MARBL Documentation

Release cesm2.1

MARBL developers

Mar 12, 2020

Contents

1 About This Guide **3**

2 Disclaimer **5**

3 Sponsorship **7**

4 Table of contents **9**

4.1 MARBL user guide 9
4.2 Examples of MARBL Implementation 31
4.3 MARBL developer's guide 57
4.4 MARBL scientific documentation 78
4.5 Rules of engagement 80

Bibliography **85**

The Marine Biogeochemistry Library, or MARBL, is a Fortran software package to be used by ocean general circulation models. It is available via github. Licensing details are provided in the top-level `LICENSE` file.

CHAPTER 1

About This Guide

This document has four major sections.

The *user's guide* is designed to support configuring and running MARBL.

We also provide *some examples* of how MARBL is implemented in GCMs.

The *developer's guide* provides techincal documentation of the MARBL code.

The *scientific guide* aims to communicate the scientific underpinnings of the formulations encoded in MARBL.

CHAPTER 2

Disclaimer

This version of MARBL has been made public solely for use in CESM 2.1. If you want to bring MARBL in to your GCM, please wait for the official MARBL 1.0.0 release.

CHAPTER 3

Sponsorship

MARBL is supported by the DOE Biological and Environmental Research office and the National Center for Atmospheric Research, which is funded by the National Science Foundation.

CHAPTER 4

Table of contents

4.1 MARBL user guide

4.1.1 How to Use MARBL in a GCM

Init

The init stage is where MARBL is configured, parameters are set, and memory is allocated. If the GCM wants to specify non-default parameter values, that needs to be done with `put_setting()` statements before calling `init()`. There are three different interfaces that can be used; all are equivalent, but different GCMs may find it easiest to call different interfaces.

1. Two arguments: a string containing the variable name, and then the variable value (in proper datatype)

```
call marbl_instance%put_setting('ciso_on', .true.)
```

2. One argument: a string containing a line from a MARBL settings file (see examples below) of the format `varname = value`

```
call marbl_instance%put_setting('ciso_on = .true.')
```

3. Three arguments: strings containing the variable name, the datatype, and the value

```
call marbl_instance%put_setting('ciso_on', 'logical', '.true.')
```

`put_setting()` calls that do not correspond to defined MARBL parameters will result in an error during `init()`. There is no check in the `put_setting()` call itself because allowable parameters may depend on other parameter values. For example, `autotrophs(3)%sname` is defined if `autotroph_cnt >= 3` but not if `autotroph_cnt < 3`.

The `init()` interface

```
subroutine init(this,                       &
     gcm_num_levels,                        &
     gcm_num_PAR_subcols,                   &
     gcm_num_elements_surface_flux,         &
     gcm_delta_z,                           &
     gcm_zw,                                &
     gcm_zt,                                &
     lgcm_has_global_ops)

  .
  .
  .

  class(marbl_interface_class), intent(inout) :: this
  integer(int_kind),            intent(in)    :: gcm_num_levels
  integer(int_kind),            intent(in)    :: gcm_num_PAR_subcols
  integer(int_kind),            intent(in)    :: gcm_num_elements_surface_flux
  real(r8),                     intent(in)    :: gcm_delta_z(gcm_num_levels) ! thickness of layer k
  real(r8),                     intent(in)    :: gcm_zw(gcm_num_levels) ! thickness of layer k
  real(r8),                     intent(in)    :: gcm_zt(gcm_num_levels) ! thickness of layer k
  logical,           optional,  intent(in)    :: lgcm_has_global_ops
```

Note the optional argument: `lgcm_has_global_ops` is a way for the GCM to inform MARBL that it can perform global operations such as finding global averages of values. There are some MARBL configurations that require this, and MARBL will abort unless the GCM verifies it can provide these values.

MARBL does not have an explicit interface to tell a GCM how many tracers are being computed. Instead, use `size(marbl_instance%tracer_metadata)` (called after `marbl_instance%init()`).

MARBL can compute surface fluxes across multiple columns simultaneously, with the number of columns supported set by `gcm_num_elements_surface_flux`. There is not a corresponding `gcm_num_elements_interior_tendency` yet, because currently MARBL computes tracer tendencies one column at a time.

Example from Stand-Alone MARBL

The stand-alone MARBL driver / test suite use settings files (`*.settings` in the `tests/input_files/settings/` directory) that are processed in the following manner:

```
settings_file_line = ''
do while(ioerr .eq. 0)
  ! (i) broadcast settings_file_line and call put_setting(); abort if error
  !     calling with empty settings_file_line on first entry to loop is okay, and
  !     this ensures we don't call put_setting with a garbage line if
  !     ioerr is non-zero
  call marbl_mpi_bcast(settings_file_line, 0)
  call marbl_instance%put_setting(settings_file_line)
  if (marbl_instance%StatusLog%labort_marbl) then
    call marbl_instance%StatusLog%log_error_trace("put_setting(settings_file_line)", subname)
    call marbl_io_print_marbl_log(marbl_instance%StatusLog)
```

(continues on next page)

(continued from previous page)

```
  end if

  ! (ii) master task reads next line in settings file
  if (my_task .eq. 0) read(97,"(A)", iostat=ioerr) settings_file_line

  ! (iii) broadcast settings file line to all tasks (along with iostat)
  call marbl_mpi_bcast(ioerr, 0)
end do

if (.not.is_iostat_end(ioerr)) then
  if (my_task .eq. 0) then
    write(*,"(A,I0)") "ioerr = ", ioerr
    write(*,"(2A)") "ERROR encountered when reading MARBL settings file ",␣
↪trim(settings_file)
  end if
  call marbl_mpi_abort()
end if
```

`init()` is then called from the individual test:

```
! Call marbl%init
call marbl_instance%init(gcm_num_levels = km,                &
                         gcm_num_PAR_subcols = 1,            &
                         gcm_num_elements_surface_flux = 1, &
                         gcm_delta_z = delta_z,              &
                         gcm_zw = zw,                        &
                         gcm_zt = zt)
if (marbl_instance%StatusLog%labort_marbl) then
  call marbl_instance%StatusLog%log_error_trace('marbl%init', subname)
  return
end if
```

Default Parameter Values

Below are the default parameter values (real variables provided to double precision). This specific page was been generated by running the `gen_settings_file` regression test with no input settings file. The test writes this output to `marbl_settings.gen`. Note that the order the variables are listed in comes from the order the variables are defined in MARBL, but the order of `put_setting()` calls does not matter.

```
PFT_defaults = 'CESM2'
ciso_on = F
lsource_sink = T
lecovars_full_depth_tavg = F
ciso_lsource_sink = T
lflux_gas_o2 = T
lflux_gas_co2 = T
lcompute_nhx_surface_emis = T
lvariable_PtoC = T
ladjust_bury_coeff = F
lo2_consumption_scalef = F
lp_remin_scalef = F
init_bury_coeff_opt = 'settings_file'
particulate_flux_ref_depth = 100
Jint_Ctot_thres_molpm2pyr =   0.10000000000000001E-08
```

(continues on next page)

(continued from previous page)

```
gQsi_0 =   0.13700000000000001E+00
gQsi_max =   0.82199999999999995E+00
gQsi_min =   0.45699999999999998E-01
gQ_Fe_kFe_thres =   0.10000000000000000E+02
gQ_Si_kSi_thres =   0.60000000000000000E+01
parm_Fe_bioavail =   0.10000000000000000E+01
parm_o2_min =   0.50000000000000000E+01
parm_o2_min_delta =   0.50000000000000000E+01
parm_kappa_nitrif_per_day =   0.59999999999999998E-01
parm_nitrif_par_lim =   0.10000000000000000E+01
parm_labile_ratio =   0.93999999999999995E+00
parm_init_POC_bury_coeff =   0.25400000000000000E+01
parm_init_POP_bury_coeff =   0.35999999999999999E+00
parm_init_bSi_bury_coeff =   0.15300000000000000E+01
parm_Fe_scavenge_rate0 =   0.22000000000000000E+02
parm_Lig_scavenge_rate0 =   0.14999999999999999E-01
parm_FeLig_scavenge_rate0 =   0.12000000000000000E+01
parm_Lig_degrade_rate0 =   0.93999999999999994E-04
parm_Fe_desorption_rate0 =   0.99999999999999995E-06
parm_f_prod_sp_CaCO3 =   0.70000000000000007E-01
parm_POC_diss =   0.10000000000000000E+05
parm_SiO2_diss =   0.65000000000000000E+05
parm_SiO2_gamma =   0.00000000000000000E+00
parm_hPOC_SiO2_ratio =   0.10000000000000000E-01
parm_CaCO3_diss =   0.50000000000000000E+05
parm_CaCO3_gamma =   0.20000000000000000E-01
parm_hPOC_CaCO3_ratio =   0.10000000000000000E-01
parm_hPOC_dust_ratio =   0.10000000000000000E-01
o2_sf_o2_range_hi =   0.45000000000000000E+02
o2_sf_o2_range_lo =   0.50000000000000000E+01
o2_sf_val_lo_o2 =   0.26000000000000001E+01
parm_sed_denitrif_coeff =   0.10000000000000000E+01
bury_coeff_rmean_timescale_years =   0.10000000000000000E+02
parm_scalelen_z(1) =   0.10000000000000000E+05
parm_scalelen_z(2) =   0.25000000000000000E+05
parm_scalelen_z(3) =   0.50000000000000000E+05
parm_scalelen_z(4) =   0.10000000000000000E+06
parm_scalelen_vals(1) =   0.10000000000000000E+01
parm_scalelen_vals(2) =   0.36000000000000001E+01
parm_scalelen_vals(3) =   0.47000000000000002E+01
parm_scalelen_vals(4) =   0.47999999999999998E+01
caco3_bury_thres_opt = 'omega_calc'
caco3_bury_thres_depth =   0.30000000000000000E+06
caco3_bury_thres_omega_calc =   0.89000000000000001E+00
PON_bury_coeff =   0.50000000000000000E+00
POM_bury_frac_max =   0.80000000000000004E+00
bSi_bury_frac_max =   0.10000000000000000E+01
ciso_fract_factors = 'Laws'
auto_mort2_exp =   0.17500000000000000E+01
zoo_mort2_exp =   0.15000000000000000E+01
QCaCO3_max =   0.40000000000000002E+00
f_graze_CaCO3_remin =   0.33000000000000002E+00
autotroph_cnt = 3
zooplankton_cnt = 1
max_grazer_prey_cnt = 3
autotroph_settings(1)%sname = 'sp'
autotroph_settings(1)%lname = 'Small Phyto'
```

(continues on next page)

(continued from previous page)

```
autotroph_settings(1)%temp_func_form_opt = 'q_10'
autotroph_settings(1)%Nfixer = F
autotroph_settings(1)%imp_calcifier = T
autotroph_settings(1)%exp_calcifier = F
autotroph_settings(1)%silicifier = F
autotroph_settings(1)%is_carbon_limited = F
autotroph_settings(1)%kFe =   0.30000000000000001E-04
autotroph_settings(1)%kPO4 =   0.10000000000000000E-01
autotroph_settings(1)%kDOP =   0.29999999999999999E+00
autotroph_settings(1)%kNO3 =   0.25000000000000000E+00
autotroph_settings(1)%kNH4 =   0.10000000000000000E-01
autotroph_settings(1)%kSiO3 =   0.00000000000000000E+00
autotroph_settings(1)%kCO2 =   0.00000000000000000E+00
autotroph_settings(1)%Qp_fixed =   0.85470085470085479E-02
autotroph_settings(1)%gQfe_0 =   0.30000000000000001E-04
autotroph_settings(1)%gQfe_min =   0.25000000000000002E-05
autotroph_settings(1)%alphaPI_per_day =   0.39000000000000001E+00
autotroph_settings(1)%PCref_per_day =   0.50000000000000000E+01
autotroph_settings(1)%thetaN_max =   0.25000000000000000E+01
autotroph_settings(1)%loss_thres =   0.10000000000000000E-01
autotroph_settings(1)%loss_thres2 =   0.00000000000000000E+00
autotroph_settings(1)%temp_thres =  -0.10000000000000000E+02
autotroph_settings(1)%mort_per_day =   0.10000000000000001E+00
autotroph_settings(1)%mort2_per_day =   0.10000000000000000E-01
autotroph_settings(1)%agg_rate_max =   0.50000000000000000E+00
autotroph_settings(1)%agg_rate_min =   0.10000000000000000E-01
autotroph_settings(1)%loss_poc =   0.00000000000000000E+00
autotroph_settings(1)%Ea =   0.32000000000000001E+00
autotroph_settings(2)%sname = 'diat'
autotroph_settings(2)%lname = 'Diatom'
autotroph_settings(2)%temp_func_form_opt = 'q_10'
autotroph_settings(2)%Nfixer = F
autotroph_settings(2)%imp_calcifier = F
autotroph_settings(2)%exp_calcifier = F
autotroph_settings(2)%silicifier = T
autotroph_settings(2)%is_carbon_limited = F
autotroph_settings(2)%kFe =   0.69999999999999994E-04
autotroph_settings(2)%kPO4 =   0.50000000000000003E-01
autotroph_settings(2)%kDOP =   0.50000000000000000E+00
autotroph_settings(2)%kNO3 =   0.50000000000000000E+00
autotroph_settings(2)%kNH4 =   0.50000000000000003E-01
autotroph_settings(2)%kSiO3 =   0.69999999999999996E+00
autotroph_settings(2)%kCO2 =   0.00000000000000000E+00
autotroph_settings(2)%Qp_fixed =   0.85470085470085479E-02
autotroph_settings(2)%gQfe_0 =   0.30000000000000001E-04
autotroph_settings(2)%gQfe_min =   0.25000000000000002E-05
autotroph_settings(2)%alphaPI_per_day =   0.28000000000000003E+00
autotroph_settings(2)%PCref_per_day =   0.50000000000000000E+01
autotroph_settings(2)%thetaN_max =   0.40000000000000000E+01
autotroph_settings(2)%loss_thres =   0.20000000000000000E-01
autotroph_settings(2)%loss_thres2 =   0.00000000000000000E+00
autotroph_settings(2)%temp_thres =  -0.10000000000000000E+02
autotroph_settings(2)%mort_per_day =   0.10000000000000001E+00
autotroph_settings(2)%mort2_per_day =   0.10000000000000000E-01
autotroph_settings(2)%agg_rate_max =   0.50000000000000000E+00
autotroph_settings(2)%agg_rate_min =   0.20000000000000000E-01
autotroph_settings(2)%loss_poc =   0.00000000000000000E+00
```

(continues on next page)

(continued from previous page)

```
autotroph_settings(2)%Ea =   0.32000000000000001E+00
autotroph_settings(3)%sname = 'diaz'
autotroph_settings(3)%lname = 'Diazotroph'
autotroph_settings(3)%temp_func_form_opt = 'q_10'
autotroph_settings(3)%Nfixer = T
autotroph_settings(3)%imp_calcifier = F
autotroph_settings(3)%exp_calcifier = F
autotroph_settings(3)%silicifier = F
autotroph_settings(3)%is_carbon_limited = F
autotroph_settings(3)%kFe =   0.45000000000000003E-04
autotroph_settings(3)%kPO4 =   0.14999999999999999E-01
autotroph_settings(3)%kDOP =   0.74999999999999997E-01
autotroph_settings(3)%kNO3 =   0.20000000000000000E+01
autotroph_settings(3)%kNH4 =   0.20000000000000001E+00
autotroph_settings(3)%kSiO3 =   0.00000000000000000E+00
autotroph_settings(3)%kCO2 =   0.00000000000000000E+00
autotroph_settings(3)%Qp_fixed =   0.27350427350427355E-02
autotroph_settings(3)%gQfe_0 =   0.60000000000000002E-04
autotroph_settings(3)%gQfe_min =   0.25000000000000002E-05
autotroph_settings(3)%alphaPI_per_day =   0.39000000000000001E+00
autotroph_settings(3)%PCref_per_day =   0.25000000000000000E+01
autotroph_settings(3)%thetaN_max =   0.25000000000000000E+01
autotroph_settings(3)%loss_thres =   0.20000000000000000E-01
autotroph_settings(3)%loss_thres2 =   0.10000000000000000E-02
autotroph_settings(3)%temp_thres =   0.15000000000000000E+02
autotroph_settings(3)%mort_per_day =   0.10000000000000001E+00
autotroph_settings(3)%mort2_per_day =   0.10000000000000000E-01
autotroph_settings(3)%agg_rate_max =   0.50000000000000000E+00
autotroph_settings(3)%agg_rate_min =   0.10000000000000000E-01
autotroph_settings(3)%loss_poc =   0.00000000000000000E+00
autotroph_settings(3)%Ea =   0.32000000000000001E+00
zooplankton_settings(1)%sname = 'zoo'
zooplankton_settings(1)%lname = 'Zooplankton'
zooplankton_settings(1)%temp_func_form_opt = 'q_10'
zooplankton_settings(1)%z_mort_0_per_day =   0.10000000000000001E+00
zooplankton_settings(1)%loss_thres =   0.74999999999999997E-01
zooplankton_settings(1)%z_mort2_0_per_day =   0.40000000000000002E+00
zooplankton_settings(1)%Ea =   0.65000000000000002E+00
grazing_relationship_settings(1,1)%sname = 'grz_sp_zoo'
grazing_relationship_settings(1,1)%lname = 'Grazing of sp by zoo'
grazing_relationship_settings(1,1)%auto_ind_cnt = 1
grazing_relationship_settings(1,1)%zoo_ind_cnt = 0
grazing_relationship_settings(1,1)%grazing_function = 1
grazing_relationship_settings(1,1)%z_umax_0_per_day =   0.32999999999999998E+01
grazing_relationship_settings(1,1)%z_grz =   0.12000000000000000E+01
grazing_relationship_settings(1,1)%graze_zoo =   0.29999999999999999E+00
grazing_relationship_settings(1,1)%graze_poc =   0.00000000000000000E+00
grazing_relationship_settings(1,1)%graze_doc =   0.59999999999999998E-01
grazing_relationship_settings(1,1)%f_zoo_detr =   0.12000000000000000E+00
grazing_relationship_settings(1,1)%auto_ind(1) = 1
grazing_relationship_settings(2,1)%sname = 'grz_diat_zoo'
grazing_relationship_settings(2,1)%lname = 'Grazing of diat by zoo'
grazing_relationship_settings(2,1)%auto_ind_cnt = 1
grazing_relationship_settings(2,1)%zoo_ind_cnt = 0
grazing_relationship_settings(2,1)%grazing_function = 1
grazing_relationship_settings(2,1)%z_umax_0_per_day =   0.31499999999999999E+01
grazing_relationship_settings(2,1)%z_grz =   0.12000000000000000E+01
```

(continues on next page)

(continued from previous page)

```
grazing_relationship_settings(2,1)%graze_zoo =   0.25000000000000000E+00
grazing_relationship_settings(2,1)%graze_poc =   0.39000000000000001E+00
grazing_relationship_settings(2,1)%graze_doc =   0.59999999999999998E-01
grazing_relationship_settings(2,1)%f_zoo_detr =   0.23999999999999999E+00
grazing_relationship_settings(2,1)%auto_ind(1) = 2
grazing_relationship_settings(3,1)%sname = 'grz_diaz_zoo'
grazing_relationship_settings(3,1)%lname = 'Grazing of diaz by zoo'
grazing_relationship_settings(3,1)%auto_ind_cnt = 1
grazing_relationship_settings(3,1)%zoo_ind_cnt = 0
grazing_relationship_settings(3,1)%grazing_function = 1
grazing_relationship_settings(3,1)%z_umax_0_per_day =   0.32999999999999998E+01
grazing_relationship_settings(3,1)%z_grz =   0.12000000000000000E+01
grazing_relationship_settings(3,1)%graze_zoo =   0.29999999999999999E+00
grazing_relationship_settings(3,1)%graze_poc =   0.10000000000000001E+00
grazing_relationship_settings(3,1)%graze_doc =   0.59999999999999998E-01
grazing_relationship_settings(3,1)%f_zoo_detr =   0.12000000000000000E+00
grazing_relationship_settings(3,1)%auto_ind(1) = 3
tracer_restore_vars(1) = ''
tracer_restore_vars(2) = ''
tracer_restore_vars(3) = ''
tracer_restore_vars(4) = ''
tracer_restore_vars(5) = ''
tracer_restore_vars(6) = ''
tracer_restore_vars(7) = ''
tracer_restore_vars(8) = ''
tracer_restore_vars(9) = ''
tracer_restore_vars(10) = ''
tracer_restore_vars(11) = ''
tracer_restore_vars(12) = ''
tracer_restore_vars(13) = ''
tracer_restore_vars(14) = ''
tracer_restore_vars(15) = ''
tracer_restore_vars(16) = ''
tracer_restore_vars(17) = ''
tracer_restore_vars(18) = ''
tracer_restore_vars(19) = ''
tracer_restore_vars(20) = ''
tracer_restore_vars(21) = ''
tracer_restore_vars(22) = ''
tracer_restore_vars(23) = ''
tracer_restore_vars(24) = ''
tracer_restore_vars(25) = ''
tracer_restore_vars(26) = ''
tracer_restore_vars(27) = ''
tracer_restore_vars(28) = ''
tracer_restore_vars(29) = ''
tracer_restore_vars(30) = ''
tracer_restore_vars(31) = ''
tracer_restore_vars(32) = ''
```

A python tool to generate input settings files is also provided: `MARBL_tools/MARBL_generate_settings_file.py`. This script creates `marbl.settings`, and organizes the output better than the Fortran test:

```
! config PFTs
PFT_defaults = "CESM2"
```

(continues on next page)

(continued from previous page)

```
autotroph_cnt = 3
max_grazer_prey_cnt = 3
zooplankton_cnt = 1

! config flags
ciso_lsource_sink = .true.
ciso_on = .false.
ladjust_bury_coeff = .false.
lcompute_nhx_surface_emis = .true.
lecovars_full_depth_tavg = .false.
lflux_gas_co2 = .true.
lflux_gas_o2 = .true.
lo2_consumption_scalef = .false.
lp_remin_scalef = .false.
lsource_sink = .true.
lvariable_PtoC = .true.

! config strings
init_bury_coeff_opt = "settings_file"

! general parameters
Jint_Ctot_thres_molpm2pyr = 1e-09
QCaCO3_max = 0.4
auto_mort2_exp = 1.75
bury_coeff_rmean_timescale_years = 10
caco3_bury_thres_depth =   3.0000000000000000e+05
caco3_bury_thres_omega_calc = 0.89
caco3_bury_thres_opt = "omega_calc"
ciso_fract_factors = "Laws"
f_graze_CaCO3_remin = 0.33
gQ_Fe_kFe_thres = 10.0
gQ_Si_kSi_thres = 6.0
gQsi_0 = 0.137
gQsi_max = 0.822
gQsi_min = 0.0457
o2_sf_o2_range_hi = 45.0
o2_sf_o2_range_lo = 5.0
o2_sf_val_lo_o2 = 2.6
parm_CaCO3_gamma = 0.02
parm_Fe_bioavail = 1.0
parm_Fe_desorption_rate0 =   9.9999999999999995e-07
parm_Lig_degrade_rate0 = 9.4e-05
parm_SiO2_gamma = 0.0
parm_f_prod_sp_CaCO3 = 0.07
parm_hPOC_CaCO3_ratio = 0.01
parm_hPOC_SiO2_ratio = 0.01
parm_hPOC_dust_ratio = 0.01
parm_labile_ratio = 0.94
parm_o2_min = 5.0
parm_o2_min_delta = 5.0
parm_sed_denitrif_coeff = 1
particulate_flux_ref_depth = 100
zoo_mort2_exp = 1.5

! general parameters (bury coeffs)
POM_bury_frac_max = 0.8
PON_bury_coeff = 0.5
```

(continues on next page)

