environment, though some local variables can relate to spatial localization (e.g., local pressure is a reasonable measure of depth).

Exported variables typically include state variables [termed "prognostic variables" in some hydrodynamic models]. For these variables, the model only specifies their local temporal derivative; calculation of their actual value requires known values at the start of the simulation, and a [temporal] integration scheme updating the variable value based on its current state and the temporal derivative. Initialization and temporal integration are expected to be handled by the physical host – the biogeochemical model only provides temporal derivatives, plus a space-independent initial value for the state variables, which in most cases will be overwritten with space-varying values by the host. Biogeochemical models must register the state variables they intend to expose during initialization.

In addition to state variables, biogeochemical models can export one or more diagnostic variables. These variables can be calculated directly from the local biotic and/or abiotic environment. Biogeochemical models must register the diagnostic variables they intend to expose during initialization.

Finally, biogeochemical models can also export one or more variables that are designated to be conserved in time and space. For instance, such variables can be the total of all atoms of a particular chemical element, e.g., total nitrogen or total carbon. These variables across all space should not change, unless fluxes over the boundaries of the model domain are permitted. Conserved quantities must be registered during model initialization.

At all times, a biogeochemical model can read the values of its own state variables and any external variables it depends on. The diagnostic variables exported by a model can also be written to at all times. State variables can only be modified by providing [changing] their local temporal derivatives; this can be done only during the call to `fabm_do`.

Recommended code structure

FABM is designed as the interface between a physical host model and biogeochemical models. Its core files should therefore *not* contain code for either of these models: both the physical host and the biogeochemical models should be independent sets of one or more files, typically containing one or more Fortran 90 modules. Adding a new biochemical model is done by selectively inserting several short (1-2 line) references to the model code in a single FABM file (`src/fabm.F90`), *never* by integrating many lines of code for model initialization or variable calculation directly in the FABM files.

To interface with FABM, a model needs to provide at least two subroutines: one that initializes the model (registers model variables and dependencies, reads runtime configurable model settings – if any – from input files, etc.), and one that calculates the local temporal derivatives of the state variables and the value of the diagnostic variables of the biogeochemical model. Additional subroutines could be defined to provide a custom light extinction coefficient, time- and/or space-varying vertical movement, fluxes over the water surface, totals of conserved quantities, and rates of change due to benthic processes.

When interfacing a biogeochemical model to FABM, the necessary subroutines can either be integrated in the code base of the biogeochemical model itself (which would make sense if the

model is expected to always run as part of FABM), or isolated in a separate file (ideally as a Fortran 90 module). The latter should then in turn interface with the biogeochemical model itself. Isolating the FABM interfaces from the biogeochemical model would be good for larger models (from the perspective of good code design), and is necessary for biogeochemical models that should be able to operate independently of FABM.

General: diagnostic messages and fatal errors

Physical host models handle diagnostic messages and fatal error in different ways. Rather than directly writing diagnostics to the console (the standard output and standard error streams) or using the `stop` statement when fatal errors occur, biogeochemical models should use the routines `log_message` and `fatal_error`, both provided in the Fortran 90 module `fabm_driver`. This module should thus be referenced by a `use` statement in the biogeochemical module if logging or error handling functionality is required.

Subroutine `log_message` takes a single argument: the string to log. Typically, this string will be written to the console, but depending on the host it could also be redirected to file. Subroutine `fatal_error` takes two arguments: the subroutine or function triggering the error (a string), and a string description of the error itself. It is guaranteed that subroutine `fatal_error` never returns; code after the call to `fatal_error` will not be executed.

In summary, a biogeochemical model may not write to the console or use the `stop` statement; the provided FABM routines must be used instead.

Minimum requirements

A biogeochemical model is typically contained in a Fortran 90 module. This module contains at minimum:

1. A derived type (declared with the Fortran 90 `type` statement) whose members store all model data that can vary at runtime. This includes variable identifiers that are provided by FABM, as well as user-configurable model parameters.

2. A subroutine that performs model initialization: it reads in user-configurable settings (if any), and registers all variables it want to expose with FABM. This routine must set the values of all members of the derived type declared in step 1; after initialization, data contained in the derived type are read-only.

3. A subroutine that calculates local temporal derivatives (biogeochemical sink and source terms) of the model's state variables. In order to be able to calculate the derivatives, the subroutine has access to the model's derived type (step 1), as well as the local state and abiotic environment.

The model can additionally implement other functionality, such as variable vertical velocities, complex formulation for light extinction, benthic variables, surface exchange, etc. This is documented below.

To get an idea of the functionality that biogeochemical models should implement, one can take the current NPZD model as an example; this model is coded in the file `src/models/npzd/npzd.F90`. Additionally, surface exchange is demonstrated for tghe inorganic carbon model coded in `src/models/co2sys/co2sys.F90`.

Step 1: grouping model data in a derived type

Every model contains data that can vary at run time. This includes the identifiers of the model's variables, as returned by FABM during initialization, and it typically also includes user-configurable data such as parameter values. Such data must be grouped together in a model-specific derived type. By placing these data into a derived type, it is guaranteed that multiple instances of the model will be able to run side-by-side without sharing (or overwriting) each other's run time data. Models may, therefore, not use global or module-level variables, unless their value is guaranteed to be the same across model instances (in that case, they can typically be declared with the `parameter` attribute).

A typical model-specific derived type could be defined as follows

```
type type_MODELNAME
   ! State variable identifiers
   _TYPE_STATE_VARIABLE_ID_ :: id_n, id_p, id_z, id_d

   ! Diagnostic variable identifiers
   _TYPE_DIAGNOSTIC_VARIABLE_ID_ :: id_primprod

   ! Environmental variable identifiers
   _TYPE_DEPENDENCY_ID_ :: id_temp

   ! Model parameters
   REALTYPE :: par1,par2,par3
end type
```

with `MODELNAME` being a short descriptive name identifying the new model.

