MARBL Documentation

Release cesm2.0

MARBL developers

Jul 20, 2018

Contents

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 2.0. 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 input file (TODO: input file link!) 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_forcing,    &
     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_forcing
  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_forcing`. There is not a corresponding `gcm_num_elements_interior_forcing` 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 input files (`*.input` in the test directory) that are processed in the following manner:

```
ioerr = 0
do while(ioerr .eq. 0)
  input_line = ''
  ! (i) master task reads next line in inputfile
  if (my_task .eq. 0) read(*,"(A)", iostat=ioerr) input_line
  ! (ii) broadcast inputfile line to all tasks (along with iostat)
  call marbl_mpi_bcast(ioerr, 0)
  call marbl_mpi_bcast(input_line, 0)
  ! (iii) call put_setting(); abort if error
  call marbl_instance%put_setting(input_line)
  if (marbl_instance%StatusLog%labort_marbl) then
    call marbl_instance%StatusLog%log_error("Error reading input file!", subname)
    call print_marbl_log(marbl_instance%StatusLog)
  end if
end do
```

(continues on next page)

(continued from previous page)

```
if (.not.is_iostat_end(ioerr)) then
  write(*,"(A,I0)") "ioerr = ", ioerr
  write(*,"(A)") "ERROR encountered when reading MARBL input file from stdin"
  call marbl_mpi_abort()
end if
```

`init()` is then called from the individual test, storing the tracer count in the local variable `nt`:

```
! Call marbl%init
call marbl_instance%init(gcm_num_levels = km,                    &
                         gcm_num_PAR_subcols = 1,                &
                         gcm_num_elements_surface_forcing = 1, &
                         gcm_delta_z = delta_z,                  &
                         gcm_zw = zw,                            &
                         gcm_zt = zt,                            &
                         marbl_tracer_cnt = nt)
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_input_file` regression test with no input file. The test writes this output to `marbl.input`. 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
ciso_lecovars_full_depth_tavg = F
lflux_gas_o2 = T
lflux_gas_co2 = T
lcompute_nhx_surface_emis = T
lvariable_PtoC = T
ladjust_bury_coeff = F
init_bury_coeff_opt = 'settings_file'
particulate_flux_ref_depth = 100
Jint_Ctot_thres_molpm2pyr =   0.10000000000000001E-08
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 =   0.11000000000000001E+01
parm_init_POP_bury_coeff =   0.11000000000000001E+01
parm_init_bSi_bury_coeff =   0.10000000000000000E+01
parm_Fe_scavenge_rate0 =   0.18000000000000000E+02
parm_Lig_scavenge_rate0 =   0.14999999999999999E-01
parm_FeLig_scavenge_rate0 =   0.13999999999999999E+01
```

(continues on next page)

(continued from previous page)

```
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 =   0.77000000000000000E+05
parm_CaCO3_diss =   0.50000000000000000E+05
parm_sed_denitrif_coeff =   0.10000000000000000E+01
bury_coeff_rmean_timescale_years =   0.100000000000000000E+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) =   0.30000000000000000E+01
parm_scalelen_vals(3) =   0.45000000000000000E+01
parm_scalelen_vals(4) =   0.55000000000000000E+01
caco3_bury_thres_opt = 'omega_calc'
caco3_bury_thres_depth =   0.30000000000000000E+06
caco3_bury_thres_omega_calc =   0.10000000000000000E+01
PON_bury_coeff =   0.50000000000000000E+00
ciso_fract_factors = 'Laws'
autotroph_cnt = 3
zooplankton_cnt = 1
max_grazer_prey_cnt = 3
autotrophs(1)%sname = 'sp'
autotrophs(1)%lname = 'Small Phyto'
autotrophs(1)%Nfixer = F
autotrophs(1)%imp_calcifier = T
autotrophs(1)%exp_calcifier = F
autotrophs(1)%silicifier = F
autotrophs(1)%kFe =   0.30000000000000001E-04
autotrophs(1)%kPO4 =   0.10000000000000000E-01
autotrophs(1)%kDOP =   0.29999999999999999E+00
autotrophs(1)%kNO3 =   0.25000000000000000E+00
autotrophs(1)%kNH4 =   0.10000000000000000E-01
autotrophs(1)%kSiO3 =   0.00000000000000000E+00
autotrophs(1)%Qp_fixed =   0.85470085470085479E-02
autotrophs(1)%gQfe_0 =   0.34999999999999997E-04
autotrophs(1)%gQfe_min =   0.27000000000000000E-05
autotrophs(1)%alphaPi_per_day =   0.39000000000000001E+00
autotrophs(1)%PCref_per_day =   0.50000000000000000E+01
autotrophs(1)%thetaN_max =   0.25000000000000000E+01
autotrophs(1)%loss_thres =   0.10000000000000000E-01
autotrophs(1)%loss_thres2 =   0.00000000000000000E+00
autotrophs(1)%temp_thres =  -0.10000000000000000E+02
autotrophs(1)%mort_per_day =   0.10000000000000001E+00
autotrophs(1)%mort2_per_day =   0.10000000000000000E-01
autotrophs(1)%agg_rate_max =   0.50000000000000000E+00
autotrophs(1)%agg_rate_min =   0.10000000000000000E-01
autotrophs(1)%loss_poc =   0.00000000000000000E+00
autotrophs(2)%sname = 'diat'
autotrophs(2)%lname = 'Diatom'
autotrophs(2)%Nfixer = F
autotrophs(2)%imp_calcifier = F
autotrophs(2)%exp_calcifier = F
autotrophs(2)%silicifier = T
autotrophs(2)%kFe =   0.60000000000000002E-04
```