(continued from previous page)

```
bSi_bury_frac_max = 1.0
parm_init_POC_bury_coeff = 2.54
parm_init_POP_bury_coeff = 0.36
parm_init_bSi_bury_coeff = 1.53

! general parameters (dissolution)
parm_CaCO3_diss =   5.0000000000000000e+04
parm_POC_diss =   1.0000000000000000e+04
parm_SiO2_diss =   6.5000000000000000e+04

! general parameters (nitrification)
parm_kappa_nitrif_per_day = 0.06
parm_nitrif_par_lim = 1.0

! general parameters (scavenging)
parm_FeLig_scavenge_rate0 = 1.2
parm_Fe_scavenge_rate0 = 22.0
parm_Lig_scavenge_rate0 = 0.015

! Scale lengths
parm_scalelen_vals(1) = 1
parm_scalelen_vals(2) = 3.6
parm_scalelen_vals(3) = 4.7
parm_scalelen_vals(4) = 4.8
parm_scalelen_z(1) =   1.0000000000000000e+04
parm_scalelen_z(2) =   2.5000000000000000e+04
parm_scalelen_z(3) =   5.0000000000000000e+04
parm_scalelen_z(4) =   1.0000000000000000e+05

! autotrophs
autotroph_settings(1)%Ea = 0.32
autotroph_settings(1)%Nfixer = .false.
autotroph_settings(1)%PCref_per_day = 5
autotroph_settings(1)%Qp_fixed =   8.5470085470085479e-03
autotroph_settings(1)%agg_rate_max = 0.5
autotroph_settings(1)%agg_rate_min = 0.01
autotroph_settings(1)%alphaPI_per_day = 0.39
autotroph_settings(1)%exp_calcifier = .false.
autotroph_settings(1)%gQfe_0 =   3.0000000000000001e-05
autotroph_settings(1)%gQfe_min = 2.5e-06
autotroph_settings(1)%imp_calcifier = .true.
autotroph_settings(1)%is_carbon_limited = .false.
autotroph_settings(1)%kCO2 = 0
autotroph_settings(1)%kDOP = 0.3
autotroph_settings(1)%kFe = 3e-05
autotroph_settings(1)%kNH4 = 0.01
autotroph_settings(1)%kNO3 = 0.25
autotroph_settings(1)%kPO4 = 0.01
autotroph_settings(1)%kSiO3 = 0
autotroph_settings(1)%lname = "Small Phyto"
autotroph_settings(1)%loss_poc = 0
autotroph_settings(1)%loss_thres = 0.01
autotroph_settings(1)%loss_thres2 = 0
autotroph_settings(1)%mort2_per_day = 0.01
autotroph_settings(1)%mort_per_day = 0.1
autotroph_settings(1)%silicifier = .false.
autotroph_settings(1)%sname = "sp"
```

(continues on next page)

(continued from previous page)

```
autotroph_settings(1)%temp_func_form_opt = "q_10"
autotroph_settings(1)%temp_thres = -10
autotroph_settings(1)%thetaN_max = 2.5
autotroph_settings(2)%Ea = 0.32
autotroph_settings(2)%Nfixer = .false.
autotroph_settings(2)%PCref_per_day = 5
autotroph_settings(2)%Qp_fixed =   8.5470085470085479e-03
autotroph_settings(2)%agg_rate_max = 0.5
autotroph_settings(2)%agg_rate_min = 0.02
autotroph_settings(2)%alphaPI_per_day = 0.28
autotroph_settings(2)%exp_calcifier = .false.
autotroph_settings(2)%gQfe_0 =   3.0000000000000001e-05
autotroph_settings(2)%gQfe_min = 2.5e-06
autotroph_settings(2)%imp_calcifier = .false.
autotroph_settings(2)%is_carbon_limited = .false.
autotroph_settings(2)%kCO2 = 0
autotroph_settings(2)%kDOP = 0.5
autotroph_settings(2)%kFe = 7e-05
autotroph_settings(2)%kNH4 = 0.05
autotroph_settings(2)%kNO3 = 0.5
autotroph_settings(2)%kPO4 = 0.05
autotroph_settings(2)%kSiO3 = 0.7
autotroph_settings(2)%lname = "Diatom"
autotroph_settings(2)%loss_poc = 0
autotroph_settings(2)%loss_thres = 0.02
autotroph_settings(2)%loss_thres2 = 0
autotroph_settings(2)%mort2_per_day = 0.01
autotroph_settings(2)%mort_per_day = 0.1
autotroph_settings(2)%silicifier = .true.
autotroph_settings(2)%sname = "diat"
autotroph_settings(2)%temp_func_form_opt = "q_10"
autotroph_settings(2)%temp_thres = -10
autotroph_settings(2)%thetaN_max = 4
autotroph_settings(3)%Ea = 0.32
autotroph_settings(3)%Nfixer = .true.
autotroph_settings(3)%PCref_per_day = 2.5
autotroph_settings(3)%Qp_fixed =   2.7350427350427355e-03
autotroph_settings(3)%agg_rate_max = 0.5
autotroph_settings(3)%agg_rate_min = 0.01
autotroph_settings(3)%alphaPI_per_day = 0.39
autotroph_settings(3)%exp_calcifier = .false.
autotroph_settings(3)%gQfe_0 =   6.0000000000000002e-05
autotroph_settings(3)%gQfe_min = 2.5e-06
autotroph_settings(3)%imp_calcifier = .false.
autotroph_settings(3)%is_carbon_limited = .false.
autotroph_settings(3)%kCO2 = 0
autotroph_settings(3)%kDOP = 0.075
autotroph_settings(3)%kFe = 4.5e-05
autotroph_settings(3)%kNH4 = 0.2
autotroph_settings(3)%kNO3 = 2
autotroph_settings(3)%kPO4 = 0.015
autotroph_settings(3)%kSiO3 = 0
autotroph_settings(3)%lname = "Diazotroph"
autotroph_settings(3)%loss_poc = 0
autotroph_settings(3)%loss_thres = 0.02
autotroph_settings(3)%loss_thres2 = 0.001
autotroph_settings(3)%mort2_per_day = 0.01
```

(continues on next page)

(continued from previous page)

```
autotroph_settings(3)%mort_per_day = 0.1
autotroph_settings(3)%silicifier = .false.
autotroph_settings(3)%sname = "diaz"
autotroph_settings(3)%temp_func_form_opt = "q_10"
autotroph_settings(3)%temp_thres = 15
autotroph_settings(3)%thetaN_max = 2.5

! zooplankton
zooplankton_settings(1)%Ea = 0.65
zooplankton_settings(1)%lname = "Zooplankton"
zooplankton_settings(1)%loss_thres = 0.075
zooplankton_settings(1)%sname = "zoo"
zooplankton_settings(1)%temp_func_form_opt = "q_10"
zooplankton_settings(1)%z_mort2_0_per_day = 0.4
zooplankton_settings(1)%z_mort_0_per_day = 0.1

! grazing
grazing_relationship_settings(1,1)%auto_ind(1) = 1
grazing_relationship_settings(1,1)%auto_ind_cnt = 1
grazing_relationship_settings(1,1)%f_zoo_detr = 0.12
grazing_relationship_settings(1,1)%graze_doc = 0.06
grazing_relationship_settings(1,1)%graze_poc = 0
grazing_relationship_settings(1,1)%graze_zoo = 0.3
grazing_relationship_settings(1,1)%grazing_function = 1
grazing_relationship_settings(1,1)%lname = "Grazing of sp by zoo"
grazing_relationship_settings(1,1)%sname = "grz_sp_zoo"
grazing_relationship_settings(1,1)%z_grz = 1.2
grazing_relationship_settings(1,1)%z_umax_0_per_day = 3.3
grazing_relationship_settings(1,1)%zoo_ind_cnt = 0
grazing_relationship_settings(2,1)%auto_ind(1) = 2
grazing_relationship_settings(2,1)%auto_ind_cnt = 1
grazing_relationship_settings(2,1)%f_zoo_detr = 0.24
grazing_relationship_settings(2,1)%graze_doc = 0.06
grazing_relationship_settings(2,1)%graze_poc = 0.39
grazing_relationship_settings(2,1)%graze_zoo = 0.25
grazing_relationship_settings(2,1)%grazing_function = 1
grazing_relationship_settings(2,1)%lname = "Grazing of diat by zoo"
grazing_relationship_settings(2,1)%sname = "grz_diat_zoo"
grazing_relationship_settings(2,1)%z_grz = 1.2
grazing_relationship_settings(2,1)%z_umax_0_per_day = 3.15
grazing_relationship_settings(2,1)%zoo_ind_cnt = 0
grazing_relationship_settings(3,1)%auto_ind(1) = 3
grazing_relationship_settings(3,1)%auto_ind_cnt = 1
grazing_relationship_settings(3,1)%f_zoo_detr = 0.12
grazing_relationship_settings(3,1)%graze_doc = 0.06
grazing_relationship_settings(3,1)%graze_poc = 0.1
grazing_relationship_settings(3,1)%graze_zoo = 0.3
grazing_relationship_settings(3,1)%grazing_function = 1
grazing_relationship_settings(3,1)%lname = "Grazing of diaz by zoo"
grazing_relationship_settings(3,1)%sname = "grz_diaz_zoo"
grazing_relationship_settings(3,1)%z_grz = 1.2
grazing_relationship_settings(3,1)%z_umax_0_per_day = 3.3
grazing_relationship_settings(3,1)%zoo_ind_cnt = 0

! tracer restoring
tracer_restore_vars(1) = ""
tracer_restore_vars(2) = ""
```

(continues on next page)

(continued from previous page)

```
tracer_restore_vars(3) = ""
tracer_restore_vars(4) = ""
tracer_restore_vars(5) = ""
tracer_restore_vars(6) = ""
tracer_restore_vars(7) = ""
tracer_restore_vars(8) = ""
tracer_restore_vars(9) = ""
tracer_restore_vars(10) = ""
tracer_restore_vars(11) = ""
tracer_restore_vars(12) = ""
tracer_restore_vars(13) = ""
tracer_restore_vars(14) = ""
tracer_restore_vars(15) = ""
tracer_restore_vars(16) = ""
tracer_restore_vars(17) = ""
tracer_restore_vars(18) = ""
tracer_restore_vars(19) = ""
tracer_restore_vars(20) = ""
tracer_restore_vars(21) = ""
tracer_restore_vars(22) = ""
tracer_restore_vars(23) = ""
tracer_restore_vars(24) = ""
tracer_restore_vars(25) = ""
tracer_restore_vars(26) = ""
tracer_restore_vars(27) = ""
tracer_restore_vars(28) = ""
tracer_restore_vars(29) = ""
tracer_restore_vars(30) = ""
tracer_restore_vars(31) = ""
tracer_restore_vars(32) = ""
```

Requirements on the GCM

After MARBL has been setup, the GCM will need to ensure that it can provide MARBL with all the data MARBL has requested and run any computations that MARBL is not capable of.

Provide data to MARBL

MARBL will need the following from the GCM:

1. *Tracer state* (including initial state)
2. *Specific forcing fields*
3. *Saved state from the previous timestep*

Computations across columns

Currently, the only time MARBL expects a GCM to set `lgcm_has_global_ops = .true.` in `init()` is when running with `ladjust_bury_coeff = .true.`. In that case, MARBL will ask the GCM to provide results from the following two functions:

1. *Global sums* of data MARBL computes column-by-column

2. *Running means* of fields (eventually MARBL will compute these internally and use saved state to maintain the mean)

Note that most typical experiments can be run with `ladjust_bury_coeff = .false.`, so for first-time implementations it is okay to skip this section.

What Tracer Tendencies will MARBL Compute?

This is an important question, because the GCM will need to provide the current state for each tracer at each timestep (including initial conditions at the beginning of the run). MARBL provides a stand-alone test in `$MARBL/tests/regression_tests/requested_tracers` that shows how the MARBL library passes this information to the GCM. For example, running

```
$ ./requested_tracers.py
```

Provides a list of the tracers in the base ecosystem module. The test output is below:

```
-----------------
Requested tracers
-----------------

1. PO4
2. NO3
3. SiO3
4. NH4
5. Fe
6. Lig
7. O2
8. DIC
9. DIC_ALT_CO2
10. ALK
11. ALK_ALT_CO2
12. DOC
13. DON
14. DOP
15. DOPr
16. DONr
17. DOCr
18. zooC
19. spChl
20. spC
21. spP
22. spFe
23. spCaCO3
24. diatChl
25. diatC
26. diatP
27. diatFe
28. diatSi
29. diazChl
30. diazC
31. diazP
32. diazFe
```

Tracer Metadata Available from the MARBL Interface

The details are found in `$MARBL/tests/driver_src/marbl.F90`:

```
call driver_status_log%log_header('Requested tracers', subname)
do n=1, size(marbl_instance%tracer_metadata)
  write(log_message, "(I0, 2A)") n, '. ',                                    &
    trim(marbl_instance%tracer_metadata(n)%short_name)
  call driver_status_log%log_noerror(log_message, subname)
end do
```

The `marbl_interface_class` contains an object `tracer_metadata`, the length of which is equal to the number of tracers MARBL is computing tendencies of. The `tracer_metadata_type` contains metadata for each tracer:

```
type, public :: marbl_tracer_metadata_type
   character(len=char_len) :: short_name
   character(len=char_len) :: long_name
   character(len=char_len) :: units
   character(len=char_len) :: tend_units
   character(len=char_len) :: flux_units
   logical                 :: lfull_depth_tavg
   character(len=char_len) :: tracer_module_name
end type marbl_tracer_metadata_type
```

The `short_name` and `long_name` are both unique to the tracer, and either can be used to inform the GCM which tracer each index refers to.

Example: Accessing Tracer Metadata in POP

Details are on the *implementation* page

What Forcing Fields has MARBL Requested?

This question is similar to "*What Tracer Tendencies will MARBL Compute?*", and so it should not be a surprise that the answer is also very similar. MARBL provides a stand-alone test in `$MARBL/tests/regression_tests/requested_forcing` as an example of how the MARBL library passes information to the GCM.

```
$ ./requested_forcings.py
```

Provides a list of the forcing fields requested with the default MARBL configuration.

```
--------------------------------
Requested surface forcing fields
--------------------------------

1. u10_sqr (units: cm^2/s^2)
2. sss (units: psu)
3. sst (units: degC)
4. Ice Fraction (units: unitless)
5. Dust Flux (units: g/cm^2/s)
6. Iron Flux (units: nmol/cm^2/s)
7. NOx Flux (units: nmol/cm^2/s)
8. NHy Flux (units: nmol/cm^2/s)
```

(continues on next page)

(continued from previous page)

```
9. Atmospheric Pressure (units: atmospheres)
10. xco2 (units: ppmv)
11. xco2_alt_co2 (units: ppmv)

---------------------------------
Requested interior forcing fields
---------------------------------

1. Dust Flux (units: g/cm^2/s)
2. Surface Shortwave (units: W/m^2)
3. Potential Temperature (units: degC)
4. Salinity (units: psu)
5. Pressure (units: bars)
6. Iron Sediment Flux (units: nmol/cm^2/s)
```

Forcing Field Metadata Available from the MARBL Interface

The details are found in `$MARBL/tests/driver_src/marbl.F90`:

```
! Log requested surface forcing fields
call driver_status_log%log_header('Requested surface forcing fields', subname)
do n=1,size(marbl_instance%surface_flux_forcings)
  write(log_message, "(I0, 5A)") n, '. ', &
        trim(marbl_instance%surface_flux_forcings(n)%metadata%varname), &
        ' (units: ', trim(marbl_instance%surface_flux_forcings(n)%metadata%field_
↪units),')'
  call driver_status_log%log_noerror(log_message, subname)
end do
! Log requested interior forcing fields
call driver_status_log%log_header('Requested interior forcing fields', subname)
do n=1,size(marbl_instance%interior_tendency_forcings)
  write(log_message, "(I0, 5A)") n, '. ', &
       trim(marbl_instance%interior_tendency_forcings(n)%metadata%varname), &
       ' (units: ', trim(marbl_instance%interior_tendency_forcings(n)%metadata%field_
↪units),')'
  call driver_status_log%log_noerror(log_message, subname)
end do
```

The `marbl_interface_class` contains two objects (`surface_flux_forcings` and `interior_tendency_forcings`) that are arrays with dimension equal to the number of surface and interior forcing fields, respectively. Both are of type `marbl_forcing_fields_type`, which contains a `metadata` object. The `marbl_forcing_fields_metadata_type` contains metadata for each tracer:

```
type :: marbl_forcing_fields_metadata_type
   ! Contains variable names and units for required forcing fields as well as
   ! dimensional information; actual forcing data is in array of
   ! marbl_forcing_fields_type
   character(len=char_len) :: varname
   character(len=char_len) :: field_units
   integer                 :: rank            ! 0d or 1d
   integer, allocatable    :: extent(:)       ! length = rank
end type marbl_forcing_fields_metadata_type
```

The `varnames` member of this data type is the only unique identifier provided.

Restoring Tracers Towards a Prior State

Some GCMs restore certain tracers towards observations in particular regions. For example, POP restores `PO4`, `NO3`, `SiO3`, `ALK`, and `ALK_ALT_CO2` in marginal seas. MARBL treats tracer restoring terms as an interior forcing, which can be seen by running `./requested_forcings.py -s ../../input_files/settings/marbl_with_restore.settings`:

```
--------------------------------
Requested surface forcing fields
--------------------------------

1. u10_sqr (units: cm^2/s^2)
2. sss (units: psu)
3. sst (units: degC)
4. Ice Fraction (units: unitless)
5. Dust Flux (units: g/cm^2/s)
6. Iron Flux (units: nmol/cm^2/s)
7. NOx Flux (units: nmol/cm^2/s)
8. NHy Flux (units: nmol/cm^2/s)
9. Atmospheric Pressure (units: atmospheres)
10. xco2 (units: ppmv)
11. xco2_alt_co2 (units: ppmv)

---------------------------------
Requested interior forcing fields
---------------------------------

1. Dust Flux (units: g/cm^2/s)
2. Surface Shortwave (units: W/m^2)
3. Potential Temperature (units: degC)
4. Salinity (units: psu)
5. Pressure (units: bars)
6. Iron Sediment Flux (units: nmol/cm^2/s)
7. SiO3 Restoring Field (units: mmol/m^3)
8. SiO3 Restoring Inverse Timescale (units: 1/s)
9. NO3 Restoring Field (units: mmol/m^3)
10. NO3 Restoring Inverse Timescale (units: 1/s)
11. PO4 Restoring Field (units: mmol/m^3)
12. PO4 Restoring Inverse Timescale (units: 1/s)
13. ALK Restoring Field (units: meq/m^3)
14. ALK Restoring Inverse Timescale (units: 1/s)
15. ALK_ALT_CO2 Restoring Field (units: meq/m^3)
16. ALK_ALT_CO2 Restoring Inverse Timescale (units: 1/s)
```

MARBL is not aware of the horizontal grid, and will request these forcings for all columns. Set a tracer's `Restoring Inverse Timescale` to 0 in columns where restoring is not desired. Note that the surface forcing fields are the same, but there are 10 additional interior forcing fields to pass to MARBL.

Example: Accessing Forcing Field Metadata in POP

Details are on the *implementation* page

How Many Saved State Fields Need to be Stored?

MARBL does not yet have a stand-alone test to provide a list of requested saved state fields, but `marbl_saved_state_init()` shows that there are four saved state fields - two for the surface and two for the interior. The two surface fields are `surface pH` and `surface pH (alternate CO2)`:

```
call surface_state%construct(num_elements_surface_flux, num_levels)

lname = 'surface pH'
sname = 'PH_SURF'
units = 'pH'
vgrid = 'none'
rank  = 2
call surface_state%add_state(lname, sname, units, vgrid, rank,            &
     surf_ind%ph_surf, marbl_status_log)
if (marbl_status_log%labort_marbl) then
  call marbl_status_log%log_error_trace("add_state(PH_SURF)", subname)
  return
end if

lname = 'surface pH (alternate CO2)'
sname = 'PH_SURF_ALT_CO2'
units = 'pH'
vgrid = 'none'
rank  = 2
call surface_state%add_state(lname, sname, units, vgrid, rank,            &
     surf_ind%ph_alt_co2_surf, marbl_status_log)
if (marbl_status_log%labort_marbl) then
  call marbl_status_log%log_error_trace("add_state(PH_SURF_ALT_CO2)", subname)
  return
end if
```

The two interior state fields are `3D pH` and `3D pH (alternate CO2)`

```
call interior_state%construct(num_elements_interior_tendency, num_levels)

lname = '3D pH'
sname = 'PH_3D'
units = 'pH'
vgrid = 'layer_avg'
rank  = 3
call interior_state%add_state(lname, sname, units, vgrid, rank,           &
     interior_ind%ph_col, marbl_status_log)
if (marbl_status_log%labort_marbl) then
  call marbl_status_log%log_error_trace("add_state(PH_3D)", subname)
  return
end if

lname = '3D pH (alternate CO2)'
sname = 'PH_3D_ALT_CO2'
units = 'pH'
vgrid = 'layer_avg'
rank  = 3
call interior_state%add_state(lname, sname, units, vgrid, rank,           &
     interior_ind%ph_alt_co2_col, marbl_status_log)
if (marbl_status_log%labort_marbl) then
  call marbl_status_log%log_error_trace("add_state(PH_3D_ALT_CO2)", subname)
```

(continues on next page)

(continued from previous page)

```
  return
end if
```

The pH computation is an iterative solver, and it has proven useful to use the pH at timestep `t` as an initial value when solving for time `t+1`.

Saved State in the Interface

MARBL splits the saved state fields between those needed for computing surface fluxes and those needed for computing interior tendencies, so on the interface we have

```
type, public :: marbl_interface_class

  .
  .
  .
  type(marbl_saved_state_type)              , public               :: surface_flux_
↪saved_state             ! input/output
  type(marbl_saved_state_type)              , public               :: interior_
↪tendency_saved_state        ! input/output
  .
  .
  .
end type marbl_interface_class
```

The `marbl_saved_state_type` is a wrapper for `marbl_single_saved_state_type`:

```
type, public :: marbl_single_saved_state_type
  integer                   :: rank
  character(len=char_len) :: long_name
  character(len=char_len) :: short_name
  character(len=char_len) :: units
  character(len=char_len) :: vertical_grid ! 'none', 'layer_avg', 'layer_iface'
  real(r8), allocatable, dimension(:)   :: field_2d  ! num_elements
  real(r8), allocatable, dimension(:,:) :: field_3d  ! num_levels, num_elements
contains
  procedure :: construct => marbl_single_saved_state_construct
end type marbl_single_saved_state_type

!*****************************************************************************

type, public :: marbl_saved_state_type
  integer :: saved_state_cnt
  integer :: num_elements
  integer :: num_levels
  type(marbl_single_saved_state_type), dimension(:), pointer :: state => NULL()
 contains
   procedure, public :: construct => marbl_saved_state_constructor
   procedure, public :: add_state => marbl_saved_state_add
end type marbl_saved_state_type
```

What Should the GCM Do?

After `marbl_instance%surface_flux_compute()` returns, the GCM needs to process `marbl_instance%surface_flux_saved_state`. That means looping through each element in the `marbl_instance%surface_flux_saved_state%state(:)` array, checking `state(n)%rank`, and then storing either `state(n)%field_2d` or `state(n)%field_3d` in a global array. Before calling `surface_flux_compute()` in the next time step, these saved values should be copied back into `marbl_intance%surface_flux_saved_state`.

Similar actions must be taken with `marbl_instance%interior_tendency_saved_state` before / after calls to `marbl_instance%interior_tendency_compute()`.

Global Sums of Fields Provided by MARBL

Documentation for this feature will not be available until the MARBL 1.0.0 release.

Running Means of Fields Provided by MARBL

Documentation for this feature will not be available until the MARBL 1.0.0 release.

Compute Surface Fluxes

`surface_flux_compute()` computes surface fluxes (and related diagnostics) over a 1D array of grid cells. The array is assumed to be length `num_elements_surface_flux`, per *The init() interface*.