It may be noted that the data types of all members listed above (`_TYPE_STATE_VARIABLE_ID_`, `REALTYPE`, etc.) are not conventional Fortran data types. Instead, they are preprocessor symbols that will resolve to actual Fortran data types during compilation. Variable identifiers will only be used during communication by FABM; their actual data type therefore need not be known (and its internal type may change in later versions of FABM). The `REALTYPE` symbol is a flexible floating point data type; it is defined by `REALTYPE` rather than `real` or `double precision` to permit compilation with varying precision (e.g., 4-byte reals or 8-byte reals). Any other data types can appear as well in the derived type, e.g., integers or Boolean values. These can be defined with their native Fortran data types (`integer` and `logical`).

Step 2: model initialization

Initialization of a biogeochemical model involves reading user-configurable model settings – if any – from file, and registering all variables exported by the model. Also, all dependencies of the model (that is, variables needed by the model, but defined by the physical host model or by another biogeochemical model that is running in parallel) must be registered here.

The typical model initialization routine looks like this:

```
subroutine MODELNAME_init(self,modelinfo,namlst)

   implicit none
```

```
   type (type_MODELNAME), intent(out)   :: self
   type (type_model_info),intent(inout) :: modelinfo
   integer,               intent(in)    :: namlst

   ...

end subroutine MODELNAME_init
```

with `MODELNAME` being replaced by the short name describing the model.

The provided arguments are the following

`self` An instance of the model-specific derived type, as described in section "Step 1: grouping model data in a derived type". This variable stores run time configuration settings for the model (e.g., parameter values obtained from namelists) and the identifiers of the model's variables, obtained by calling FABM variable registration functions.

`modelinfo` An instance of the generic derived type (`type_model_info`, declared in the Fortran 90 module `fabm_types`) that contains model information. This variable is managed by FABM; a model should not modify it directly, but instead call various FABM functions for variable and dependency registration. These are described below.

`namlst` The unit of an open file from which configuration settings can be read. *This file is exclusively managed by FABM* – biogeochemical models may read from the provided unit, but they may not open or close it. A typical biogeochemical model gets its runtime configuration from one or more namelists; in that case, these can be read from the provided unit. Models must use the provided unit, rather than opening a specific hard-coded path themselves. The reason for this is that FABM allows for multiple instances of a biogeochemical model to run side-by-side, with different runtime configuration settings. If a model uses hard-coded paths for initialization, all instances will share the same settings (as far as they are read from the hard-coded path), thus breaking this functionality. If a model needs additional input files, it is recommended that it obtains the path to these input files dynamically, that is, from a namelist read from the unit provided by FABM.

During initialization (and only then!) the model can register its variables by calling several functions provided by FABM in the Fortran 90 module `fabm_types`. Each of these routines takes the instance of the derived type `type_model_info` as first argument, variable `modelinfo` above. All routines return identifiers that can later be used to refer to the variable. These identifiers are guaranteed to remain the same during a simulation, but they may vary between simulations. Models should store the identifiers in their derived type, in order to be able to use it later to read or write the values of the variables. The table below lists the FABM functions that can be called to register variables and dependencies. It should be stressed that these routines return identifiers of different types (state variable identifiers, diagnostic variable identifiers, generic variable identifiers, conserved quantity identifiers), which can *only* be used in the manner described.

`register_state_variable` This function registers a state variable that the model will provide (i.e., the model calculates local temporal

derivatives).

The function returns a *state variable identifier* (type `_TYPE_STATE_VARIABLE_ID_`), which can later be used to obtain the current variable value with `_GET_STATE_` (alternatively, `_GET_STATE_HZ_` for benthic variables, which must be registered with argument `benthic=.true.`). Also, the identifier can be used to index the (writeable) array of temporal derivatives (see next section).

`register_diagnostic_variable` This function registers a diagnostic variable, that is, a variable that can be calculated directly from the current state of the biotic and/or abiotic environment, and the model parameters.

The function returns a *diagnostic variable identifier* (type `_TYPE_DIAGNOSTIC_VARIABLE_ID_`), which can later be used to set the current variable value with `_SET_DIAG_` (alternatively, `_SET_DIAG_HZ_` for variables defined only on horizontal slices of the spatial domain; these must be registered with argument `shape=shape_hz`). The identifier returned may be `id_not_used` (a parameter defined in `fabm_types.F90`), which indicates that the host model requests that this variable is not provided; its value should then never be set.

`register_conserved_quantity` This function registers a conserved quantity. The model should provide the local value of this quantity through a separate subroutine that calculates conserved quantities, called from `fabm_get_conserved_quantities`.

The function returns a *conserved quantity identifier* (type `_TYPE_CONSERVED_QUANTITY_ID_`), which can later be used to set the current value of the conserved quantity during the call from `fabm_get_conserved_quantities`.

`register_dependency` This function registers a read-only dependency on an external variable.

The function returns a *dependency identifier* (type `_TYPE_DEPENDENCY_ID_`), which can later be used to obtain the current variable value with `_GET_DEPENDENCY_` (alternatively `_GET_DEPENDENCY_HZ_` for variables defined only on horizontal slices of the spatial domain; these must be

	registered with argument `shape=shape_hz`).
`register_state_dependency`	This function registers that the model reads *and changes* the value of a specific external state variable. This must be a state variable exported by another biogeochemical model. The model can change this variable by modifying its temporal derivative (it cannot set its value directly). The function returns a *state variable identifier* (type `_TYPE_STATE_VARIABLE_ID_`), which can later be used to obtain the current variable value with `_GET_STATE_` (alternatively, with `_GET_STATE_BEN_` for benthic variables, which must be registered with argument `benthic=.true.`). Also, the identifier can be used to index the (writeable) array of temporal derivatives (see next section).

