| | | | |
|---|---|---|---|
| **Project** | NECCTON No 101081273 | **Start / Duration** | 2023-2026 |
| **Dissemination** | Public | **Nature** | Report |
| **Date** | 31 January 2024 | **Version** | 1.1 |

# Deliverable 3.1

# Best practices for implementing new BGC process modules in CMEMS/FABM

| **Deliverable Contributors:** | Name | Organisation | Role / Title |
|---|---|---|---|
| Deliverable Leader | Jorn Bruggeman | BB | Task leader |
| Contributing Author(s) | Veli Çağlar Yumruktepe | NERSC | WP3 contributor |
| | Paolo Lazzari | OGS | WP3 contributor |
| | Sarah Albernhe | CLS | WP3 contributor |
| | Ute Daewel | HEREON | WP7 contributor |
| | Nicolas Azaña Schnedler-Meyer | BB | WP3 contributor |
| Reviewer(s) | Diego Macías | JRC | External reviewer |
| | | | |
| Final review and approval | Stefano Ciavatta | MOi | Project lead |

This project has received funding from Horizon Europe RIA under Grant Number 101081273

| | | | |
|---|---|---|---|
| **Project** | NECCTON No 101081273 | **Deliverable** | 3.1 |
| **Dissemination** | Public | **Nature** | Report |
| **Date** | 1 February 2024 | **Version** | 1.1 |

**Document History:**

| Release | Date | Reason for Change | Status | Distribution |
|---|---|---|---|---|
| 0.0 | 22/01/2024 | Initial document, to be reviewed | Released | Internal |
| 1.0 | 31/01/2024 | Final revised report | Released | Public |
| 1.1 | 01/02/2024 | Fixed formatting/referencing issue | Released | Public |

**To cite this document**

Bruggeman, J., Yumruktepe, VC., Lazzari, P., Albernhe, S., Daewel, U., Schnedler-Meyer, NA (2024). D3.1 “Best practices for implementing new BGC process modules in CMEMS/FABM”. Deliverable report of project Horizon Europe NECCTON (grant 101081273), Doi 10.5281/zenodo.10604683

## TABLE OF CONTENTS

| Project | NECCTON No 101081273 | Deliverable | 3.1 |
|---|---|---|---|
| Dissemination | Public | Nature | Report |
| Date | 1 February 2024 | Version | 1.1 |

# 1. Executive Summary

A central objective of NECCTON is the construction of a new interlinked model framework for the Copernicus Marine Service (CMEMS). Existing CMEMS models, new models ported in WP3, and new process descriptions delivered by WP5-8 will build on this framework to ensure their interoperability and usability across CMEMS Monitoring and Forecasting Centres (MFCs). The framework chosen for this purpose is the Framework for Aquatic Biogeochemical Models (FABM), which underpins a wide range of marine biogeochemical models. This currently includes implementations of most biogeochemical models in use within CMEMS (ERSEM, BFM, PISCES, ECOSMO, ERGOM?) and more CMEMS biogeochemical models will be included in NECCTON (BAMHBI). A core strength of FABM is its ability to interface with numerous hydrodynamic models, including those used within CMEMS (NEMO, HYCOM).

FABM provides a well-documented minimal foundation for biogeochemical/ecosystem models, on top of which different developers have over the years introduced their own conventions for describing new biogeochemical and ecological processes. For example, they have introduced ad-hoc approaches for modularisation (e.g., ERSEM, BFM) and coupling to higher trophic level models (MIZER, ECOSMO E2E). In many cases these approaches use undocumented FABM functionality. In NECCTON, we have reviewed existing FABM-based models used by project partners, within CMEMS and beyond. Based on this review, we have (1) identified and consolidated a common set of development conventions ("best practices") that can support NECCTON's needs for process model development, interoperability and exchange, (2) refined and documented the previously-unpublished FABM functionality needed to support these conventions, (3) introduced new functionality ("connectors") to support two-way coupling to higher trophic level models, and (4) made numerous improvements in the FABM code that increase performance for configurations with very large numbers (> 500) of coupled model variables.

This document outlines "best practices" for biogeochemical/ecological model development within NECCTON and documents the FABM functionality introduced to support this. Specifically, (1) it describes the concepts of modular model development and its practical implementation in FABM, (2) it gives recommendations for the implementations of common types of process models, and (3) it provides guidance for model testing and distribution of code. These best practices will guide the implementation of new process models in NECCTON, e.g., FABM implementations of the remaining CMEMS models (BAMHBI, SEAPODYM-LMTL, the latest ERGOM release) in WP3. This will optimize the interoperability of NECCTON model components and the potential for exchanging/sharing them across the CMEMS MFCs.

# 2. Scope

This document formulates conventions for modular ecosystem model development and best practices for the implementation of specific types of process models developed by NECCTON partners, as well as by possible external stakeholders and future users of the NECCTON outputs, such as the Copernicus Marine Service developers. As part of this, it documents the Application

Programming Interfaces (APIs) in FABM that make this possible – some newly introduced for NECCTON, some previously available but undocumented. This document does not repeat the documentation of FABM existing core APIs, which is available from its wiki.

All functionality described here is currently available from the "neccton" branch of the public FABM repository (https://fabm.net/code). It will be integrated in FABM's main branch by the end of 2024. This deliverable will provide the basis for a live document that is to become part of the FABM wiki and will be further developed over the course of NECCTON in response to feedback and requests from partners.

# 3. Introduction

## About FABM

The Framework for Aquatic Biogeochemical Models (FABM, https://fabm.net, Bruggeman & Bolding, 2014) is a Fortran programming framework for biogeochemical models of marine and freshwater systems. Models written in this framework are directly usable in any hydrodynamic model with a FABM interface. This includes a wide variety of models with different spatial dimensions (0D, 1D, 3D), horizontal grids (unstructured and structured) and vertical coordinates (e.g., z, sigma, isopycnal). Notably, both 3D hydrodynamic models used within the Copernicus Marine Service (CMEMS; https://marine.copernicus.eu) – NEMO (Madec, 2008) and HYCOM (Bleck, 2002) – can interface with FABM.

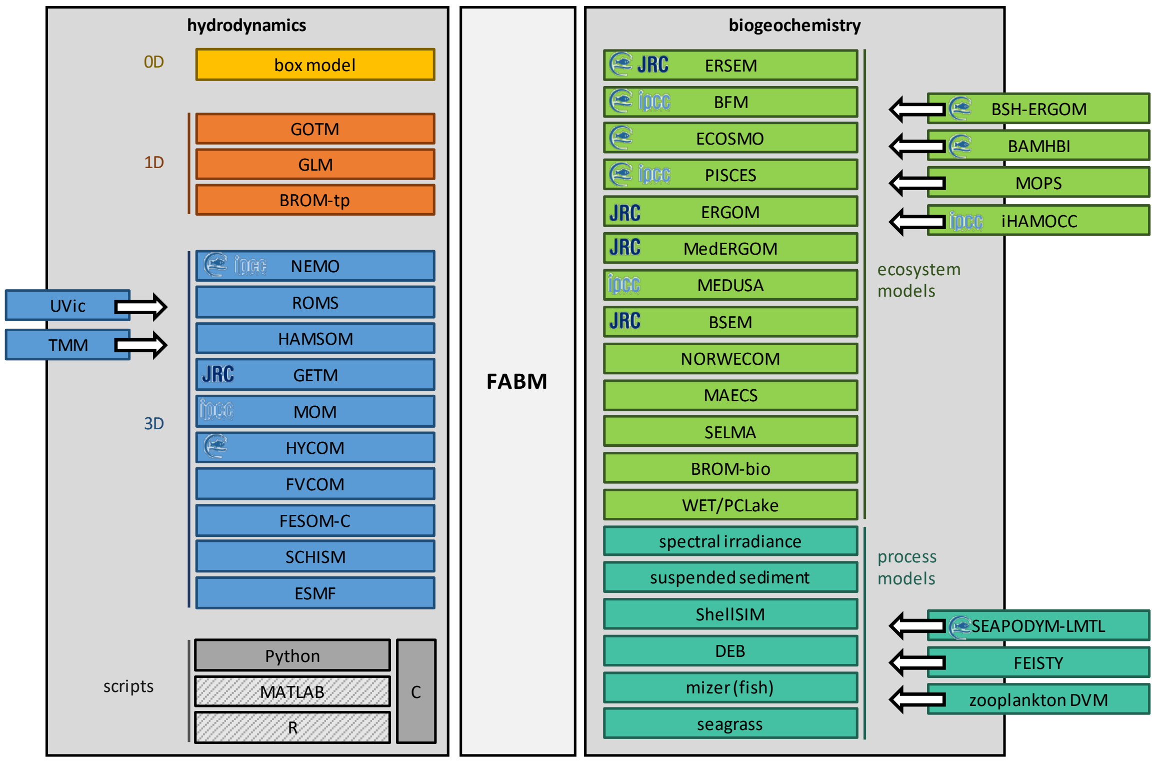


**Figure 1**. Hydrodynamic and biogeochemical models currently linked to FABM. Models shown with arrows are currently being developed within either NECCTON or in the Horizon Europe OceanICU project. Icons to the left of model names indicate their use in CMEMS ( ), earth system models ( ipcc ), and the JRC marine modelling framework ( JRC).

By design, FABM places minimal constraints on the design of biogeochemical models: while these need to use specific interfaces to define their variables, parameters and source terms, FABM does not specify which processes or variables should be represented, what units they use, and how the underlying code is structured.

## Modularity

A key design feature of FABM is that it allows multiple biogeochemical/ecosystem model components – from complete ecosystem models to individual processes – to be active at the same time. These processes, called “instances” in FABM, can freely exchange information. This functionality can be used to create modular ecosystem models, in which for example phytoplankton and zooplankton are coded separately as stand-alone building blocks. These can then be combined and coupled at runtime in FABM’s configuration file (fabm.yaml). The same functionality can also be used to combine ecosystem building blocks from different origins and authors, for example, to share components for oxygen, the carbonate system, suspended particular matter, or underwater radiative fluxes, or to combine lower and higher trophic level ecosystem models in two-way coupled setting, as we will do in NECCTON

As mentioned before, FABM does not specify the internal organization of biogeochemical/ecosystem models, and thus, does not *require* modular design. It is and will always remain possible to write monolithic biogeochemical models in FABM, with all processes coded in a single source file (or even subroutine). This is common practice for models with smaller numbers of variable and processes, e.g., the NPZD, Fasham, MedERGOM and BSEM models in FABM. However, some degree of modularisation of ecosystem models in FABM has proven benefits.

The first benefit of modularisation is that it enables distributed development (different processes are coded by different authors/institutes) and code sharing. Some processes lend themselves well for implementation in separate modules: those processes that have few, well-defined links to the rest of the ecosystem, and relatively complex internal processes and/or calculations. This commonly applies to models for the carbonate system, suspended particular matter, or underwater radiative fluxes. By coding these *once*, in a modular fashion, they become reusable by others – no source code changes necessary. For example, the operational version of ECOSMO (Yumruktepe et al., 2022) used in the Copernicus Marine Service leverages the ERSEM (Butenschön et al., 2016) carbonate system to add dissolved inorganic carbon, alkalinity and pH.

The second benefit of modularisation in FABM is that is allows for a single codebase to support different ecosystem configurations. In complex models, extensive modularisation allows you to write generic building blocks for a functional type, and then to re-use those multiple times with different parametrisations to compose the runtime ecosystem. For example, the present FABM implementations of ERSEM, BFM (https://www.bfm-community.eu, Vichi et al., 2020) and PISCES (https://www.pisces-community.org, Aumont et al., 2015) each have coded single generalized "phytoplankton" and "zooplankton" components, which are instantiated multiple times to create an ecosystem with multiple phytoplankton and multiple zooplankton types. ERSEM additionally used the same code-once, instantiate-multiple-times approach to add multiple types of benthic fauna and bacteria. This level of modularisation is particularly appealing for models that experiment with different numbers of species/populations/functional types, e.g., to test different representations of biodiversity.

Modularisation of biogeochemical/ecosystem models in FABM generally does not come at the expense of computational performance. Splitting up a code over multiple modules does not increase the number of spatially explicit fields, or the amount of data read and written whenever FABM is called. The main consequence of modularisation is that single spatial loops (e.g., those that calculate source terms) are split into multiple loops. As a result, the interior of individual loops becomes simpler. This often allows the compiler to perform additional optimizations. For example, it is not uncommon for compilers to give up on vectorizing loops due to the presence of many conditional statements ("if"). Splitting such a loop into multiple ones can allow the compiler to vectorize at least a subset of calculations; it may even allow the compiler to vectorize all new loops, e.g. because the number of conditional statements per loop was reduced.

## Aim of this document

This document is aimed at developers of biogeochemical/ecosystem models in FABM, either within the NECCTON consortium, or within the external community of potential users of the NECCTON outputs, such as developers of the Copernicus Marine Service MFCs. It outlines how one can develop modular and interoperable model components that can be shared among institutes, individuals, and setups.

This document exemplifies the existing and new functionality offered by of FABM by focusing on NECCTON and CMEMS needs, i.e. by referring to models (e.g. ERSEM, BFM, PISCES) and processes (e.g., modules for oxygen, carbonate chemistry, irradiance, suspended particulate matter, plankton functional types and higher trophic levels, among others) that will be revised or newly developed within the project.