Example from Stand-Alone MARBL

The GCM needs to make sure the MARBL instance has all the data it needs to compute surface fluxes correctly. The following blocks of code can all be found in `tests/driver_src/marbl_call_compute_subroutines_drv.F90`:

Step 1. Set global scalars

Currently, there are no MARBL configurations requiring *global scalars* to compute the surface fluxes. To prepare for future updates where that is no longer the case, it is recommended that the GCM calls

```
! 5. Call surface_flux_compute() (all columns simultaneously)
!    (a) call set_global_scalars() for consistent setting of time-varying scalars
!        [surface_flux computation doesn't currently have any time-varying scalars]
call marbl_instances(n)%set_global_scalars('surface_flux')
```

Step 2. Copy data into MARBL

MARBL needs surface tracer values, surface flux forcings, and saved state values. These are copied to `marbl_instances(n)%tracers_at_surface`, `marbl_instances(n)%surface_flux_forcings`, and `marbl_instances(n)%surface_flux_saved_state%state`, respectively.

```
do col_id_loc = 1, col_cnt(n)
  col_id = col_start(n)+col_id_loc

  ! (b) copy surface tracer values into marbl_instances(n)%tracers_at_surface
  marbl_instances(n)%tracers_at_surface(col_id_loc, :) = tracer_initial_vals(:, 1,␣
↪col_id)

  ! (c) copy surface flux forcings into marbl_instances(n)%surface_flux_forcings
  do m=1, size(marbl_instances(n)%surface_flux_forcings)
    if (associated(marbl_instances(n)%surface_flux_forcings(m)%field_0d)) then
      marbl_instances(n)%surface_flux_forcings(m)%field_0d(col_id_loc) = surface_flux_
↪forcings(m)%field_0d(col_id)
    else
      marbl_instances(n)%surface_flux_forcings(m)%field_1d(col_id_loc,:) = surface_
↪flux_forcings(m)%field_1d(col_id,:)
    end if
  end do
end do

!    (d) populate marbl_instances(n)%surface_flux_saved_state (with 0s)
do m=1, size(marbl_instances(n)%surface_flux_saved_state%state)
  marbl_instances(n)%surface_flux_saved_state%state(m)%field_2d(:) = 0._r8
end do
```

Step 3. Call `surface_flux_compute()`

Since all the data is available on the class object, the call to the routine does not require any arguments:

```
!    (e) call surface_flux_compute()
call marbl_instances(n)%surface_flux_compute()
```

Step 4. Copy values MARBL will need later into a local buffer

After computing the surface fluxes, MARBL returns several different fields to the GCM. Each GCM will handle these fields in its own way, but it is important to know where to find them. In the descriptions below, `num_elements_surface_flux` is the number of grid cells MARBL computes surface fluxes for simultaneously.

1. The surface fluxes themselves, which are needed in the source term of the advection solver
   - `marbl_instance%surface_fluxes(:,:)` is a `num_elements_surface_flux` by `marbl_tracer_cnt` array
2. Surface forcing fields, which may be requested by the GCM
3. Saved state, which the GCM should store and then provide to MARBL on the next time step
   - `marbl_instance%surface_flux_saved_state` is `marbl_saved_state_type`
   - `marbl_instance%surface_flux_saved_state%state(:)` is array of `marbl_single_saved_state_type` containing data GCM should store for next time step
4. Values that need a global operation performed (per *above*, there are not yet any of these)
   - `marbl_instance%glo_avg_fields_surface_flux(:,:)` is a `num_elements_surface_flux` by `glo_avg_field_cnt_surface_flux` array

- GCM should store fields in global array and compute an average prior to calling `interior_tendency_compute()`
- Recommended to do global average as soon as all surface fluxes have been computed

5. Diagnostics for the GCM to provide to the user
   - `marbl_instance%surface_flux_diags` is `marbl_diagnostics_type`
   - `marbl_instance%surface_flux_diags%diags(:)` is array of `marbl_single_diagnostic_type` containing data GCM should add to diagnostic output

The stand-alone driver does not hold on to saved state (there is no time stepping involved). It also does not request any surface forcing fields or compute global averages. After the call to `surface_flux_compute()`, the standalone driver copies diags into the buffer and stores the surface fluxes (which are also written to the netCDF output file).

```
!    (f) write to diagnostic buffers
!        Note: passing col_start and col_cnt => surface flux diagnostic buffer
call marbl_io_copy_into_diag_buffer(col_start(n), col_cnt(n), marbl_instances(n))
surface_fluxes((col_start(n)+1):(col_start(n)+col_cnt(n)),:) = marbl_instances(n)
↪%surface_fluxes(:,:)
```

A more complete example can be found in *how POP handles MARBL output*.

Compute Interior Tracer Tendencies

`interior_tendency_compute()` computes interior tracer tendencies (and related diagnostics) for a single column. (Recall that `num_elements_interior_tendency = 1`, per *The init() interface*.)

Example from Stand-Alone MARBL

The GCM needs to make sure the MARBL instance has all the data it needs to compute surface fluxes correctly. The following blocks of code can all be found in `tests/driver_src/marbl_call_compute_subroutines_drv.F90`:

Step 1. Set global scalars

If MARBL is configured with `ladjust_bury_coeff = .true.` then it will request running means of global averages of a few fields.

```
!  (a) call set_global_scalars() for consistent setting of time-varying scalars
!      [necessary when running ladjust_bury_coeff, since GCM is responsible
!       for computing running means of values needed to compute burial coefficients]
call marbl_instances(n)%set_global_scalars('interior_tendency')
```

Note that at this point, the GCM is responsible for both the global averaging and keeping the running means; in the future running means will be computed in MARBL (and requested as part of saved state). At present there is not an example of this behavior in the stand-alone driver.

Step 2. Copy data into MARBL

Interior tracer tendencies are computed for a single column in MARBL. For each column, MARBL needs to know the following:

1. Domain information (level depths, layer thicknesses, number of active levels, etc)
2. Tracer values (for each level and each tracer)
3. Interior forcing data
4. Saved state

```
!  (b) set domain information
!      In this example, the vertical grid is the same from column to column and
!      therefore set during initialization. The columns vary in depth, so
!      the index of the bottom layer must be updated for each column.
marbl_instances(n)%domain%kmt = active_level_cnt(col_id)

!  (c) copy tracer values into marbl_instances(n)%tracers
marbl_instances(n)%tracers = tracer_initial_vals(:,:,col_id)

!  (d) copy interior tendency forcings into marbl_instances(n)%interior_tendency_
↪forcings
do m=1, size(marbl_instances(n)%interior_tendency_forcings)
  if (associated(marbl_instances(n)%interior_tendency_forcings(m)%field_0d)) then
    marbl_instances(n)%interior_tendency_forcings(m)%field_0d(1) = &
         interior_tendency_forcings(m)%field_0d(col_id)
  else
    marbl_instances(n)%interior_tendency_forcings(m)%field_1d(1,:) = &
         interior_tendency_forcings(m)%field_1d(col_id,:)
  end if
end do

!  (e) populate marbl_instances(n)%interior_tendency_saved_state (with 0s)
do m=1, size(marbl_instances(n)%interior_tendency_saved_state%state)
  if (allocated(marbl_instances(n)%interior_tendency_saved_state%state(m)%field_2d))␣
↪then
    marbl_instances(n)%interior_tendency_saved_state%state(m)%field_2d(:) = 0._r8
  else
    marbl_instances(n)%interior_tendency_saved_state%state(m)%field_3d(:,1) = 0._r8
  end if
end do
```

Step 3. Call `interior_tendency_compute()`

Since all the data is available on the class object, the call to the routine does not require any arguments:

```
!  (f) call interior_tendency_compute()
call marbl_instances(n)%interior_tendency_compute()
```

Step 4. Copy values MARBL will need later into a local buffer

After computing the tracer tendencies, MARBL returns several fields to the GCM. Each GCM will handle these fields in its own way, but it is important to know where to find them. MARBL returns all these fields on a per-column basis. Some variables may have a `num_elements_interior_tendency` dimension, but that dimension is hard-coded to be 1. POP checks to ensure that MARBL does set any values to `NaN`, other GCMs may or may not want to do so as well.

1. The interior tendencies themselves, which are needed in the source term of the advection solver

- `marbl_instance%interior_tendencies(:,:)` is a `marbl_tracer_cnt` by `num_levels` array

2. Saved state, which the GCM should store and then provide to MARBL on the next time step
   - `marbl_instance%interior_tendency_saved_state` is `marbl_saved_state_type`
   - `marbl_instance%interior_tendency_saved_state%state(:)` is array of `marbl_single_saved_state_type` containing data GCM should store for next time step
3. Values that need a global operation performed
   - `marbl_instance%glo_avg_fields_interior_tendency(:)` is an array of length `glo_avg_field_cnt_interior_tendency`
   - The global average should be computed prior to the next `interior_tendency_compute()` call
4. Diagnostics for the GCM to provide to the user
   - `marbl_instance%interior_tendency_diags` is `marbl_diagnostics_type`
   - `marbl_instance%interior_tendency_diags%diags(:)` is array of `marbl_single_diagnostic_type` containing data GCM should add to diagnostic output

The stand-alone driver does not hold on to saved state (there is no time stepping involved) or compute global averages. After the call to `interior_tendency_compute()`, the standalone driver copies diags into the buffer and stores the tendencies (which are also written to the netCDF output file).

```
!  (g) write to diagnostic buffer
!        Note: passing just col_id => interior tendency diagnostic buffer
call marbl_io_copy_into_diag_buffer(col_id, marbl_instances(n))
interior_tendencies(:,:,col_id) = marbl_instances(n)%interior_tendencies(:,:)
```

A more complete example can be found in *how POP handles MARBL output*.

Shutdown

The shutdown stage is where MARBL deallocates memory (including memory allocated inside of derived types, such as the diagnostic indexing types). The only object still accessible after shutdown is `marbl_interface%timer_summary`, so GCMs can still access performance timers.

The `shutdown()` interface

```
subroutine shutdown(this)

  class(marbl_interface_class), intent(inout) :: this
```

No additional arguments are needed for calls to `marbl_instance%shutdown()`.

4.2 Examples of MARBL Implementation

4.2.1 The MARBL Standalone Driver

The MARBL stand alone driver currently exists solely as a way to test the MARBL code base. Testing is broken into three categories:

1. Does the Fortran code *build* correctly? We test both the MARBL library and the test framework driver. These tests can be found in `$MARBLROOT/tests/bld_tests`.
2. *Unit testing*: do specific subroutines return the correct value? These tests can be found in `$MARBLROOT/tests/unit_tests`.
3. *Regression testing*: do specific call sequences continue to return the same value? These tests can be found in `$MARBLROOT/tests/regression_tests`.

All testing can be run via python scripts that import code from `$MARBLROOT/tests/python_for_tests`. This directory contains a class used to control how to build MARBL and what options should be available to the user. It also maintains settings for running the standalone tests on a handful of super computers (loading proper modules, etc).

Testing the MARBL Build System

There are two build test scripts in `$MARBLROOT/tests/bld_tests`: `bld_lib.py` and `bld_exe.py`. The former just builds `libmarbl.a` while the latter also builds the standalone driver that runs the *unit tests* and *regression tests*. For known super computers, these scripts build with all defined compilers. For unknown computers, these scripts check to see what supported compilers are in `$PATH` and test with each of them.

```
$ ./bld_lib.py
(bld_lib): No machine specified and [machine name] is not recognized
(bld_lib): This test will assume you are not running on a supported cluster
(bld_lib): Override with the --mach option if this is not correct
(bld_lib): Found the following compilers in $PATH: ['gnu', 'pgi']
```

Unit Tests in MARBL

There are two subdirectories in `$MARBLROOT/tests/unit_tests`: `get_put/` and `utils_routines/`. Each contains a python script that launches a test to check the correctness of a small piece of the MARBL code base.

Testing the `get()` and `put()` subroutines

This test is designed to ensure the variable names passed to `marbl_instance%put()` alter the correct variable in memory. All logical variables are set to `.true.`, string variables are set to `'-1'`, and numeric variables are set to `-1`. Then values are checked via `marbl_instance%get()` – any values that are not `.true.` or `-1` were not set properly.

This builds upon error checks in the MARBL code base itself. Here is output from a failing test, where `gQsi_min` inadvertently points to `gQsi_max` in memory:

```
Beginning get_put test...
MARBL ERROR (marbl_settings_mod:add_var): gQsi_min and gQsi_max both point to same␣
↪variable in memory.
MARBL ERROR (marbl_settings_mod:marbl_settings_define_general_parms): Error reported␣
↪from this%add_var(gQsi_min)
MARBL ERROR (marbl_init_mod:marbl_init_parameters_pre_tracers): Error reported from␣
↪marbl_settings_define_general_parms()
MARBL ERROR (marbl_interface:init): Error reported from marbl_init_parameters_pre_
↪tracers
MARBL ERROR (marbl_get_put_drv:test): Error reported from marbl_loc%init
STOP 1
(run_exe): ERROR in executable
```

Perhaps this unit test has been deprecated, as the above error appears when running any executable that initializes a MARBL instance. The expected output from the test is:

```
Beginning get_put test...
Setting variables to .true. or -1 ...
... Done!
Making sure variables are .true. or -1
... Done!

-------------
Timer summary
-------------

There are 4 timers being returned
----
MARBL Init:       0.141 seconds
MARBL surface_flux_compute:       0.000 seconds
MARBL interior_tendency_compute:       0.000 seconds
MARBL carbonate chemistry:       0.000 seconds
```

Testing additional small subroutines

There are two small utility functions in `marbl_utils_mod` that MARBL relies on, and both of them are tested for correctness.

The first is `marbl_utils_linear_root` – given two points in `(x, y)` space, this function returns a linear root between the point if one exists. It is used to compute the saturation depth for calcite and argonite. The unit tests ensure that several conditions are handled correctly:

1. Finding a root when increasing from a negative value to a positive value
2. The same, but passing in the point with the larger `x` value first
3. Finding a root when decreasing from a positive value to a negative value
4. The same, but passing in the point with the larger `x` value first
5. Finding a root when the `y` value at the left endpoint is 0 (should return the left endpoint `x` value)
6. Finding a root when the `y` value at the right endpoint is 0 (should return the right endpoint `x` value)
7. Finding a root when the `y` value at both endpoints are 0 (should return the right endpoint `x` value)
8. Determining that no root exists if both `y` values are positive
9. Determining that no root exists if both `y` values are negative

```
-----------------
Linear Root Tests
-----------------

PASS: Test 1 [linear root: root between (1.0, -1.0) and (2.0, 1.0)]
      Root at x = 1.5
PASS: Test 2 [linear root: root between (2.0, 1.0) and (1.0, -1.0)]
      Root at x = 1.5
PASS: Test 3 [linear root: root between (1.0, 1.0) and (2.0, -1.0)]
      Root at x = 1.5
PASS: Test 4 [linear root: root between (2.0, -1.0) and (1.0, 1.0)]
      Root at x = 1.5
```

(continues on next page)

(continued from previous page)

```
PASS: Test 5 [linear root: root between (5.0, .0) and (7.0, 3.0)]
      Root at x = 5.0
PASS: Test 6 [linear root: root between (5.0, 3.0) and (7.0, .0)]
      Root at x = 7.0
PASS: Test 7 [linear root: root between (5.0, .0) and (7.0, .0)]
      Root at x = 7.0
PASS: Test 8 (linear root: root between (1.0, 1.0) and (2.0, 3.0)]
      No root found
PASS: Test 9 (linear root: root between (1.0, -1.0) and (2.0, -3.0)]
      No root found

** passed all linear root tests **
```

There second function, `marbl_utils_str_to_substrs()`, takes a string and a delimiter as arguments and returns an array of substrings where each substring ends before a delimiter and the next substring starts after. This is useful when the settings file defines values for an array, such as over-riding the first five elements of `tracer_restore_vars`:

```
tracer_restore_vars = '','','','',''
```

It is also useful for ignoring comments in the settings file, which use the `!` character. In both of these cases, it is important to ensure that if the delimiter appears within a string itself then it is not treated as a delimiter. The tests for this subroutine are:

1. If the delimiter is `,` then a string with no `,` returns a single substring equal to the original string
2. If the delimiter is `,` then a string with a `,` is broken into two substrings
3. If the delimiter is `,` then a `,` appearing between two `'` is ignored
4. If the delimiter is `!` then the first substring only contains text leading up to the first `!`
5. If the delimiter is `!` then a `!` appearing between two `'` is ignored

```
-------------------------
String -> Substrings Tests
-------------------------

PASS: Test 1 (str_to_substrs: .true.)
      substr(1): .true.
PASS: Test 2 (str_to_substrs: 123, 456)
      substr(1): 123
      substr(2):  456
PASS: Test 3 (str_to_substrs: 'ABC, DEF', 'GHI')
      substr(1): 'ABC, DEF'
      substr(2):  'GHI'

** passed all string to substrings tests **

----------------------
Comment-Stripping Tests
----------------------

PASS: Test 1 (strip_comments: ciso_on = .true.  ! Turn on ciso)
      string: ciso_on = .true.
PASS: Test 2 (strip_comments: autotrophs(1)%lname='Small Phytoplankton!')
      string: autotrophs(1)%lname='Small Phytoplankton!'
```

(continues on next page)

(continued from previous page)

```
** passed all comment stripping tests **

All unit tests passed!
```

If more utility functions are added in the future, they will also be tested here. Further, if we encounter an edge case where one of the functions misbehaves, that edge case will be added to the tests after the function has been fixed.

Regression Tests in MARBL

There are eight subdirectories in `tests/regression_tests`, though only one is truely a regression test. The other seven subdirectories can be better thought of as examples of specific MARBL functionality, and indeed may be moved from the `tests/` directory to an `examples/` directory in a future update.

`call_compute_subroutines/`

`call_compute_subroutines/` is the only regression test provided. It computes surface fluxes and interior tendencies based on forcing from five columns of the standard POP configuration. Initial conditions are forcing fields for these columns can be found in `tests/input_files/initial_conditions/call_compute_subroutines.20190718.nc`.

All the default diagnostic values are output to a netCDF file The test can be run with one, two, or five MARBL instances, and regardless of instance count the results should be bit-for-bit identical. Additionally, there is a baseline provided in `tests/input_files/baselines/call_compute_subroutines.history.nc`.

The `MARBL_tools/netcdf_comparison.py` script can be used to compare netCDF output between instance counts (use `--strict exact` to check for bit-for-bit match) or between a test run and the baseline (us `--strict loose` to account for round-off level changes that may creep in due to hardware differences). The `--strict loose` option sets bounds on the allowable differences between the new run and the baseline – by default, absolute differences under 10^{-16} are allowed in variables within 10^{-16} of 0, and relative differences under 10^{-11} are allowed in all other variables. These different boundaries are necessary because round-off level diffences in variables that are numerically close to 0 may have relative errors that are O(1).

Other Examples

There are five subdirectories that provide information on how MARBL is configured, and three are documented elsewhere on on this site:

1. `gen_settings_file/` generates a *settings file* for a later MARBL run.
2. `requested_forcings/` lists the *forcing fields MARBL needs* in a given configuration.
3. `requested_tracers/` lists the *tracer tendencies computed* in a given configuration.

The stand-alone driver can also report what tracers are being restored (without *looking at the rest of the forcing fields*) as well as what diagnostics are being returned to the GCM.

`requested_restoring/`

Running `./requested_restoring.py` returns a list of tracers to be restored. By default, it relies on the settings file `tests/input_files/settings/marbl_with_restore.settings`, which sets a few elements of `tracer_restore_vars`.

```
---------------------------
Requested tracers to restore
---------------------------

1. SiO3
2. NO3
3. PO4
4. ALK
5. ALK_ALT_CO2
```

If no tracers are to be restored (as per `./requested_restoring.py -s None`) the output should be

```
---------------------------
Requested tracers to restore
---------------------------

No tracers to restore!
```

The details are found in `$MARBL/tests/driver_src/marbl.F90`:

```
! Log tracers requested for restoring
call driver_status_log%log_header('Requested tracers to restore', subname)
cnt = 0
do n=1,size(marbl_instances(1)%interior_tendency_forcings)
  varname = marbl_instances(1)%interior_tendency_forcings(n)%metadata%varname
  if (index(varname, 'Restoring Field').gt.0) then
    cnt = cnt + 1
    varname = varname(1:scan(varname,' ')-1)
    write(log_message, "(I0, 2A)") cnt, '. ', trim(varname)
call driver_status_log%log_noerror(log_message, subname)
  end if
end do
if (cnt.eq.0) then
  call driver_status_log%log_noerror('No tracers to restore!', subname)
end if
```

The driver looks at metadata for `interior_tendency_forcings(:)` and tracks forcings containing `'Restoring Field'`.

requested_diags/

Running `./requested_diags.py` returns a list of diagnostics returned from MARBL. The list is split between surface flux diagnostics and interior tendency diagnostics.

```
------------------------
Surface flux diagnostics
------------------------

1. ECOSYS_IFRAC: Ice Fraction for ecosys fluxes (units: fraction)
2. ECOSYS_XKW: XKW for ecosys fluxes (units: cm/s)
3. ECOSYS_ATM_PRESS: Atmospheric Pressure for ecosys fluxes (units: atmospheres)
4. PV_O2: PV_O2 (units: cm/s)
5. SCHMIDT_O2: O2 Schmidt Number (units: 1)
6. O2SAT: O2 Saturation (units: mmol/m^3)
7. CO2STAR: CO2 Star (units: mmol/m^3)
8. DCO2STAR: D CO2 Star (units: mmol/m^3)
```

(continues on next page)

(continued from previous page)

```
9. pCO2SURF: surface pCO2 (units: ppmv)
10. DpCO2: D pCO2 (units: ppmv)
11. PV_CO2: CO2 Piston Velocity (units: cm/s)
12. SCHMIDT_CO2: CO2 Schmidt Number (units: 1)
13. FG_CO2: DIC Surface Gas Flux (units: mmol/m^3 cm/s)
14. PH: Surface pH (units: 1)
15. ATM_CO2: Atmospheric CO2 (units: ppmv)
16. CO2STAR_ALT_CO2: CO2 Star, Alternative CO2 (units: mmol/m^3)
17. DCO2STAR_ALT_CO2: D CO2 Star, Alternative CO2 (units: mmol/m^3)
18. pCO2SURF_ALT_CO2: surface pCO2, Alternative CO2 (units: ppmv)
19. DpCO2_ALT_CO2: D pCO2, Alternative CO2 (units: ppmv)
20. FG_ALT_CO2: DIC Surface Gas Flux, Alternative CO2 (units: mmol/m^3 cm/s)
21. PH_ALT_CO2: Surface pH, Alternative CO2 (units: 1)
22. ATM_ALT_CO2: Atmospheric Alternative CO2 (units: ppmv)
23. IRON_FLUX: Atmospheric Iron Flux (units: mmol/m^2/s)
24. DUST_FLUX: Dust Flux (units: g/cm^2/s)
25. NOx_FLUX: Flux of NOx from Atmosphere (units: nmol/cm^2/s)
26. NHy_FLUX: Flux of NHy from Atmosphere (units: nmol/cm^2/s)
27. NHx_SURFACE_EMIS: Emission of NHx to Atmosphere (units: nmol/cm^2/s)

------------------------------
Interior tendency diagnostics
------------------------------

1. zsatcalc: Calcite Saturation Depth (units: cm)
2. zsatarag: Aragonite Saturation Depth (units: cm)
3. O2_ZMIN: Vertical Minimum of O2 (units: mmol/m^3)
4. O2_ZMIN_DEPTH: Depth of Vertical Minimum of O2 (units: cm)
5. photoC_TOT_zint: Total C Fixation Vertical Integral (units: mmol/m^3 cm/s)
6. photoC_TOT_zint_100m: Total C Fixation Vertical Integral, 0-100m (units: mmol/m^3␣
↪cm/s)
7. photoC_NO3_TOT_zint: Total C Fixation from NO3 Vertical Integral (units: mmol/m^3␣
↪cm/s)
8. photoC_NO3_TOT_zint_100m: Total C Fixation from NO3 Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
9. DOC_prod_zint: Vertical Integral of DOC Production (units: mmol/m^3 cm/s)
10. DOC_prod_zint_100m: Vertical Integral of DOC Production, 0-100m (units: mmol/m^3␣
↪cm/s)
11. DOC_remin_zint: Vertical Integral of DOC Remineralization (units: mmol/m^3 cm/s)
12. DOC_remin_zint_100m: Vertical Integral of DOC Remineralization, 0-100m (units:␣
↪mmol/m^3 cm/s)
13. DOCr_remin_zint: Vertical Integral of DOCr Remineralization (units: mmol/m^3 cm/s)
14. DOCr_remin_zint_100m: Vertical Integral of DOCr Remineralization, 0-100m (units:␣
↪mmol/m^3 cm/s)
15. Jint_Ctot: Vertical Integral of Conservative Subterms of Source Sink Term for␣
↪Ctot (units: mmol/m^3 cm/s)
16. Jint_Ntot: Vertical Integral of Conservative Subterms of Source Sink Term for␣
↪Ntot (units: mmol/m^3 cm/s)
17. Jint_Ptot: Vertical Integral of Conservative Subterms of Source Sink Term for␣
↪Ptot (units: mmol/m^3 cm/s)
18. Jint_Sitot: Vertical Integral of Conservative Subterms of Source Sink Term for␣
↪Sitot (units: mmol/m^3 cm/s)
19. Jint_Fetot: Vertical Integral of Conservative Subterms of Source Sink Term for␣
↪Fetot (units: mmol/m^3 cm/s)
20. calcToFloor: CaCO3 Flux Hitting Sea Floor (units: nmol/cm^2/s)
21. calcToSed: CaCO3 Flux to Sediments (units: nmol/cm^2/s)
22. calcToSed_ALT_CO2: CaCO3 Flux to Sediments, Alternative CO2 (units: nmol/cm^2/s)
```