(continues on next page)

(continued from previous page)

```
autotrophs(2)%kPO4 =   0.50000000000000003E-01
autotrophs(2)%kDOP =   0.50000000000000000E+00
autotrophs(2)%kNO3 =   0.50000000000000000E+00
autotrophs(2)%kNH4 =   0.50000000000000003E-01
autotrophs(2)%kSiO3 =   0.69999999999999996E+00
autotrophs(2)%Qp_fixed =   0.85470085470085479E-02
autotrophs(2)%gQfe_0 =   0.34999999999999997E-04
autotrophs(2)%gQfe_min =   0.27000000000000000E-05
autotrophs(2)%alphaPi_per_day =   0.28999999999999998E+00
autotrophs(2)%PCref_per_day =   0.50000000000000000E+01
autotrophs(2)%thetaN_max =   0.40000000000000000E+01
autotrophs(2)%loss_thres =   0.20000000000000000E-01
autotrophs(2)%loss_thres2 =   0.00000000000000000E+00
autotrophs(2)%temp_thres =  -0.10000000000000000E+02
autotrophs(2)%mort_per_day =   0.10000000000000001E+00
autotrophs(2)%mort2_per_day =   0.10000000000000000E-01
autotrophs(2)%agg_rate_max =   0.50000000000000000E+00
autotrophs(2)%agg_rate_min =   0.20000000000000000E-01
autotrophs(2)%loss_poc =   0.00000000000000000E+00
autotrophs(3)%sname = 'diaz'
autotrophs(3)%lname = 'Diazotroph'
autotrophs(3)%Nfixer = T
autotrophs(3)%imp_calcifier = F
autotrophs(3)%exp_calcifier = F
autotrophs(3)%silicifier = F
autotrophs(3)%kFe =   0.45000000000000003E-04
autotrophs(3)%kPO4 =   0.14999999999999999E-01
autotrophs(3)%kDOP =   0.74999999999999997E-01
autotrophs(3)%kNO3 =   0.20000000000000000E+01
autotrophs(3)%kNH4 =   0.20000000000000001E+00
autotrophs(3)%kSiO3 =   0.00000000000000000E+00
autotrophs(3)%Qp_fixed =   0.27350427350427355E-02
autotrophs(3)%gQfe_0 =   0.69999999999999994E-04
autotrophs(3)%gQfe_min =   0.54000000000000000E-05
autotrophs(3)%alphaPi_per_day =   0.39000000000000001E+00
autotrophs(3)%PCref_per_day =   0.25000000000000000E+01
autotrophs(3)%thetaN_max =   0.25000000000000000E+01
autotrophs(3)%loss_thres =   0.20000000000000000E-01
autotrophs(3)%loss_thres2 =   0.10000000000000000E-02
autotrophs(3)%temp_thres =   0.15000000000000000E+02
autotrophs(3)%mort_per_day =   0.10000000000000001E+00
autotrophs(3)%mort2_per_day =   0.10000000000000000E-01
autotrophs(3)%agg_rate_max =   0.50000000000000000E+00
autotrophs(3)%agg_rate_min =   0.10000000000000000E-01
autotrophs(3)%loss_poc =   0.00000000000000000E+00
zooplankton(1)%sname = 'zoo'
zooplankton(1)%lname = 'Zooplankton'
zooplankton(1)%z_mort_0_per_day =   0.10000000000000001E+00
zooplankton(1)%loss_thres =   0.74999999999999997E-01
zooplankton(1)%z_mort2_0_per_day =   0.40000000000000002E+00
grazing(1,1)%sname = 'grz_sp_zoo'
grazing(1,1)%lname = 'Grazing of sp by zoo'
grazing(1,1)%auto_ind_cnt = 1
grazing(1,1)%zoo_ind_cnt = 0
grazing(1,1)%grazing_function = 1
grazing(1,1)%z_umax_0_per_day =   0.32999999999999998E+01
grazing(1,1)%z_grz =   0.12000000000000000E+01
```