Accessing the value of state variables and external dependencies

At any moment after initialization, a biogeochemical model can request the current (local) value of the model state variables and external dependencies. To this end, the following expressions (preprocessor macros) can be used:

macro	**usage**
`_GET_STATE_(id)`	To obtain the current local value of a state variable with identifier `id`. This state variable is defined on the full model domain – i.e., it is pelagic. The identifier must have been obtained with a call to `register_state_variable` or `register_state_dependency`, as described in the previous section.
`_GET_STATE_BEN_(id)`	To obtain the current local value of the benthic state variable with identifier `id`. As it is benthic, this variable is defined on a horizontal slice of the spatial domain only. The identifier must have been obtained with a call to `register_state_variable` or `register_state_dependency`, with argument `benthic` set to `.true.`.
`_GET_DEPENDENCY_(id)`	To obtain the current local value of an external variable with identifier `id`. This state variable is defined on the full model domain – i.e., it is pelagic. The identifier must have been obtained with a call to `register_dependency`, as described in the previous section.
`_GET_DEPENDENCY_HZ_(id)`	To obtain the current local value of an external variable with identifier `id`, defined on a horizontal slice of the spatial domain only. The identifier must have been obtained with a call to

register_dependency, with argument shape set to shape_hz.

Providing values for diagnostic variables

At any moment after initialization, a biogeochemical model can provide values for its diagnostic variables. To this end, the following expressions (preprocessor macros) can be used:

macro	usage
_SET_DIAG_(id,value)	To set the current local value of the diagnostic variable with identifier id. This variable is defined on the full model domain – i.e., it is pelagic. The identifier must have been obtained with a call to register_diagnostic_variable, as described in the section "Step 2: model initialization".
_SET_DIAG_HZ_(id,value)	To set the current local value of the diagnostic variable with identifier id, defined on a horizontal slice of the spatial domain only. The identifier must have been obtained with a call to register_diagnostic_variable, with argument shape set to shape_hz.

Step 3: providing local temporal derivatives

A typical routine providing temporal derivatives looks like:

```
subroutine MODELNAME_do(self,_FABM_ARGS_DO_RHS_)

   implicit none

   ! Variable declarations
   type (type_MODELNAME),intent(in) :: self
   _DECLARE_FABM_ARGS_DO_RHS_

   ! Declare local variables (if any)

   ! Enter spatial loops
   _FABM_LOOP_BEGIN_

   ...

   ! Leave spatial loops
   _FABM_LOOP_END_

end subroutine MODELNAME_do
```

The routine takes two arguments: self, an instance of the model-specific derived type that typically contains parameter values and variable identifiers (section "Step 1: grouping model data in a derived type"), and _FABM_ARGS_DO_RHS_, a preprocessor symbol that will resolve to a set of one or more arguments needed and managed by FABM. These arguments are declared subsequently by the single preprocessor macro _DECLARE_FABM_ARGS_DO_RHS_.

After variable declaration, the preprocessor macro `_FABM_LOOP_BEGIN_` *must* appear as the first statement in the body of the subroutine. This macro defines the start of a loop over one or more spatial dimensions for vectorized versions of FABM. Similarly, the very last statement in the subroutine must be `_FABM_LOOP_END_`, which closes the spatial loop. Note that if the model requires additional local variables, for instance to store intermediate results, these are best declared in between `_DECLARE_FABM_ARGS_DO_RHS_` and `_FABM_LOOP_BEGIN_`.

In the body of the subroutine, the current local temporal derivatives of the model's state variables are calculated. These generally require the current local state, which may be retrieved with macros `_GET_STATE_` and `_GET_STATE_BEN_`, and potentially also the value of one or more external variables, e.g., temperature; values for such external variables can be retrieved with macros `_GET_DEPENDENCY_` and `_GET_DEPENDENCY_HZ_`. These macros are documented in the earlier section "Accessing the value of state variables and external dependencies". There are no limit to the Fortran constructs that may be used in the function body (between `_FABM_LOOP_BEGIN_` and `_FABM_LOOP_END_`); specifically, calls to model-specific functions and subroutines can be used.

The final values of the temporal derivatives are transferred to FABM by calling (preprocessor macro) `_SET_ODE_`, with the state variable identifier as first argument, and the value of the temporal derivative as second argument, e.g., `_SET_ODE_(self%id_n, value)`.

Vertical movement

Biogeochemical state variables can display vertical movement unrelated to the displacement of the water itself (e.g., by sinking, floating or active swimming). If the rate of movement is constant in time and space, it can be specified during model initialization, with the argument `vertical_movement` (m/s, negative for downward movement, positive for upward movement) in the call to `register_state_variable` that registers the variable. This argument defaults to zero, i.e., by default, variables do not display vertical movement.

If the sinking rate of any of the state variables of a biogeochemical model varies in time and/or space, a subroutine that provides the sinking rates (m/s, negative for downward movement, positive for upward movement) for all state variables must be added as option to the "select" statement in `fabm_get_vertical_movement` in `src/fabm.F90`. In that case, that subroutine must set the sinking rates of all variables, even if some do not vary in time and space.

Light attenuation

Many biogeochemical state variables attenuate light. These include chlorophyll and dissolved organic matter, for instance. In many cases, the contribution of biogeochemical variables to attenuation can be completely described by variable-specific light extinction coefficients (/m/variable unit). These specific extinction coefficients can be specified during model initialization, in the call to `register_state_variable` that registers the variable. This argument defaults to zero, i.e., by default, variables do not attenuate light.