(continues on next page)

(continued from previous page)

```
23. pocToFloor: POC Flux Hitting Sea Floor (units: nmol/cm^2/s)
24. pocToSed: POC Flux to Sediments (units: nmol/cm^2/s)
25. ponToSed: nitrogen burial Flux to Sediments (units: nmol/cm^2/s)
26. SedDenitrif: nitrogen loss in Sediments (units: nmol/cm^2/s)
27. OtherRemin: non-oxic,non-dentr remin in Sediments (units: nmol/cm^2/s)
28. popToSed: phosphorus Flux to Sediments (units: nmol/cm^2/s)
29. bsiToSed: biogenic Si Flux to Sediments (units: nmol/cm^2/s)
30. dustToSed: dust Flux to Sediments (units: g/cm^2/s)
31. pfeToSed: pFe Flux to Sediments (units: nmol/cm^2/s)
32. sp_N_lim_surf: Small Phyto N Limitation, Surface (units: 1)
33. sp_N_lim_Cweight_avg_100m: Small Phyto N Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
34. sp_P_lim_surf: Small Phyto P Limitation, Surface (units: 1)
35. sp_P_lim_Cweight_avg_100m: Small Phyto P Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
36. sp_Fe_lim_surf: Small Phyto Fe Limitation, Surface (units: 1)
37. sp_Fe_lim_Cweight_avg_100m: Small Phyto Fe Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
38. sp_light_lim_surf: Small Phyto Light Limitation, Surface (units: 1)
39. sp_light_lim_Cweight_avg_100m: Small Phyto Light Limitation, carbon biomass␣
↪weighted average over 0-100m (units: 1)
40. photoC_sp_zint: Small Phyto C Fixation Vertical Integral (units: mmol/m^3 cm/s)
41. photoC_sp_zint_100m: Small Phyto C Fixation Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
42. photoC_NO3_sp_zint: Small Phyto C Fixation from NO3 Vertical Integral (units:␣
↪mmol/m^3 cm/s)
43. sp_CaCO3_form_zint: Small Phyto CaCO3 Formation Vertical Integral (units: mmol/m^
↪3 cm/s)
44. sp_CaCO3_form_zint_100m: Small Phyto CaCO3 Formation Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
45. graze_sp_zint: Small Phyto Grazing Vertical Integral (units: mmol/m^3 cm/s)
46. graze_sp_zint_100m: Small Phyto Grazing Vertical Integral, 0-100m (units: mmol/m^
↪3 cm/s)
47. graze_sp_poc_zint: Small Phyto Grazing to POC Vertical Integral (units: mmol/m^3␣
↪cm/s)
48. graze_sp_poc_zint_100m: Small Phyto Grazing to POC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
49. graze_sp_doc_zint: Small Phyto Grazing to DOC Vertical Integral (units: mmol/m^3␣
↪cm/s)
50. graze_sp_doc_zint_100m: Small Phyto Grazing to DOC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
51. graze_sp_zoo_zint: Small Phyto Grazing to ZOO Vertical Integral (units: mmol/m^3␣
↪cm/s)
52. graze_sp_zoo_zint_100m: Small Phyto Grazing to ZOO Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
53. sp_loss_zint: Small Phyto Loss Vertical Integral (units: mmol/m^3 cm/s)
54. sp_loss_zint_100m: Small Phyto Loss Vertical Integral, 0-100m (units: mmol/m^3 cm/
↪s)
55. sp_loss_poc_zint: Small Phyto Loss to POC Vertical Integral (units: mmol/m^3 cm/s)
56. sp_loss_poc_zint_100m: Small Phyto Loss to POC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
57. sp_loss_doc_zint: Small Phyto Loss to DOC Vertical Integral (units: mmol/m^3 cm/s)
58. sp_loss_doc_zint_100m: Small Phyto Loss to DOC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
59. sp_agg_zint: Small Phyto Aggregation Vertical Integral (units: mmol/m^3 cm/s)
60. sp_agg_zint_100m: Small Phyto Aggregation Vertical Integral, 0-100m (units: mmol/
↪m^3 cm/s)
```

(continues on next page)

(continued from previous page)

```
61. diat_N_lim_surf: Diatom N Limitation, Surface (units: 1)
62. diat_N_lim_Cweight_avg_100m: Diatom N Limitation, carbon biomass weighted average␣
↪over 0-100m (units: 1)
63. diat_P_lim_surf: Diatom P Limitation, Surface (units: 1)
64. diat_P_lim_Cweight_avg_100m: Diatom P Limitation, carbon biomass weighted average␣
↪over 0-100m (units: 1)
65. diat_Fe_lim_surf: Diatom Fe Limitation, Surface (units: 1)
66. diat_Fe_lim_Cweight_avg_100m: Diatom Fe Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
67. diat_SiO3_lim_surf: Diatom SiO3 Limitation, Surface (units: 1)
68. diat_SiO3_lim_Cweight_avg_100m: Diatom SiO3 Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
69. diat_light_lim_surf: Diatom Light Limitation, Surface (units: 1)
70. diat_light_lim_Cweight_avg_100m: Diatom Light Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
71. photoC_diat_zint: Diatom C Fixation Vertical Integral (units: mmol/m^3 cm/s)
72. photoC_diat_zint_100m: Diatom C Fixation Vertical Integral, 0-100m (units: mmol/m^
↪3 cm/s)
73. photoC_NO3_diat_zint: Diatom C Fixation from NO3 Vertical Integral (units: mmol/m^
↪3 cm/s)
74. graze_diat_zint: Diatom Grazing Vertical Integral (units: mmol/m^3 cm/s)
75. graze_diat_zint_100m: Diatom Grazing Vertical Integral, 0-100m (units: mmol/m^3␣
↪cm/s)
76. graze_diat_poc_zint: Diatom Grazing to POC Vertical Integral (units: mmol/m^3 cm/
↪s)
77. graze_diat_poc_zint_100m: Diatom Grazing to POC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
78. graze_diat_doc_zint: Diatom Grazing to DOC Vertical Integral (units: mmol/m^3 cm/
↪s)
79. graze_diat_doc_zint_100m: Diatom Grazing to DOC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
80. graze_diat_zoo_zint: Diatom Grazing to ZOO Vertical Integral (units: mmol/m^3 cm/
↪s)
81. graze_diat_zoo_zint_100m: Diatom Grazing to ZOO Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
82. diat_loss_zint: Diatom Loss Vertical Integral (units: mmol/m^3 cm/s)
83. diat_loss_zint_100m: Diatom Loss Vertical Integral, 0-100m (units: mmol/m^3 cm/s)
84. diat_loss_poc_zint: Diatom Loss to POC Vertical Integral (units: mmol/m^3 cm/s)
85. diat_loss_poc_zint_100m: Diatom Loss to POC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
86. diat_loss_doc_zint: Diatom Loss to DOC Vertical Integral (units: mmol/m^3 cm/s)
87. diat_loss_doc_zint_100m: Diatom Loss to DOC Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
88. diat_agg_zint: Diatom Aggregation Vertical Integral (units: mmol/m^3 cm/s)
89. diat_agg_zint_100m: Diatom Aggregation Vertical Integral, 0-100m (units: mmol/m^3␣
↪cm/s)
90. diaz_N_lim_surf: Diazotroph N Limitation, Surface (units: 1)
91. diaz_N_lim_Cweight_avg_100m: Diazotroph N Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
92. diaz_P_lim_surf: Diazotroph P Limitation, Surface (units: 1)
93. diaz_P_lim_Cweight_avg_100m: Diazotroph P Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
94. diaz_Fe_lim_surf: Diazotroph Fe Limitation, Surface (units: 1)
95. diaz_Fe_lim_Cweight_avg_100m: Diazotroph Fe Limitation, carbon biomass weighted␣
↪average over 0-100m (units: 1)
96. diaz_light_lim_surf: Diazotroph Light Limitation, Surface (units: 1)
97. diaz_light_lim_Cweight_avg_100m: Diazotroph Light Limitation, carbon biomass␣
↪weighted average over 0-100m (units: 1)
```

(continues on next page)

(continued from previous page)

```
98. photoC_diaz_zint: Diazotroph C Fixation Vertical Integral (units: mmol/m^3 cm/s)
99. photoC_diaz_zint_100m: Diazotroph C Fixation Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
100. photoC_NO3_diaz_zint: Diazotroph C Fixation from NO3 Vertical Integral (units:␣
↪mmol/m^3 cm/s)
101. graze_diaz_zint: Diazotroph Grazing Vertical Integral (units: mmol/m^3 cm/s)
102. graze_diaz_zint_100m: Diazotroph Grazing Vertical Integral, 0-100m (units: mmol/
↪m^3 cm/s)
103. graze_diaz_poc_zint: Diazotroph Grazing to POC Vertical Integral (units: mmol/m^
↪3 cm/s)
104. graze_diaz_poc_zint_100m: Diazotroph Grazing to POC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
105. graze_diaz_doc_zint: Diazotroph Grazing to DOC Vertical Integral (units: mmol/m^
↪3 cm/s)
106. graze_diaz_doc_zint_100m: Diazotroph Grazing to DOC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
107. graze_diaz_zoo_zint: Diazotroph Grazing to ZOO Vertical Integral (units: mmol/m^
↪3 cm/s)
108. graze_diaz_zoo_zint_100m: Diazotroph Grazing to ZOO Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
109. diaz_loss_zint: Diazotroph Loss Vertical Integral (units: mmol/m^3 cm/s)
110. diaz_loss_zint_100m: Diazotroph Loss Vertical Integral, 0-100m (units: mmol/m^3␣
↪cm/s)
111. diaz_loss_poc_zint: Diazotroph Loss to POC Vertical Integral (units: mmol/m^3 cm/
↪s)
112. diaz_loss_poc_zint_100m: Diazotroph Loss to POC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
113. diaz_loss_doc_zint: Diazotroph Loss to DOC Vertical Integral (units: mmol/m^3 cm/
↪s)
114. diaz_loss_doc_zint_100m: Diazotroph Loss to DOC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
115. diaz_agg_zint: Diazotroph Aggregation Vertical Integral (units: mmol/m^3 cm/s)
116. diaz_agg_zint_100m: Diazotroph Aggregation Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
117. CaCO3_form_zint: Total CaCO3 Formation Vertical Integral (units: mmol/m^3 cm/s)
118. CaCO3_form_zint_100m: Total CaCO3 Formation Vertical Integral, 0-100m (units:␣
↪mmol/m^3 cm/s)
119. zoo_loss_zint: Zooplankton Loss Vertical Integral (units: mmol/m^3 cm/s)
120. zoo_loss_zint_100m: Zooplankton Loss Vertical Integral, 0-100m (units: mmol/m^3␣
↪cm/s)
121. zoo_loss_poc_zint: Zooplankton Loss to POC Vertical Integral (units: mmol/m^3 cm/
↪s)
122. zoo_loss_poc_zint_100m: Zooplankton Loss to POC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
123. zoo_loss_doc_zint: Zooplankton Loss to DOC Vertical Integral (units: mmol/m^3 cm/
↪s)
124. zoo_loss_doc_zint_100m: Zooplankton Loss to DOC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
125. graze_zoo_zint: Zooplankton Grazing Vertical Integral (units: mmol/m^3 cm/s)
126. graze_zoo_zint_100m: Zooplankton Grazing Vertical Integral, 0-100m (units: mmol/
↪m^3 cm/s)
127. graze_zoo_poc_zint: Zooplankton Grazing to POC Vertical Integral (units: mmol/m^
↪3 cm/s)
128. graze_zoo_poc_zint_100m: Zooplankton Grazing to POC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
129. graze_zoo_doc_zint: Zooplankton Grazing to DOC Vertical Integral (units: mmol/m^
↪3 cm/s)
```

(continues on next page)

(continued from previous page)

```
130. graze_zoo_doc_zint_100m: Zooplankton Grazing to DOC Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
131. graze_zoo_zoo_zint: Zooplankton Grazing to ZOO Vertical Integral (units: mmol/m^
↪3 cm/s)
132. graze_zoo_zoo_zint_100m: Zooplankton Grazing to ZOO Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
133. x_graze_zoo_zint: Zooplankton Grazing Gain Vertical Integral (units: mmol/m^3 cm/
↪s)
134. x_graze_zoo_zint_100m: Zooplankton Grazing Gain Vertical Integral, 0-100m␣
↪(units: mmol/m^3 cm/s)
135. insitu_temp: in situ temperature (units: degC)
136. CO3: Carbonate Ion Concentration (units: mmol/m^3)
137. HCO3: Bicarbonate Ion Concentration (units: mmol/m^3)
138. H2CO3: Carbonic Acid Concentration (units: mmol/m^3)
139. pH_3D: pH (units: 1)
140. CO3_ALT_CO2: Carbonate Ion Concentration, Alternative CO2 (units: mmol/m^3)
141. HCO3_ALT_CO2: Bicarbonate Ion Concentration, Alternative CO2 (units: mmol/m^3)
142. H2CO3_ALT_CO2: Carbonic Acid Concentration, Alternative CO2 (units: mmol/m^3)
143. pH_3D_ALT_CO2: pH, Alternative CO2 (units: 1)
144. co3_sat_calc: CO3 concentration at calcite saturation (units: mmol/m^3)
145. co3_sat_arag: CO3 concentration at aragonite saturation (units: mmol/m^3)
146. NITRIF: Nitrification (units: mmol/m^3/s)
147. DENITRIF: Denitrification (units: mmol/m^3/s)
148. O2_PRODUCTION: O2 Production (units: mmol/m^3/s)
149. O2_CONSUMPTION: O2 Consumption (units: mmol/m^3/s)
150. AOU: Apparent O2 Utilization (units: mmol/m^3)
151. PAR_avg: PAR Average over Model Cell (units: W/m^2)
152. graze_auto_TOT: Total Autotroph Grazing (units: mmol/m^3/s)
153. photoC_TOT: Total C Fixation (units: mmol/m^3/s)
154. photoC_NO3_TOT: Total C Fixation from NO3 (units: mmol/m^3/s)
155. DOC_prod: DOC Production (units: mmol/m^3/s)
156. DOC_remin: DOC Remineralization (units: mmol/m^3/s)
157. DOCr_remin: DOCr Remineralization (units: mmol/m^3/s)
158. DON_prod: DON Production (units: mmol/m^3/s)
159. DON_remin: DON Remineralization (units: mmol/m^3/s)
160. DONr_remin: DONr Remineralization (units: mmol/m^3/s)
161. DOP_prod: DOP Production (units: mmol/m^3/s)
162. DOP_remin: DOP Remineralization (units: mmol/m^3/s)
163. DOPr_remin: DOPr Remineralization (units: mmol/m^3/s)
164. DOP_loss_P_bal: DOP loss, due to P budget balancing (units: mmol/m^3/s)
165. Fe_scavenge: Iron Scavenging (units: mmol/m^3/s)
166. Fe_scavenge_rate: Iron Scavenging Rate (units: 1/y)
167. Lig_prod: Production of Fe-binding Ligand (units: mmol/m^3/s)
168. Lig_loss: Loss of Fe-binding Ligand (units: mmol/m^3/s)
169. Lig_scavenge: Loss of Fe-binding Ligand from Scavenging (units: mmol/m^3/s)
170. Fefree: Fe not bound to Ligand (units: mmol/m^3)
171. Lig_photochem: Loss of Fe-binding Ligand from UV radiation (units: mmol/m^3/s)
172. Lig_deg: Loss of Fe-binding Ligand from Bacterial Degradation (units: mmol/m^3/s)
173. FESEDFLUX: Iron Sediment Flux (units: nmol/cm^2/s)
174. POC_FLUX_100m: POC Flux at 100m (units: mmol/m^3 cm/s)
175. POP_FLUX_100m: POP Flux at 100m (units: mmol/m^3 cm/s)
176. CaCO3_FLUX_100m: CaCO3 Flux at 100m (units: mmol/m^3 cm/s)
177. SiO2_FLUX_100m: SiO2 Flux at 100m (units: mmol/m^3 cm/s)
178. P_iron_FLUX_100m: P_iron Flux at 100m (units: mmol/m^3 cm/s)
179. POC_PROD_zint: Vertical Integral of POC Production (units: mmol/m^3 cm/s)
180. POC_PROD_zint_100m: Vertical Integral of POC Production, 0-100m (units: mmol/m^3␣
↪cm/s)
```

(continues on next page)

(continued from previous page)

```
181. POC_REMIN_DOCr_zint: Vertical Integral of POC Remineralization routed to DOCr␣
↪(units: mmol/m^3 cm/s)
182. POC_REMIN_DOCr_zint_100m: Vertical Integral of POC Remineralization routed to␣
↪DOCr, 0-100m (units: mmol/m^3 cm/s)
183. POC_REMIN_DIC_zint: Vertical Integral of POC Remineralization routed to DIC␣
↪(units: mmol/m^3 cm/s)
184. POC_REMIN_DIC_zint_100m: Vertical Integral of POC Remineralization routed to DIC,
↪ 0-100m (units: mmol/m^3 cm/s)
185. CaCO3_PROD_zint: Vertical Integral of CaCO3 Production (units: mmol/m^3 cm/s)
186. CaCO3_PROD_zint_100m: Vertical Integral of CaCO3 Production, 0-100m (units: mmol/
↪m^3 cm/s)
187. CaCO3_REMIN_zint: Vertical Integral of CaCO3 Remineralization (units: mmol/m^3␣
↪cm/s)
188. CaCO3_REMIN_zint_100m: Vertical Integral of CaCO3 Remineralization, 0-100m␣
↪(units: mmol/m^3 cm/s)
189. POC_FLUX_IN: POC Flux into Cell (units: mmol/m^3 cm/s)
190. POC_sFLUX_IN: POC sFlux into Cell (units: mmol/m^3 cm/s)
191. POC_hFLUX_IN: POC hFlux into Cell (units: mmol/m^3 cm/s)
192. POC_PROD: POC Production (units: mmol/m^3/s)
193. POC_REMIN_DOCr: POC Remineralization routed to DOCr (units: mmol/m^3/s)
194. POC_REMIN_DIC: POC Remineralization routed to DIC (units: mmol/m^3/s)
195. POP_FLUX_IN: POP Flux into Cell (units: mmol/m^3 cm/s)
196. POP_PROD: POP Production (units: mmol/m^3/s)
197. POP_REMIN_DOPr: POP Remineralization routed to DOPr (units: mmol/m^3/s)
198. POP_REMIN_PO4: POP Remineralization routed to PO4 (units: mmol/m^3/s)
199. PON_REMIN_DONr: PON Remineralization routed to DONr (units: mmol/m^3/s)
200. PON_REMIN_NH4: PON Remineralization routed to NH4 (units: mmol/m^3/s)
201. CaCO3_FLUX_IN: CaCO3 Flux into Cell (units: mmol/m^3 cm/s)
202. CaCO3_PROD: CaCO3 Production (units: mmol/m^3/s)
203. CaCO3_REMIN: CaCO3 Remineralization (units: mmol/m^3/s)
204. CaCO3_ALT_CO2_FLUX_IN: CaCO3 Flux into Cell, Alternative CO2 (units: mmol/m^3 cm/
↪s)
205. CaCO3_ALT_CO2_PROD: CaCO3 Production, Alternative CO2 (units: mmol/m^3/s)
206. CaCO3_ALT_CO2_REMIN: CaCO3 Remineralization, Alternative CO2 (units: mmol/m^3/s)
207. SiO2_FLUX_IN: SiO2 Flux into Cell (units: mmol/m^3 cm/s)
208. SiO2_PROD: SiO2 Production (units: mmol/m^3/s)
209. SiO2_REMIN: SiO2 Remineralization (units: mmol/m^3/s)
210. dust_FLUX_IN: Dust Flux into Cell (units: g/cm^2/s)
211. dust_REMIN: Dust Remineralization (units: g/cm^3/s)
212. P_iron_FLUX_IN: P_iron Flux into Cell (units: mmol/m^3 cm/s)
213. P_iron_PROD: P_iron Production (units: mmol/m^3/s)
214. P_iron_REMIN: P_iron Remineralization (units: mmol/m^3/s)
215. sp_Qp: Small Phyto P:C ratio (units: 1)
216. photoC_sp: Small Phyto C Fixation (units: mmol/m^3/s)
217. photoC_NO3_sp: Small Phyto C Fixation from NO3 (units: mmol/m^3/s)
218. photoFe_sp: Small Phyto Fe Uptake (units: mmol/m^3/s)
219. photoNO3_sp: Small Phyto NO3 Uptake (units: mmol/m^3/s)
220. photoNH4_sp: Small Phyto NH4 Uptake (units: mmol/m^3/s)
221. DOP_sp_uptake: Small Phyto DOP Uptake (units: mmol/m^3/s)
222. PO4_sp_uptake: Small Phyto PO4 Uptake (units: mmol/m^3/s)
223. graze_sp: Small Phyto Grazing (units: mmol/m^3/s)
224. graze_sp_poc: Small Phyto Grazing to POC (units: mmol/m^3/s)
225. graze_sp_doc: Small Phyto Grazing to DOC (units: mmol/m^3/s)
226. graze_sp_zootot: Small Phyto Grazing to ZOO TOT (units: mmol/m^3/s)
227. graze_sp_zoo: Small Phyto Grazing to Zooplankton (units: mmol/m^3/s)
228. sp_loss: Small Phyto Loss (units: mmol/m^3/s)
229. sp_loss_poc: Small Phyto Loss to POC (units: mmol/m^3/s)
```

(continues on next page)