(continues on next page)

(continued from previous page)

```
grazing(1,1)%graze_zoo =   0.29999999999999999E+00
grazing(1,1)%graze_poc =   0.00000000000000000E+00
grazing(1,1)%graze_doc =   0.59999999999999998E-01
grazing(1,1)%f_zoo_detr =   0.12000000000000000E+00
grazing(1,1)%auto_ind(1) = 1
grazing(2,1)%sname = 'grz_diat_zoo'
grazing(2,1)%lname = 'Grazing of diat by zoo'
grazing(2,1)%auto_ind_cnt = 1
grazing(2,1)%zoo_ind_cnt = 0
grazing(2,1)%grazing_function = 1
grazing(2,1)%z_umax_0_per_day =   0.31000000000000001E+01
grazing(2,1)%z_grz =   0.12000000000000000E+01
grazing(2,1)%graze_zoo =   0.25000000000000000E+00
grazing(2,1)%graze_poc =   0.38000000000000000E+00
grazing(2,1)%graze_doc =   0.59999999999999998E-01
grazing(2,1)%f_zoo_detr =   0.23999999999999999E+00
grazing(2,1)%auto_ind(1) = 2
grazing(3,1)%sname = 'grz_diaz_zoo'
grazing(3,1)%lname = 'Grazing of diaz by zoo'
grazing(3,1)%auto_ind_cnt = 1
grazing(3,1)%zoo_ind_cnt = 0
grazing(3,1)%grazing_function = 1
grazing(3,1)%z_umax_0_per_day =   0.32500000000000000E+01
grazing(3,1)%z_grz =   0.12000000000000000E+01
grazing(3,1)%graze_zoo =   0.29999999999999999E+00
grazing(3,1)%graze_poc =   0.10000000000000001E+00
grazing(3,1)%graze_doc =   0.59999999999999998E-01
grazing(3,1)%f_zoo_detr =   0.12000000000000000E+00
grazing(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) = ''
```

(continues on next page)

(continued from previous page)

```
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.input`, and organizes the output better than the Fortran test:

```
! config PFTs
PFT_defaults = "CESM2"
autotroph_cnt = 3
max_grazer_prey_cnt = 3
zooplankton_cnt = 1

! config flags
ciso_lecovars_full_depth_tavg = .false.
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.
lsource_sink = .true.
lvariable_PtoC = .true.

! config strings
init_bury_coeff_opt = "settings_file"

! general parameters
Jint_Ctot_thres_molpm2pyr = 1e-09
bury_coeff_rmean_timescale_years = 10
caco3_bury_thres_depth =   3.0000000000000000e+05
caco3_bury_thres_omega_calc = 1.0
caco3_bury_thres_opt = "omega_calc"
ciso_fract_factors = "Laws"
parm_Fe_bioavail = 1.0
parm_Fe_desorption_rate0 =   9.9999999999999995e-07
parm_Lig_degrade_rate0 = 9.4e-05
parm_f_prod_sp_CaCO3 = 0.07
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