If any of the biogeochemical state variables attenuate light, and this effect cannot be captured by a specific extinction coefficient for these model variables, a function that calculates the light extinction coefficient (/m) from the current model state must be added as option to the `select case` statement in routine `fabm_get_light_extinction` in `src/fabm.F90`. This allows for complete customization of extinction due to biogeochemical variables. This is demonstrated in

the NPZD model ("npzd") provided with FABM. It may be noted that the physical host model must handle light attenuation due to the medium (water) itself. This effect will be added to the attenuation by biogeochemical variables, and should therefore *not* be included in the custom extinction coefficients.

Surface fluxes

Some biogeochemical state variables may be exchanged across the air-water interface. This is typical for dissolved gasses such as oxygen and carbon dioxide, but may also affect certain abiotic compounds (e.g., nutrients) entering the water through atmospheric dust deposition or precipitation. To allow for this, FABM allows a biogeochemical model to prescribe fluxes of its state variables across the air-water interface. If a model allows surface fluxes of one or more of its state variables, it must provide a subroutine that calculates these fluxes. A call to this routine must be added to the `select case` statement in subroutine `fabm_get_surface_exchange` in `src/fabm.F90`. Fluxes must be specified in state variable units × m/s; positive values indicate fluxes entering the water, negative values indicate fluxes leaving the water. This demonstrated in the CO_2 system model ("co2sys") provided with FABM.

Bottom fluxes and the benthos

Similar to surface fluxes, biogeochemical models may also allow for fluxes or one or more state variables across the bottom interface of the water column. This often goes combined with a representation of the benthos, i.e., the set of biotic and abiotic processes that are localized at the bottom of the water column (e.g., sea floor biota). If a model allows exchange of one or more pelagic state variables across the bottom interface, or needs to represent the benthos with one or more state variables (registered with argument `benthic=.true.` in `register_state_variable`), or both, it must provide a subroutine that calculates the bottom fluxes of the pelagic state variables and/or the temporal derivatives of the benthic state variables. A call to this routine must be added to the `select case` statement in subroutine `fabm_do_benthos` in `src/fabm.F90`. Fluxes of pelagic state variables must be specified in state variable units × m/s; temporal derivatives must be specified in state variable units/s. In both cases, the typical unit would be mol/m²/s.

Mass and energy conservation

If (some of) the model state variables are composed of conserved quantities (energy and/or chemical elements), a function that provides the local sum of these quantities given the model state can be added as option to the `select case` statement in `get_conserved_quantities_fabm` in `src/fabm.F90`. In that case, the totals of these conserved quantities over the spatial domain can be diagnosed at run-time, and is included in the output of the physical host model. Note that the model should have registered any conserved quantities during initialization. This is demonstrated in the NPZD model ("npzd") provided with FABM.

Registering the new model

A new biogeochemical model only needs to be registered in `fabm.F90`. For the impatient: throughout `src/fabm.F90`, locations where additional biogeochemical models may be referenced are indicated by the comment `ADD_NEW_MODEL_HERE`. You can search for this string to get started quickly.

The following steps are required to register a new biogeochemical model with FABM:

1. If your model is contained in a Fortran 90 module (recommended!), add a `use` statement that references that module in `src/fabm.F90`. For instance, the module containing the NPZD example model (`src/models/npzd/npzd.F90`) is referenced by

   ```
   use fabm_npzd
   ```

2. Define a unique integer identifier for the new biogeochemical model in `fabm.F90`. For instance, the NPZD example model is assigned a model identifier as follows:

   ```
   integer, parameter :: npzd_id = 1
   ```

   This defines a named constant (`npzd_id`) that represents the actual integer value. Throughout the code, this named constant should be used to refer to the new model – *the actual numeric value may never be used* (this allows for later changes to the numeric value without necessitating any other code change).

3. Define a unique short string (max. 64 characters, lower case letters, digits and underscores only) that describes the model. This string can be used by the user at run time to select the model, and is also used in model output as prefix of variable names. It typically is identical to the model identifier used to prefix all model routines and as postfix of the module name.

   Register the identifier string by coupling it to the integer identifier of the model (step 2) in the subroutine `register_models` in `src/fabm.F90`. This is done by adding a call to `register_model` with the integer identifier and short name of your choice. For the NPZD example model, this is done as follows:

   ```
   call register_model(npzd_id,'npzd')
   ```

   Note that `npzd_id` is the integer identifier defined in step 2.

4. If your model groups data in a derived type, add an instance of this type as member to the derived type `type_model` defined in `fabm.F90`. For instance, the NPZD model defines the derived type `type_npzd` containing variable identifiers and model parameters in its module `npzd` (source file `models/npzd/npzd.F90`). This derived type is added to `type_model` as follows:

   ```
   type type_model
      ...
      type (type_npzd) :: npzd
      ...
   end type type_model
   ```

   The name given to the instance of the derived type (here: “npzd”) is arbitrary, as it is only referenced in FABM by lines that you will add later. However, it is strongly recommended that it is equal to the short name defined in step 3.

5. Write a subroutine that initializes the biogeochemical model. This subroutine will be part of the biogeochemical model (or of the interface between that model and FABM), not of FABM itself. Initialization involves registration of all model variables (state variables, diagnostic variables, conserved quantities), as well as reading the values of user-configurable parameters from file. For more details, see the section "Model initialization" below.

   The subroutine that initializes the biogeochemical model must be called in the subroutine `fabm_init` in `src/fabm.F90`. This is done by inserting the call in its `select case` statement, which calls initialization routines based on the integer model identifier.

   For the NPZD example model, the call to its initialization subroutine `npzd_init` is inserted as follows:

```
select case (model%id)
   ...
   case (npzd_id)
      call npzd_init(model%npzd,model%info,nmlunit)
   ...
end select
```

   Note that `npzd_id` is the identifier defined in step 2, `model%npzd` refers to the model-specific derived type defined in step 4, `model%info` is a structure that will contain model metadata such as variable names and units (managed by FABM), and `nmlunit` is the unit of the open text file from which the model may read configuration information (the opening and closing of the unit is managed by FABM).