(continued from previous page)

```
230. sp_loss_doc: Small Phyto Loss to DOC (units: mmol/m^3/s)
231. sp_agg: Small Phyto Aggregation (units: mmol/m^3/s)
232. sp_CaCO3_form: Small Phyto CaCO3 Formation (units: mmol/m^3/s)
233. diat_Qp: Diatom P:C ratio (units: 1)
234. photoC_diat: Diatom C Fixation (units: mmol/m^3/s)
235. photoC_NO3_diat: Diatom C Fixation from NO3 (units: mmol/m^3/s)
236. photoFe_diat: Diatom Fe Uptake (units: mmol/m^3/s)
237. photoNO3_diat: Diatom NO3 Uptake (units: mmol/m^3/s)
238. photoNH4_diat: Diatom NH4 Uptake (units: mmol/m^3/s)
239. DOP_diat_uptake: Diatom DOP Uptake (units: mmol/m^3/s)
240. PO4_diat_uptake: Diatom PO4 Uptake (units: mmol/m^3/s)
241. graze_diat: Diatom Grazing (units: mmol/m^3/s)
242. graze_diat_poc: Diatom Grazing to POC (units: mmol/m^3/s)
243. graze_diat_doc: Diatom Grazing to DOC (units: mmol/m^3/s)
244. graze_diat_zootot: Diatom Grazing to ZOO TOT (units: mmol/m^3/s)
245. graze_diat_zoo: Diatom Grazing to Zooplankton (units: mmol/m^3/s)
246. diat_loss: Diatom Loss (units: mmol/m^3/s)
247. diat_loss_poc: Diatom Loss to POC (units: mmol/m^3/s)
248. diat_loss_doc: Diatom Loss to DOC (units: mmol/m^3/s)
249. diat_agg: Diatom Aggregation (units: mmol/m^3/s)
250. diat_bSi_form: Diatom Si Uptake (units: mmol/m^3/s)
251. diaz_Qp: Diazotroph P:C ratio (units: 1)
252. photoC_diaz: Diazotroph C Fixation (units: mmol/m^3/s)
253. photoC_NO3_diaz: Diazotroph C Fixation from NO3 (units: mmol/m^3/s)
254. photoFe_diaz: Diazotroph Fe Uptake (units: mmol/m^3/s)
255. photoNO3_diaz: Diazotroph NO3 Uptake (units: mmol/m^3/s)
256. photoNH4_diaz: Diazotroph NH4 Uptake (units: mmol/m^3/s)
257. DOP_diaz_uptake: Diazotroph DOP Uptake (units: mmol/m^3/s)
258. PO4_diaz_uptake: Diazotroph PO4 Uptake (units: mmol/m^3/s)
259. graze_diaz: Diazotroph Grazing (units: mmol/m^3/s)
260. graze_diaz_poc: Diazotroph Grazing to POC (units: mmol/m^3/s)
261. graze_diaz_doc: Diazotroph Grazing to DOC (units: mmol/m^3/s)
262. graze_diaz_zootot: Diazotroph Grazing to ZOO TOT (units: mmol/m^3/s)
263. graze_diaz_zoo: Diazotroph Grazing to Zooplankton (units: mmol/m^3/s)
264. diaz_loss: Diazotroph Loss (units: mmol/m^3/s)
265. diaz_loss_poc: Diazotroph Loss to POC (units: mmol/m^3/s)
266. diaz_loss_doc: Diazotroph Loss to DOC (units: mmol/m^3/s)
267. diaz_agg: Diazotroph Aggregation (units: mmol/m^3/s)
268. diaz_Nfix: Diazotroph N Fixation (units: mmol/m^3/s)
269. bSi_form: Total Si Uptake (units: mmol/m^3/s)
270. CaCO3_form: Total CaCO3 Formation (units: mmol/m^3/s)
271. Nfix: Total N Fixation (units: mmol/m^3/s)
272. zoo_loss: Zooplankton Loss (units: mmol/m^3/s)
273. zoo_loss_poc: Zooplankton Loss to POC (units: mmol/m^3/s)
274. zoo_loss_doc: Zooplankton Loss to DOC (units: mmol/m^3/s)
275. graze_zoo: Zooplankton grazing loss (units: mmol/m^3/s)
276. graze_zoo_poc: Zooplankton grazing loss to POC (units: mmol/m^3/s)
277. graze_zoo_doc: Zooplankton grazing loss to DOC (units: mmol/m^3/s)
278. graze_zoo_zootot: Zooplankton grazing loss to ZOO TOT (units: mmol/m^3/s)
279. graze_zoo_zoo: Zooplankton grazing loss to Zooplankton (units: mmol/m^3/s)
280. x_graze_zoo: Zooplankton grazing gain (units: mmol/m^3/s)
281. PO4_RESTORE_TEND: Dissolved Inorganic Phosphate Restoring Tendency (units: mmol/
↪m^3/s)
282. NO3_RESTORE_TEND: Dissolved Inorganic Nitrate Restoring Tendency (units: mmol/m^
↪3/s)
283. SiO3_RESTORE_TEND: Dissolved Inorganic Silicate Restoring Tendency (units: mmol/
↪m^3/s)
```

(continues on next page)

(continued from previous page)

```
284. NH4_RESTORE_TEND: Dissolved Ammonia Restoring Tendency (units: mmol/m^3/s)
285. Fe_RESTORE_TEND: Dissolved Inorganic Iron Restoring Tendency (units: mmol/m^3/s)
286. Lig_RESTORE_TEND: Iron Binding Ligand Restoring Tendency (units: mmol/m^3/s)
287. O2_RESTORE_TEND: Dissolved Oxygen Restoring Tendency (units: mmol/m^3/s)
288. DIC_RESTORE_TEND: Dissolved Inorganic Carbon Restoring Tendency (units: mmol/m^3/
↪s)
289. DIC_ALT_CO2_RESTORE_TEND: Dissolved Inorganic Carbon, Alternative CO2 Restoring␣
↪Tendency (units: mmol/m^3/s)
290. ALK_RESTORE_TEND: Alkalinity Restoring Tendency (units: meq/m^3/s)
291. ALK_ALT_CO2_RESTORE_TEND: Alkalinity, Alternative CO2 Restoring Tendency (units:␣
↪meq/m^3/s)
292. DOC_RESTORE_TEND: Dissolved Organic Carbon Restoring Tendency (units: mmol/m^3/s)
293. DON_RESTORE_TEND: Dissolved Organic Nitrogen Restoring Tendency (units: mmol/m^3/
↪s)
294. DOP_RESTORE_TEND: Dissolved Organic Phosphorus Restoring Tendency (units: mmol/m^
↪3/s)
295. DOPr_RESTORE_TEND: Refractory DOP Restoring Tendency (units: mmol/m^3/s)
296. DONr_RESTORE_TEND: Refractory DON Restoring Tendency (units: mmol/m^3/s)
297. DOCr_RESTORE_TEND: Refractory DOC Restoring Tendency (units: mmol/m^3/s)
298. zooC_RESTORE_TEND: Zooplankton Carbon Restoring Tendency (units: mmol/m^3/s)
299. spChl_RESTORE_TEND: Small Phyto Chlorophyll Restoring Tendency (units: mg/m^3/s)
300. spC_RESTORE_TEND: Small Phyto Carbon Restoring Tendency (units: mmol/m^3/s)
301. spP_RESTORE_TEND: Small Phyto Phosphorus Restoring Tendency (units: mmol/m^3/s)
302. spFe_RESTORE_TEND: Small Phyto Iron Restoring Tendency (units: mmol/m^3/s)
303. spCaCO3_RESTORE_TEND: Small Phyto CaCO3 Restoring Tendency (units: mmol/m^3/s)
304. diatChl_RESTORE_TEND: Diatom Chlorophyll Restoring Tendency (units: mg/m^3/s)
305. diatC_RESTORE_TEND: Diatom Carbon Restoring Tendency (units: mmol/m^3/s)
306. diatP_RESTORE_TEND: Diatom Phosphorus Restoring Tendency (units: mmol/m^3/s)
307. diatFe_RESTORE_TEND: Diatom Iron Restoring Tendency (units: mmol/m^3/s)
308. diatSi_RESTORE_TEND: Diatom Silicon Restoring Tendency (units: mmol/m^3/s)
309. diazChl_RESTORE_TEND: Diazotroph Chlorophyll Restoring Tendency (units: mg/m^3/s)
310. diazC_RESTORE_TEND: Diazotroph Carbon Restoring Tendency (units: mmol/m^3/s)
311. diazP_RESTORE_TEND: Diazotroph Phosphorus Restoring Tendency (units: mmol/m^3/s)
312. diazFe_RESTORE_TEND: Diazotroph Iron Restoring Tendency (units: mmol/m^3/s)
```

The details are found in `$MARBL/tests/driver_src/marbl.F90`:

```
! Log surface flux diagnostics passed back to driver
associate(diags => marbl_instances(1)%surface_flux_diags%diags)
  call driver_status_log%log_header('Surface flux diagnostics', subname)
  do n=1, size(diags)
    write(log_message, "(I0,7A)") n, '. ', trim(diags(n)%short_name), ': ',␣
↪trim(diags(n)%long_name), &
                                  ' (units: ', trim(diags(n)%units),')'
    call driver_status_log%log_noerror(log_message, subname)
  end do
end associate
! Log interior tendency diagnostics passed back to driver
associate(diags => marbl_instances(1)%interior_tendency_diags%diags)
  call driver_status_log%log_header('Interior tendency diagnostics', subname)
  do n=1, size(diags)
    write(log_message, "(I0,7A)") n, '. ', trim(diags(n)%short_name), ': ',␣
↪trim(diags(n)%long_name), &
                                  ' (units: ', trim(diags(n)%units),')'
    call driver_status_log%log_noerror(log_message, subname)
  end do
end associate
```

The marbl_interface_class contains two objects (surface_flux_diags and interior_tendency_diags). Both are of type marbl_diagnostics_type, and contain an array named diags. The length of this array is equal to the number of surface flux or interior tendency diagnostics. This marbl_single_diagnostic_type contains the data and metadata for each diagnostic:

```
type, private :: marbl_single_diagnostic_type
   ! marbl_single_diagnostic :
   ! a private type, this contains both the metadata
   ! and the actual diagnostic data for a single
   ! diagnostic quantity. Data must be accessed via
   ! the marbl_diagnostics_type data structure.
   character (len=char_len)                    :: long_name
   character (len=char_len)                    :: short_name
   character (len=char_len)                    :: units
   character (len=char_len)                    :: vertical_grid ! 'none', 'layer_avg',
↪ 'layer_iface'
   logical   (log_kind)                        :: compute_now
   logical   (log_kind)                        :: ltruncated_vertical_extent
   integer   (int_kind)                        :: ref_depth ! depth that diagnostic␣
↪nominally resides at
   real      (r8), allocatable, dimension(:)   :: field_2d
   real      (r8), allocatable, dimension(:,:) :: field_3d

 contains
   procedure :: initialize  => marbl_single_diag_init
end type marbl_single_diagnostic_type

!*****************************************************************************

type, public :: marbl_diagnostics_type
   ! marbl_diagnostics :
   ! used to pass diagnostic information from marbl back to
   ! the driver.
   integer :: num_elements
   integer :: num_levels
   type(marbl_single_diagnostic_type), dimension(:), pointer :: diags

 contains
   procedure, public :: construct      => marbl_diagnostics_constructor
   procedure, public :: set_to_zero    => marbl_diagnostics_set_to_zero
   procedure, public :: add_diagnostic => marbl_diagnostics_add
   procedure, public :: deconstruct    => marbl_diagnostics_deconstructor
end type marbl_diagnostics_type
```

The short name of the diagnostic is the recommended name for the variable if writing a netCDF output file. The long name is a more descriptive name and, as the example output shows, the units are also included in the metadata.

4.2.2 MARBL examples in POP

POP Interacting with MARBL Tracers

POP will read the initial state for each tracers and store the data in a global array (dimensions nx by ny by nz by n_tracers). To know what tracer to read into each n_tracers index, POP copies the information from marbl_instace%tracer_metadata to a local type during initialization. The data type is defined as

```
type :: tracer_field
   character(char_len) :: short_name
   character(char_len) :: long_name
   character(char_len) :: units
   character(char_len) :: tend_units
   character(char_len) :: flux_units
   real(r8)            :: scale_factor
   logical :: lfull_depth_tavg
end type
```

An array of the above type is then populated; note that MARBL expects the GCM to apply any necessary scale factor so the POP datatype is set to 1.

```
! Initialize tracer_d_module input argument (needed before reading
! tracers from restart file)
do n = 1, ecosys_tracer_cnt
   tracer_d_module(n)%short_name       = marbl_instances(1)%tracer_metadata(n)%short_
↪name
   tracer_d_module(n)%long_name        = marbl_instances(1)%tracer_metadata(n)%long_
↪name
   tracer_d_module(n)%units            = marbl_instances(1)%tracer_metadata(n)%units
   tracer_d_module(n)%tend_units       = marbl_instances(1)%tracer_metadata(n)%tend_
↪units
   tracer_d_module(n)%flux_units       = marbl_instances(1)%tracer_metadata(n)%flux_
↪units
   tracer_d_module(n)%lfull_depth_tavg = marbl_instances(1)%tracer_metadata(n)%lfull_
↪depth_tavg
   tracer_d_module(n)%scale_factor     = c1
end do
```

POP combines the `tracer_d_module` object with information regarding what files contain tracer initial conditions (and what the netCDF variable name corresponds to each tracer) to properly initialize each tracer.

Reading Tracer Initial Conditions

All tracer initial conditions are read from a file. If the run is a continuation run, the initial tracer values are found in a restart file. Otherwise they are read from an initial condition.

POP has a specific data type to manage the metadata of a tracer it is reading from a file.

```
!-----------------------------------------------------------------------
!  derived type for reading tracers from a file
!-----------------------------------------------------------------------

   type, public :: tracer_read
      character(char_len) :: mod_varname, filename, file_varname, file_fmt
      real(r8) :: scale_factor, default_val
   end type
```

Metadata such as the tracer name and the name of the tracer as it appears in the file is copied from `tracer_d_module` into `tracer_inputs` (an array of type `tracer_read`). The rest of `tracer_inputs` (file name, file format, etc) is also set and then each tracer state is read into the correct index of the tracer array by looping over `tracer_inputs`.

POP Interacting with MARBL Requested Forcings

POP mirrors the MARBL datatypes for forcing fields and the associated metadata, but expands the metadata class to also manage the source of the data (read from a file, provided by POP, provided by the flux coupler, etc). In the code below, `marbl_req_surface_flux_forcings(:)` is the MARBL data provided through the interface, and `surface_flux_forcings(:)` is the copy into the POP datatype.

```
do n=1,size(surface_flux_forcings)
  marbl_varname = marbl_req_surface_flux_forcings(n)%metadata%varname
  units         = marbl_req_surface_flux_forcings(n)%metadata%field_units
  select case (trim(marbl_req_surface_flux_forcings(n)%metadata%varname))
    case ('d13c')
      d13c_ind = n
      call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
           marbl_varname=marbl_varname, field_units=units,                      &
           driver_varname='d13C', rank=2, id=n)

    case ('d14c')
      d14c_ind = n
      call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
           marbl_varname=marbl_varname, field_units=units,                      &
           driver_varname='D14C', rank=2, id=n)

    case ('u10_sqr')
      u10sqr_ind = n
      call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
           marbl_varname=marbl_varname, field_units=units,                      &
           driver_varname='U10_SQR', rank=2, id=n)

    case ('sst')
      sst_ind = n
      call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
           marbl_varname=marbl_varname, field_units=units,                      &
           driver_varname='SST', rank=2, id=n)

    case ('sss')
      sss_ind = n
      call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
           marbl_varname=marbl_varname, field_units=units,                      &
           driver_varname='SSS', rank=2, id=n)

    case ('xco2')
      if (trim(atm_co2_opt).eq.'const') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='const', &
             marbl_varname=marbl_varname, field_units=units,                   &
             field_constant=atm_co2_const, rank=2, id=n)
      else if (trim(atm_co2_opt).eq.'drv_prog') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='named_field', &
             marbl_varname=marbl_varname, field_units=units,                        ⌐
↪    &
             named_field='ATM_CO2_PROG', rank=2, id=n)
      else if (trim(atm_co2_opt).eq.'drv_diag') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='named_field', &
             marbl_varname=marbl_varname, field_units=units,                        ⌐
↪    &
             named_field='ATM_CO2_DIAG', rank=2, id=n)
      else if (trim(atm_co2_opt).eq.'box_atm_co2') then
        box_atm_co2_ind = n
```

(continues on next page)

(continued from previous page)

```
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,                      ␣
↪ &
             driver_varname='box_atm_co2', rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(atm_co2_opt),                    &
             'is not a valid option for atm_co2_opt'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('xco2_alt_co2')
      if (trim(atm_alt_co2_opt).eq.'const') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='const', &
             marbl_varname=marbl_varname, field_units=units,                      &
             field_constant=atm_alt_co2_const, rank=2, id=n)
      else if (trim(atm_alt_co2_opt).eq.'box_atm_co2') then
        if (trim(atm_co2_opt).eq.'box_atm_co2') then
          box_atm_co2_dup_ind = n
        else
          box_atm_co2_ind = n
        end if
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,                      ␣
↪ &
             driver_varname='box_atm_co2', rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(atm_alt_co2_opt),                &
             'is not a valid option for atm_alt_co2_opt'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('Ice Fraction')
      ifrac_ind = n
      if (trim(gas_flux_forcing_opt).eq.'drv') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,                      ␣
↪ &
             driver_varname='ICE Fraction', rank=2, id=n)
      else if (trim(gas_flux_forcing_opt).eq.'file') then
        file_details => fice_file_loc
        call init_monthly_surface_flux_forcing_metadata(file_details)
        call surface_flux_forcings(n)%add_forcing_field(                    &
             field_source='POP monthly calendar',                                &
             marbl_varname=marbl_varname, field_units=units,                     &
             forcing_calendar_name=file_details, rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(gas_flux_forcing_opt),           &
             'is not a valid option for gas_flux_forcing_opt'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('Atmospheric Pressure')
      ap_ind = n
      if (trim(gas_flux_forcing_opt).eq.'drv') then
```

(continues on next page)

(continued from previous page)

```
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,
↪ &
             driver_varname='AP_FILE_INPUT', rank=2, id=n)
      else if (trim(gas_flux_forcing_opt).eq.'file') then
        file_details => ap_file_loc
        call init_monthly_surface_flux_forcing_metadata(file_details)
        call surface_flux_forcings(n)%add_forcing_field(                    &
             field_source='POP monthly calendar',                           &
             marbl_varname=marbl_varname, field_units=units,                &
                            forcing_calendar_name=file_details, rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(gas_flux_forcing_opt),              &
             'is not a valid option for gas_flux_forcing_opt'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('Dust Flux')
      dust_dep_ind = n
      if (trim(dust_flux_source).eq.'driver') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,
↪ &
             driver_varname='DUST_FLUX', rank=2, id=n)
      else if (trim(dust_flux_source).eq.'monthly-calendar') then
        file_details => dust_flux_file_loc
        call init_monthly_surface_flux_forcing_metadata(file_details)
        call surface_flux_forcings(n)%add_forcing_field(                    &
             field_source='POP monthly calendar',                           &
             marbl_varname=marbl_varname, field_units=units,                &
             forcing_calendar_name=file_details, rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(dust_flux_source),                  &
             'is not a valid option for dust_flux_source'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('Iron Flux')
      if (trim(iron_flux_source).eq.'driver-derived') then
        bc_dep_ind = n
        call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
             marbl_varname=marbl_varname, field_units=units,
↪ &
             driver_varname='BLACK_CARBON_FLUX', rank=2, id=n)
      else if (trim(iron_flux_source).eq.'monthly-calendar') then
        Fe_dep_ind = n
        file_details => iron_flux_file_loc
        call init_monthly_surface_flux_forcing_metadata(file_details)
        call surface_flux_forcings(n)%add_forcing_field(                    &
             field_source='POP monthly calendar',                           &
             marbl_varname=marbl_varname, field_units=units,                &
             forcing_calendar_name=file_details, rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(iron_flux_source),                  &
             'is not a valid option for iron_flux_source'
```

(continues on next page)

(continued from previous page)

```
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('NOx Flux')
      if (trim(ndep_data_type).eq.'shr_stream') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='shr_stream', &
             strdata_inputlist_ptr=surface_strdata_inputlist_ptr,
↪    &
             marbl_varname=marbl_varname, field_units=units,
↪    &
             unit_conv_factor=ndep_shr_stream_scale_factor,
↪    &
             file_varname='NOy_deposition',
↪    &
             year_first = ndep_shr_stream_year_first,
↪    &
             year_last = ndep_shr_stream_year_last,
↪    &
             year_align = ndep_shr_stream_year_align,
↪    &
             filename = ndep_shr_stream_file,
↪    &
             rank = 2, id = n)
      else if (trim(ndep_data_type).eq.'monthly-calendar') then
        file_details => nox_flux_monthly_file_loc
        call init_monthly_surface_flux_forcing_metadata(file_details)
        call surface_flux_forcings(n)%add_forcing_field(                    &
             field_source='POP monthly calendar',                           &
             marbl_varname=marbl_varname, field_units=units,                &
             forcing_calendar_name=file_details, rank=2, id=n)
      else if (trim(ndep_data_type).eq.'driver') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='named_field', &
             marbl_varname=marbl_varname, field_units=units,
↪    &
             named_field='ATM_NOy', rank=2, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(ndep_data_type),                    &
             'is not a valid option for ndep_data_type'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('NHy Flux')
      if (trim(ndep_data_type).eq.'shr_stream') then
        call surface_flux_forcings(n)%add_forcing_field(field_source='shr_stream', &
             strdata_inputlist_ptr=surface_strdata_inputlist_ptr,
↪    &
             marbl_varname=marbl_varname, field_units=units,
↪    &
             unit_conv_factor=ndep_shr_stream_scale_factor,
↪    &
             file_varname='NHx_deposition',
↪    &
             year_first = ndep_shr_stream_year_first,
↪    &
             year_last = ndep_shr_stream_year_last,
↪    &
```

(continues on next page)

(continued from previous page)

```
              year_align = ndep_shr_stream_year_align,                                 ␣
↪    &
              filename = ndep_shr_stream_file,                                         ␣
↪    &
              rank = 2, id = n)
       else if (trim(ndep_data_type).eq.'monthly-calendar') then
         file_details => nhy_flux_monthly_file_loc
         call init_monthly_surface_flux_forcing_metadata(file_details)
         call surface_flux_forcings(n)%add_forcing_field(                    &
              field_source='POP monthly calendar',                                &
              marbl_varname=marbl_varname, field_units=units,                     &
              forcing_calendar_name=file_details, rank=2, id=n)
       else if (trim(ndep_data_type).eq.'driver') then
         call surface_flux_forcings(n)%add_forcing_field(field_source='named_field', &
              marbl_varname=marbl_varname, field_units=units,                          ␣
↪    &
              named_field='ATM_NHx', rank=2, id=n)
       else
         write(err_msg, "(A,1X,A)") trim(ndep_data_type),                  &
              'is not a valid option for ndep_data_type'
         call document(subname, err_msg)
         call exit_POP(sigAbort, 'Stopping in ' // subname)
       end if

     case ('external C Flux')
       ext_C_flux_ind = n
       call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
            marbl_varname=marbl_varname, field_units=units,                        &
            driver_varname='ext_C_flux', rank=2, id=n)

     case ('external P Flux')
       ext_P_flux_ind = n
       call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
            marbl_varname=marbl_varname, field_units=units,                        &
            driver_varname='ext_P_flux', rank=2, id=n)

     case ('external Si Flux')
       ext_Si_flux_ind = n
       call surface_flux_forcings(n)%add_forcing_field(field_source='internal', &
            marbl_varname=marbl_varname, field_units=units,                        &
            driver_varname='ext_Si_flux', rank=2, id=n)

     case DEFAULT
       write(err_msg, "(A,1X,A)") trim(marbl_req_surface_flux_forcings(n)%metadata
↪%varname), &
                     'is not a valid surface flux forcing field name.'
       call document(subname, err_msg)
       call exit_POP(sigAbort, 'Stopping in ' // subname)
   end select
end do
```