! general parameters (bury coeffs)
PON_bury_coeff = 0.5
parm_init_POC_bury_coeff = 1.1
parm_init_POP_bury_coeff = 1.1
parm_init_bSi_bury_coeff = 1.0

! general parameters (dissipation)
parm_CaCO3_diss =   5.0000000000000000e+04
parm_POC_diss =   1.0000000000000000e+04
```

(continues on next page)

(continued from previous page)

```
parm_SiO2_diss =   7.7000000000000000e+04

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

! general parameters (scavenging)
parm_FeLig_scavenge_rate0 = 1.4
parm_Fe_scavenge_rate0 = 18.0
parm_Lig_scavenge_rate0 = 0.015

! Scale lengths
parm_scalelen_vals(1) = 1
parm_scalelen_vals(2) = 3.0
parm_scalelen_vals(3) = 4.5
parm_scalelen_vals(4) = 5.5
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
autotrophs(1)%Nfixer = .false.
autotrophs(1)%PCref_per_day = 5
autotrophs(1)%Qp_fixed =   8.5470085470085479e-03
autotrophs(1)%agg_rate_max = 0.5
autotrophs(1)%agg_rate_min = 0.01
autotrophs(1)%alphaPI_per_day = 0.39
autotrophs(1)%exp_calcifier = .false.
autotrophs(1)%gQfe_0 =   3.4999999999999997e-05
autotrophs(1)%gQfe_min = 2.7e-06
autotrophs(1)%imp_calcifier = .true.
autotrophs(1)%kDOP = 0.3
autotrophs(1)%kFe = 3e-05
autotrophs(1)%kNH4 = 0.01
autotrophs(1)%kNO3 = 0.25
autotrophs(1)%kPO4 = 0.01
autotrophs(1)%kSiO3 = 0
autotrophs(1)%lname = "Small Phyto"
autotrophs(1)%loss_poc = 0
autotrophs(1)%loss_thres = 0.01
autotrophs(1)%loss_thres2 = 0
autotrophs(1)%mort2_per_day = 0.01
autotrophs(1)%mort_per_day = 0.1
autotrophs(1)%silicifier = .false.
autotrophs(1)%sname = "sp"
autotrophs(1)%temp_thres = -10
autotrophs(1)%thetaN_max = 2.5
autotrophs(2)%Nfixer = .false.
autotrophs(2)%PCref_per_day = 5
autotrophs(2)%Qp_fixed =   8.5470085470085479e-03
autotrophs(2)%agg_rate_max = 0.5
autotrophs(2)%agg_rate_min = 0.02
autotrophs(2)%alphaPI_per_day = 0.29
autotrophs(2)%exp_calcifier = .false.
autotrophs(2)%gQfe_0 =   3.4999999999999997e-05
autotrophs(2)%gQfe_min = 2.7e-06
```

(continues on next page)

(continued from previous page)

```
autotrophs(2)%imp_calcifier = .false.
autotrophs(2)%kDOP = 0.5
autotrophs(2)%kFe = 6e-05
autotrophs(2)%kNH4 = 0.05
autotrophs(2)%kNO3 = 0.5
autotrophs(2)%kPO4 = 0.05
autotrophs(2)%kSiO3 = 0.7
autotrophs(2)%lname = "Diatom"
autotrophs(2)%loss_poc = 0
autotrophs(2)%loss_thres = 0.02
autotrophs(2)%loss_thres2 = 0
autotrophs(2)%mort2_per_day = 0.01
autotrophs(2)%mort_per_day = 0.1
autotrophs(2)%silicifier = .true.
autotrophs(2)%sname = "diat"
autotrophs(2)%temp_thres = -10
autotrophs(2)%thetaN_max = 4
autotrophs(3)%Nfixer = .true.
autotrophs(3)%PCref_per_day = 2.5
autotrophs(3)%Qp_fixed =   2.7350427350427355e-03
autotrophs(3)%agg_rate_max = 0.5
autotrophs(3)%agg_rate_min = 0.01
autotrophs(3)%alphaPI_per_day = 0.39
autotrophs(3)%exp_calcifier = .false.
autotrophs(3)%gQfe_0 =   6.9999999999999994e-05
autotrophs(3)%gQfe_min = 5.4e-06
autotrophs(3)%imp_calcifier = .false.
autotrophs(3)%kDOP = 0.075
autotrophs(3)%kFe = 4.5e-05
autotrophs(3)%kNH4 = 0.2
autotrophs(3)%kNO3 = 2
autotrophs(3)%kPO4 = 0.015
autotrophs(3)%kSiO3 = 0
autotrophs(3)%lname = "Diazotroph"
autotrophs(3)%loss_poc = 0
autotrophs(3)%loss_thres = 0.02
autotrophs(3)%loss_thres2 = 0.001
autotrophs(3)%mort2_per_day = 0.01
autotrophs(3)%mort_per_day = 0.1
autotrophs(3)%silicifier = .false.
autotrophs(3)%sname = "diaz"
autotrophs(3)%temp_thres = 15
autotrophs(3)%thetaN_max = 2.5