6. Define a subroutine that calculates the temporal derivatives of all state variables exported by the biogeochemical model. Most models that export diagnostic variables will set their values in this routine as well, as the calculation of state variable derivatives and diagnostic variables often shares code. However, values for diagnostic variables can also be provided in any other model routine called by FABM, if appropriate. For more details, see the section "Providing variable values" below.

   The subroutine that calculates the values of all variables of the biogeochemical model must be called in the subroutine `fabm_do_rhs` in `src/fabm.F90`. This is done by inserting the call in its `select case` statement, which calls model-specific routines based on the integer model identifier. For the NPZD example model, the call to its subroutine `npzd_do` is inserted as follows:

```
select case (model%id)
   ...
   case (npzd_id)
      call npzd_do(model%npzd,_INPUT_ARGS_DO_RHS_)
   ...
end select
```

   Note that `npzd_id` is the identifier defined in step 2, `npzd_do` is the (user-chosen)

name of the subroutine that initializes the model, `model%npzd` refers to the model-specific derived type defined in step 4, and `_INPUT_ARGS_DO_RHS_` is a preprocessor macro that defines a set of one or more additional arguments needed and managed by FABM.

Coupling to a physical host model

Prologue: variable storage in the physical host model

There are no explicit requirements on the physical model, except for the use of the interfaces documented below. However, in order to use FABM *efficiently*, data for environmental and biotic variables should be stored in a particular way by the host.

Specifically: At any given point in time, all variable values (at all points in space) of any single environmental or biotic variable must be accessible through a single Fortran 90 pointer. For spatial models, all data for a single variable must, thus, be stored in a single array (e.g., a one-dimensional array for column models, or a three-dimensional array for global circulation models). Ultimately, the variable will be represented by a Fortran 90 array slice, which means that the variable for storing is allowed to use additional dimensions (e.g., for time), which are then set to some index chosen by the host, and ignored by FABM.

Note that this does not require that the values for *all variables combined* are stored in a single contiguous block of memory. In practice, it does mean that the values for a *single* variable are best stored in a contiguous block of memory.

Variable values will be accessed through Fortran 90 pointers. Therefore, the Fortran variables at the side of the host that contain environmental or biotic data must be able to serve as the target of a Fortran 90 pointer. That means that it should either have the `target` or `pointer` attribute, or be a member of a derived type with one of these attributes.

The arrays on the side of the host could thus be defined with any of the following:

Three examples for a 3D model:

```
real,dimension(:,:,:),allocatable,target  :: temp,salt
real,dimension(nx,ny,nz),         target  :: temp,salt
real,dimension(:,:,:),            pointer :: temp,salt
```

Three examples for a 1D model:

```
real,dimension(:),allocatable,target  :: temp,salt
real,dimension(nz),           target  :: temp,salt
real,dimension(:),            pointer :: temp,salt
```

They can also be stored as members of a derived type:

```
type type_data
   real,dimension(:,:,:),allocatable  :: temp,salt
end type
```

In that case, any instance of type `type_data` needs to have the `target` or `pointer` attribute, for instance:

```
type (type_data),pointer :: dat
```

In this case, members of the derived type, `temp` and `salt`, do *not* need to have the `target` or `pointer` attribute.

Also, the data arrays may contain additional dimensions that should be set to a fixed index when used by FABM, for instance:

```
real,dimension(nx,ny,nz,3),target :: temp,salt
```

When sending data to FABM, the last dimension can be ignored by providing slices such as

```
temp(:,:,:,1) and salt(:,:,:,1)
```

This can be needed for models that use an additional dimension for the time step index, e.g., MOM4.

If one or more environmental or biotic variables are not stored in the required manner, it will be upon the coupling layer between FABM and its physical host to create arrays of the required structure for these variables, and make sure that the contained values are current upon each call to FABM. It may be clear that this is feasible for variables that have one or more dimensions fewer than the main grid, such as bottom/surface layers, but it will be (very!) computationally expensive if it is to be done for many variables that exist on the full grid.

Who does what? Requirements on the physical host model

At present, the physical host is responsible for maintaining (declaring and allocating) spatially explicit arrays for all state variables and diagnostic variables exported by FABM, and all external variables required by FABM. In the case of state variables, the host must also handle advection and diffusion (if modelled), as well as temporal integration.

State variable values may change through advection and diffusion (if represented in the host model), and by biogeochemical processes represented by FABM. Changes through advection and diffusion must be handled by the host model, that is, if it represents these processes it is also responsible for applying them to the FABM state variables, typically by invoking specific numerical algorithms. In some cases it is needed to provide these algorithms with the range that can be taken by the biogeochemical variables represented by FABM, or their positive-definiteness. This can be obtained from FABM structure describing the model (specifically, members `minimum` and `maximum` of the elements in array `model%info%state_variables`).

FABM provides the changes to the biogeochemical state variables in the form of temporal derivatives. That is, it provides the instantaneous rate of change, rather than updated state variable values at the end of a time step. It is the responsibility of the physical host model to perform the (numerical) temporal integration that produces updated state variable values[3].

[3] This is by design. By providing the temporal derivatives, the host model is free to perform temporal integration in any way it sees fit. That may involve splitting of the advection, diffusion and local sink/source operators, or integrate all of these together in a single operation. The latter is typically more accurate, but more difficult to implement, and it can be more difficult to respect constraints on variable ranges (e.g., positive definiteness).

Diagnostic variables can be calculated directly from the current model state (typically in combination with the local environment and the value of model parameters). They are, thus, not subject to advection and diffusion, and do not require temporal integration (although some models may want to provide time-averaged values of diagnostic variables, requiring temporal integration of a sort).