Over the past 15 years, FABM has proven to be a stable basis for biogeochemical model development, flexible enough to accommodate a wide variety of processes and modelling approaches. This is demonstrated by the fact that some of the most complex biogeochemical models – e.g., ERSEM, BFM, PISCES used by NECCTON partners as well as by the Copernicus Marine Service – have successfully been implemented in FABM. Part of such implementation occurred in the framework of the EU Horizon 2020 SEAMLESS project (grant agreement 101004032), which was a Copernicus Service evolution project developing a new prototype (EAT) for new ensemble data assimilation methods to be implemented in the CMEMS MFCs (Bruggeman, Bolding, Nerger, Teruzzi, Spada, Skákala, et al., 2023; Bruggeman, Bolding, Nerger, Teruzzi, Spada, Wakamatsu, et al., 2023).

FABM's core functionality is extensively documented. We are not aiming to replicate this documentation in this document. However, over time, several conventions have emerged that build on top of FABM's core functionality to facilitate modular development of biogeochemical/ecosystem models. In practice, these conventions take the form of additional Application Programming Interfaces (APIs) that, for example, reduce the number of explicit coupling instructions users need to provide (in fabm.yaml) and that enable FABM to perform automatic unit conversions (e.g., between mg carbon m$^{-3}$, mol carbon m$^{-3}$, and mmol carbon m$^{-3}$) between models/variables being coupled. In general, the additional APIs make it easier to code self-contained process modules, without prior knowledge about the context they will be used in. Over the past 5 years, these APIs have become more widely adopted, and will form a cornerstone for development within NECCTON. However, these APIs have until now been poorly documented. This document aims to rectify this by:

1. reiterating the concept of model coupling within FABM;
2. describing how the new APIs fit into this and how they can be used;
3. providing specific recommendations for the base structure of new modules for a range of processes.

The latter in particular aims at increasing interoperability between model components developed by different authors, and more specifically, between processes descriptions developed within NECCTON for the different CMEMS Monitoring Forecasting Centres (MFCs) and beyond.

# 4. Coupling and modularity

## Common ingredients of FABM-based models

### Models, instances and modules

FABM places emphasis on the runtime configurability of biogeochemical/ecological models. Its runtime configuration file, fabm.yaml, does not only specify the values of parameters, but also which model components are to be activated. That list may be short, for example "NPZD and carbonate system", or, in more modular models, very long: "4 phytoplankton types, 2 microzooplankton types, 1 mesozooplankton types, …". The user specifies the list of components to activate as well as the connections between them. In FABM, each active component is called an "instance". In fabm.yaml, each instance is given a name, a pointer to the code that contains its actual implementation (e.g., all calculations), and any options that the code declares as user-configurable (e.g., parameter values). For example, the section in ERSEM's fabm.yaml that adds an instance "P1" begins with:

```
P1:
  long_name: diatoms
  model: ersem/primary_producer
  parameters:
    sum: 1.375
    …
```

Here, “model: ersem/primary_producer” points FABM to the code (specifically, the Fortran derived type) that implements the behaviour appropriate for the diatom instance. In this case, this is a generic primary producer code, flexible enough to describe any type of phytoplankton. Accordingly, new phytoplankton instances (P2, P3, P4) introduced later in this specific configuration file use the same “model: ersem/primary_producer” implementation. The takeaway message is that *multiple instances can use the same code*, though they will typically set different parameter values. In the sections below, we use “module” to refer to the implementation of model instances in source code.

### Variables

The central components of a FABM module are its *variables*: a model’s own state variables and diagnostics, and any external dependencies that describe the model environment:

- State variables are initialized at the start of the simulation and are changed by providing sources and sinks (instantaneous rates of change). In the case of pelagic (3D) state variables, they additionally change due to transport (advection, diffusion) calculated by the hydrodynamic model that FABM is embedded in. The current value of a state variable is determined by its initial value and its entire history of sources and sinks.
- Diagnostic variables can be calculated at any time from the model state and environment.
- Dependencies are variables that need to be provided by an outside component, e.g., temperature from the hydrodynamic model, or pH from a carbonate system module.

A variable is linked to a specific domain upon its declaration: the “interior” (pelagic), the water surface, or the bottom. More details about the types of variables and their handling is available on the FABM wiki.

### Routines

The routines that calculate source terms and diagnostics in a FABM-based model are specific to a single domain: “do” operates over the pelagic, “do_bottom” over the bottom, “do_surface” over the water surface. Each of these routines may operate on slices of the spatial domain, since they contain placeholders (preprocessor macros) for spatial loops, but they are not aware of the dimensionality of the host model (0D, 1D, 2D, 3D), the interpretation of different dimensions (e.g. depth vs. horizontal), their ordering (e.g., surface-to-bottom or bottom-to-surface), or land-sea masking. This is intentional: biogeochemical logic can then be written as local (grid-cell-specific) operations, and the host model and FABM can internally optimize spatial looping.

Where grid-aware operations are needed, they typically involved vertical loops, for example to loop from the water surface to the bottom to calculate the light field. For these, FABM supports one additional routine: “do_column”. This processes a single column in the pelagic, with explicit control over the direction (surface-to-bottom or bottom-to-surface). It should be noted that the “do_column” routine is often more computationally expensive to use than “do”; its use should therefore be limited to case where iteration over the vertical is truly *essential*.

### Source terms and surface/bottom fluxes

FABM-based models change their state variables by providing “sources”: the instantaneous rate of change of the variable due to the biogeochemical/ecological processes described by the model. In

FABM, these source terms can be positive (the variable increases) or negative (the variable decreases), so in practice they are equivalent to the sources-minus-sinks tracked in e.g. NEMO.

For interior [pelagic] state variables, FABM-based models can additionally prescribe surface and bottom fluxes: the instantaneous flux of the variable over the surface and bottom interface of the water column. These are positive for inward fluxes (i.e., fluxes that increase the pelagic variable).

### Aggregate variables

FABM-based models can register any variable they like, in whatever unit they like. However, to facilitate coupling between models, and to allow FABM to verify conservation of mass (e.g., totals of chemical elements), FABM asks models to declare how their variables contribute to "standard variables", e.g., total carbon, total nitrogen. These links are registered as part of model initialization by calling `add_to_aggregate_variable`. For example, the nitrogen-based NPZD example would use

```
call self%add_to_aggregate_variable(standard_variables%total_nitrogen, self%id_p)
```

in its phytoplankton module.

In models that track multiple chemical elements, this routine would be called multiple times for a biomass pool with constant stoichiometry. For example,

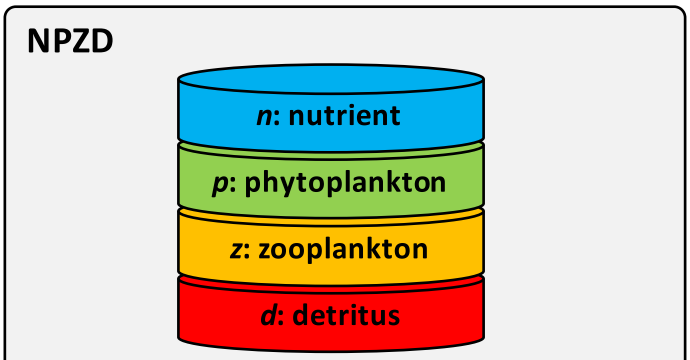
```
call self%add_to_aggregate_variable(standard_variables%total_nitrogen, self%id_p)
call self%add_to_aggregate_variable(standard_variables%total_phopshorus,
self%id_p, scale_factor=1.0_rk/16.0_rk)
call self%add_to_aggregate_variable(standard_variables%total_carbon, self%id_p,
scale_factor=106.0_rk/16.0_rk)
```

This would be appropriate for a phytoplankton pool represented in mmol N m$^{-3}$, and a carbon : nitrogen : phosphorus elemental ratio of 106 : 16 : 1. Note that the unit of standard variables is defined as part of the standard variable and therefore fixed (typically mmol m$^{-3}$); models that internally use different units must use the `scale_factor` argument to convert to standard variable units.

## A modularisation example

Every FABM module contains an `initialize` routine that registers its state variables, diagnostics and dependencies. In the simple Nutrient-Phytoplankton-Zooplankton-Detritus (NPZD) model shown in Figure 2, this routine calls FABM's `register_state_variable` routine four times:

```
call self%register_state_variable(self%id_n, 'n', 'mmol m-3', 'nutrients')
call self%register_state_variable(self%id_p, 'p', 'mmol m-3', 'phytoplankton')
call self%register_state_variable(self%id_z, 'z', 'mmol m-3', 'zooplankton')
call self%register_state_variable(self%id_d, 'd', 'mmol m-3', 'detritus')
```

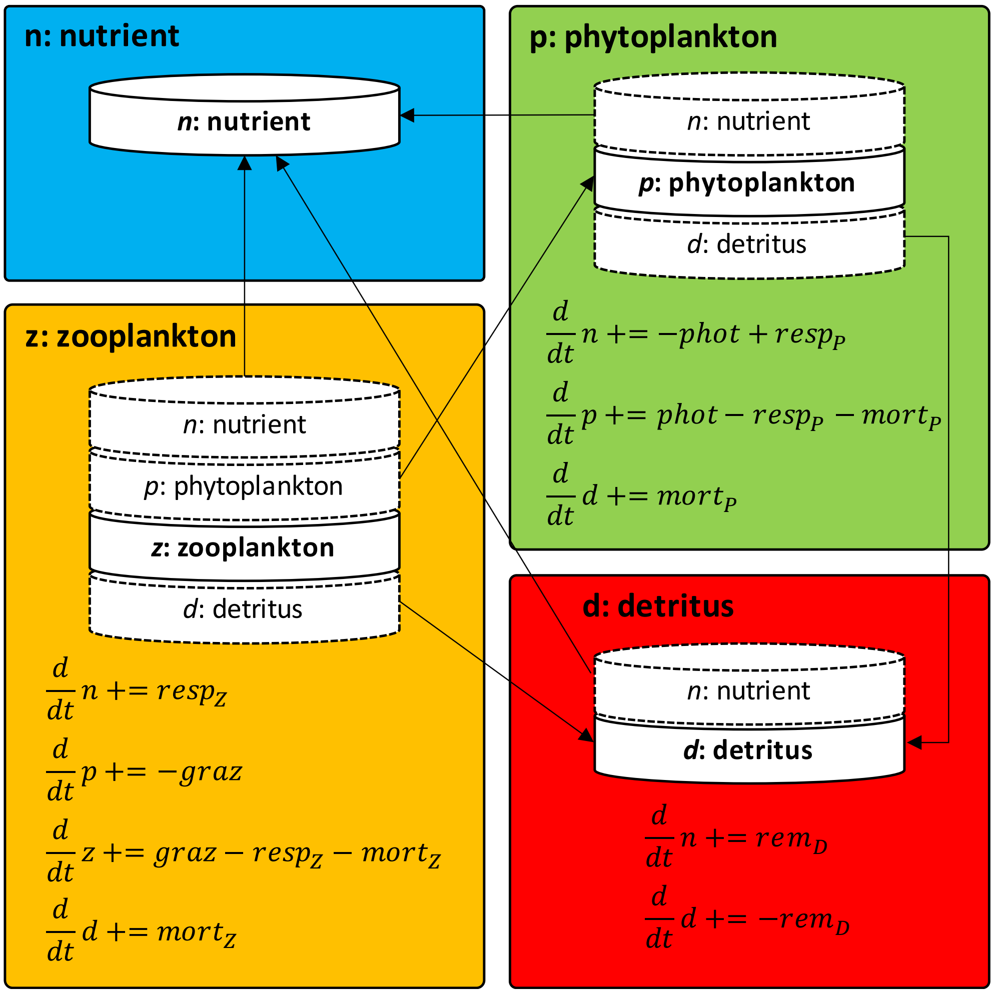


$$\frac{d}{dt} n \mathrel{+}= -phot + resp_P + resp_Z + rem_D$$
$$\frac{d}{dt} p \mathrel{+}= phot - resp_P - mort_P - graz$$
$$\frac{d}{dt} z \mathrel{+}= graz - resp_Z - mort_Z$$
$$\frac{d}{dt} d \mathrel{+}= mort_P + mort_Z - rem_D$$

**Figure 2.** The state variables and source terms of a simple NPZD model. In practice, such a model will also contain environmental dependencies, diagnostics and parameters. These are not shown here.

The source terms for each of these four state variables must be provided by the developer in an accompanying “do” subroutine, which processes the pelagic point by point. While doing so, it has read access to all registered state variables and dependencies, increment access to the state variables’ source terms, and write access to any registered diagnostics. Note that in FABM, source terms are *accumulated*: they are reset to 0 by FABM when the host (hydrodynamic) model asks for the source terms and then incremented by all model instances that are active.

If we were to modularise the NPZD model, that is, to split it in multiple components coded in different sources files, one intuitive approach would be to separate its four constituents: nutrients, phytoplankton, zooplankton, detritus. Each then gets its own source file, its own “initialize” routine, and its own “do” routine to specify pelagic source terms. The result could look like Figure 3. Source calculations are now distributed over three of the four modules (p, z, d). This also shows why in FABM, each module *increments* its source terms instead *setting* them; this ensures the total sources for each of the four state variables are accumulated to ultimately equal the equations shown in Figure 2.



n: nutrient
n: nutrient
p: phytoplankton
n: nutrient
p: phytoplankton
d: detritus
$\frac{d}{dt}n \mathrel{+}= -phot + resp_P$
$\frac{d}{dt}p \mathrel{+}= phot - resp_P - mort_P$
$\frac{d}{dt}d \mathrel{+}= mort_P$
z: zooplankton
n: nutrient
p: phytoplankton
z: zooplankton
d: detritus
$\frac{d}{dt}n \mathrel{+}= resp_Z$
$\frac{d}{dt}p \mathrel{+}= -graz$
$\frac{d}{dt}z \mathrel{+}= graz - resp_Z - mort_Z$
$\frac{d}{dt}d \mathrel{+}= mort_Z$
d: detritus
n: nutrient
d: detritus
$\frac{d}{dt}n \mathrel{+}= rem_D$
$\frac{d}{dt}d \mathrel{+}= -rem_D$

**Figure 3**. The first iteration of a modular NPZD model. Each coloured box represents a separate module, coded in a stand-alone source file. Cylinders represent state variables. Those with solid borders and bold font are "owned" by the module itself, those with dashed borders are state dependencies. Arrows represent the coupling links make at runtime based on the specification in fabm.yaml. For example, the nutrient state dependency of the "phytoplankton" instance is fulfilled by coupling to the nutrient state variable in the "nutrient" instance.