! zooplankton
zooplankton(1)%lname = "Zooplankton"
zooplankton(1)%loss_thres = 0.075
zooplankton(1)%sname = "zoo"
zooplankton(1)%z_mort2_0_per_day = 0.4
zooplankton(1)%z_mort_0_per_day = 0.1

! grazing
grazing(1,1)%auto_ind(1) = 1
grazing(1,1)%auto_ind_cnt = 1
grazing(1,1)%f_zoo_detr = 0.12
grazing(1,1)%graze_doc = 0.06
grazing(1,1)%graze_poc = 0
```

(continues on next page)

(continued from previous page)

```
grazing(1,1)%graze_zoo = 0.3
grazing(1,1)%grazing_function = 1
grazing(1,1)%lname = "Grazing of sp by zoo"
grazing(1,1)%sname = "grz_sp_zoo"
grazing(1,1)%z_grz = 1.2
grazing(1,1)%z_umax_0_per_day = 3.3
grazing(1,1)%zoo_ind_cnt = 0
grazing(2,1)%auto_ind(1) = 2
grazing(2,1)%auto_ind_cnt = 1
grazing(2,1)%f_zoo_detr = 0.24
grazing(2,1)%graze_doc = 0.06
grazing(2,1)%graze_poc = 0.38
grazing(2,1)%graze_zoo = 0.25
grazing(2,1)%grazing_function = 1
grazing(2,1)%lname = "Grazing of diat by zoo"
grazing(2,1)%sname = "grz_diat_zoo"
grazing(2,1)%z_grz = 1.2
grazing(2,1)%z_umax_0_per_day = 3.1
grazing(2,1)%zoo_ind_cnt = 0
grazing(3,1)%auto_ind(1) = 3
grazing(3,1)%auto_ind_cnt = 1
grazing(3,1)%f_zoo_detr = 0.12
grazing(3,1)%graze_doc = 0.06
grazing(3,1)%graze_poc = 0.1
grazing(3,1)%graze_zoo = 0.3
grazing(3,1)%grazing_function = 1
grazing(3,1)%lname = "Grazing of diaz by zoo"
grazing(3,1)%sname = "grz_diaz_zoo"
grazing(3,1)%z_grz = 1.2
grazing(3,1)%z_umax_0_per_day = 3.25
grazing(3,1)%zoo_ind_cnt = 0

! tracer restoring
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) = ""
```

(continues on next page)

(continued from previous page)

```
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

If running with `ladjust_bury_coeff = .true.` then MARBL will ask the GCM to provide results from the two functions mentioned below. Note that the typical runcase sets `ladjust_bury_coeff = .false.`, so for first-time implementations it is okay to skip this section.

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)

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
```

(continues on next page)

(continued from previous page)

```
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,nt
  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 similar to the question *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)
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_input_forcings)
  write(log_message, "(I0, 5A)") n, '. ', &
        trim(marbl_instance%surface_input_forcings(n)%metadata%varname), &
```

(continues on next page)

(continued from previous page)

```
        ' (units: ', trim(marbl_instance%surface_input_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_input_forcings)
  write(log_message, "(I0, 5A)") n, '. ',                                    &
       trim(marbl_instance%interior_input_forcings(n)%metadata%varname), &
       ' (units: ', trim(marbl_instance%interior_input_forcings(n)%metadata%field_
↪units),')'
  call driver_status_log%log_noerror(log_message, subname)
end do
```

The `marbl_interface_class` contains two objects (`surface_input_forcings` and `interior_input_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`, and contain the `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.

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_surface_elements, 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
```