FABM generally depends on external variables such as temperature and light; the exact dependencies depend on the set of biogeochemical models that is active. The host model must maintain spatially-explicit arrays with the values of all external variables required by FABM. These may be variables that are already part of the physical host model (e.g., the array in which it stores the current temperature); if they are not part of the physical host model, the driver between FABM and host must declare and allocate the array, and obtain its values from an external data source.

Specifying the spatial context

Biogeochemical models operate on local conditions only. For performance reasons, however, they are always provided with the full spatially-explicit environment, as well as a set of integer indices telling the model which location it is operating upon. Thus, although biogeochemical models are in principle unaware of the non-local environment, they still require knowledge on the dimensionality of the environment and the indices needed to specify a single point in space, in order to access local variable values. To make this possible, preprocessor definitions are used.

The physical host must define the preprocessor variables that specify its spatial dimensions. These definitions are combined in a separate "include" file, which is per definition specific to the physical host, and is located at `drivers/$(FABMHOST)/fabm_driver.h`. Here, `$(FABMHOST)` is the name of the host model, e.g., gotm. In turn, this host-specific inluce file must reference (through `#include`) the FABM include file `include/fabm.h`.

The following preprocessor variables are used by FABM (specifically, in `include/fabm.h`), and can therefore be used in the host's `fabm_driver.h`:

symbol	meaning	1D	3D
`_FABM_DIMENSION_COUNT_`	The number of dimensions [rank] of fully spatially-explicit arrays.	`1`	`3`
`_LOCATION_`	A comma-separated list of variable names that will be used to specify a single point in the spatial domain. The number of variables must match `_FABM_DIMENSION_COUNT_`. The variable names defined here *cannot* be used in any routine of the biogeochemical models; they should thus not be commonly used names.	`kk`	`ii,jj,kk`
`_LOCATION_DIMENSIONS_`	A Fortran shape specifyer (comma-separated list of colons) that will be used to specify the dimensionality of spatially-	`:`	`:,:,:`

symbol	meaning	1D	3D
	explicit arrays. The number of colons must match _FABM_DIMENSION_COUNT_.		
_HORIZONTAL_IS_SCALAR_	The presence of this symbol specifies that a horizontal slice of the spatial domain is represented by a scalar. This is typical for column models.		
_LOCATION_HZ_	[only used if _HORIZONTAL_IS_SCALAR_ is not defined] A comma-separated list of variable names that will be used to specify a single point in a horizontal slice of the spatial domain. The variable names *must* be a subset of the variable names defined in _LOCATION_. The number of variables typically equals _FABM_DIMENSION_COUNT_ minus one (for depth), but it would equal _FABM_DIMENSION_COUNT_ for horizontal-only models. In that case, _LOCATION_HZ_ equals _LOCATION_.		ii,jj
_LOCATION_DIMENSIONS_HZ_	[only used if _HORIZONTAL_IS_SCALAR_ is not defined] A Fortran shape specifyer (comma-separated list of colons) that will be used to specify the dimensionality of a horizontal slice of the spatial domain. The number of colons typically equals FABM_DIMENSIONS minus one (for depth), expect for horizontal-only models, for which _LOCATION_DIMENSIONS_HZ_ equals _LOCATION_DIMENSIONS_.		:,:
_VARIABLE_1DLOOP_	[only used if _FABM_DIMENSION_COUNT_ exceeds 1] The name of variable that should be varied when iterating over the (outer dimension of) the spatial domain. This should be a variable name (typically the first, for performance reasons) used in the definition of _LOCATION_. [currently, the dimension that is iterated over must be depth]		ii

symbol	meaning	1D	3D
_LOCATION_1DLOOP_	[only used if _FABM_DIMENSION_COUNT_ exceeds 1] A comma-separated list of names of variables that should be kept constant when iterating over the (outer dimension of) the spatial domain. These variables names must match those defined in _LOCATION_, with the variable defined in _VARIABLE_1DLOOP_ omitted. [currently, the dimensions that are kept constant must be those describing the horizontal location; only depth can be iterated over]		jj,kk

Thus, for a 1D column model, a minimal `fabm_driver.h` would look like this:

```
#define _FABM_DIMENSIONS_ 1
#define _FABM_HORIZONTAL_IS_SCALAR_

#define _LOCATION_ kk
#define _LOCATION_DIMENSIONS_ :

#include "fabm.h"
```

Similarly, for a 3D model, a minimal `fabm_driver.h` could look like this:

```
#define _FABM_DIMENSIONS_ 3

#define _LOCATION_ ii,jj,kk
#define _LOCATION_DIMENSIONS_ :,:,:

#define _LOCATION_HZ_ ii,jj
#define _LOCATION_DIMENSIONS_HZ_ :,:

#define _VARIABLE_1DLOOP_ kk
#define _LOCATION_1DLOOP_ ii,jj

#include "fabm.h"
```

Note that the symbols may be defined in any order, but the inclusion of `fabm.h` must happen after all FABM-specific symbols are defined.

Providing routines for logging and error handling

Physical host models handle log messages and errors in different ways. In the simplest case, log messages are simply written to the console, and errors are handled by writing an error message

to the console and stopping the process with the stop statement. However, some models may want to redirect log messages (e.g., to an open file), or to perform additional clean-up when errors are encountered (e.g., by bringing down a parallel model running on multiple cores or machines gracefully).

Therefore, host models must provide a separate, typically small Fortran 90 module `fabm_driver`, contained in file `src/drivers/$(FABMHOST)/fabm_driver.F90`, that contains two subroutines: `log_message` and `fatal_error`. Subroutine `log_message` takes a single argument: the string to log. Subroutine `fatal_error` takes two arguments: the subroutine or function triggering the error (a string), and a string description of the error itself. It should be stressed that subroutine `fatal_error` may never return (generally by using the Fortran `stop` statement directly or indirectly). This prevents data corruption causes by modules continuing to operate after a fatal error occurred.