Each module now makes just one call to `register_state_variable`: the nutrient module registers "n", the phytoplankton module register "p", the zooplankton module register "z" and the detritus module registers "d". Thus, the final combined model still has only four state variables (tracers). However, the modules compute source terms that affect not just their own state variable, but several others as well. For example, in the phytoplankton module, the difference between photosynthesis and respiration comes at the expense of the nutrients; the loss of phytoplankton due to mortality increases detritus. To represent this, FABM has the concept of *state variable dependencies*: state variables that are not "owned" by the module itself but must come from another active model instance. For example, in addition to its own state variable, the phytoplankton module registers two state dependencies by calling `register_state_dependency`:

```
call self%register_state_variable(self%id_p, 'p', 'mmol m-3', 'phytoplankton')
call self%register_state_dependency(self%id_n, 'n', 'mmol m-3', 'nutrients')
call self%register_state_dependency(self%id_d, 'd', 'mmol m-3', 'detritus')
```

It is then free to increment source terms for each of these three state variables.

There is no direct link (e.g., a Fortran use statement) made from the phytoplankton source code to the nutrient and detritus source codes. Instead, the phytoplankton module is a stand-alone code that registers its dependencies on nutrients and phytoplankton, but then leaves these to be resolved at runtime. In FABM, the specification of which modules to activate, and how they are to be coupled, is deferred till runtime; there is no global compile-time register of active modules and available variables. One benefit of this is that each of the four modules is self-contained and can be compiled independently (if desired, all four can be compiled in parallel). It also means that the conversion of the original NPZD code to N, P, Z, and D components can be done in a straightforward manner:

1. create four copies of the original code
2. replace `register_state_variable` by `register_state_dependency` for all but the "owned" state variable
3. drop redundant variables and source terms
4. rename externally visible Fortran modules and types (e.g., npzd → p).

How should we allocate the individual source terms to the new stand-alone modules? For example, why is the change in detritus due to plankton mortality ($mort_P$ and $mort_Z$ ) specified in the phytoplankton and zooplankton modules, rather than to the detritus module? Part of the answer is that we want to compute each process just once, as the underlying calculation may be complex and computationally expensive; we then use the result of the calculation in as few places (modules) as possible to minimize code complexity. However, this rule of thumb would still permit mortalities $mort_P$ and $mort_Z$ to be calculated in the detritus module, and their impact on p, z and d to be represented there. The argument against doing this is that it would make the detritus module dependent on the presence of the phytoplankton and zooplankton modules. Similarly, if we were to move the calculation of the grazing rate to the phytoplankton module, it would become dependent on the presence of the zooplankton module. The current partitioning of source terms has the appealing feature that it is possible to build mini-ecosystems with a subset of modules, without any code changes: it is perfectly feasible to compose an NPD model where zooplankton has been removed, or even an ND model that describes remineralization of detritus only (although in the absence of a detritus source, its behaviour will not be very interesting) – all without changes to the source code. In practice, we have found that basing modules on "integral physical entities" (chemical compounds such as nutrients and oxygen, generalized compounds such as classes of particulate and dissolved organic matter, plankton functional types) provides a degree of modularity that is both useful and intuitive. Nevertheless, it remains perfectly possible to write modules for larger part of an ecosystem (e.g., the complete NPZD example) or for individual processes.

The above is sufficient to deliver a modular NPZD model. However, the individual components in Fig. 3 still clearly expect to embedded within the original NPZD context. This is most obvious in their naming of dependencies. For example, zooplankton still refers to "phytoplankton" as it prey, even though it could in principle consume any type of prey: it might be coupled to detritus as "prey" to make it a detritivore. Another loose end is the naming of the "owned" state variables. Since the instance name already describes the component (nutrient, phytoplankton, etc.), these is no need for the internal state variable to repeat that name – since in output, variables are named <instance>_<variable>, it would lead to redundancy in variable names, e.g., short names such as "p_p" and long names such as "phytoplankton_phytoplankton". Therefore, the final stage in modularizing a FABM-based model is often the renaming of variables and dependencies to (1) indicate the newly acquired flexibility, and (2) avoid repletion within variable names. This is shown for the NPZD model in Fig. 4. The changed variable names reflect that the model has in essence become a collection of four generic building blocks, which can be put together in completely new configurations. For example, is very feasible to add a second zooplankton instance that feeds on detritus instead of phytoplankton.

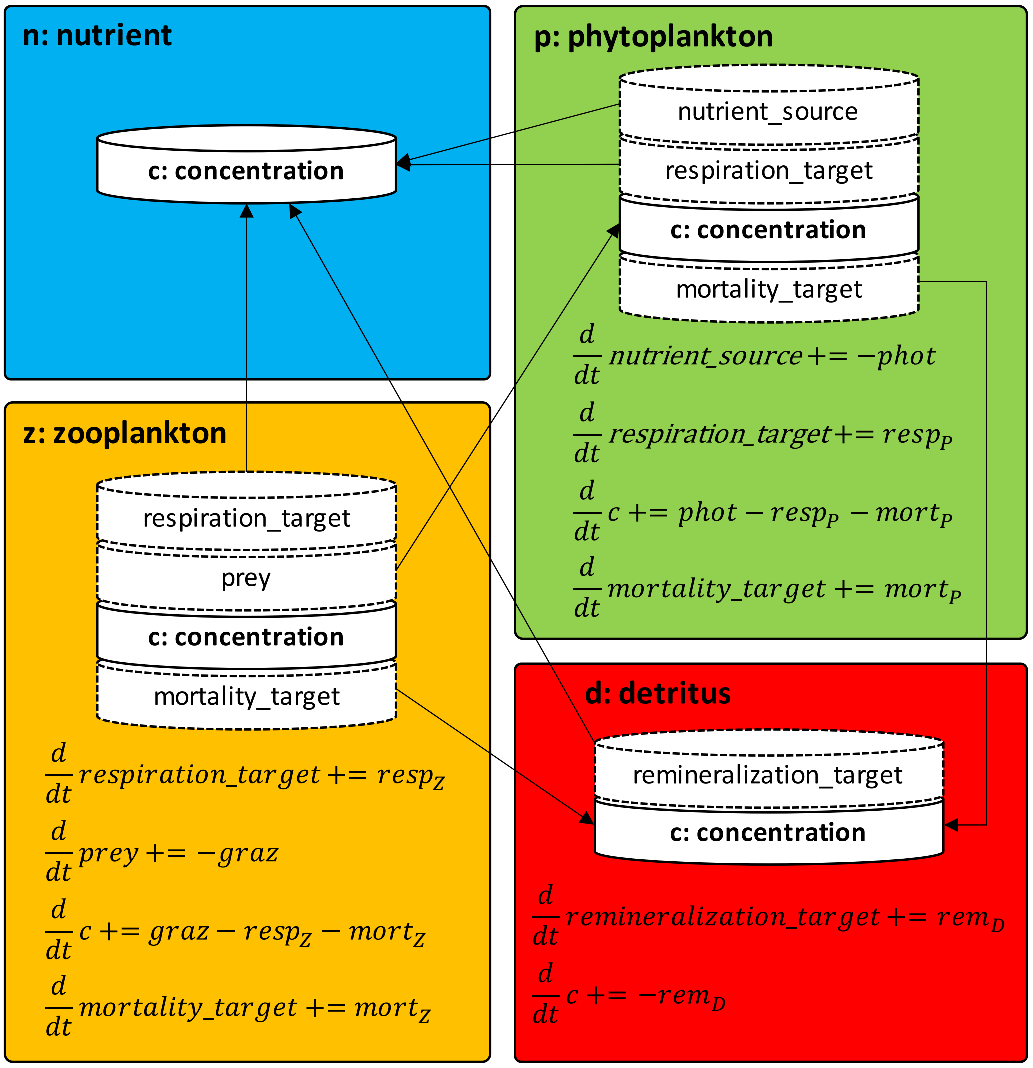

n: nutrient
c: concentration
p: phytoplankton
nutrient_source
respiration_target
c: concentration
mortality_target
$\frac{d}{dt}$ nutrient_source += −phot
$\frac{d}{dt}$ respiration_target += $resp_P$
$\frac{d}{dt}$ c += phot − $resp_P$ − $mort_P$
$\frac{d}{dt}$ mortality_target += $mort_P$
z: zooplankton
respiration_target
prey
c: concentration
mortality_target
$\frac{d}{dt}$ respiration_target += $resp_Z$
$\frac{d}{dt}$ prey += −graz
$\frac{d}{dt}$ c += graz − $resp_Z$ − $mort_Z$
$\frac{d}{dt}$ mortality_target += $mort_Z$
d: detritus
remineralization_target
c: concentration
$\frac{d}{dt}$ remineralization_target += $rem_D$
$\frac{d}{dt}$ c += −$rem_D$

**Figure 4.** The final iteration of the modularized NPZD model, in which variables and dependencies have been renamed to emphasize the flexibility of the four building blocks.

## Conservation checks

One of the few overarching principles in the development of biogeochemical models is that they should conserve mass. This makes mass conservation checks one of the most valuable tools in the toolbox of biogeochemical model developers. These checks become even more powerful when models are modularised.

Mass conservation is often verified by summing the total of each represented chemical element (e.g., nitrogen) across all biogeochemical state variables, and ascertaining whether this total stays the same (or if allowing for numerical inaccuracies and drift: “nearly the same”) over time within a closed model domain.

An equivalent mass conservation check can be made at the level of source terms in the biogeochemical model: if we sum the contributions of all source terms[1] to the total change of a chemical element, this sum should be numerically indistinguishable from 0 (in practice: less than $10^{-15}$ times the largest source term, if using double precision). For example, summing all source terms in the nitrogen-based NPZD model (Fig 2) shows that the change in total nitrogen (n+p+z+d) equals 0. Such checks on the rate of change are generally more precise, and therefore more informative, than checks on the model state, because unlike the latter, (a) they are not influenced by external sinks and sources (e.g., open boundaries, rivers, precipitation), (b) they are not influenced by inaccuracies or conservation issues in numerical schemes for transport and time integration, or any clipping that these apply, and (c) any errors do not accumulate over time.

Mass conservation applies to all sources combined (e.g., Fig. 2), but also at the level of individual model instances: the sum of all sources of a chemical element within a single instance should also equal 0. It is easily verified in Figs 3 and 4 that the sum of all source terms is indeed zero within each of the four instances. Such checks are very useful in complex models with tens to hundreds of source terms. If these models show a gap in the mass balance, it is often tedious and time-consuming to determine which of the many source terms is at fault. However, if such a model is split over tens of model instances, mass conservation of sources can be checked on each of these instances individually. The problem then reduces to first identifying the offending module (e.g., “it is due to the pelagic source terms computed in the diatom module”), and then, within that, the offending term. This is a much quicker procedure.

Fortunately, FABM makes such granular mass conservation checking easy: if you add a single “check_conservation: true” switch at the top level of fabm.yaml, FABM will calculate the total change in all known conserved quantities for every active module, e.g., the change in total nitrogen due to phytoplankton, the change in total nitrogen due to zooplankton, etc. This is done separately for the pelagic (i.e., sources computed from the “do” routine), the bottom (“do_bottom” routine) and surface (“do_surface”). Each of the resulting sums is available as a diagnostic for output, with names of <instance_name>_change_in_<standard_variable_name><_domain>_calculator_result,

[1] In FABM, sources can be positive or negative as described under “Source terms and surface/bottom fluxes”. Thus “source terms” here is equivalent to the “sources minus sinks” used elsewhere (Madec, 2008).

with <domain> being empty (for the pelagic), “at_surface”, or “at_bottom”. This greatly increases the number of metrics available to for checking model validity.

For FABM to generate appropriate conservation diagnostics, it needs to be aware of the link between model variables (e.g., “phytoplankton”) and conserved quantities (e.g., “total nitrogen”). These links are registered as part of model initialization as described in section “Aggregate variables”.

Finally, it should be noted that the generation of source conservation diagnostics is computationally costly: First, it requires FABM to trap each source term before it is added to the accumulated per-variable sources. Second, it will result in the calculation of large numbers of sums (number of model instances × number of conserved quantities × 3 for interior, bottom and state). Thus, this feature would typically be active during testing, but deactivated in production/operational simulations.

## Coupling specification

As mentioned above, the coupling between modules is deferred until runtime. Specifically, the links between modules are set in the “coupling” sections of FABM’s runtime configuration file, fabm.yaml. For the NPZD example from Fig. 4, that would look like:

```yaml
instances:
  n:
    model: examples/npzd/n
  p:
    model: examples/npzd/p
    coupling:
      nutrient_source: n/c
      respiration_target: n/c
      mortality_target: d/c
  z:
    model: examples/npzd/z
    coupling:
      prey: p/c
      respiration_target: n/c
      mortality_target: d/c
  d:
    model: examples/npzd/d
    coupling:
      mineralization_target: n/c
```

FABM also supports an implicit form of coupling through the use of “standard variable” identities. If one module specifies that it depends on a standard variable named “x” and another module registers one of its state or diagnostic variables with the same identify “x”, FABM will couple them automatically. In this case, no entries in fabm.yaml are necessary.