Note that POP uses `field_source` to denote where it will be getting the forcing field. Not shown in this example is where POP actually populates the data. The code for interior forcing fields looks similar, although there are far fewer fields to handle and that results in a shorter code snippet. Again, `marbl_req_interior_tendency_forcings` is provided through the MARBL interface and `interior_tendency_forcings` is a POP construct.

```
do n=1,size(interior_tendency_forcings)
  marbl_varname = marbl_req_interior_tendency_forcings(n)%metadata%varname
  units = marbl_req_interior_tendency_forcings(n)%metadata%field_units

  var_processed = .false.
  ! Check to see if this forcing field is tracer restoring
  if (index(marbl_varname,'Restoring Field').gt.0) then
    tracer_name = trim(marbl_varname(1:scan(marbl_varname,' ')))
    do m=1,marbl_tracer_cnt
      if (trim(tracer_name).eq.trim(restorable_tracer_names(m))) then
        ! Check to make sure restore_data_filenames and
        ! restore_data_file_varnames have both been provided by namelist
        if (len_trim(restore_data_filenames(m)).eq.0) then
          write(err_msg, "(3A)") "No file provided to read restoring ",    &
                                 "field for ", trim(tracer_name)
          call document(subname, err_msg)
          call exit_POP(sigAbort, 'Stopping in ' // subname)
        end if
        if (len_trim(restore_data_file_varnames(m)).eq.0) then
          write(err_msg, "(3A)") "No variable name provided to read ",     &
                                 "restoring field for ", trim(tracer_name)
          call document(subname, err_msg)
          call exit_POP(sigAbort, 'Stopping in ' // subname)
        end if
        if (my_task.eq.master_task) then
          write(stdout, "(6A)") "Will restore ", trim(tracer_name),        &
                          " with ", trim(restore_data_file_varnames(m)),   &
                          " from ", trim(restore_data_filenames(m))
        end if
        call interior_tendency_forcings(n)%add_forcing_field(                 &
                   field_source='shr_stream',                                 &
                   strdata_inputlist_ptr=interior_strdata_inputlist_ptr,      &
                   marbl_varname=marbl_varname, field_units=units,            &
                   filename=restore_data_filenames(m),                        &
                   file_varname=restore_data_file_varnames(m),                &
                   year_first=restore_year_first(m),                          &
                   year_last=restore_year_last(m),                            &
                   year_align=restore_year_align(m),                          &
                   unit_conv_factor=restore_scale_factor(m),                  &
                   rank=3, dim3_len=km, id=n)
        var_processed = .true.
        exit
      end if
    end do
  end if

  ! Check to see if this forcing field is a restoring time scale
  if (index(marbl_varname,'Restoring Inverse Timescale').gt.0) then
    select case (trim(restore_inv_tau_opt))
      case('const')
        call interior_tendency_forcings(n)%add_forcing_field(                 &
                   field_source='const',                                      &
                   marbl_varname=marbl_varname, field_units=units,            &
                   field_constant=restore_inv_tau_const,                      &
                   rank=3, dim3_len=km, id=n)
      case('file_time_invariant')
        call interior_tendency_forcings(n)%add_forcing_field(                 &
```

(continues on next page)

(continued from previous page)

```
                    field_source='file_time_invariant',                    &
                    marbl_varname=marbl_varname, field_units=units,        &
                    filename=restore_inv_tau_input%filename,               &
                    file_varname=restore_inv_tau_input%file_varname,       &
                    unit_conv_factor=restore_inv_tau_input%scale_factor,   &
                    rank=3, dim3_len=km, id=n)
      case DEFAULT
        write(err_msg, "(A,1X,A)") trim(restore_inv_tau_opt),              &
             'is not a valid option for restore_inv_tau_opt'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
    end select
    var_processed = .true.
  end if

  if (.not.var_processed) then
    select case (trim(marbl_req_interior_tendency_forcings(n)%metadata%varname))
      case ('Dust Flux')
        dustflux_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪&
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='dust_flux', rank=2, id=n)
      case ('PAR Column Fraction')
        PAR_col_frac_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪&
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='PAR_col_frac', rank=3, dim3_len=mcog_nbins,  &
                    ldim3_is_depth=.false., id=n)
      case ('Surface Shortwave')
        surf_shortwave_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪ &
                    marbl_varname=marbl_varname, field_units=units,               &
                    driver_varname='surf_shortwave', rank=3, dim3_len=mcog_nbins, &
                    ldim3_is_depth=.false., id=n)
      case ('Potential Temperature')
        potemp_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪&
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='temperature', rank=3, dim3_len=km, id=n)
      case ('Salinity')
        salinity_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪&
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='salinity', rank=3, dim3_len=km, id=n)
      case ('Pressure')
        pressure_ind = n
        call interior_tendency_forcings(n)%add_forcing_field(field_source='internal',␣
↪&
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='pressure', rank=3, dim3_len=km, id=n)
      case ('Iron Sediment Flux')
        fesedflux_ind = n
```

(continues on next page)

(continued from previous page)

```
        call interior_tendency_forcings(n)%add_forcing_field(                &
                      field_source='file_time_invariant',                    &
                      marbl_varname=marbl_varname, field_units=units,        &
                      filename=fesedflux_input%filename,                     &
                      file_varname=fesedflux_input%file_varname,             &
                      unit_conv_factor=fesedflux_input%scale_factor,         &
                      rank=3, dim3_len=km, id=n)
      case ('O2 Consumption Scale Factor')
        select case (trim(o2_consumption_scalef_opt))
        case ('const')
          call interior_tendency_forcings(n)%add_forcing_field(field_source='const', &
               marbl_varname=marbl_varname, field_units=units, &
               field_constant=o2_consumption_scalef_const, rank=3, dim3_len=km, id=n)
        case ('file_time_invariant')
          call interior_tendency_forcings(n)%add_forcing_field( &
               field_source='file_time_invariant', &
               marbl_varname=marbl_varname, field_units=units, &
               filename=o2_consumption_scalef_input%filename, &
               file_varname=o2_consumption_scalef_input%file_varname, &
               unit_conv_factor=o2_consumption_scalef_input%scale_factor, &
               rank=3, dim3_len=km, id=n)
        case default
          call document(subname, 'unknown o2_consumption_scalef_opt', o2_consumption_
↪scalef_opt)
          call exit_POP(sigAbort, 'Stopping in ' // subname)
        end select
      case ('Particulate Remin Scale Factor')
        select case (trim(p_remin_scalef_opt))
        case ('const')
          call interior_tendency_forcings(n)%add_forcing_field(field_source='const', &
               marbl_varname=marbl_varname, field_units=units, &
               field_constant=p_remin_scalef_const, rank=3, dim3_len=km, id=n)
        case ('file_time_invariant')
          call interior_tendency_forcings(n)%add_forcing_field( &
               field_source='file_time_invariant', &
               marbl_varname=marbl_varname, field_units=units, &
               filename=p_remin_scalef_input%filename, &
               file_varname=p_remin_scalef_input%file_varname, &
               unit_conv_factor=p_remin_scalef_input%scale_factor, &
               rank=3, dim3_len=km, id=n)
        case default
          call document(subname, 'unknown p_remin_scalef_opt', p_remin_scalef_opt)
          call exit_POP(sigAbort, 'Stopping in ' // subname)
        end select
      case DEFAULT
        write(err_msg, "(A,1X,A)") trim(marbl_req_interior_tendency_forcings(n)
↪%metadata%varname), &
                     'is not a valid interior tendency forcing field name.'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
    end select
  end if

end do ! do n=1,size(interior_tendency_forcings)
```

Processing MARBL Output

Surface Flux Output

After calling `surface_flux_compute()`, the GCM needs to *manage all the MARBL output*. The block of code below shows how POP accesses `surface_flux_saved_state`, `surface_flux_output`, `surface_fluxes`, `surface_flux_diags`, and `glo_avg_fields_surface_flux`. Note that POP aborts if any values in `surface_fluxes` are `NaN`.

```
!-----------------------------------------------------------------------
! Copy data from marbl output column to pop slab data structure
!-----------------------------------------------------------------------

do index_marbl = 1, marbl_col_cnt(iblock)
   i = marbl_col_to_pop_i(index_marbl,iblock)
   j = marbl_col_to_pop_j(index_marbl,iblock)

   do n=1,size(surface_flux_saved_state)
     surface_flux_saved_state(n)%field_2d(i,j,iblock) = &
       marbl_instances(iblock)%surface_flux_saved_state%state(n)%field_2d(index_marbl)
   end do

   do n=1,sfo_cnt
     surface_flux_outputs(i,j,iblock,n) = &
        marbl_instances(iblock)%surface_flux_output%sfo(n)%forcing_field(index_marbl)
   end do

   !-------------------------------------------------------------
   ! before copying surface fluxes, check to see if any are NaNs
   !-------------------------------------------------------------

   if (any(shr_infnan_isnan(marbl_instances(iblock)%surface_fluxes(index_marbl,:)))␣
↪then
      write(stdout, *) subname, ': NaN in stf_module, (i,j)=(', &
         this_block%i_glob(i), ',', this_block%j_glob(j), ')'
      write(stdout, *) '(lon,lat)=(', TLOND(i,j,iblock), ',', TLATD(i,j,iblock), ')'
      do n = 1, ecosys_tracer_cnt
         write(stdout, *) trim(marbl_instances(1)%tracer_metadata(n)%short_name), ' ',
↪ &
            marbl_instances(iblock)%tracers_at_surface(index_marbl,n), ' ', &
            marbl_instances(iblock)%surface_fluxes(index_marbl,n)
      end do
      do n = 1, size(surface_flux_forcings)
         associate (forcing_field => surface_flux_forcings(n))
            write(stdout, *) trim(forcing_field%metadata%marbl_varname)
            if (forcing_field%rank == 2) then
               write(stdout, *) forcing_field%field_0d(i,j,iblock)
            else
               write(stdout, *) forcing_field%field_1d(i,j,:,iblock)
            end if
         end associate
      end do
      call exit_POP(sigAbort, 'Stopping in ' // subname)
   end if

   do n = 1,ecosys_tracer_cnt
      stf_module(i,j,n) = &
```

(continues on next page)

(continued from previous page)

```
            marbl_instances(iblock)%surface_fluxes(index_marbl,n)
   end do

   do n=1,size(marbl_instances(1)%surface_flux_diags%diags)
      surface_flux_diags(i,j,n,iblock) = &
            marbl_instances(iblock)%surface_flux_diags%diags(n)%field_2d(index_marbl)
   end do

   ! copy values to be used in computing requested global averages
   ! arrays have zero extent if none are requested
   glo_avg_fields_surface(i,j,iblock,:) = marbl_instances(iblock)%glo_avg_fields_
↪surface_flux(index_marbl,:)
end do
```

Interior Tendency Output

After calling `interior_tendency_compute()`, the GCM needs to *manage all the MARBL output*. The block of code below shows how POP accesses `interior_tendency_saved_state`, `interior_tendencies` and `glo_avg_fields_interior_tendency`. Diagnostics are accumulated in `ecosys_tavg_accumulate_interior()`, which is a wrapper to handle column-wise accumulation (POP typically accumulates level by level). Note that POP aborts if any values in `interior_tendencies` are `NaN`.

```
do c = this_block%jb,this_block%je
   do i = this_block%ib,this_block%ie

      if (land_mask(i,c,bid)) then

         ! [snipped prestage calls]

         !-----------------------------------------------------------
         ! copy marbl column data back to slab
         !-----------------------------------------------------------

         call timer_start(ecosys_interior_marbl_to_pop, block_id=bid)

         do n=1,size(interior_tendency_saved_state)
           interior_tendency_saved_state(n)%field_3d(:,i,c,bid) =                &
             marbl_instances(bid)%interior_tendency_saved_state%state(n)%field_3d(:,1)
         end do

         !-----------------------------------------------------------
         ! before copying tendencies, check to see if any are NaNs
         !-----------------------------------------------------------

         do k = 1, KMT(i, c, bid)
            if (any(shr_infnan_isnan(marbl_instances(bid)%interior_tendencies(:,␣
↪k)))) then
               write(stdout, *) subname, ': NaN in dtracer_module, (i,j,k)=(', &
                  this_block%i_glob(i), ',', this_block%j_glob(c), ',', k, ')'
               write(stdout, *) '(lon,lat)=(', TLOND(i,c,bid), ',', TLATD(i,c,bid), ')
↪'
               do n = 1, ecosys_tracer_cnt
                  write(stdout, *) trim(marbl_instances(1)%tracer_metadata(n)%short_
↪name), ' ', &
```

(continues on next page)

(continued from previous page)

```
                    marbl_instances(bid)%tracers(n, k), ' ', &
                    marbl_instances(bid)%interior_tendencies(n, k)
             end do
             do n = 1, size(interior_tendency_forcings)
                associate (forcing_field => interior_tendency_forcings(n))
                   write(stdout, *) trim(forcing_field%metadata%marbl_varname)
                   if (forcing_field%rank == 2) then
                      write(stdout, *) forcing_field%field_0d(i,c,bid)
                   else
                      if (forcing_field%ldim3_is_depth) then
                         write(stdout, *) forcing_field%field_1d(i,c,k,bid)
                      else
                         write(stdout, *) forcing_field%field_1d(i,c,:,bid)
                      end if
                   end if
                end associate
             end do
             call exit_POP(sigAbort, 'Stopping in ' // subname)
          end if
       end do

       do n = 1, ecosys_tracer_cnt
          dtracer_module(i, c, 1:KMT(i, c, bid), n) = marbl_instances(bid)%interior_
↪tendencies(n, 1:KMT(i, c, bid))
       end do

       ! copy values to be used in computing requested global averages
       ! arrays have zero extent if none are requested
       glo_avg_fields_interior(i, c, bid, :) = marbl_instances(bid)%glo_avg_fields_
↪interior_tendency(:)
       call timer_stop(ecosys_interior_marbl_to_pop, block_id=bid)

       !-----------------------------------------------------------
       ! Update pop tavg diags
       !-----------------------------------------------------------

       call timer_start(ecosys_interior_marbl_tavg, block_id=bid)

       call ecosys_tavg_accumulate_interior(i, c, marbl_instances(bid), bid)

       call timer_stop(ecosys_interior_marbl_tavg, block_id=bid)

    end if ! end if land_mask > 0

 end do ! do i
end do ! do c
```

4.3 MARBL developer's guide

This document provides technical documentation of the MARBL code.

Contributing to MARBL

- *Guidelines for participation*
- *Developing code*
- *Working on the documentation*

4.3.1 Introduction to MARBL framework

Tracer equation view of biogeochemistry

MARBL is designed to be a modular implementation of ocean biogoechemistry suitable for coupling to ocean general circulation models (OGCM). In the OGCM context, the prognostic equation governing the evolution of an arbitrary tracer χ in the ocean is

$$\frac{\partial \chi}{\partial t} + \nabla \cdot (\mathbf{u}\chi) - \nabla \cdot (K \cdot \nabla \chi) = B_\chi(\mathbf{x}) \tag{4.1}$$

where and $B_\chi(\mathbf{x})$ is the sum of sources minus sinks for χ, computed as a function of the model state vector, $\mathbf{x}$.

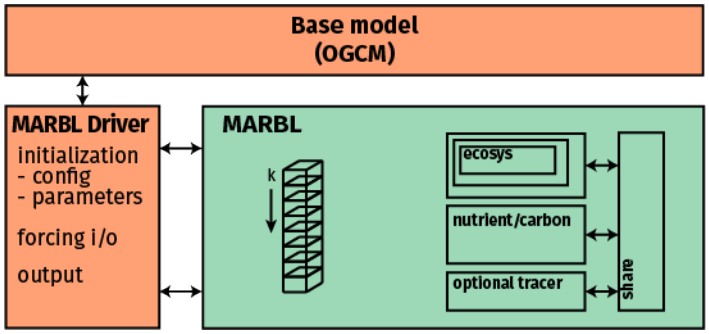
The OGCM computes the lefthand-side of (4.1) (time tendency, advection, diffusion); MARBL's role is to compute the righthand-side ($B_\chi(\mathbf{x})$). MARBL returns this tendency to the OGCM, which steps (4.1) forward in time. MARBL also computes air-sea fluxes for constituents like CO_2; the OGCM is responsible for handling these in a manner consistent with its numerics. MARBL also returns diagnostic output, including intermediate terms used in the computation of $B_\chi(\mathbf{x})$, such as net primary productivity or grazing of phytoplankton. The OGCM must compute diagnostics relating to the total tracer concentration and terms from the righthand-side of (4.1).

Implementation overview

MARBL is compiled as a standalone library with no explicit dependencies on aspects of the OGCM code. The OGCM includes a *MARBL driver*, which is responsible for all communication with MARBL making use of MARBL's interface layer.

MARBL is configured to operate on vertical columns. The OGCM passes data into MARBL on these columns, including information describing the domain (i.e., *dz* or the vertical layer thickness, which may vary in time).

MARBL returns data on columns, which must then be remapped back to the OGCM's data format.



4.3.2 Coding conventions in MARBL

MARBL is written in Fortran. A few constructs commonly used in MARBL may be unfamiliar to scientific programmers.

Object-oriented programming features

Object-oriented programming constructs permit the definition of classes that both contain data and methods which can perform operations on that data.

Example of an Object-oriented Class

The class used to time blocks of code inside MARBL, `marbl_internal_timers_type`, is object-oriented:

```
!*****************************************************************************
! Internal timer types

type :: marbl_single_timer_type
  character(len=char_len) :: name
  logical                 :: is_running
  logical                 :: is_threaded
  real(r8)                :: cur_start
  real(r8)                :: cumulative_runtime
contains
  procedure :: init => init_single_timer
end type marbl_single_timer_type

type, public :: marbl_internal_timers_type
  integer :: num_timers
  type(marbl_single_timer_type), allocatable :: individual_timers(:)
contains
  procedure :: add => add_new_timer
  procedure :: start => start_timer
  procedure :: stop => stop_timer
  procedure :: extract => extract_timer_data
  procedure :: setup => setup_timers
  procedure :: reset => reset_timers
  procedure :: shutdown => shutdown_timers
end type marbl_internal_timers_type
```

This type includes several *methods*, such as the `start()` routine, referenced to the subroutine `start_timer`.

How to Call an Object-oriented Subroutine

A subroutine inside a class is referenced just like any other member, via the `%` character. For example, MARBL times the call to the subroutine `marbl_compute_carbonate_chemistry()` from the subroutine `marbl_interior_tendency_compute()` (part of `marbl_interior_tendency_mod.F90`). The timer calls look like this:

```
subroutine marbl_interior_tendency_compute( &
  .
  .
  .
  type(marbl_internal_timers_type),                        intent(inout) :: marbl_
↪timers
  .
  .
  .
  call marbl_timers%start(marbl_timer_indices%carbonate_chem_id,             &
```

(continues on next page)

(continued from previous page)

```
                          marbl_status_log)
  call compute_carbonate_chemistry(domain, temperature, pressure,           &
       salinity, tracer_local(:, :), marbl_tracer_indices, carbonate,       &
       ph_prev_col(:), ph_prev_alt_co2_col(:), marbl_status_log)
  call marbl_timers%stop(marbl_timer_indices%carbonate_chem_id,             &
                          marbl_status_log)
```

Example of a Subroutine Inside an Object-oriented Class

The subroutine header looks like this:

```
subroutine start_timer(self, id, marbl_status_log)

  class(marbl_internal_timers_type), intent(inout) :: self
  integer,                           intent(in)    :: id
  type(marbl_log_type),              intent(inout) :: marbl_status_log
```

One key thing to note here is the use of `self`, the first argument to the subroutine. In this case, `self` stands for the particular instance of an object of type `marbl_internal_timers_type`. The subroutine is actually part of this object, which lets it access members of the class without explicitly passing them through the interface. So this subroutine can change members of `self%individual_timers(:)` (e.g. `individual_timers(:)%cur_start`).

Associate construct

The `associate` construct allows complex variables or expression to be denoted by a simple name or "alias." The association between the name and the underlying variable is terminated at the end of the associate block.

If we look closer at the `start_timer` routine, we see an example:

```
associate(timer => self%individual_timers(id))
  if (timer%is_running) then
    log_message = 'Timer has already been started!'
    call marbl_status_log%log_error(log_message, subname)
    return
  end if

  timer%is_running = .true.
  .
  .
  .
  timer%cur_start = get_time()
 end associate
```

In this case, `timer` replaces all instances of the more complicated expression `self%individual_timers(id)`. The assocation is terminated at `end associate`.

4.3.3 MARBL interface

GCMs should use the MARBL interface class to call MARBL routines. The class definition is shown below:

```
type, public :: marbl_interface_class

```

(continues on next page)

(continued from previous page)

```
  ! public data - general
  type(marbl_domain_type)                          , public  :: domain
  type(marbl_tracer_metadata_type)  , allocatable, public  :: tracer_metadata(:)
  ! Pointer so that destructor doesn't need to reset all inds to 0
  ! (that happens automatically when new tracer indexing type is allocated)
  type(marbl_tracer_index_type)     , pointer    , public  :: tracer_indices =>␣
↪NULL()
  type(marbl_log_type)                             , public  :: StatusLog

  type(marbl_saved_state_type)                , public               :: surface_flux_
↪saved_state             ! input/output
  type(marbl_saved_state_type)                , public               :: interior_
↪tendency_saved_state        ! input/output
  type(marbl_surface_flux_saved_state_indexing_type), public       :: surf_state_ind
  type(marbl_interior_tendency_saved_state_indexing_type), public  :: interior_state_
↪ind
  type(marbl_timers_type)                     , public               :: timer_summary

  ! public data related to computing interior tendencies
  real (r8), allocatable                                , public  :: tracers(:,:)␣
↪           ! input
  type(marbl_forcing_fields_type), allocatable       , public  :: interior_tendency_
↪forcings(:) ! input
  real (r8), allocatable                                , public  :: interior_
↪tendencies(:,:)      ! output
  type(marbl_interior_tendency_forcing_indexing_type), public  :: interior_tendency_
↪forcing_ind ! FIXME #311: should be private
  type(marbl_diagnostics_type)                          , public  :: interior_tendency_
↪diags       ! output

  ! public data related to computing surface fluxes
  real (r8)                                          , public, allocatable  :: tracers_
↪at_surface(:,:)     ! input
  type(marbl_forcing_fields_type)                    , public, allocatable  :: surface_
↪flux_forcings(:)    ! input
  type(marbl_surface_flux_forcing_indexing_type) , public               :: surface_
↪flux_forcing_ind    ! FIXME #311: should be private
  real (r8)                                          , public, allocatable  :: surface_
↪fluxes(:,:)         ! output
  type(marbl_surface_flux_output_type)               , public               :: surface_
↪flux_output         ! output
  type(marbl_diagnostics_type)                       , public               :: surface_
↪flux_diags          ! output

  ! public data - global averages
  real (r8)                                       , public, allocatable  :: glo_avg_fields_
↪interior_tendency(:)   ! output (nfields)
  real (r8)                                       , public, allocatable  :: glo_avg_
↪averages_interior_tendency(:) ! input (nfields)
  real (r8)                                       , public, allocatable  :: glo_avg_fields_
↪surface_flux(:,:)      ! output (num_elements,nfields)
  real (r8)                                       , public, allocatable  :: glo_avg_
↪averages_surface_flux(:)      ! input (nfields)

  ! FIXME #77: for now, running means are being computed in the driver
  !            they will eventually be moved from the interface to inside MARBL
  real (r8)                                       , public, allocatable  :: glo_scalar_
↪interior_tendency(:)
```

(continues on next page)

(continued from previous page)

```
   real (r8)                                    , public, allocatable  :: glo_scalar_
↪surface_flux(:)

   type(marbl_running_mean_0d_type)             , public, allocatable  :: glo_avg_rmean_
↪interior_tendency(:)
   type(marbl_running_mean_0d_type)             , public, allocatable  :: glo_avg_rmean_
↪surface_flux(:)
   type(marbl_running_mean_0d_type)             , public, allocatable  :: glo_scalar_
↪rmean_interior_tendency(:)
   type(marbl_running_mean_0d_type)             , public, allocatable  :: glo_scalar_
↪rmean_surface_flux(:)

   ! private data
   type(marbl_PAR_type),                        private :: PAR
   type(autotroph_derived_terms_type),          private :: autotroph_derived_terms
   type(autotroph_local_type),                  private :: autotroph_local
   type(zooplankton_derived_terms_type),        private :: zooplankton_derived_terms
   type(zooplankton_local_type),                private :: zooplankton_local
   type(zooplankton_share_type),                private :: zooplankton_share
   type(marbl_particulate_share_type),          private :: particulate_share
   type(marbl_interior_tendency_share_type), private :: interior_tendency_share
   type(dissolved_organic_matter_type),         private :: dissolved_organic_matter
   type(carbonate_type),                        private :: carbonate
   type(marbl_surface_flux_share_type),         private :: surface_flux_share
   type(marbl_surface_flux_internal_type),      private :: surface_flux_internal
   logical,                                     private :: lallow_glo_ops
   type(marbl_internal_timers_type),            private :: timers
   type(marbl_timer_indexing_type),             private :: timer_ids
   type(marbl_settings_type),                   private :: settings

 contains

   procedure, public  :: init
   procedure, public  :: reset_timers
   procedure, public  :: extract_timing
   procedure, private :: glo_vars_init
   procedure, public  :: get_tracer_index
   procedure, public  :: interior_tendency_compute
   procedure, public  :: surface_flux_compute
   procedure, public  :: set_global_scalars
   procedure, public  :: shutdown
   generic            :: inquire_settings_metadata => inquire_settings_metadata_by_
↪name, &
                                                      inquire_settings_metadata_by_id
   generic            :: put_setting => put_real,            &
                                        put_integer,         &
                                        put_logical,         &
                                        put_string,          & ! This routine checks␣
↪to see if string is actually an array
                                        put_input_file_line, & ! This line converts␣
↪string "var = val" to proper put()
                                        put_all_string
   generic            :: get_setting => get_real,    &
                                        get_integer, &
                                        get_logical, &
                                        get_string
   procedure, public  :: get_settings_var_cnt
```

(continues on next page)

(continued from previous page)

```
   procedure, private :: inquire_settings_metadata_by_name
   procedure, private :: inquire_settings_metadata_by_id
   procedure, private :: put_real
   procedure, private :: put_integer
   procedure, private :: put_logical
   procedure, private :: put_string
   procedure, private :: put_input_file_line
   procedure, private :: put_all_string
   procedure, private :: get_real
   procedure, private :: get_integer
   procedure, private :: get_logical
   procedure, private :: get_string

end type marbl_interface_class
```

4.3.4 Development Examples

Examples of common development tasks.