A simple `fabm_driver` model could look like:

```
module fabm_driver

   implicit none

   public

contains

   subroutine fatal_error(routine,errormsg)
     character(len=*), intent(in) :: routine,errormsg

     write (*,*) trim(routine)//': '//trim(errormsg)
     stop 1
   end subroutine fatal_error

   subroutine log_message(msg)
     character(len=*), intent(in) :: msg

     write (*,*) trim(msg)
   end subroutine log_message

end module fabm_driver
```

Integrating FABM in the build process of the host model

There are two ways to integrate FABM in the physical host:

1. Compile FABM as library, then reference it during linking phase of the host model. During compilation, the FABM make file must be provided by the name of the host model (environment variable `FABMHOST`), which should match the name of a directory in `src/drivers`. Note that this way of using FABM works only if the driver (specifically, `src/drivers/$(FABMHOST)/fabm_driver.F90`) does not depend on any part of the host model. If it does, FABM must be integrated in the build process of the host.

2. Integrate FABM directly in the compilation process of the host. This typically means adding the required source files (`src/fabm.F90`, `src/fabm_types.F90`,

`src/drivers/$(FABMHOST)/fabm_driver.F90`, `src/models/*/*.F90`) and the location of include files (`include`, `src/drivers/$(FABMHOST)`) to the scripts that compile the physical host model. The physical host model can then be compiled as usual, with support for FABM integrated.

Accessing FABM

1. Add a use statement that references the main FABM module, `fabm` defined in `src/fabm.F90`:

   ```
   use fabm
   ```

2. Define a pointer to an instance of the derived type `type_model`:

   ```
   type (type_model), pointer :: model
   ```

   This instance will hold all information on the selected biogeochemical model(s), including descriptive strings, state variable information, and parameter values.

3. Create the root model instance by calling `fabm_create_model`. This returns a pointer to an instance of `type_model`. Typically, the root model will be a model container that is created by calling `fabm_create_model` without arguments:

   ```
   model => fabm_create_model()
   ```

   After the root model is created, it is empty. It is populated with one or more child models by calling `fabm_create_model` once per child model, with the existing root model as parent argument:

   ```
   childmodel => fabm_create_model('co2sys',parent=model)
   childmodel => fabm_create_model('npzd',parent=model)
   ```

   The first argument to `fabm_create_model` is the model identifier, which can be a short model name (e.g., “npzd”, “mnemiopsis”) or an integer identifier (e.g., 1, 2). Thus, the two calls above add child models “co2sys” and “npzd” to the root. The driver of the physical host must allow the user to specify the biogeochemical models to run at runtime (e.g., by providing their names or integer identifiers in a namelist).

   Note: it is also possible to create a tree structure of models, nesting deeper than the one level demonstrated above. By creating empty child models (omitting the model identifier in `fabm_create_model`), and then using these container child models as parent in other calls to `fabm_create_model`, arbitrary model trees can be created. This can simplify model configuration, as models try to resolve the names of external dependencies locally, starting with itself, and every time moving one level up the tree only if the variable was not found among siblings. At present, FABM is coded such that a set of models arranged in a more complex tree topology (with deeper nested models) will get the same performance as a simple topology where all models reside directly below the root (this is possible because models are accessed as a flattened list at run

time). Users are therefore encouraged to use tree topologies wherever that makes conceptual sense.

4. Initialize the model tree by calling `fabm_init` on the root model:

   ```
   call fabm_init(model,nmlunit)
   ```

   with `nmlunit` being a unit specifier (integer) of a configuration file that has already been opened. FABM biogeochemical models may read namelists from this file during initialization. After initialization the caller (i.e., the driver between FABM and host) is responsible for closing the file.

5. Provide the model with (pointers to) the arrays that will hold the current values of environmental variables. Each array must contain all data for the variable across its relevant spatial domain. Variables may be defined on the full spatial domain (e.g., temperature), or on a horizontal slice only (typically for bottom or surface, e.g., surface wind speed, bottom roughness).

   Variables defined on the full model domain are provided by calling subroutine `fabm_link_data`, with the model, the variable identifier (string or integer) and the array slice that will hold the data. For instance, a 3D model might have an array with temperature and salinity values allocated as

   ```
   real,dimension(nx,ny,nz),target :: temp,salt
   ```

   These should be provided to FABM as follows:

   ```
   call fabm_link_data(model,varname_temp,temp)
   call fabm_link_data(model,varname_salt,salt)
   ```

   Note that `model` refers to the root of the model tree, created above, and `varname_temp` and `varname_salt` refer to standard variable identifiers defined by FABM.

   Similarly, variables defined on a horizontal slice of the domain are provided by calling subroutine `fabm_link_data_hz`. For instance, a 3D model might have an array with surface wind speed values allocated as

   ```
   real,dimension(nx,ny),target :: wind
   ```

   This should be provided to FABM as follows:

   ```
   call fabm_link_data_hz(model,varname_wind_sf,wind)
   ```

   For an overview of all environmental variables that ideally are provided in this manner (and their standard identifiers), see the list in `fabm_types.F90`.

   It is worth noting that `fabm_link_data` and `fabm_link_data_hz` may be called at any time during program execution, and that they may be called repeatedly with the same

variable identifier. Thus, it is possible to redirect FABM to another in-memory array slice during simulation, if applicable.

6. Create spatially-explicit arrays that will hold the values for the state variables exported by FABM. These typically include state variables defined on the full (3D pelagic) model domain, but they may also include benthic variables, which are defined on a horizontal slice of the domain only.

   A 3D model could define and allocate the following arrays to store state variable values:

   ```
   real,dimension(:,:,:,:),target,allocatable :: state
   real,dimension(:,:,:)  ,target,allocatable :: state_ben

   allocate(state    (nx,ny,nz,ubound(model%info%state_variables,    1))
   allocate(state_ben(nx,ny,   ubound(model%info%state_variables_ben,1))
   ```

   A 1D model would not include the x and y dimensions, but otherwise could do it the same way.