## Particles: grouped couplings and generalized access

The coupling functionality described above is suitable for many purposes, but it suffers from two problems:

1. All variable links between models have to be specified explicitly in the FABM configuration file, fabm.yaml. For more complex models, e.g., those with many functional types and multiple elemental cycles (e.g., Fig 5), this can quickly result in coupling statements dominating the configuration file altogether. For example, ERSEM's mesozooplankton couples to 9 prey types, and for each it needs access to 5 constituents (C, N, P, Si, $CaCO_3$). Thus, its coupling specification in fabm.yaml would require 9×5=45 coupling statements, each on its own line. This becomes difficult to manage and error prone.
2. Variable links can only be made if the supplying and receiving model both use the same units. This is fine when components are all designed and implemented by the same institute/author, but problematic when they come from different institutes/authors. That is because different authors often prefer different units (e.g., ERSEM uses mg C, mmol N, mmol P where PISCES uses mol C throughout). This would hinder interoperability of model components.

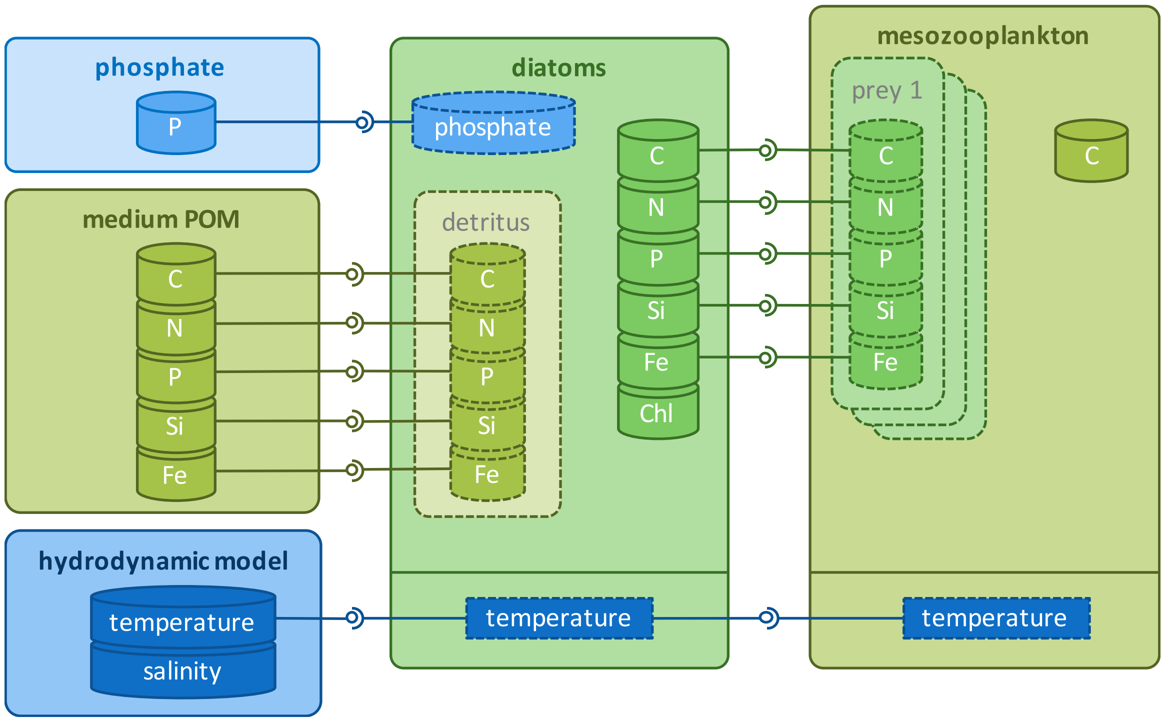


**Figure 5.** Coupling in more complex models with multiple functional types and multiple chemical elements. Rectangles with rounded corners represent model instances, cylinders represent state variables, rectangles read-only dependencies. Variables with solid borders are “owned” by the instance that contains them; those with dashed borders are dependencies. Simple variable dependencies (e.g., phosphate in diatoms, temperature in diatoms and mesozooplankton) are resolved as before (see “Coupling specification”). Other dependencies are grouped. For example, the diatom module registers dependencies on the individual components of detritus (where waste is to be deposited). These components are grouped together by a single dependency on a model instance named “detritus”. As a result, the user only needs to specify a target for “detritus” (e.g., to point it to the medium-size particulate organic matter pool). Connections to its individual C, N, P, Si, Fe constituents are made automatically.  Similarly, mesozooplankton accepts multiple prey, each quantified by several constituents (C, N, etc.). For each prey, these are grouped via a dependency on a single model instance “prey1”, “prey2”, etc. These high-level couplings (“prey1: diatoms”) are the only ones the user needs to specify in fabm.yaml.

To address these issues, FABM introduces the concept of a “particle model”, representing a coherent physical entity. It is implemented in **`type_particle_model`**, defined in the **`fabm_particle`** module. Models that inherit from this type acquire the ability to couple to entire model instances by name, and to automatically set up links to variables from such a coupled instance. For example, ERSEM’s mesozooplankton couples to 9 prey instances (set in fabm.yaml), and from each of those requests (in code) its total carbon, nitrogen, phosphorus, silicon and calcite. To make this possible, a new section was added at the end of the mesozooplankton initialize routine:

```
do iprey = 1, self%nprey
   write (index,'(i0)') iprey
```

```
    call self%register_model_dependency(self%id_prey(iprey), 'prey' // trim(index))
    call self%request_coupling_to_model(self%id_preyc(iprey), self%id_prey(iprey),standard_variables%total_carbon)
    call self%request_coupling_to_model(self%id_preyn(iprey), self%id_prey(iprey),standard_variables%total_nitrogen)
    call self%request_coupling_to_model(self%id_preyp(iprey), self%id_prey(iprey),standard_variables%total_phosphorus)
    call self%request_coupling_to_model(self%id_preys(iprey), self%id_prey(iprey),standard_variables%total_silicate)
    call self%request_coupling_to_model(self%id_preyl(iprey), self%id_prey(iprey),total_calcite_in_biota)
end do
```

The APIs `request_model_dependency` and `request_coupling_to_model` were introduced with type_particle_model. **`request_model_dependency`** registers a dependency on an entire model instance, identified by name and to be coupled at runtime in fabm.yaml. For instance, in the above, the mesozooplankton module registers a dependency on “prey1”, “prey2”. Each of these is a model instance (e.g. “diatoms”), i.e., not a single variable but a collection of variables. In the next step, **`request_coupling_to_model`** is called to request specific variables from the coupled instance (e.g., “diatom carbon”). These requests use standard variables (e.g., total_carbon), which allows them to work independent of implementation details (e.g., variable names and units) of the coupled instance. In the above example, totals of different chemical elements (`total_carbon`, etc.) from each prey instance are linked those totals to previously registered dependencies (`self%id_preyc`, etc.). Thus, the above lines supplement but not replace the variable-specific registration commands in mesozooplankton’s “initialize” routine. The lines are added to group couplings together (the user can couple “prey1” in one go in fabm.yaml, instead of micromanaging links to “prey1c”, “prey1n”, etc.) and to link them based on standard identities (total_carbon, etc.) with known units.

In the example above, total contained carbon, nitrogen, phosphorus, silicon and calcite are obtained for every prey type. The standard variables used for this purpose, e.g., total_carbon, have fixed units, typically mmol m$^{-3}$. FABM *guarantees* that the requesting model (mesozooplankton) will receive the requested variable in these fixed units. This is non-trivial, as a few examples show:

- Many prey types do not have silicon or calcite. This is known to FABM, as the prey instance (e.g., microzooplankton) then does not call add_to_aggregate_variable for total_silicate or total_calcite_in_biota. In response, FABM couples mesozooplankton’s corresponding id_preys(iprey) and/or id_preyl(iprey) to a field filled with zeros.
- ERSEM’s functional types express carbon in mg C m$^{-3}$, not mmol C m$^{-3}$ implied by the total_carbon standard variable. This is known to FABM, as the prey instance (e.g., microzooplankton) calls add_to_aggregate_variable for total_carbon with a scale factor of 1/12.011 (12.011 being the atomic mass of carbon). In response, FABM creates a temporary variable for prey in mmol C m$^{-3}$ in the background. FABM calculates the value of this new variable on demand by dividing prey carbon in mg C m$^{-3}$ by 12.011 and providing the result to mesozooplankton’s id_preyc(iprey).
- If a prey contains multiple state variables that contribute to total carbon, FABM calculates the sum of these variables in mmol C m$^{-3}$ on demand and provides that to mesozooplankton.

Thanks to such automatic unit conversions, the predator does not need to be aware which elements the prey does or does not contain. Correct links between predator and prey are made even if they internally operate in different units.

The particle-specific coupling described above allows the requesting model to always see (i.e., read) all desired prey properties in predictable units. However, this functionality is not sufficient to apply grazing losses to prey. Predators generally calculate a specific loss rate (e.g., in $d^{-1}$) that applies to prey biomass as a whole, i.e., to all its constituents. If that prey is defined in carbon units only, with fixed C:N and C:P, that specific loss rate should apply only to its carbon state variable. If the prey has variable stoichiometry with explicit state variables for C, N and P, the specific loss rate should apply to all three state variables. In both cases, the predator sees non-zero values for prey carbon, nitrogen and phosphorus – it cannot tell the prey types apart. How can it apply the correct loss rates without knowing the internal structure of the coupled prey instance? The current solution to this is to have the predator loop over all state variables of the prey, without knowing their identity, and to then apply the same specific loss rate to each. For example,

```
do iprey = 1, self%nprey
   do istate = 1, size(self%id_prey(iprey)%state)
      _GET_(self%id_prey(iprey)%state(istate), preyP)
      _ADD_SOURCE_(self%id_prey(iprey)%state(istate), -sprey(iprey)*preyP)
   end do
end do
```

Here, `sprey(iprey)` is the previously calculated specific loss rate, different for each prey type.

An added bonus of this approach is that any non-tracked state variables of the prey are destroyed along with all others, since they are included in the “state” array in the example. In ERSEM, that means that chlorophyll of phytoplankton functional types is (correctly) destroyed by predation, even though the predator code does not handle chlorophyll explicitly.

## Child models and mapping across domains

Even for a module for a single physical particle (e.g., a phytoplankton or zooplankton functional type), it can be convenient to split calculations over different routines. In some cases, it is preferable or even unavoidable to place these routines into separate modules. For example:

ERSEM’s mesozooplankton switches between active and overwintering behaviour based on the quantity of depth-integrated prey. All its source terms and diagnostics are pelagic and thus implemented in a “do” routine. For the overwintering behaviour, calculations in this routine have a dependency on the depth integral of the sum of all its 9 prey types. This dependency is registered at module (mesozooplankton) level. In FABM, calculation of depth integrals must be done in a “do_column” routine; summation of pelagic variables is best done in a “do” routine. It is worth recalling that the “do” routine cannot do the job on its own, as it performs local operations in the pelagic without being of different spatial dimensions and their interpretation; moreover, it typically only sees a fraction of the pelagic at a time, and this fraction typically does not encompass all of the vertical.

Thus, a combination of “do” and “do_column” is unavoidable. The ideal processing order would be:

1. a “do” routine calculates the sum over all prey;
2. a “do_column” routine depth-integrates this sum;
3. a “do” routine uses the depth integral to determine whether to activate overwintering; based on that, it calculates all source terms and diagnostics.

This three-step logic cannot be implemented in a single mesozooplankton type, since that only has a single “do” routine. The solution to this issue is to place the calculation of the sum and of the depth integral in separate modules, created as children of the mesozooplankton module (i.e., they are created and configured in code, not by the user through fabm.yaml). To support this, FABM allows modules to allocate and configure model instances of any type and to add them by calling the FABM’s built-in **add_child** subroutine.

FABM already has built-in modules for summation and depth integration. Therefore, a simple implementation in mesozooplankton’s “initialize” could look like:

```
use fabm_builtin_sum, only: type_weighted_sum
use fabm_builtin_depth_integral, only: type_depth_integral

class (type_weighted_sum), pointer :: totprey_summation
class (type_depth_integral), pointer :: totprey_integrator

! Set up the sum over all prey types
allocate(totprey_summation)
do iprey = 1, self%nprey
   write (index,'(i0)') iprey
   call totprey_summation%add_component('../prey' // trim(index) // 'c')
end do
call self%add_child(totprey_summation, 'totprey_summation')

! Set up the depth integral
allocate(totprey_integrator)
call self%add_child(totprey_integrator, 'totprey_integrator')
call totprey_integrator%request_coupling('source', '../ totprey_summation/result')

! Couple our dependency on depth-integrated prey to the child model calculating it
call self%register_dependency(self%id_intprey, 'intprey', 'mg C m-2', 'depth-
integrated prey')
call self%request_coupling(self%id_intprey, '../totprey_integrator/result')
```

This creates two children of mesozooplankton: totprey_summation calculates the sum of all prey across the pelagic, totprey_integrator subsequently calculates its depth integral. The terms to include in the summation are set by calling add_component. This is a custom routine of type_weighted_sum that ensures a dependency on the term will be set up and coupled to the specified named variable (the first argument to add_component). The field to integrate is set by coupling the "source" of the integrator to the "result" of the sum, by calling **`request_coupling`**. The final dependency on depth-integrated prey is linked to the result of the depth integration by another call to request_coupling.