Adding a Diagnostic

This is a five step process. There are three changes to make in the Fortran code. The indexing type is in `marbl_interface_private_types.F90`, and the rest of the code is in `marbl_diagnostics_mod.F90`. (If your diagnostic is part of the carbon isotope tracer module, that code belongs in `marbl_ciso_diagnostics_mod.F90`.) There are also two steps to make sure the diagnostic is known the GCM so it is included in the output.

For this example, we follow the in situ temperature, which uses the `insitu_temp` index.

Step 1. Add to MARBL diagnostic indexing type

To reduce the number of string comparisons inside routines called every time-step, MARBL uses integer indices to track many different variables. These indices are packed into datatypes to group common indices together. So the indices for diagnostics variables are split into `marbl_surface_flux_diagnostics_indexing_type` and `marbl_interior_tendency_diagnostics_indexing_type`. `insitu_temp` is an interior forcing diagnostic.

```
type, public :: marbl_interior_tendency_diagnostics_indexing_type
  ! General 2D diags
  integer(int_kind) :: zsatcalc
  integer(int_kind) :: zsatarag
  .
  .
  .
  ! General 3D diags
  integer(int_kind) :: insitu_temp
  .
  .
  .
end type marbl_interior_tendency_diagnostics_indexing_type
```

Step 2. Add to diagnostic structure

Another common feature among MARBL datatypes is the idea of adding an element to a derived type to contain all the data. Most derived types, including as `marbl_diagnostics_type`, are "reallocating": when a field is added, a new array of size `N+1` is created, the existing array is copied into the first `N` elements and then deallocated, and the new entry becomes element `N+1`. In these situations, pointers are used instead of allocatable arrays so that `marbl_instance%{surface,interior}_forcing_diags%diags` can point to the new array.

```
lname = 'in situ temperature'
sname = 'insitu_temp'
units = 'degC'
vgrid = 'layer_avg'
truncate = .false.
call diags%add_diagnostic(lname, sname, units, vgrid, truncate,     &
     ind%insitu_temp, marbl_status_log)
if (marbl_status_log%labort_marbl) then
  call marbl_logging_add_diagnostics_error(marbl_status_log, sname, subname)
  return
end if
```

Step 3. Populate diagnostic type with data

The purpose of the `marbl_diagnostics_type` structure is to allow an easy way to pass diagnostics through the interface. This step copies data only available in MARBL into the datatype that is available to the GCM.

```
associate( &
     kmt   => domain%kmt, &
     diags => marbl_interior_tendency_diags%diags, &
     ind   => marbl_interior_tendency_diag_ind &
     )
diags(ind%insitu_temp)%field_3d(1:kmt, 1) = temperature(1:kmt)
end associate
```

Note: In situ temperature is copied to the diagnostic type in `marbl_diagnostics_interior_tendency_compute()`. This subroutine also calls many different `store_diagnostics_*` subroutines, but in a future release the `store_diagnostics` routines will be condensed into a smaller subset of routines. Regardless, find the routine that makes the most sense for your diagnostic variable. (Surface forcing fields are copied to the diagnostic type in `marbl_diagnostics_surface_flux_compute()`.)

Step 4. Update the Diagnostics YAML files

We use a YAML file to provide an easy-to-edit and human-readable text file containing a list of all diagnostics and the recommended frequency of output. Developers adding or removing diagnostics should make changes to `defaults/diagnostics_latest.yaml`.

```
insitu_temp :
   longname : in situ temperature
   units : degC
   vertical_grid : layer_avg
   frequency : medium
   operator : average
```

Note that `insitu_temp` matches what we used for the short name in *Step 2. Add to diagnostic structure*. The frequency `medium` means "we recommend outputting this variable monthly". Other acceptable frequencies are `never`, `low` (annual), and `high` (daily).

The operator means "average over this time period." Other acceptable operators are `instantaneous`, `minimum`, and `maximum`. You can recommend multiple frequencies by adding a list to the YAML, as long as the operator key is a list of the same size:

```
CaCO3_form_zint :
   longname : Total CaCO3 Formation Vertical Integral
   units : mmol/m^3 cm/s
   vertical_grid : none
   frequency :
      - medium
      - high
   operator :
      - average
      - average
```

Step 5. Convert the YAML file to JSON

We prefer editing YAML files to editing JSON files because they are much easier to maintain (and allow user comments). Unfortunately, python does not include a YAML parser in the default distributions. Rather than require all users to install `pyYAML`, we require that of MARBL developers and then ask them to convert the YAML files to JSON. The `MARBL_tools/yaml_to_json.py` script is provided to do just that:

```
$ cd MARBL_tools
$ ./yaml_to_json.py
```

The rest of the python scripts provided in the `MARBL_tools/` subdirectory rely on the JSON file rather than the YAML. `MARBL_tools/MARBL_generate_diagnostics_file.py` will turn the JSON file into a list for the GCM to parse:

```
# This file contains a list of all diagnostics MARBL can compute for a given␣
↪configuration,
# as well as the recommended frequency and operator for outputting each diagnostic.
# The format of this file is:
#
# DIAGNOSTIC_NAME : frequency_operator
#
# And fields that should be output at multiple different frequencies will be comma-
↪separated:
#
# DIAGNOSTIC_NAME : frequency1_operator1, frequency2_operator2, ..., frequencyN_
↪operatorN
#
# Frequencies are never, low, medium, and high.
# Operators are instantaneous, average, minimum, and maximum.
.
.
.
CaCO3_form_zint : medium_average, high_average
.
.
.
insitu_temp : medium_average
```

It is then up to the GCM to convert this text file into a format it recognizes for output (e.g. POP will add to the `tavg_contents` file).

Adding a MARBL Parameter

This is a five step process. There are three changes to make in the Fortran code, all of which are made in `marbl_settings_mod.F90*`[0]. There are also two steps to make sure the parameter is picked up by the python tools to allow non-default values to be set.

Settings parameters are sorted into four categories that are processed in the following order:

```
general_parms
PFT_counts
PFT_derived_types
tracer_dependent
```

This is necessary because the general parameter settings may affect the number of PFTs being modeled, which changes the dimensions of the PFT derived types, which may affect the tracer count. During initialization, parameters will be defined and the default values will be set for each category. The defaults will be overwritten if the GCM called `marbl_instance%put_setting()` before calling `init()`.

For this example, we follow the minimum O2 needed for prodution and consumption parameter, which is `parm_o2_min` in the code. `parm_o2_min` is a general parameter.

Step 1. Create new module-level variable

All parameter settings are module variables in `marbl_settings_mod.F90`. Further, all subroutines and module variables are public in this module.

```
real(kind=r8), target :: &
real(r8), target :: &
     Jint_Ctot_thres_molpm2pyr,  & ! MARBL will abort if abs(Jint_Ctot) exceeds this␣
↪threshold
     .
     .
     .
     parm_Fe_bioavail,           & ! fraction of Fe flux that is bioavailable
     parm_o2_min,                & ! min O2 needed for prod & consump. (nmol/cm^3)
     parm_o2_min_delta,          & ! width of min O2 range (nmol/cm^3)
     .
     .
     .
```

Note: The `target` attribute is necessary because MARBL's internal parameter registry makes use of pointers.

Step 2. Add the parameter to MARBL registry

This registry allows the parameter to be both set by and returned to the GCM. Parameters in each of the four possible categories are defined separately from each other, in one of the following subroutines:

[0] The only exception is for parameters dealing with PFTs, those are in `marbl_pft_mod.F90`.

```
subroutine marbl_settings_define_general_parms(this, marbl_status_log)
subroutine marbl_settings_define_PFT_counts(this, marbl_status_log)
subroutine marbl_settings_define_PFT_derived_types(this, marbl_status_log)
subroutine marbl_settings_define_tracer_dependent(this, marbl_status_log)
```

`parm_o2_min` is registered in `marbl_settings_define_general_parms`:

```
subroutine marbl_settings_define_general_parms(this, marbl_status_log)
.
.
.
  sname     = 'parm_o2_min'
  lname     = 'Minimum O2 needed for production and consumption'
  units     = 'nmol/cm^3'
  datatype  = 'real'
  rptr      => parm_o2_min
  call this%add_var(sname, lname, units, datatype, category,       &
                    marbl_status_log, rptr=rptr)
  call check_and_log_add_var_error(marbl_status_log, sname, subname, labort_marbl_loc)
```

Step 3. Set default value

MARBL convention is to have a reasonable default value defined in case the variable is not changed via an input settings file. Defaults for each of the four categories are set separately from each other, immediately after parameters for that category are defined. The subroutines for setting defaults are

```
subroutine marbl_settings_set_defaults_general_parms()
subroutine marbl_settings_set_defaults_PFT_counts(marbl_status_log)
subroutine marbl_settings_set_defaults_PFT_derived_types(marbl_status_log)
subroutine marbl_settings_set_defaults_tracer_dependent(marbl_status_log)
```

`parm_o2_min` is set in `marbl_settings_set_defaults_general_parms`:

```
subroutine marbl_settings_set_defaults_general_parms()
  .
  .
  .
  parm_Fe_bioavail              = 1.0_r8          ! CESM USERS - DO NOT CHANGE HERE!␣
↪POP calls put_setting() for this var, see CESM NOTE above
  parm_o2_min                   = 5.0_r8          ! CESM USERS - DO NOT CHANGE HERE!␣
↪POP calls put_setting() for this var, see CESM NOTE above
  parm_o2_min_delta             = 5.0_r8          ! CESM USERS - DO NOT CHANGE HERE!␣
↪POP calls put_setting() for this var, see CESM NOTE above
```

Step 4. Update the settings YAML files

We use a YAML file to provide an easy-to-edit and human-readable text file containing a list of all parameters and their default values. On the `development` branch, make changes to `defaults/settings_latest.yaml`. Release branches may only offer specific versions of this file, such as `defaults/settings_cesm2.1.yaml`.

```
# ABOUT THIS FILE
# ---------------
# MARBL users can change settings values for runtime-configurable variables via a␣
↪settings
```

(continues on next page)

(continued from previous page)

```
# input file. MARBL provides a python script that can generate an input file by␣
↪reading a
# JSON file containing the configurable variables and default values, but JSON does␣
↪not allow
# comments in the file format so the workflow is to edit this YAML file and then␣
↪generate
# the JSON file via $MARBL/MARBL_tools/yaml_to_json.py
#
# Parameters in MARBL are divided into four different stages, based on the order in␣
↪which they are set
# 1. General Parameters: variables that have no dependencies on other stages
#    (note that init_bury_coeff_opt is alone in general_parms2 because it depends on␣
↪ladjust_bury_coeff)
# 2. PFT Counts: variables that can not be set until after PFT_defaults (in General␣
↪Parameters) is known
# 3. PFT Derived Types: variables that can not be set until PFT Counts are known
#                       (autotroph_cnt, zooplankton_cnt, and max_grazer_prey_cnt)
# 4. Post-Tracer: variables that can not be set until the tracer count is known
#                 (tracer count depends on PFT Derived Types)
#
# All variables need to provide the following metadata:
# 1. longname: a description of the variable
# 2. subcategory: when writing parameters to the log, MARBL will group variables by␣
↪subcategory
# 3. units: physical units (use "unitless" for pure numbers and "non-numeric" for␣
↪strings / logicals)
# 4. datatype: integer, real, logical, or string
# 5. default_value: Value to use unless overwritten by the MARBL input file
#       NOTE: some parameters provide different default values for different␣
↪configurations;
#             e.g. in CESM, the value of some parameters is resolution-dependent. In␣
↪these
#             cases, default_value should be a dictionary with a "default" key and␣
↪then keys
#             for whatever resolutions differ from the default.
#
#             Accepted keys:
#                1. default
#                2. CESM_x3
#
# There are also some optional metadata options:
# 1. valid_values: only values that MARBL will accept (default_value must be in valid_
↪values!)
# 2. cannot change:
# 3. must set:
# 4. _append_to_config_keywords: if default values of variables processed later␣
↪depend on the
#                                value of another variable, then that variable needs␣
↪to have
#                                _append_to_config_keywords = True
#
.
.
.
###############################################################################
#                        Category 1: General Parameters                       #
###############################################################################
```

(continues on next page)

(continued from previous page)

```
general_parms :
   .
   .
   .
   parm_o2_min :
      longname : Minimum O2 needed for production & consumption
      subcategory : 4. general parameters
      units : nmol/cm^3
      datatype : real
      default_value : 5.0
```

Step 5. Convert the YAML file to JSON

We prefer editing YAML files to editing JSON files because they are much easier to maintain (and allow user comments). Unfortunately, python does not include a YAML parser in the default distributions. Rather than require all users to install `pyYAML`, we require that of MARBL developers and then ask them to convert the YAML files to JSON. The `MARBL_tools/yaml_to_json.py` script is provided to do just that:

```
$ cd MARBL_tools
$ ./yaml_to_json.py
```

The rest of the python scripts provided in the `MARBL_tools/` subdirectory rely on the JSON file rather than the YAML. `MARBL_tools/MARBL_generate_settings_file.py` will turn the JSON file into a list for the GCM to parse:

```
! general parameters
.
.
.
parm_o2_min = 5.0
parm_o2_min_delta = 5.0
```

It is then up to the GCM to read this text file and pass it line by line to `marbl_instance%put_setting()`

Adding a Tracer

The steps needed to add a new tracer depend greatly on what the tracer is, so this page will not use a single tracer as an example. Also, a significant portion of the code shown in these examples will be cleaned up prior to the MARBL 1.0.0 release (sorry!).

MARBL Code Changes

This is an eight step process.

Step 1. Add to MARBL tracer index type

As mentioned in *Step 1. Add to MARBL diagnostic indexing type*, the `indexing_type` is a common structure in MARBL. Due to the many ways to introduce tracers (different modules, living tracers, etc), the tracer indexing type is a little more complex than others.

```
type, public :: marbl_tracer_index_type
  ! Book-keeping (tracer count and index ranges)
  integer (int_kind) :: total_cnt = 0
  type (marbl_tracer_count_type) :: ecosys_base
  type (marbl_tracer_count_type) :: ciso

  ! General tracers
  integer (int_kind) :: po4_ind         = 0 ! dissolved inorganic phosphate
  .
  .
  .
  ! CISO tracers
  integer (int_kind) :: di13c_ind       = 0 ! dissolved inorganic carbon 13
  .
  .
  .
  ! Living tracers
  type(marbl_living_tracer_index_type), allocatable :: auto_inds(:)
  .
  .
  .
contains
  procedure, public :: add_tracer_index
  procedure, public :: construct => tracer_index_constructor
  procedure, public :: destruct => tracer_index_destructor
end type marbl_tracer_index_type
```

For this example, assume we are adding a single tracer in the base ecosys module (regretfully referred to as "general tracers" in most comments).

Note: This data type does not conform to MARBL naming conventions and will be renamed `marbl_tracer_indexing_type` in a future update.

Step 2. Update `tracer_index_constructor`

If you are adding a tracer that is only active in certain configurations, you would include an if statement around the following code. At this point in time, all the base ecosystem tracers are present in all configurations, so there is no such restriction. For example, here we set in index for the refractory DOC tracer:

```
subroutine tracer_index_constructor(this, ciso_on, lvariable_PtoC, autotroph_settings,
↪ &
           zooplankton_settings, marbl_status_log)
  .
  .
  .
  ! General ecosys tracers
  .
  .
  .
  call this%add_tracer_index('docr', 'ecosys_base', this%docr_ind, marbl_status_log)
  .
  .
  .
end subroutine tracer_index_constructor
```

Note: There is an issue ticket to refer to objects as `self` instead of `this`. *Example of an Object-oriented Class* has it right.

Step 3. Set tracer metadata

MARBL provides the following metadata to describe each tracer:

```
type, public :: marbl_tracer_metadata_type
   character(len=char_len) :: short_name
   character(len=char_len) :: long_name
   character(len=char_len) :: units
   character(len=char_len) :: tend_units
   character(len=char_len) :: flux_units
   logical                 :: lfull_depth_tavg
   character(len=char_len) :: tracer_module_name
end type marbl_tracer_metadata_type
```

There are a few different subroutines in `marbl_init_mod.F90` to define the metadata for different classes of tracers. (Metadata for carbon isotope tracers is handled in `marbl_ciso_init_mod::marbl_ciso_init_tracer_metadata`.)

```
subroutine marbl_init_tracer_metadata
subroutine marbl_init_non_autotroph_tracer_metadata
subroutine marbl_init_non_autotroph_tracers_metadata
subroutine marbl_init_zooplankton_tracer_metadata
subroutine marbl_init_autotroph_tracer_metadata
```

The last three subroutines above are called from `marbl_init_tracer_metadata()`, and `marbl_init_non_autotroph_tracer_metadata()` is called from `marbl_init_non_autotroph_tracers_metadata()` Prior to those calls, `marbl_init_tracer_metadata()` sets two attributes in the metadata type:

```
marbl_tracer_metadata(:)%lfull_depth_tavg   = .true.
marbl_tracer_metadata(:)%tracer_module_name = 'ecosys'
```

Metadata for all base ecosystem non-living tracers is set in `marbl_init_non_autotroph_tracers_metadata()`. For example, here is where the dissolved inorganic phosphate index is set:

```
subroutine marbl_init_non_autotroph_tracers_metadata(marbl_tracer_metadata, &
           marbl_tracer_indices)
  .
  .
  .
  call marbl_init_non_autotroph_tracer_metadata('PO4', 'Dissolved Inorganic Phosphate
↪', &
              marbl_tracer_metadata(marbl_tracer_indices%po4_ind))
```

Step 4. Compute surface flux for new tracer (if necessary)

Not all tracers return a surface flux, so this may not be necessary for your tracer. For this example, we will follow the oxygen tracer. Surface fluxes are computed in `marbl_surface_flux_mod::marbl_surface_flux_compute`:

```
subroutine marbl_surface_flux_compute( &
  .
  .
  .
  associate(                                                                    ↵
↪             &
    .
    .
    .
    o2_ind             => marbl_tracer_indices%o2_ind,                          ↵
↪      &
    .
    .
    .
    )

  !-----------------------------------------------------------------------
  !  fluxes initially set to 0
  !-----------------------------------------------------------------------

  surface_fluxes(:, :) = c0
  .
  .
  .
  !-----------------------------------------------------------------------
  !  compute CO2 flux, computing disequilibrium one row at a time
  !-----------------------------------------------------------------------

  if (lflux_gas_o2 .or. lflux_gas_co2) then
     .
     .
     .
     if (lflux_gas_o2) then
       .
       .
       .
       pv_o2(:) = xkw_ice(:) * sqrt(660.0_r8 / schmidt_o2(:))
       o2sat(:) = ap_used(:) * o2sat_1atm(:)
       flux_o2_loc(:) = pv_o2(:) * (o2sat(:) - tracers_at_surface(:, o2_ind))
       surface_fluxes(:, o2_ind) = surface_fluxes(:, o2_ind) + flux_o2_loc(:)
```

Step 5. Compute tracer tendency

The tracer tendencies are computed in a two step process - MARBL computes the tracer tendency terms from a variety of processes and then combines the terms in the end. Given the modular nature of MARBL, the tendencies from each process are computed in their own routine. This is done in `marbl_interior_tendency_mod::interior_tendency_compute`:

```
subroutine marbl_interior_tendency_compute( &
  .
  .
  .
  call compute_PAR(domain, interior_tendency_forcings, interior_tendency_forcing_
↪indices, &
                   totalChl_local, PAR)
```

(continues on next page)

(continued from previous page)

```
  call compute_autotroph_elemental_ratios(km, autotroph_local, marbl_tracer_indices,␣
↪tracer_local, &
       autotroph_derived_terms)

  call compute_function_scaling(temperature, Tfunc)
  .
  .
  .
  do k = 1, km

     call compute_scavenging(k, km, marbl_tracer_indices, tracer_local(:,:), &
          POC, P_CaCO3, P_SiO2, dust, Fefree(:), Fe_scavenge_rate(:), &
          Fe_scavenge(:), Lig_scavenge(:), marbl_status_log)

     if (marbl_status_log%labort_marbl) then
        call marbl_status_log%log_error_trace('compute_scavenging()', subname)
        return
     end if

     call compute_large_detritus_prod(k, domain, marbl_tracer_indices, zooplankton_
↪derived_terms, &
          autotroph_derived_terms, Fe_scavenge(k),                    &
          POC, POP, P_CaCO3, P_CaCO3_ALT_CO2, P_SiO2, dust, P_iron,   &
          dissolved_organic_matter%DOP_loss_P_bal(k), marbl_status_log)

     ! FIXME #28: need to pull particulate share out
     !            of compute_particulate_terms!
     call compute_particulate_terms(k, domain,                   &
          marbl_particulate_share, p_remin_scalef(k),            &
          POC, POP, P_CaCO3, P_CaCO3_ALT_CO2,                    &
          P_SiO2, dust, P_iron, PON_remin(k), PON_sed_loss(k),   &
          QA_dust_def(k),                                        &
          tracer_local(:, k), carbonate, sed_denitrif(k),        &
          other_remin(k), fesedflux(k), marbl_tracer_indices,    &
          glo_avg_fields_interior_tendency, marbl_status_log)

     if (marbl_status_log%labort_marbl) then
        call marbl_status_log%log_error_trace('compute_particulate_terms()', subname)
        return
     end if

     if  (k < km) then
        call update_particulate_terms_from_prior_level(k+1, POC, POP, P_CaCO3, &
             P_CaCO3_ALT_CO2, P_SiO2, dust, P_iron, QA_dust_def(:))
     endif

  end do ! k
  .
  .
  .
  call compute_denitrif(km, marbl_tracer_indices, tracer_local(:, :), &
       dissolved_organic_matter%DOC_remin(:), &
       dissolved_organic_matter%DOCr_remin(:), &
       POC%remin(:), other_remin(:), sed_denitrif(:), denitrif(:))

  call compute_local_tendencies(km, marbl_tracer_indices, autotroph_derived_terms, &
```

(continues on next page)

(continued from previous page)

```
       zooplankton_derived_terms, &
       dissolved_organic_matter, &
       nitrif(:), denitrif(:), sed_denitrif(:), &
       Fe_scavenge(:), Lig_prod(:), Lig_loss(:), &
       P_iron%remin(:), POC%remin(:), POP%remin(:), &
       P_SiO2%remin(:), P_CaCO3%remin(:), P_CaCO3_ALT_CO2%remin(:), &
       other_remin(:), PON_remin(:), &
       tracer_local(:,:), &
       o2_consumption_scalef(:), &
       o2_production(:), o2_consumption(:), &
       interior_tendencies(:,:))
```

The tendencies are combined in `compute_local_tendencies` while subroutines like `compute_PAR`, `compute_autotroph_uptake`, and `compute_denitrif` are the per-process computations. So you will need to update `compute_local_tendencies` to compute the tracer tendency for your new tracer correctly:

```
subroutine compute_local_tendencies(km, marbl_tracer_indices, autotroph_derived_terms,
↪ &
     zooplankton_derived_terms, dissolved_organic_matter, nitrif, denitrif, sed_
↪denitrif, &
     Fe_scavenge, Lig_prod, Lig_loss, P_iron_remin, POC_remin, POP_remin, P_SiO2_
↪remin, &
     P_CaCO3_remin, P_CaCO3_ALT_CO2_remin, other_remin, PON_remin, tracer_local, &
     o2_consumption_scalef, o2_production, o2_consumption, interior_tendencies)
  .
  .
  .
  do k=1, km
    .
    .
    .
    o2_consumption(k) = (O2_loc(k) - parm_o2_min) / parm_o2_min_delta
    o2_consumption(k) = min(max(o2_consumption(k), c0), c1)
    o2_consumption(k) = o2_consumption(k) * ((POC_remin(k) * (c1 - POCremin_refract)␣
↪+ DOC_remin(k) + DOCr_remin(k) &
                                             - (sed_denitrif(k) * denitrif_C_N) -␣
↪other_remin(k) &
                                             + sum(zoo_loss_dic(:,k)) + sum(zoo_
↪graze_dic(:,k))  &
                                             + sum(auto_loss_dic(:,k)) + sum(auto_
↪graze_dic(:,k))) &
                                            / parm_Remin_D_C_O2 + (c2 * nitrif(k)))
    o2_consumption(k) = o2_consumption_scalef(k) * o2_consumption(k)

    interior_tendencies(o2_ind,k) = o2_production(k) - o2_consumption(k)

  end do
```

Step 6. Add any necessary diagnostics

By default, MARBL's diagnostics include the interior restoring tendency for each tracer. Otherwise, it is assumed that the GCM will provide tracer diagnostics itself. MARBL does compute the vertical integral of the conservative terms in the source-sink computation of many tracers. If your tracer affects these integrals, you should update the appropriate subroutine in `marbl_diagnostics_mod.F90`:

```
private :: store_diagnostics_carbon_fluxes
private :: store_diagnostics_nitrogen_fluxes
private :: store_diagnostics_phosphorus_fluxes
private :: store_diagnostics_silicon_fluxes
private :: store_diagnostics_iron_fluxes
```

If you want to provide a specific diagnostic related to your tracer, see *Adding a Diagnostic*.