   After defining and allocating the arrays, they must linked to FABM by calling subroutines `fabm_link_state_data` and `fabm_link_benthos_state_data`, for pelagic and benthic variables, respectively. Each variable must be registered separately[4]. If arrays are defined as in the above example, these could be transferred as follows:

   ```
   do i=1,ubound(model%info%state_variables,1)
      call fabm_link_state_data(model,i,state(:,:,:,i))
   end do
   do i=1,ubound(model%info%state_variables_ben,1)
      call fabm_link_benthos_state_data(model,i,state_ben(:,:,i))
   end do
   ```

   It is worth noting that `fabm_link_state_data` and `fabm_link_benthos_state_data` may be called at any time during program execution, and that they may be called repeatedly with the same variable identifier. Thus, it is possible to redirect FABM to another in-memory array slice during simulation, if applicable.

7. Create spatially-explicit arrays that will hold the values for the diagnostic variables exported by FABM. These may include variables defined on the full (3D pelagic) model domain, as well as variables defined on a horizontal slice of the domain only, e.g., to diagnose air-sea exchange or benthic processes.

   A 3D model could define and allocate the following arrays to store the values of diagnostic variables:

   ```
   real,dimension(:,:,:,:),target,allocatable :: diag
   real,dimension(:,:,:)  ,target,allocatable :: diag_hz
   ```

[4] This allows different state variables to be stored in separate arrays, if that is appropriate in the physical host model.

```
allocate(diag    (nx,ny,nz,ubound(model%info%diagnostic_variables,   1))
allocate(diag_hz(nx,ny,   ubound(model%info%diagnostic_variables_hz,1))
```

After defining the arrays that will store state variable data, these must linked to FABM by calling subroutines `fabm_link_diagnostic_data` and `fabm_link_diagnostic_data_hz`, for variables defined on the full domain and on horizontal slices, respectively. Each variable must be registered separately. If arrays are defined as in the above example, these could be transferred as follows:

```
do i=1,ubound(model%info%diagnostic_variables,1)
   call fabm_link_diagnostic_data(model,i,diag(:,:,:,i))
end do
do i=1,ubound(model%info%diagnostic_variables_hz,1)
   call fabm_link_diagnostic_data_hz(model,i,diag_hz(:,:,i))
end do
```

It is worth noting that `fabm_link_diagnostic_data` and `fabm_link_diagnostic_data_hz` may be called at any time during program execution, and that they may be called repeatedly with the same variable identifier. Thus, it is possible to redirect FABM to another in-memory array slice during simulation, if applicable.

8. Access the model by the following subroutines:

    a. `fabm_do`: to get local temporal derivatives of pelagic state variables (unit: variable units/s).

    b. `fabm_get_vertical_movement`: to get local vertical movement rates (one value per pelagic state variable, unit: m/s, negative for downward movement, positive for upward movement) for the pelagic state variables. This is movement *additional* to any passive transport due to advection of the medium. Typically, it is used to specify sinking or floating of the variable(s).

    c. `fabm_get_light_extinction`: to get the local light extinction coefficient due to biogeochemical variables. This routine returns a single extinction coefficient (unit: /m) which describes attenuation by biogeochemical variables only – *not* by the medium, which is expected to be prescribed by the host model.

    d. `fabm_get_conserved_quantities`: to get the local value of the conserved quantities described by the model (names, long names and units of conserved quantities are available from the `type_model%info%conserved_quantities` array). This routine returns one value per conserved quantity, in units defined by the biogeochemical models during their initialization phase.

    e. `fabm_get_surface_exchange`: to get updated fluxes over the air-water interface. This routine returns one value (unit: variable units × m/s) per pelagic state variable.

f. `fabm_do_benthos`: to get updated fluxes between the benthos and the bottom layer of the pelagic. This routine returns one value per each model state variable (units: quantity per surface area per time, that is, variable units × m/s for pelagic variables, variable units/s for benthic variables, in both case positive for variable increases and negative for decreases), in separate arrays for the benthos and the pelagic.

g. `fabm_check_state`: to check the validity of current state variable values, and repair these if desired. By default, "repairing" implies clipping values to the valid range defined by the biogeochemical model, but a biogeochemical model can also provided custom logic for state variable repair.

   This routine can both be used to check the validity of the biogeochemical variables, for instance, to adaptively reduce the time step of integration if variable values become invalid, and to obtain "corrected" values needed to continue a simulation.

Each of these subroutines operates on a single location in the spatial domain. They are, therefore, always provided with a set of spatial indices that specify the current location (one index for a 1D model, three indices for a 3D model, none for a non-spatial 0D model). This includes horizontal-only routines `fabm_get_surface_exchange` and `fabm_do_benthos`; depth-explicit models must provide these with the depth index that corresponds to the surface or bottom layer[5].

Additional information on the model (e.g. long names, units for its variables) is present in the `info` member of the model, after the model has been initialized. For an overview of available metadata, see the definition of the info type (`type_model_info`) in `fabm_types.F90`, as well as the definitions of its contained types (`type_state_variable_info`, `type_diagnostic_variable_info`, etc.)

Acknowledgments

This work was funded by the European Project "Marine Ecosystem Evolution in a Changing Environment (MEECE)", supported within Theme 6 Environment of the Seventh Framework Programme for Research and Technological Development.

[5] This takes away the requirement that a single index value can be used to index the surface or bottom layer across the entire model domain. That requirement cannot usually be met by z-coordinate models with varying bathymetry: the index of the bottom layer (or alternatively, the surface layer) then typically varies in the horizontal, as deeper cells are not used in shallow regions.