These operations happen under the hood and are not visible to the user of your model, or configurable in fabm.yaml, since the responsible models are created and added (with add_child) and coupled (with request_coupling) automatically.

You may note that the above design can only work if FABM respects the intended call order: summation, then depth integration, then mesozooplankton sources. This is the case. FABM infers from the dependencies between the three modules which needs to be called first and guarantees that this will be done correctly.

## Your diagnostic, my state variable

The standard rules for coupling variables between modules are straightforward: models can either register a *dependency* (access: read-only) that is fulfilled by coupling to a state variable or diagnostic from another model, or they can register a *state variable dependency* (access: read value, increment sources) that must be fulfilled by coupling to a state variable from another model (e.g., Fig 3). FABM enforces this: you cannot couple a state variable dependency to a diagnostic variable, as there is then no destination for the source terms associated with the state variable dependency.

However, in some scenarios it is helpful if these rules can be relaxed. For example, many biogeochemical models calculate the change in alkalinity that is associated by their uptake or exudation of dissolved compounds. Thus, they register a state variable dependency on total alkalinity. Still, someone may want to use these model components with a simple carbonate system in which alkalinity is parameterized as function of salinity: it is a diagnostic rather than a state variable. The simple carbonate module can prepare for this scenario by explicitly stating that other modules may try to provide source terms for its alkalinity variable. This is done by registering it with argument **`act_as_state_variable=.true.`** FABM will then allow another module (e.g., phytoplankton) with a state variable dependency on alkalinity to couple to this diagnostic. Any source terms (or surface/bottom fluxes) that it provides for alkalinity will then simply be discarded.

A more interesting scenario is when the source terms (or surface/bottom fluxes) for such a diagnostic-masquerading-as-state-variable should *not* be discarded but collected and processed in some way. For example, when ERSEM is configured to use parameterized alkalinity, it calculates an abiotic reference value of alkalinity from salinity, but it simultaneously tracks the change in alkalinity due to biological processes in a new state variable called "bioalkalinity". The (diagnostic) total alkalinity is computed as the sum of the salinity-based reference value and the current bioalkalinity. In this way, the impact of biogeochemical processes on alkalinity is still represented. To achieve this in FABM, the diagnostic total alkalinity is registered with `act_as_state_variable=.true.` ERSEM's modules will couple to this variable and calculate source terms for it. These terms are now no longer discarded, but applied instead to the bioalkalinity state variable. That is achieved in a few lines:

```
call self%register_diagnostic_variable(self%id_TA_diag,'TA','mmol/m^3','total
alkalinity', act_as_state_variable=.true.,
standard_variable=standard_variables%alkalinity_expressed_as_mole_equivalent)

call
self%register_state_variable(self%id_bioalk,'bioalk','mmol/m^3','bioalkalinity')

call copy_fluxes(self,self%id_TA_diag,self%id_bioalk)
```

Here, the **`copy_fluxes`** routine takes the total sources for the diagnostic total alkalinity and redirects them to state variable bioalkalinity.

The benefit of this implementation is that all other modules (phytoplankton etc.) do not need to know whether alkalinity is purely diagnostic, diagnostic with representation of bioalkalinity, or fully prognostic (a normal state variable). Their code, configuration and coupling can stay exactly the same.

The ability to make diagnostics masquerade as state variable may seem exotic, but it provides vital functionality that would be difficult to implement in any other way. It underpins, among others, ERSEM's implicit representation the vertical structure of the sediment. For example, ERSEM represent a class of organic matter in the sediment with just two state variables: the depth-integrated density (mmol C $m^{-2}$, mmol N $m^{-2}$, etc) and a mean penetration depth (m). The concentration of organic matter is assumed to be exponentially distributed in depth (the highest concentration at the sediment surface, asymptotically approaching 0 at depth); this allows us to diagnose the quantity of organic matter over a given depth interval from the density and penetration depth. Each type of benthic fauna has a habitat in the sediment defined by depth range; this defines among other the amount of particulate organic matter available as food. Thus, its food is a diagnostic calculated from the organic matter integrated over the sediment column, the penetration depth, and depth range that the fauna inhabits. From the perspective of the fauna, the food is a state variable. Accordingly, organic matter calculated over a specified depth interval is registered with `act_as_state_variable=.true.` The change in food calculated by benthic fauna is not discarded but used twice: the (negative) source term is directly applied to the state variable for column-integrated organic matter, and additionally converted into a rate of change of penetration depth.

# 5. New functionality for coupling 2D and 3D

The coupling functionality in FABM was foremost designed to enable *local* interaction between multiple model instances. Here, each instance is active in the same domain: in the pelagic, at the water surface, and/or at the bottom. However, numerous applications in NECCTON and beyond would benefit from the ability to couple depth-integrated variables (2D) with depth-explicit [pelagic] ones (3D). Notably, several higher trophic level (HTL) models featured in NECCTON, including ECOSMO E2E (Daewel et al., 2019), SEAPODYM-LMTL (Lehodey et al., 2010, 2015) and a spatially explicit implementation of the Community Size Spectrum Model (Cheung et al., 2018), represent HTL biomass by depth-integrated quantities that interact with an arbitrary part of the water column, for example, to obtain prey and dissolved oxygen, or to inject waste products such as $CO_2$. To underpin this interaction, the predator has a vertical "distribution", "habitat" or "search range", specified by weights that indicate what proportional of time the HTL spends in each layer. Such logic previously was difficult to implement in FABM: while possible (https://doi.org/10.5281/zenodo.4593394), it required use of poorly documented APIs (e.g., "do_column") and, due to the need for explicit and complex 2D-3D mapping, it led to hard-to-read code.

In the "neccton" branch of FABM, new functionality has been added to facilitate the mapping between the 3D pelagic environment (e.g., temperature, net primary production, prey) and 2D higher trophic levels, while respecting the vertical distribution of the predator and guaranteeing mass conservation. This is specifically designed to facilitate the coupling between lower and higher trophic level models – a central pillar in NECCTON.

## Base type

The primary way to use this new functionality is to have your predator module inherit from the new **`type_depth_integrated_particle`** type introduced in new Fortran module "fabm_builtin_depth_mapping":

```
type, extends(type_depth_integrated_particle) :: type_depth_integrated_predator
   …
end type
```

The `type_depth_integrated_particle` type adds several subroutines that give access to depth integrals and depth averages of depth-explicit dependencies. These routines can be used, for example, to access temperature or prey averaged over a predator's vertical habitat. The depth integration logic needs to be provided with weights for every layer, which define the depth distribution of your predator: to have the predator inhabit only the top 100 m of the water column, you would set these weights to 1 in cells at depths ≤ 100 m, and to 0 in all other ones. The weights are allowed to vary in time and space, and thus can account for behaviours such as diurnal or seasonal vertical migration. The weights are not restricted to 0 and 1: they can take any real value. Moreover, within a water column, only the *relative* value of the weights matters (i.e., scaling the weights with an arbitrary depth-independent constant has no impact on results). Thus, to have the predator distribution track prey availability, the weights can be set equal to the prey concentration. In that case, the fraction of time spent in each layer becomes proportional to prey concentration (formally, this fraction equals $w_i h_i / \sum_k w_k h_k$, with $w_i$ being the weight in layer $i$, and $h_i$ the layer's thickness).

In many cases, you will want to create your own simple child model to calculate these weights as a function of environmental variables, depth and/or time. That allows you to prescribe custom recipes, e.g., "make the predator distribution proportional to pelagic prey but avoid areas with oxygen ≤ 5 mmol $m^{-3}$".

## Dependencies on depth-averaged variables

To add a dependency on depth-averaged temperature to your predator, you add a surface (i.e., depth-independent) dependency to your type as usual:

```
type (type_surface_dependency_id) :: id_temp
```

This is registered in your "initialize" routine as usual:

```
call self%register_dependency(self%id_temp, 'temp', 'degrees_Celsius', 'depth-averaged
temperature')
```

The one new step is to couple the new dependency to the actual depth-average of temperature, which is done with a single call to **`request_mapped_coupling`**:

```
call self%request_mapped_coupling(self%id_temp, standard_variables%temperature,
average=.true.)
```

This sets up a child model (see also "Child models and mapping across domains") that performs the depth-averaging, as shown in Fig 6. `request_mapped_coupling` will ensure that this child model will respect the vertical weights that specify the predator habitat. The argument **`average=.true.`** ensures that `id_temp` will receive the depth *average* $\sum_k w_k c_k h_k / \sum_k w_k h_k$, with $w_k$, $c_k$ and $h_k$ representing local distribution weights, temperature, and layer thickness, respectively. If the average argument is omitted or set to `.false.`, `id_temp` receives the depth integral $\sum_k w_k c_k h_k$ instead.

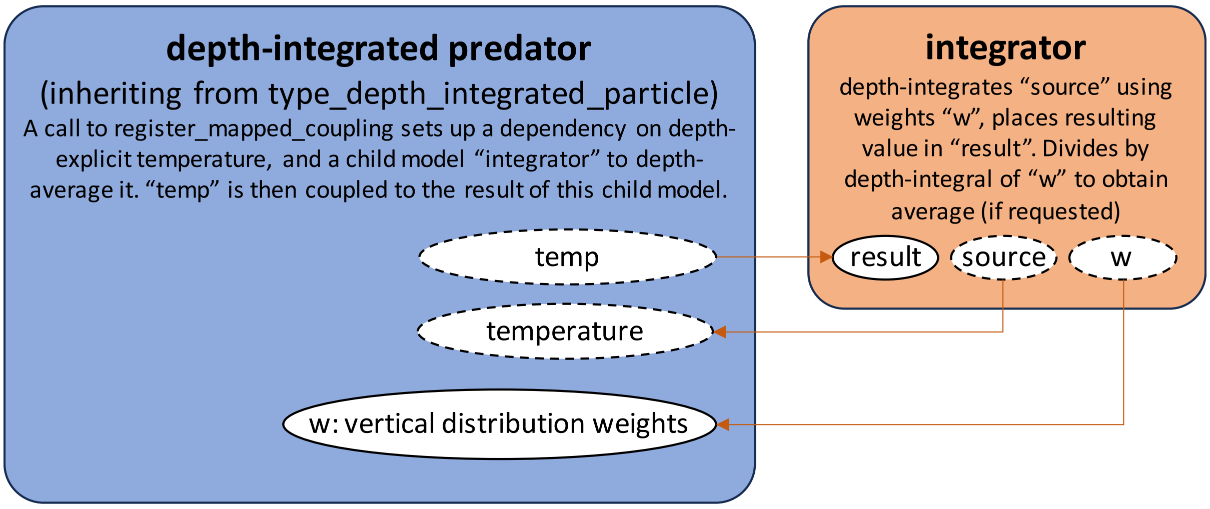

depth-integrated predator
(inheriting from type_depth_integrated_particle)
A call to register_mapped_coupling sets up a dependency on depth-explicit temperature, and a child model "integrator" to depth-average it. "temp" is then coupled to the result of this child model.
temp
temperature
w: vertical distribution weights
integrator
depth-integrates "source" using weights "w", places resulting value in "result". Divides by depth-integral of "w" to obtain average (if requested)
result
source
w

**Figure 6**. A depth-integrated predator type with a dependency on depth-averaged temperature. All ellipses indicate variables; dependencies are indicated by ellipses with dashed border. The depth-integrated predator would have its own entry in fabm.yaml; conversely, the child model "integrator" is automatically created by the call to `register_mapped_coupling`.

The very similar **`request_mapped_coupling_to_model`** provides access to the depth average or depth integral of a variable from another model instance. For example, the following links `id_prey_c` to the depth-averaged total carbon from a coupled model instance named "prey".

```
call self%request_mapped_coupling_to_model(self%id_prey_c, 'prey',
standard_variables%total_carbon, average=.true.)
```

This “prey” instance must be coupled to an actual model instance in the “coupling” section of the predator in fabm.yaml. An example where the predator is coupled to a fixed-stoichiometry zooplankton instance “Z4” is shown in Fig 7.

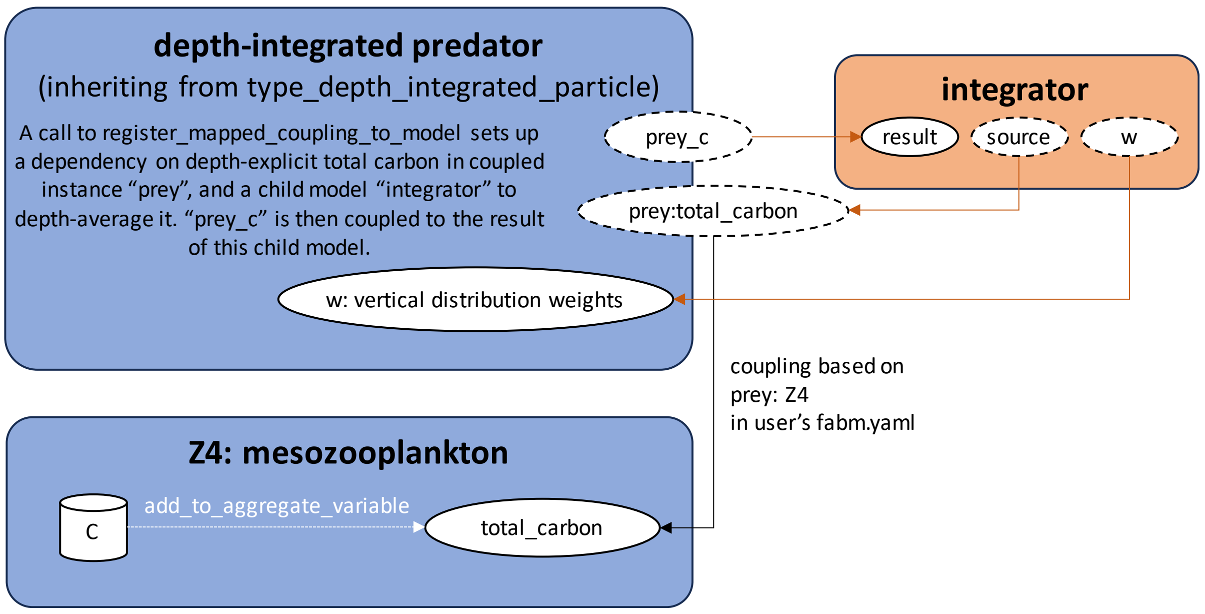

depth-integrated predator
(inheriting from type_depth_integrated_particle)
A call to register_mapped_coupling_to_model sets up a dependency on depth-explicit total carbon in coupled instance “prey”, and a child model “integrator” to depth-average it. “prey_c” is then coupled to the result of this child model.
prey_c
prey:total_carbon
w: vertical distribution weights
integrator
result
source
w
coupling based on
prey: Z4
in user’s fabm.yaml
Z4: mesozooplankton
C
add_to_aggregate_variable
total_carbon

**Figure 7.** A depth-integrated predator type with a dependency on depth-averaged prey carbon. All ellipses indicate variables; dependencies are indicated by ellipses with dashed border. Both the depth-integrated predator and the mesozooplankton would have its own entry in fabm.yaml; conversely, the child model “integrator” is automatically created by the call to `register_mapped_coupling_to_model`.