Step 7. Update the settings YAML files

The `defaults/settings_*.yaml` files also contain a list of all defined tracers. On the `development` branch, make changes to `defaults/settings_latest.yaml`. Release branches may only offer specific versions of this file, such as `defaults/settings_cesm2.1.yaml`. The block of code defining the tracers looks like this:

```
# ABOUT THIS FILE
# ---------------
.
.
.
# Tracer count
_tracer_list :
   # Non-living tracers
   PO4 :
      long_name : Dissolved Inorganic Phosphate
      units : mmol/m^3
   NO3 :
      long_name : Dissolved Inorganic Nitrate
      units : mmol/m^3
.
.
.
```

This list needed because some parameters (such as `tracer_restore_vars(:)`) depend on the tracer count. Additionally, it makes it easy for GCMs to see a list of all tracers being returned by MARBL to help configure diagnostic output.

Step 8. Convert the YAML file to JSON

We prefer editing YAML files to editing JSON files because they are much easier to maintain (and allow user comments). Unfortunately, python does not include a YAML parser in the default distributions. Rather than require all users to install `pyYAML`, we require that of MARBL developers and then ask them to convert the YAML files to JSON. The `MARBL_tools/yaml_to_json.py` script is provided to do just that:

```
$ cd MARBL_tools
$ ./yaml_to_json.py
```

There is not a tracer-specific python script to run, but the `MARBL_settings_class` has `get_tracer_names()` and `get_tracer_cnt()` routines.

GCM Code Changes

The GCM will need to provide initial conditions for this new tracer, and may also need to output additional tracer-specific diagnostics. The MARBL guide is not able to offer guidance on how to do that, as it will vary from GCM to

GCM.

4.3.5 Code Snippets

The Fortran code included in this section has been linked from elsewhere in the *User Guide* or *Developer's Guide*.

Global Scalars

In some configurations, MARBL needs to know the value of some globally-averaged scalars. Currently, the only example of such is setting `ladjust_bury_coeff = .true.` to allow MARBL to recompute various burial coefficients. These burial coefficients are used to *Compute Interior Tracer Tendencies*.

Note: In these configurations, the GCM must explicitly tell MARBL it can perform global operations *during initialization* otherwise MARBL will abort.

Subroutine on the Interface

This is the subroutine called prior to calling *surface_flux_compute()* or *interior_tendency_compute()*. If `field_source == 'surface_flux'` then the subroutine returns without doing anything.

```
subroutine set_global_scalars(this, field_source)

  use marbl_interior_tendency_mod, only : marbl_interior_tendency_adjust_bury_coeff

  class(marbl_interface_class), intent(inout) :: this
  character(len=*),             intent(in)    :: field_source ! 'interior_tendency'␣
↪or 'surface_flux`'

  if (field_source == 'interior_tendency') then
     call marbl_interior_tendency_adjust_bury_coeff(                                  ␣
↪  &
          marbl_particulate_share            = this%particulate_share,                ␣
↪  &
          glo_avg_rmean_interior_tendency    = this%glo_avg_rmean_interior_tendency, ␣
↪  &
          glo_avg_rmean_surface_flux         = this%glo_avg_rmean_surface_flux,       ␣
↪  &
          glo_scalar_rmean_interior_tendency = this%glo_scalar_rmean_interior_
↪tendency, &
          glo_scalar_interior_tendency       = this%glo_scalar_interior_tendency)
  end if

end subroutine set_global_scalars
```

Global scalars for interior tendency

If `field_source == 'interior_tendency'` then `marbl_interior_tendency_adjust_bury_coeff()` is called. If `ladjust_bury_coeff == .false.` then this subroutine returns immediately. For runs where MARBL is adjusting the burial coefficients, though, values in `marbl_instance%particulate_share` as well as some global averages of running means are updated. Note that there is an assumption that the `marbl_glo_avg_mod` values have been *updated by the GCM*.

```
subroutine marbl_interior_tendency_adjust_bury_coeff(marbl_particulate_share, &
     glo_avg_rmean_interior_tendency, glo_avg_rmean_surface_flux, &
     glo_scalar_rmean_interior_tendency, glo_scalar_interior_tendency)

  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_CaCO3_bury
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_POC_bury
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_POP_bury
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_bSi_bury
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_d_POC_bury_d_bury_
↪coeff
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_d_POP_bury_d_bury_
↪coeff
  use marbl_glo_avg_mod, only : glo_avg_field_ind_interior_tendency_d_bSi_bury_d_bury_
↪coeff
  use marbl_glo_avg_mod, only : glo_avg_field_ind_surface_flux_C_input
  use marbl_glo_avg_mod, only : glo_avg_field_ind_surface_flux_P_input
  use marbl_glo_avg_mod, only : glo_avg_field_ind_surface_flux_Si_input
  use marbl_glo_avg_mod, only : glo_scalar_ind_interior_tendency_POC_bury_coeff
  use marbl_glo_avg_mod, only : glo_scalar_ind_interior_tendency_POP_bury_coeff
  use marbl_glo_avg_mod, only : glo_scalar_ind_interior_tendency_bSi_bury_coeff

  type (marbl_particulate_share_type), intent(inout) :: marbl_particulate_share
  type (marbl_running_mean_0d_type)  , intent(in)    :: glo_avg_rmean_interior_
↪tendency(:)
  type (marbl_running_mean_0d_type)  , intent(in)    :: glo_avg_rmean_surface_flux(:)
  type (marbl_running_mean_0d_type)  , intent(in)    :: glo_scalar_rmean_interior_
↪tendency(:)
  real (r8)                          , intent(inout) :: glo_scalar_interior_
↪tendency(:)

  !-----------------------------------------------------------------------

  if (.not. ladjust_bury_coeff) return

  associate( &
       POC_bury_coeff => marbl_particulate_share%POC_bury_coeff, &
       POP_bury_coeff => marbl_particulate_share%POP_bury_coeff, &
       bSi_bury_coeff => marbl_particulate_share%bSi_bury_coeff, &

       rmean_CaCO3_bury_avg => glo_avg_rmean_interior_tendency(glo_avg_field_ind_
↪interior_tendency_CaCO3_bury)%rmean, &
       rmean_POC_bury_avg   => glo_avg_rmean_interior_tendency(glo_avg_field_ind_
↪interior_tendency_POC_bury)%rmean, &
       rmean_POP_bury_avg   => glo_avg_rmean_interior_tendency(glo_avg_field_ind_
↪interior_tendency_POP_bury)%rmean, &
       rmean_bSi_bury_avg   => glo_avg_rmean_interior_tendency(glo_avg_field_ind_
↪interior_tendency_bSi_bury)%rmean, &

       rmean_POC_bury_deriv_avg => &
            glo_avg_rmean_interior_tendency(glo_avg_field_ind_interior_tendency_d_POC_
↪bury_d_bury_coeff)%rmean, &
       rmean_POP_bury_deriv_avg => &
            glo_avg_rmean_interior_tendency(glo_avg_field_ind_interior_tendency_d_POP_
↪bury_d_bury_coeff)%rmean, &
       rmean_bSi_bury_deriv_avg => &
            glo_avg_rmean_interior_tendency(glo_avg_field_ind_interior_tendency_d_bSi_
↪bury_d_bury_coeff)%rmean, &
```

(continues on next page)

(continued from previous page)

```
      rmean_C_input_avg  => glo_avg_rmean_surface_flux(glo_avg_field_ind_surface_
↪flux_C_input)%rmean, &
      rmean_P_input_avg  => glo_avg_rmean_surface_flux(glo_avg_field_ind_surface_
↪flux_P_input)%rmean, &
      rmean_Si_input_avg => glo_avg_rmean_surface_flux(glo_avg_field_ind_surface_
↪flux_Si_input)%rmean, &

      rmean_POC_bury_coeff => &
           glo_scalar_rmean_interior_tendency(glo_scalar_ind_interior_tendency_POC_
↪bury_coeff)%rmean, &
      rmean_POP_bury_coeff => &
           glo_scalar_rmean_interior_tendency(glo_scalar_ind_interior_tendency_POP_
↪bury_coeff)%rmean, &
      rmean_bSi_bury_coeff => &
           glo_scalar_rmean_interior_tendency(glo_scalar_ind_interior_tendency_bSi_
↪bury_coeff)%rmean &
  )

   ! Newton's method for POC_bury(coeff) + CaCO3_bury - C_input = 0
   POC_bury_coeff = rmean_POC_bury_coeff &
                  - (rmean_POC_bury_avg + rmean_CaCO3_bury_avg - rmean_C_input_avg) &
                  / rmean_POC_bury_deriv_avg

   ! Newton's method for POP_bury(coeff) - P_input = 0
   POP_bury_coeff = rmean_POP_bury_coeff &
                  - (rmean_POP_bury_avg - rmean_P_input_avg) / rmean_POP_bury_deriv_
↪avg

   ! Newton's method for bSi_bury(coeff) - Si_input = 0
   bSi_bury_coeff = rmean_bSi_bury_coeff &
                  - (rmean_bSi_bury_avg - rmean_Si_input_avg) / rmean_bSi_bury_deriv_
↪avg

   ! copy computed bury coefficients into output argument
   glo_scalar_interior_tendency(glo_scalar_ind_interior_tendency_POC_bury_coeff) =␣
↪POC_bury_coeff
   glo_scalar_interior_tendency(glo_scalar_ind_interior_tendency_POP_bury_coeff) =␣
↪POP_bury_coeff
   glo_scalar_interior_tendency(glo_scalar_ind_interior_tendency_bSi_bury_coeff) =␣
↪bSi_bury_coeff

  end associate

end subroutine marbl_interior_tendency_adjust_bury_coeff
```

4.4 MARBL scientific documentation

The default MARBL configuration invokes the Biogeochemical Elemental Cycling (BEC) model *[MDL04]*. which is an ecosystem/biogeochemistry model designed to run within the ocean circulation component of CESM.

The ecosystem includes multiple phytoplankton functional groups (diatoms, diazotrophs, small phytoplankton, and coccolithophores) and multiple potentially growth limiting nutrients (nitrate, ammonium, phosphate, silicate, and iron) *[MDK+02][MDL04]*. There is one zooplankton group, dissolved organic material (semi-labile), sinking particulate pools and explicit simulation of the biogeochemical cycling of key elements (C, N, P, Fe, Si, O, plus alkalinity)

[MDL04]. The ecosystem component is coupled with a carbonate chemistry module based on the Ocean Carbon Model Intercomparison Project (OCMIP) *[DLM+09]* allowing dynamic computation of surface ocean pCO2 and air-sea CO2 flux.

The model allows for water column denitrication, whereby nitrate is consumed during remineralization in place of O2 once ambient O2 concentrations fall below 4 micro-molar *[MD07]*. Photoadaptation is calculated as a variable phytoplankton ratio of chlorophyll to nitrogen based on Geider et al. *[GMK98]*. The model allows for variable Fe/C and Si/C ratios with an optimum and minimum value prescribed. As ambient Fe (or Si for diatoms) concentrations decline the phytoplankton lower their cellular quotas. Phytoplankton N/P ratios are fixed at the Redfield value of 16, but the diazotroph group has a higher N/P atomic ratio of 50 (see detailed description of the model in Moore et al., 2002 *[MDK+02]*, and Moore et al., 2004 *[MDL04]*). Thus, community N/P uptake varies with the phytoplankton community composition.

The ecosystem model results have been compared extensively against in situ data (e.g., JGOFS time series stations) and SeaWiFS satellite ocean color observations in a global mixed layer only variant and coupled with a full-depth, global 3-D general circulation model *[MDK+02][MDL04][DLM+09]*. In both cases, the simulated output is in generally good agreement with bulk ecosystem observations (e.g., total biomass, productivity, nutrients, export) across diverse ecosystems that include both macro-nutrient and iron-limited regimes as well as very different physical environments from high latitude sites to the mid-ocean gyres. The model also incorporates the work of Moore and Braucher *[MB08]*, who incorporated an improved sedimentary iron source and scavenging parameterization, greatly improving simulated iron fields relative to observations, and the work of Krishnamurthy et al. *[KMZL07]*, who describe the impact of atmospheric deposition of nitrogen.

Note: This description is still under development.

4.4.1 Phytoplankton growth

Light propagation

Nutrient limitation

4.4.2 Light attenuation formulation

Starting point is equation 6 of *[MM01]*

$$Z_e = \begin{cases} 912.5\,[\mathrm{Chl_{tot}}]^{-0.839} & 10 < Z_e < 102 \\ 426.3\,[\mathrm{Chl_{tot}}]^{-0.547} & 102 < Z_e < 180 \end{cases} \tag{4.2}$$

In this approximation, $[\mathrm{Chl_{tot}}]$ is $[\mathrm{Chl}(z)]$ integrated from the surface to Z_e, the depth of the euphotic zone.

We will convert this formula to one based on $[\mathrm{Chl}(z)]$ averaged over the euphotic zone

$$[\mathrm{Chl_{tot}}] = Z_e \overline{[\mathrm{Chl}]} \tag{4.3}$$

The crossover point $Z_e = 102$ corresponds to $[\mathrm{Chl_{tot}}] = 13.65$, which is equivalent to $\overline{[\mathrm{Chl}]} = 0.1338$.

Substituting equation (4.3) into equation (4.2) and solving for Z_e yields

$$Z_e = \begin{cases} 40.710\overline{[\mathrm{Chl}]}^{-0.4562} & \overline{[\mathrm{Chl}]} > 0.1338 \\ 50.105\overline{[\mathrm{Chl}]}^{-0.3536} & \overline{[\mathrm{Chl}]} < 0.1338 \end{cases} \tag{4.4}$$

The euphotic zone depth is defined to be the depth where PAR is 1% of its surface value *[Mor88]*.

We denote the attenuation coefficient of PAR as K, and its effective average over the euphotic zone as $\overline{K}$.

So we have

$$0.01 = e^{-Z_e \overline{K}}.$$

Solving for Z_e yields

$$Z_e = -\log 0.01/\overline{K} = \log 100/\overline{K}. \tag{4.5}$$

Substituting equation (4.5) into equation (4.4) and solving for $\overline{K}$ yields

$$\overline{K} = \begin{cases} 0.1131\overline{[\mathrm{Chl}]}^{0.4562} & \overline{[\mathrm{Chl}]} > 0.1338 \\ 0.0919\overline{[\mathrm{Chl}]}^{0.3536} & \overline{[\mathrm{Chl}]} < 0.1338 \end{cases} \tag{4.6}$$

In the model implementation, this equation relating $\overline{K}$ to $\overline{[\mathrm{Chl}]}$ is applied to each model layer.

The crossover point was recomputed to be where the curves cross, yielding $\overline{[\mathrm{Chl}]} = 0.13224$.

The units of K in equation (4.6) are 1/m.

Model units are cm, so the model implementation includes multiplication by 0.01.

4.5 Rules of engagement

4.5.1 Developer Guidelines

This document provides guidance for individuals contributing to the MARBL project. The MARBL code is hosted on github at https://github.com/marbl-ecosys/MARBL.

Repository Structure

Users interested in developing new features should fork the repository into their own github account and make branches off `development`. Submit a pull request back to `development` when your modifications are complete. The `stable` branch in the MARBL repository contains code from scientifically-vetted releases.

Publishing research with development code

Your contributions to the MARBL project will likely be used by others on the development team. This section aims to ensure that authors receive appropriate credit for contributions to the MARBL code base. Publishing work based on MARBL development code is encouraged; however, this project relies on contributions from a broad group and people deserve appropriate recognition for their intellectual efforts.

You are expected to communicate publication plans to other members of the MARBL development team if your work involves their contributions.

You must offer co-authorship (or other mutually agreed upon recognition) on papers or other scientific communications to authors of MARBL development code if that code meets all of the following conditions:

- The research used the code by invoking it at run time;
- The code impacted solutions, such that results and/or conclusions would differ without it; and
- The code was not available on `stable` or a release branch.

MARBL code that has been released is not subject to these restrictions.

Mailing List

We recommend signing up for the MARBL developer's mailing list. This mailing list is used by the repository administrators to send out announcements about changes to the repository that may effect development. We also encourage developers to use the list to ask questions about the code or to communicate with each other about projects that could benefit from collaboration.

4.5.2 Github workflow

4.5.3 Working on the documentation

Here are guidelines for installing and using Sphinx to develop documentation.

Local environment

The MARBL documentation is built in Sphinx from content written in reStructuredText.

The following extensions are required.

- sphinxcontrib-bibtex

Python must be available locally.

Miniconda is a nice tool for maintaining Python: https://conda.io/miniconda.html

It's helpful to setup an environment. See here for more on conda environments.

With conda installed, do the following (the last command assumes you are in the root of your MARBL repository):

```
$ conda create --name marbl-docs pip
$ conda activate marbl-docs
(marbl-docs)$ pip install -r docs/py_requirements.txt
```

This creates an environment call "marbl-docs" and ensures that `pip install` commands are local to the environment rather than global.

To deactivate the "marbl-docs" environment run

```
(marbl-docs)$ conda deactivate
```

Documentation workflow

Here's some notes on how to modify the documentation.

Do all development work on a branch

Checkout a new local branch using

```
(marbl-docs)$ git checkout -b my_branch
```

to create a branch or omit the `-b` to checkout an existing branch.

Edit documentation source

Modify and/or add reStructuredText files.

The documentation has three major sections

	Developer's guide	`MARBL/docs/src/dev-guide`
	Scientific manual	`MARBL/docs/src/sci-guide`
	User guide	`MARBL/docs/src/usr-guide`

The file `index.html` in each of these directories includes the table of contents for each section; this file must be modified when new pages are added.

Begin each `rst` file with a label that is the same as the file name

```
.. _myfilename:
```

Note the position of the underscore and ending colon. This enables referencing this page from elsewhere in the project using

```
:ref:`Name of link<myfilename>`
```

Build the documentation

Once changes are complete, build from *src* using

```
(marbl-docs)[docs/src]$ make clean html
```

The compiled documentation ends up in `MARBL/docs/html`. You can view the files there in a browser locally as you work.

Commit changes

You can check the status of your modification using

```
[MARBL]$ git status
```

When you are ready to commit

```
[MARBL/docs]$ git add .
[MARBL/docs]$ git commit -m 'message describing changes'
```

Headers in ReStructuredText

reStructuredText parses special characters to create titles, subtitles, and other headers in a non-unique way, which is to say that there are multiple ways to produce the same set of headers. Any non-alphanumeric [7-bit] character repeated for the entire length of the line above it will turn the line above it into a header. If you desire, you can also overline the header text with the same string. The order you use the special characters must be consistent within a file (the first character choice produces a title, the second character choice produces a subtitle, and so on). For example, the following two blocks of code translate into the same page:

```
Title
-----

Subtitle
~~~~~~~~

Subsubtitle
===========
```

and

```
Title
+++++

^^^^^^^^
Subtitle
^^^^^^^^

Subsubtitle
___________
```

For consistency, MARBL documentation should use the same pattern across all files. (Again, this is not a requirement of reStructuredText.) The preferred pattern is

```
=====
Title
=====

--------
Subtitle
--------

~~~~~~~~~~
Subsubtitle
~~~~~~~~~~
```

Note that this convention is entirely arbitrary, but should make reading `.rst` files a little easier. If you find a need for a Subsubsubtitle, choose your favorite special character that is not already in use and then edit this page accordingly.

---

reStructuredText resource

The authoritative reStructuredText User Documentation.

Bibliography

[Mor88] André Morel. Optical modeling of the upper ocean in relation to its biogenous matter content (case I waters). *J. Geophys. Res.*, 93(C9):10749–10768, 1988. doi:10.1029/jc093ic09p10749.

[MM01] André Morel and Stéphane Maritorena. Bio-optical properties of oceanic waters: a reappraisal. *J. Geophys. Res.*, 106(C4):7163–7180, Apr 2001. doi:10.1029/2000jc000319.

[DLM+09] S. C. Doney, I. Lima, J.K. Moore, K. Lindsay, M.J. Behrenfeld, T.K. Westberry, N. Mahowald, D.M. Glover, and T. Takahashi. Skill metrics for confronting global upper ocean ecosystem-biogeochemistry models against field and remote sensing data. *J. Mar. Systems*, 76:95–112, 2009.

[GMK98] R. Geider, H. MacIntyre, and T. Kana. A dynamic regulatory model of phytoplankton acclimation to light, nutrients, and temperature. *Limnology and Oceanography*, 43:679–694, 1998.

[KMZL07] A. Krishnamurthy, J. K. Moore, C. S. Zender, and C. Luo. The effects of atmospheric inorganic nitrogen deposition on ocean biogeochemistry. *J. Geophys. Res.*, 2007.

[MB08] J. K. Moore and O. Braucher. Sedimentary and mineral dust sources of dissolved iron to the world ocean. *Biogeosciences*, 5:631–656, 2008.

[MDK+02] J. K. Moore, S. Doney, J. Kleypas, D. Glover, and I. Fung. An intermediate complexity marine ecosystem model for the global domain. *Deep-Sea Res. II*, 49:403–462, 2002.

[MD07] J. K. Moore and S. C. Doney. Iron availability limits the ocean nitrogen inventory stabilizing feedbacks between marine denitrification and nitrogen fixation. *Global Biogeochem. Cycles*, 2007.

[MDL04] J. K. Moore, S. C. Doney, and K. Lindsay. Upper ocean ecosystem dynamics and iron cycling in a global three-dimensional model. *Global Biogeochem. Cycles*, 2004.