More information about coupling to model instances and the use of standard variables such as “total_carbon” can be found in section “Particles: grouped couplings and generalized access”.

## Distributing depth-integrated source terms over the pelagic

The above examples set up read-only dependencies: they allow you to easily obtain the value of the depth average or integral of a depth-explicit variable. What if you also want to return source terms for this variable? In that case, the syntax is the same, except that the receiving variable should be a state variable dependency (access: read, increment sources), rather than a plain dependency (read access only). For example, to link to a pelagic waste pool to which you want to direct a depth-integrated carbon flux, the dependency would be declared like:

```
type (type_surface_state_variable_id) :: id_waste_c
```

It would be registered in your `initialize` subroutine as usual:

```
call self%register_state_dependency(self%id_waste_c, 'waste_c', 'mmol C m-2', 'depth-
integrated carbon waste')
```

Finally, it would be coupled to the depth-integrated total carbon of a “waste” instance with

```
call self%request_mapped_coupling_to_model(self%id_waste_c, 'waste',
standard_variables%total_carbon)
```

In this case, we do not specify argument `average=.true.`, since we will provide a depth-*integrated* waste source. Accordingly, the dependency has units mmol C m$^{-2}$.

The above syntax is all nearly identical to the treatment of depth-averaged temperature and prey described earlier. However, the fact that `id_waste_c` is a *state variable* changes the behaviour of the depth-integrating child model, as shown in Fig 8. It now sets up a child instance that collects source terms for the depth-integrated carbon waste and redistributes them over the original depth-explicit waste; all while respecting the predator’s depth distribution.

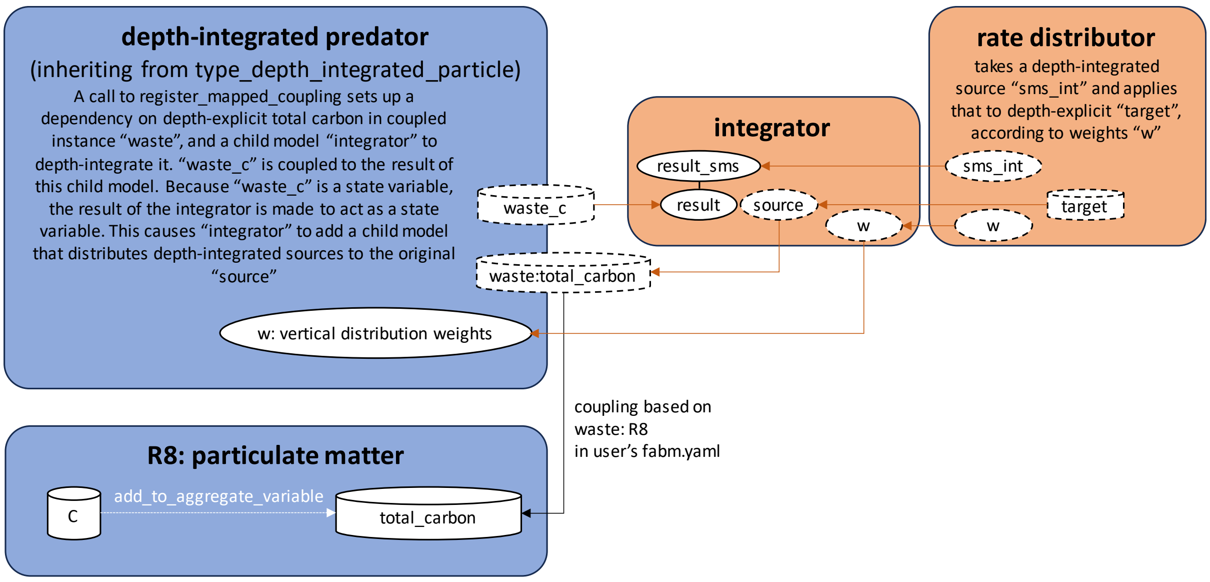

depth-integrated predator
(inheriting from type_depth_integrated_particle)
A call to register_mapped_coupling sets up a dependency on depth-explicit total carbon in coupled instance “waste”, and a child model “integrator” to depth-integrate it. “waste_c” is coupled to the result of this child model. Because “waste_c” is a state variable, the result of the integrator is made to act as a state variable. This causes “integrator” to add a child model that distributes depth-integrated sources to the original “source”
waste_c
waste:total_carbon
w: vertical distribution weights
integrator
result_sms
result
source
w
rate distributor
takes a depth-integrated source “sms_int” and applies that to depth-explicit “target”, according to weights “w”
sms_int
target
w
coupling based on waste: R8 in user’s fabm.yaml
R8: particulate matter
C
add_to_aggregate_variable
total_carbon

**Figure 8**. A depth-integrated predator type with a dependency on depth-integrated carbon in a pelagic waste pool. Unlike previous examples (Figs 6, 7), this *state variable* dependency allows the predator to feed back a depth-integrated source term, which then gets redistributed over the original depth-explicit waste pool. All ellipses indicate variables; dependencies are indicated by ellipses with dashed border. Both the depth-integrated predator and the particulate matter would have its own entry in fabm.yaml; the child models “integrator” and “rate_distributor” are automatically created by the call to `register_mapped_coupling_to_model`.

The default rule for distributing a depth-integrated fluxes over a pelagic state variable is to direct fraction $w_i h_i / \sum_k w_k h_k$ of the depth-integrated flux $f$ to layer $i$. Here, $w$ represents the weights that define the predator’s vertical distribution and $h$ the layer thickness. Expression, $\sum_k w_k h_k$ is the depth-integral of weights for the local water column. The local change in the pelagic state variable is the depth-integrated flux divided by layer thickness, i.e., $f w_i / \sum_k w_k h_k$. Note that this ensures that the flux is zero where the predator is absent (i.e., where $w_i = 0$). Additionally, only the *relative* value of weights $w$ matter for the redistribution rule: scaling the weights with an arbitrary depth-independent constant does not change the final redistributed source term.

The rate distributor module supports one alternative redistribution rule, under which the flux directed into each layer is proportional to the local concentration of the pelagic state variable (as well as being proportional to predator weights $w$). Such proportionality between the local flux and the local concentration is appropriate for certain loss terms, notably the loss of prey due to predation. The local redirected flux is now proportional to $w_i c_i$, with $c_i$ denoting the pelagic concentration. To ensure mass conservation (the depth integral of redirected flux must equal the originally specified depth-integrated flux), it follows that the local change must equal $f w_i c_i / \sum_k w_k c_i h_k$. To use this alternative redistribution rule, `register_mapped_coupling_to_model` must be called with argument **`proportional_change=.true.`**

## Applying loss terms to pelagic prey

The final new component is the ability to access (loop over) all depth-integrated state variables of a coupled pelagic instance, and to provide depth-integrated rates of change for each. This is typically used to apply the same relative loss rate to all constituents of a prey (e.g., carbon, nitrogen, phosphorus, chlorophyll, etc.), without having to know the prey's implementation details. It allows the same prey treatment as described near the end of section "Particles: grouped couplings and generalized access". To use this functionality, you first add a model dependency to the model type:

```
type (type_model_id) :: id_prey_int
```

This is registered from `initialize` by calling the new **`register_mapped_model_dependency`**:

```
call self%register_mapped_model_dependency(self%id_prey_int, 'prey',
proportional_change=.true., domain=domain_surface)
```

Here, the use of `proportional_change=.true.` specifies that when redistributing prey loss terms over the vertical, the local change in prey must be proportional to the local prey concentration (see the discussion of the alternative distribution rule above). By specifying `domain=domain_surface`, we ensure that all depth-integrated prey variables become available as `id_prey_int%surface_state`. These variables can then be read and changed by specifying source terms from `do_surface`:

```
do istate = 1, size(self%id_prey_int%surface_state)
   _GET_SURFACE_(self%id_prey_int%surface_state(istate), p)
   _ADD_SURFACE_SOURCE_(self%id_prey_int%surface_state(istate), -prey_loss_rate * p)
end do
```

Here, `prey_loss_rate` is the specific loss rate of the depth-integrated prey, in s$^{-1}$. In the background, `registered_mapped_model_dependency` takes care of setting up depth-integrated variables for each of the prey's original pelagic state variables, along with the appropriate integrator child models and rate distributors, as shown in Fig. 9. It is worth emphasizing, however, that all new functionality can be used without being aware of the underlying implementation.

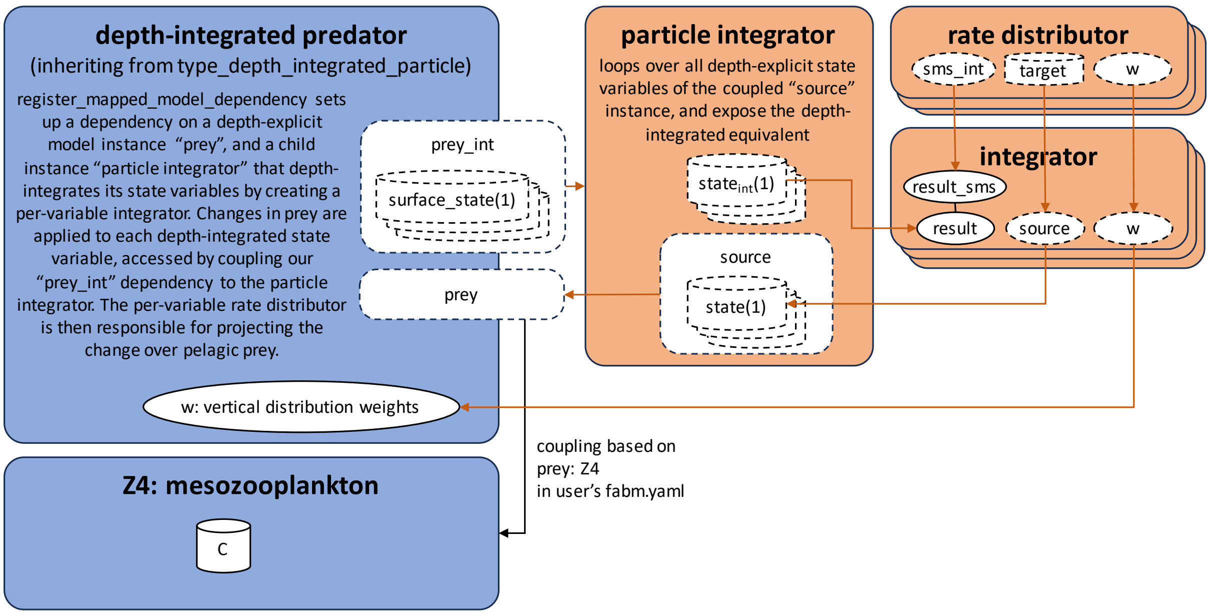

depth-integrated predator
(inheriting from type_depth_integrated_particle)
register_mapped_model_dependency sets up a dependency on a depth-explicit model instance "prey", and a child instance "particle integrator" that depth-integrates its state variables by creating a per-variable integrator. Changes in prey are applied to each depth-integrated state variable, accessed by coupling our "prey_int" dependency to the particle integrator. The per-variable rate distributor is then responsible for projecting the change over pelagic prey.
prey_int
surface_state(1)
prey
w: vertical distribution weights
particle integrator
loops over all depth-explicit state variables of the coupled "source" instance, and expose the depth-integrated equivalent
state_int(1)
source
state(1)
rate distributor
sms_int
target
w
integrator
result_sms
result
source
w
coupling based on
prey: Z4
in user's fabm.yaml
Z4: mesozooplankton
C

**Figure 9**. A depth-integrated predator type with a dependency on a pelagic prey instance. By calling `register_mapped_model_dependency`, the predator gets access to all depth-integrated state variables of the prey, irrespective of their identity. It is then free to prescribe rates of change for each, e.g., to apply the same specific loss rate. These rates get redistributed over the original depth-explicit state variables of the prey.  All ellipses indicate variables; dependencies are indicated by ellipses with dashed border. Both the depth-integrated predator and the mesozooplankton would have its own entry in fabm.yaml; the child models "particle_integrator" and all necessary instances of "integrator" and "rate_distributor" are automatically created by the call to `register_mapped_model_dependency`.

A comprehensive example that uses combines all functionality described above is included in the "neccton" branch of the FABM repository. Its current version is repeated verbatim here:

```
#include "fabm_driver.h"

module templates_depth_integrated_predator

   ! This module describes a predator that feeds across (part of) the water column.
   ! It is represented by a depth-integrated biomass and a prescribed vertical
distribution.
   ! This would be appropriate for predators that move very fast in the vertical.
   !
   ! The predator has constant C:N:P stoichiometry
   ! Its elemental ratios are defined by parameters NC and PC below.
   ! It accepts one prey type that may have variable C:N:P stoichiometry.
   ! At each point in time, ingested fluxes of C, N and P are calculated.
   ! Based on the most limiting of these, a growth rate is calculated.
   ! Unused ingested fluxes and dead biomass are sent to a coupled waste pool.
   !
   ! Extensions:
   ! * Different vertical distributions, including time-varying ones:
   !   create a new depth distribution type (your equivalent of type_vertical_depth_range
   !   defined in the fabm_builtin_depth_mapping module) and calculate your custom
   !   vertical distribution weights in its "do" routine. These can depend on any
   !   environmental input or parameter, e.g. temperature, light, time of day,
   !   depth, prey availability.
   ! * Multiple prey types: declare id_prey_c and the like as allocatable arrays,
   !   get number of prey types as parameter, allocate identifier arrays (id_prey_c etc.)
   !   and move all prey handling to the inside of a loop over all prey
   ! * Additional prey constituents (e.g., silicon, calcium carbonate): as for
   !   current treatment of N and P. If not incorporated in predator biomass,
   !   their ingested fluxes can be sent directly to the waste pool.
   ! * More complex dynamics for predation, predator population growth, etc.:
   !   modify logic in do_surface accordingly. Additional parameters should be added
   !   to the type_depth_integrated_predator type and their value retrieved from
   !   initialize. Additional environental inputs can be handled just like
   !   temperature. Any additional waste pools (e.g., to distinguish dissolved
   !   and particulate wastes) can be implemented analogous to the current waste.

   use fabm_types
   use fabm_particle
   use fabm_builtin_depth_mapping

   implicit none

   private

   type, extends(type_depth_integrated_particle), public :: type_depth_integrated_predator
      type (type_surface_state_variable_id)      :: id_c
      type (type_surface_dependency_id)          :: id_prey_c, id_prey_n, id_prey_p
      type (type_surface_state_variable_id)      :: id_waste_c, id_waste_n, id_waste_p
      type (type_surface_dependency_id)          :: id_temp
      type (type_surface_diagnostic_variable_id) :: id_net_growth, id_prey_loss_rate
      type (type_model_id)                       :: id_prey_int

      real(rk) :: clearance_rate
      real(rk) :: mortality
   contains
      procedure :: initialize
      procedure :: do_surface
   end type

   ! Redfieldian N:C and P:C ratios of predator biomass
   real(rk), parameter :: NC = 16.0_rk/ 106.0_rk
   real(rk), parameter :: PC = 1.0_rk / 106.0_rk

contains

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

      class (type_vertical_depth_range), pointer :: depth_distribution

      allocate(depth_distribution)
      call self%set_vertical_distribution(depth_distribution)

      ! Predator biomass and its contribution to different elemental pools
      call self%register_state_variable(self%id_c, 'c', 'mmol C m-2', 'density')
      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=NC)
      call self%add_to_aggregate_variable(standard_variables%total_phosphorus, self%id_c,
scale_factor=PC)

      ! Parameters
      ! NB rates are in d-1 in fabm.yaml and scaled here to s-1 using the scale_factor
argument
      call self%get_parameter(self%clearance_rate, 'clearance_rate', 'm3 d-1 mmol-1',
'clearance rate', scale_factor=1.0_rk / 86400.0_rk)
      call self%get_parameter(self%mortality, 'mortality', 'd-1', 'mortality',
scale_factor=1.0_rk / 86400.0_rk)

      ! Diagnostics
      call self%register_diagnostic_variable(self%id_net_growth, 'net_growth', 'mmol C m-2
d-1', 'net population growth rate')
      call self%register_diagnostic_variable(self%id_prey_loss_rate, 'prey_loss_rate', 'd-
1', 'specific prey loss rate')

      ! Depth-averaged dependencies
      call self%register_dependency(self%id_temp, 'temp', 'degrees_Celsius', 'depth-
averaged temperature')
      call self%register_dependency(self%id_prey_c, 'prey_c', 'mmol C m-3', 'depth-averaged
prey carbon')
      call self%register_dependency(self%id_prey_n, 'prey_n', 'mmol N m-3', 'depth-averaged
prey nitrogen')
      call self%register_dependency(self%id_prey_p, 'prey_p', 'mmol P m-3', 'depth-averaged
prey phosphorus')
      call self%register_state_dependency(self%id_waste_c, 'waste_c', 'mmol C m-2', 'depth-
integrated carbon waste')
      call self%register_state_dependency(self%id_waste_n, 'waste_n', 'mmol N m-2', 'depth-
integrated nitrogen waste')
      call self%register_state_dependency(self%id_waste_p, 'waste_p', 'mmol P m-2', 'depth-
integrated phosphorus waste')

      ! Derive depth-averaged dependencies from depth-explicit sources
      ! * Environmental variables are typically depth-averaged over the predator habitat,
      !   as temperature is below (note average=.true.)
      ! * Prey concentrations are here depth-averaged as well; they will be multiplied with
a clearance rate
      !   (volume searched per unit time per predator) to obtain ingestion rates.
      !   Prey destruction in handled separately by coupling to all prey state variables at
once
      !   (see the call to register_mapped_model_dependency below)
      ! * Waste pools are depth-integrated as they will receive depth-integrated fluxes of
waste
      !   produced by the predator population. These fluxes will be vertically distributed
      !   accordingly to the predator's vertical distribution, i.e., the waste flux
injected
      !   locally will be proportional to the local weight of the predators's
      !   vertical distribution.
      call self%request_mapped_coupling(self%id_temp, standard_variables%temperature,
average=.true.)
      call self%request_mapped_coupling_to_model(self%id_prey_c, 'prey',
standard_variables%total_carbon, average=.true.)
      call self%request_mapped_coupling_to_model(self%id_prey_n, 'prey',
standard_variables%total_nitrogen, average=.true.)
      call self%request_mapped_coupling_to_model(self%id_prey_p, 'prey',
standard_variables%total_phosphorus, average=.true.)
      call self%request_mapped_coupling_to_model(self%id_waste_c, 'waste',
standard_variables%total_carbon)
      call self%request_mapped_coupling_to_model(self%id_waste_n, 'waste',
standard_variables%total_nitrogen)
      call self%request_mapped_coupling_to_model(self%id_waste_p, 'waste',
standard_variables%total_phosphorus)

      ! Access depth-integrated prey state that we will apply specific loss rates to.
      ! The local (depth-explicit) prey loss is expected to be proportional to prey
biomass.
      ! This is specified by proportional_change=.true. The result of this is that the
      ! same specific prey loss rate (multiplied by distribution weights) will be applied
over
      ! the predator depth range.
      call self%register_mapped_model_dependency(self%id_prey_int, 'prey',
proportional_change=.true., domain=domain_surface)
   end subroutine

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

      real(rk) :: c, temp, prey_c, prey_n, prey_p, prey_s, w_int
      real(rk) :: ingestion_c, ingestion_n, ingestion_p, prey_loss_rate, p, net_growth
      integer  :: istate

      _SURFACE_LOOP_BEGIN_
         ! Get depth-integrated predator biomass
         _GET_SURFACE_(self%id_c, c)

         ! Depth-averaged environmental dependencies and prey concentrations
         _GET_SURFACE_(self%id_temp, temp)
         _GET_SURFACE_(self%id_prey_c, prey_c)
         _GET_SURFACE_(self%id_prey_n, prey_n)
         _GET_SURFACE_(self%id_prey_p, prey_p)

         ! Calculate ingested fluxes of different chemical elements
         ! Predator population growth will be based on the most limiting of these
         ingestion_c = self%clearance_rate * c * prey_c
         ingestion_n = self%clearance_rate * c * prey_n
         ingestion_p = self%clearance_rate * c * prey_p
         net_growth = min(ingestion_c, ingestion_n / NC, ingestion_p / PC) - self%mortality
* c

         ! The specific loss rate of prey is the depth-integrated ingestion,
         ! divided by depth-integrated prey biomass, e.g., ingestion_c / prey_c_int.
         ! In turn, prey_c_int is related to depth-averaged prey as prey_c = prey_c_int /
w_int,
         ! with w_int representing the depth-integral weights of the predator's vertical
distribution.
         ! Thus, the specific loss rate is ingestion_c / (prey_c * w_int), which simplifies
to
         ! clearance_rate * c / w_int (see expression for ingestion_c above)
         _GET_SURFACE_(self%id_w_int, w_int)
         prey_loss_rate = self%clearance_rate * c / w_int

         ! Source term for predator
         _ADD_SURFACE_SOURCE_(self%id_c, net_growth)

         ! Apply the same specific loss rate of all state variables of the prey
         do istate = 1, size(self%id_prey_int%surface_state)
            _GET_SURFACE_(self%id_prey_int%surface_state(istate), p)
            _ADD_SURFACE_SOURCE_(self%id_prey_int%surface_state(istate), -prey_loss_rate *
p)
         end do

         ! Send unused ingested matter and dead biomass to waste pools
         _ADD_SURFACE_SOURCE_(self%id_waste_c, ingestion_c - net_growth)
         _ADD_SURFACE_SOURCE_(self%id_waste_n, ingestion_n - net_growth * NC)
         _ADD_SURFACE_SOURCE_(self%id_waste_p, ingestion_p - net_growth * PC)
         ! Save diagnostics
         _SET_SURFACE_DIAGNOSTIC_(self%id_net_growth, net_growth * 86400.0_rk)
         _SET_SURFACE_DIAGNOSTIC_(self%id_prey_loss_rate, prey_loss_rate * 86400.0_rk)

      _SURFACE_LOOP_END_
   end subroutine

end module
```

# 6. Recommendations

## In general

### Modularity

The *minimum* level of modularity that you need follows from the components you want to exchange with others, now or in the future. To support developments planned in NECCTON, modules for oxygen, carbonate chemistry, irradiance, suspended particulate matter and higher trophic levels are best written as stand-alone module, as this will allow them to be used in the different CMEMS Monitoring and Forecasting Centres.

The *optimum* level of modularity depends on the intended uses of the model. In practice, many have found it useful to use one module instance per “integral physical particle”, e.g., a chemical compound (e.g., dissolved nitrate, oxygen), generalized compound (e.g., a class of particulate or dissolved organic matter) or organism (e.g., a plankton functional type). Each instance has a separate entry in fabm.yaml. In this scenario, the model code would typically include modules (Fortran types, each coded in a separate source file) for a generalized passive tracer, generalized phytoplankton, and generalized zooplankton. In fabm.yaml, the tracer module is used for nutrients, dissolved organic matter classes and particulate organic matter classes, the phytoplankton module is used for every instantiated phytoplankton type, and the zooplankton module is used for every instantiated zooplankton type.

Particle-based modularisation has several benefits:

1. the same code can be used to construct different ecosystem configurations (by adding or removing plankton functional types in fabm.yaml);
2. consolidated codes for generic phytoplankton and generic zooplankton are written once (no separate copies for pico-, nano-, and microphytoplankton, diatoms, califyers, etc.), which reduces code size, guarantees consistency among plankton functional types, and reduces the risk for bugs;

mass conservation can be diagnosed per module, which makes conservation checking much easier (see “Conservation checks”).

### Registering variables

- Each module can register any mixture of state variables, diagnostic variables, dependencies and state dependencies that it needs (see also "Variables").
- To simplify implicit coupling (see "Coupling specification"), link your state variables and diagnostic variables to a predefined standard variable identity where appropriate. This is done by supplying the `standard_variable` argument during registration, e.g., standard_variable=standard_variables%alkalinity_expressed_as_mole_equivalent when you register alkalinity. Note that if you do this, you promise that units of your variable and the predefined standard variable are identical.
- Register the contributions of your state and diagnostic variables to aggregate variables (see "Aggregate variables"). This is *essential* for conserved quantities such as total carbon, nitrogen, phosphorus, silicon, iron, since aggregate variable are use both for model coupling (see "Particles: grouped couplings and generalized access"). However, it is also useful for additional aggregate quantities such as total chlorophyll, which is widely used by irradiance models to compute attenuation, and net primary production, which is used by higher trophic level models (e.g., SEAPODYM-LMTL).
- If you see a need for additional standard variable identities, please contact the authors or post a message on the FABM discussion forum.

### Coupling to other modules

- Where possible, couple based on predefined standard variable identities. These can be used to access fields from the hosting hydrodynamic model (e.g., temperature), but also fields from other modules, e.g., alkalinity. Note that standard variables have specific defined units.
- When coupling to other "particles", e.g., nutrients, a class of particulate organic matter, another functional type (e.g., prey), use the standard variable-based coupling (see "Particles: grouped couplings and generalized access"). For example, register a dependence on a "pom" instance (couplable in fabm.yaml) and then use the request_coupling_to_model API to couple (in code) to individual constituents such as total carbon, nitrogen, etc. Note that the resulting variables have specific defined units.

### Providing source terms and diagnostics

Use the standard routines where possible: "do" for the pelagic, "do_bottom" for the bottom, "do_surface" for the water surface (see also "Routines"). These are designed to operate efficiently, and easy to implement as they always operate locally (given the local environment and state, calculate local sources and diagnostics). Only use "do_column" if there is no other way to achieve what you need, as it is relatively inefficient. If you do use it, consider doing so only for the minimum of calculations possible, by moving these into a stand-alone child module (see "Child models and mapping across domains").

### Vertical movement of pelagic state variables

To apply constant vertical velocities (e.g., sinking), register your state variable with the vertical_movement argument. To use time- and/or space varying velocities, implement the "get_vertical_movement" routine. This is also needed to add vertical velocities (constant or variable) to a state variable dependency, e.g., to add migration behaviour. Remember to apply the same vertical velocity to all state variables.

## Specific component models

### Phytoplankton

- Register your contribution to total chlorophyll, even if you do not normally track it, and would only assume it to be proportional to biomass. That is because chlorophyll is a crucial quantity in many models you may be coupled to (e.g., irradiance)
- Register you contribution to net primary production to support selected higher trophic level models (e.g., SEAPODYM)

### Predators (e.g., zooplankton)

- Use particle conventions to obtain prey biomass (read-only) and destroy your prey by applying the same specific loss rate to all state variables (see “Particles: grouped couplings and generalized access”)
- Consider generalizing the zooplankton type of accept any number of prey, configurable and couplable in fabm.yaml, instead of using a fixed set of prey types with hardcoded names. Together with the particle convention (previous item), this would enable your zooplankton implementation to work within other biogeochemical/lower trophic level models.

### Carbonate system

- If using parametrized alkalinity, register it with act_as_state_variable=.true. (see “Your diagnostic, my state variable”)
- If you are using not-trivial logic to calculate alkalinity, consider placing it in a stand-alone alkalinity module, separate from the rest of the carbonate system. That module can then be re-used with other carbonate system solvers.

### Irradiance

- For more complex models (e.g., spectrally resolving), consider separating the atmospheric component (if any), the underwater radiative transfer model, and the calculation of (broadband/wavelength-explicit) absorption/scattering/attenuation properties summed across Inherent Optical Properties (IOPs). This allows the first two components to work unmodified with new parametrisations (e.g., new absorption and scattering spectra) and new IOPs.

### Higher trophic levels (2D)

This section discusses organisms that move fast in the vertical, to the extent that that they can be represented by depth-integrated biomass and a prescribed vertical distribution. This distribution may be variable, for example, dependent on time, depth, environmental variables such as temperature, or prey availability.

- If you are implementing such a model in FABM anew, we recommend you review the new functionality introduced for this purpose (see “New functionality for coupling 2D and 3D”). It provides a flexible method for mapping 2D predators over the pelagic, and ensures their interactions with pelagic variables are mass conserving.

- The recommendations for generic predators ("Predators (e.g., zooplankton)") apply: by permitting the number and names of prey to be defined at runtime in fabm.yaml, and by accessing prey constituents through standard names ("total_carbon", etc.), your HTL model will be compatible with a wide range of biogeochemical/lower trophic level models, and thus shareable across CMEMS Monitoring and Forecasting Centres.

# 7. Model testing

## Debugging

Best practices for debugging biogeochemical models in FABM are not unique to NECCTON and they are mostly documented on the FABM wiki. Nevertheless, it is worth listing the main steps worth taking:

1. Make FABM perform additional runtime checks by compiling it in Debug mode. This verifies, among others, whether your biogeochemical model actually provides values for the diagnostics it has registered, and whether the source terms that your model provides are finite (not NaN). To compile FABM in debug mode, run provide `-DCMAKE_BUILD_TYPE=Debug` when you run cmake.
2. Make the compiler perform additional runtime checks. This can catch additional issues, notably the scenario where a biogeochemical model reads or writes variables that it has not registered. Different compilers require different flags to activate runtime checks, for example, gfortran needs `-fcheck=all`, Intel fortran needs `-check all`. These flags can be set through environment variable `FFLAGS` before you run cmake, or by passing `-DCMAKE_Fortran_FLAGS=<FLAGS>` to cmake. Further information is available on the wiki.
3. Verify mass conservation per model instance, per chemical element, and per domain (interior, surface, bottom) by setting `check_conservation: true` in fabm.yaml. Further information is provided in the section "Conservation checks".

Each of these three options comes at the expense of performance. Activating them all at once can easily increase runtime by a factor 4. Therefore, it is good practice to activate them during the early stages of model development, but none of these options should be used for production/operation simulations.

## Light-weight testing options

Before running newly coded biogeochemical/ecosystem models or process descriptions in production setups (e.g., within computationally expensive 3D models configured for CMEMS domains), it is good practice to evaluate their behaviour in more lightweight setting, for example, in a 0D box model, a 1D water column model, or a coarse resolution 3D model. This is very feasible in FABM, as it has been interfaced to a wide range of physical models (Fig 1), including some specifically designed for low computational cost, short simulation times, and repeated runs. The following are often useful:

- pyfabm: the Python interface to FABM can be used to run simulations in box model setting (0D). Instructions on how to build and use pyfabm are available on the wiki. A specific example for running simulations from Python is included.
- GOTM (Burchard et al., 2006; Li et al., 2021): a 1D water column model that can perform multi-year simulations under realistic forcing in minutes. Setups for any location across the globe can be generated on the fly using https://igotm.bolding-bruggeman.com/. Instructions for using GOTM with FABM are available on the wiki. For GOTM-FABM simulations, dedicated packages are available for sensitivity analysis and calibration (parsac) and for data assimilation (EAT), which was developed in the framework of the European Horizon 2020 SEAMLESS project (grant agreement No 101004032).
- fabmos: the FABM Offline Simulator that is being developed within NECCTON WP3 supports efficient 3D simulation with prescribed [offline] transports. It has a flexible architecture into which new transport engines can be added. The EU project OceanICU has delivered an efficient global simulation engine based on the Transport Matrix Method (Khatiwala, 2007), which can perform efficient (~ 1 simulated year/minute) coarse resolution global simulations. Instructions on how to build fabmos are available on its wiki. Examples for specific biogeochemical models including ERSEM and PISCES are available from its repository.

## Production use

To embed your developments in simulations comparable to those run within CMEMS, you will want to use the corresponding hydrodynamic models: HYCOM (Bleck, 2002) for the Artic MFC, NEMO (Madec, 2008) for all other ones. Both of these have been coupled to FABM:

- For HYCOM, a FABM coupler is maintained and developed by the Nansen Environmental and Remote Sensing Center. This codebase is available on GitHub.
- For NEMO, FABM couplers for several NEMO versions are maintained and developed by the Plymouth Marine Laboratory. These are available from GitHub for NEMO 3.6 and NEMO 4.0. A FABM coupler for NEMO 4.2 is currently being developed by a consortium of institutes including NECCTON partners (e.g., BSH, PML).

For advice on model configurations and code versions, we recommend contacting NECCTON partners that actively contribute to the Monitoring and Forecasting Centre(s) that you are interested in.

It is worth noting that NEMO-FABM couplers are widely used, both within NECCTON and beyond. Internationally, at least 10 different organizations currently use NEMO with FABM included, that is, they use the modified NEMO source codes distributed by PML. These codes do not always track the central NEMO repository and its updates, and even if they did, that would come with some delay. This is suboptimal. For the many organizations that use both NEMO and FABM, and for NECCTON and CMEMS in particular, it would clearly be beneficial if the FABM coupling layer were included within the authoritative NEMO repository. This is an option we recommend the NEMO consortium to consider.

## 8. Code distribution

There are three ways to distribute new FABM-based biogeochemical/ecological models:

1. Integrate them directly in the FABM repository. This is suitable for compact model codes without external dependencies. It is the way that various ERGOM strains and the operational version of ECOSMO are currently distributed. To distribute your code like this: fork the FABM repository, commit your changes, and create a pull request. After this pull request is accepted, all FABM users will have access to your model(s).
2. Place your source code (the equivalent of your "institute directory" in FABM) in a stand-alone repository. This is the way the FABM-based implementations of ERSEM, BFM and PISCES are currently distributed. Users obtain your code by cloning your repository; they then integrate it in their FABM builds by providing additional arguments to cmake (e.g., `-DEXTRA_FABM_INSTITUTES=pisces -DFABM_PISCES_BASE=<piscesdir>`)
3. As the previous option, but additionally your repository is included in the fabm-plus repository as a submodule. This repository is the basis for public distributions of GOTM and fabmos, among others (see "Light-weight testing options"), which means that users of these distributions will automatically get access to your model(s). The fabm-plus repository currently includes ERSEM and PISCES, among others.

## 9. References

Aumont, O., Ethé, C., Tagliabue, A., Bopp, L., & Gehlen, M. (2015). PISCES-v2: An ocean biogeochemical model for carbon and ecosystem studies. *Geoscientific Model Development*, *8*(8), 2465–2513. https://doi.org/10.5194/gmd-8-2465-2015

Bleck, R. (2002). An oceanic general circulation model framed in hybrid isopycnic-Cartesian coordinates. *Ocean Modelling*, *4*(1), 55–88. https://doi.org/10.1016/S1463-5003(01)00012-9

Bruggeman, J., & Bolding, K. (2014). A general framework for aquatic biogeochemical models. *Environmental Modelling and Software*, *61*, 249–265. https://doi.org/10.1016/j.envsoft.2014.04.002

Bruggeman, J., Bolding, K., Nerger, L., Teruzzi, A., Spada, S., Skákala, J., & Ciavatta, S. (2023). EAT v0.9.6: a 1D testbed for physical-biogeochemical data assimilation in natural waters. *Geoscientific Model Development Discussions*, *under review*. https://doi.org/10.5194/gmd-2023-238

Bruggeman, J., Bolding, K., Nerger, L., Teruzzi, A., Spada, S., Wakamatsu, T., Yumruktepe, Ç., Skákala, J., & Ciavatta, S. (2023). *SEAMLESS Public release and full documentation of the SEAMLESS prototype. D2.4*. https://doi.org/10.5281/zenodo.10581313

Burchard, H., Bolding, K., Kühn, W., Meister, A., Neumann, T., & Umlauf, L. (2006). Description of a flexible and extendable physical-biogeochemical model system for the water column. *Journal of Marine Systems*, *61*(3–4), 180–211. https://doi.org/10.1016/j.jmarsys.2005.04.011

Butenschön, M., Clark, J., Aldridge, J. N., Allen, J. I., Artioli, Y., Blackford, J., Bruggeman, J., Cazenave, P., Ciavatta, S., Kay, S., Lessin, G., van Leeuwen, S., van der Molen, J., de Mora, L., Polimene, L., Sailley, S., Stephens, N., & Torres, R. (2016). ERSEM 15.06: a generic model for marine biogeochemistry and the ecosystem dynamics of the lower trophic levels. *Geoscientific Model Development*, *9*(4), 1293–1339. https://doi.org/10.5194/gmd-9-1293-2016

Cheung, W. W. L., Bruggeman, J., & Butenschön, M. (2018). Projected changes in global and national potential marine fisheries catch under climate change scenarios in the twenty-first century. In M. Barange, T. Bahri, M. C. M. Beveridge, K. L. Cochrane, S. Funge-Smith, & F. Poulain (Eds.), *Impacts of Climate Change on fisheries and aquaculture: Synthesis of current knowledge, adaptation and mitigation options. FAO Fisheries Technical Paper 627*. FAO. http://www.fao.org/3/I9705EN/i9705en.pdf

Daewel, U., Schrum, C., & Macdonald, J. I. (2019). Towards end-to-end (E2E) modelling in a consistent NPZD-F modelling framework (ECOSMO E2E_v1.0): application to the North Sea and Baltic Sea. *Geoscientific Model Development*, *12*(5), 1765–1789. https://doi.org/10.5194/gmd-12-1765-2019

Khatiwala, S. (2007). A computational framework for simulation of biogeochemical tracers in the ocean. *Global Biogeochemical Cycles*, *21*(3), n/a-n/a. https://doi.org/10.1029/2007GB002923

Lehodey, P., Conchon, A., Senina, I., Domokos, R., Calmettes, B., Jouanno, J., Hernandez, O., & Kloser, R. (2015). Optimization of a micronekton model with acoustic data. *ICES Journal of Marine Science*, *72*(5), 1399–1412. https://doi.org/10.1093/icesjms/fsu233

Lehodey, P., Murtugudde, R., & Senina, I. (2010). Bridging the gap from ocean models to population dynamics of large marine predators: A model of mid-trophic functional groups. *Progress in Oceanography*, *84*(1–2), 69–84. https://doi.org/10.1016/j.pocean.2009.09.008

Li, Q., Bruggeman, J., Burchard, H., Klingbeil, K., Umlauf, L., & Bolding, K. (2021). Integrating CVMix into GOTM (v6.0): a consistent framework for testing, comparing, and applying ocean mixing schemes. *Geoscientific Model Development*, *14*(7), 4261–4282. https://doi.org/10.5194/gmd-14-4261-2021

Madec, G. (2008). NEMO ocean engine. In *Note du Pole de modélisation* (Vol. 27). Institut Pierre-Simon Laplace (IPSL).

Vichi, M., Lovato, T., Butenschön, M., Tedesco, L., Lazzari, P., Cossarini, G., Masina, S., Pinardi, N., Solidoro, C., & Zavatarelli, M. (2020). *The Biogeochemical Flux Model (BFM): Equation Description and User Manual. BFM version 5.2.* http://bfm-community.eu/

Yumruktepe, V. Ç., Samuelsen, A., & Daewel, U. (2022). ECOSMO II(CHL): a marine biogeochemical model for the North Atlantic and the Arctic. *Geoscientific Model Development*, *15*(9), 3901–3921. https://doi.org/10.5194/gmd-15-3901-2022