(continues on next page)

(continued from previous page)

```
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_interior_forcing, 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)
  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

```
marbl_instance%surface_saved_state
marbl_instance%interior_saved_state
```

Both of these are of type `marbl_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%set_surface_forcing()` returns, the GCM needs to process `marbl_intance%surface_saved_state`. That means looping through each element in the `marbl_instance%surface_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 `set_surface_forcing()` in the next time step, these saved values should be copied back into `marbl_intance%surface_saved_state`.

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

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

`set_surface_forcing()` computes surface fluxes (and related diagnostics) over a 1D array of grid cells. The array is assumed to be length `num_surface_forcing_elements`, per *The init() interface*. The stand-alone test suite does not yet call this routine, so examples come from the POP driver. The call to the routine is straightforward:

```
call marbl_instances(iblock)%set_surface_forcing()
```

The details are in the surrounding calls.

What MARBL needs prior to calling set_surface_forcing

The GCM needs to make sure the MARBL instance has all the data it needs to compute surface fluxes correctly. Specifically, it needs to do the following.

Step 1. Set global scalars

Currently, there are no global scalars that need to be set for this stage. To prepare for future updates where that is no longer the case, it is recommended that the GCM calls

```
call marbl_instances(iblock)%set_global_scalars('surface')
```

Step 2. Copy data into MARBL

In POP, 2D `(nx_block, ny_block)` data is reshaped into an array of length `num_surface_forcing_elements = nx_block*ny_block`. Data for `surface_input_forcings`, `surface_vals` (tracer values at the surface), and saved state (MARBL data from previous time step) are all copied in.

```
!-----------------------------------------------------------------------
! Copy data from slab data structure to column input for marbl
!-----------------------------------------------------------------------

do j = 1, ny_block
   do i = 1,nx_block
      index_marbl = i + (j-1)*nx_block

      do n = 1,size(surface_forcing_fields)
         marbl_instances(iblock)%surface_input_forcings(n)%field_0d(index_marbl) = &
              surface_forcing_fields(n)%field_0d(i,j,iblock)
      end do

      do n = 1,ecosys_tracer_cnt
         marbl_instances(iblock)%surface_vals(index_marbl,n) = &
              p5*(surface_vals_old(i,j,n,iblock) + surface_vals_cur(i,j,n,iblock))

      end do

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

   end do
end do
```

What the GCM needs after MARBL returns

MARBL returns surface fluxes (and any requested surface forcing fields) for all the columns, and they need to be stored in the GCM. Additionally, saved state needs to be saved so it is available in the next time step and any fields

that are globally averaged also need to be stored. Lastly, MARBL will return values for fields that need to be globally averaged.

```
!-----------------------------------------------------------------------
! 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(saved_state_surf)
     saved_state_surf(n)%field_2d(i,j,iblock) = &
       marbl_instances(iblock)%surface_saved_state%state(n)%field_2d(index_marbl)
   end do

   do n=1,sfo_cnt
     surface_forcing_outputs(i,j,n,iblock) = &
        marbl_instances(iblock)%surface_forcing_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_tracer_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)%surface_vals(index_marbl,n), ' ', &
          marbl_instances(iblock)%surface_tracer_fluxes(index_marbl,n)
    end do
    do n = 1, size(surface_forcing_fields)
       associate (forcing_field => surface_forcing_fields(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) = &
         marbl_instances(iblock)%surface_tracer_fluxes(index_marbl,n)
  end do

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

(continues on next page)

(continued from previous page)

```
  ! 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(index_marbl,:)
end do
```

Compute Interior Tracer Tendencies

`set_interior_forcing()` computes interior tracer tendencies (and related diagnostics) for a single column. (Recall that `num_interior_forcing_elements = 1`, per *The init() interface*.) The stand-alone test suite does not yet call this routine, so examples come from the POP driver. The call to the routine is straightforward:

```
call marbl_instances(iblock)%set_interior_forcing()
```

As with *Compute Surface Fluxes*, the details are in the surrounding calls.

What MARBL needs prior to calling set_interior_forcing

The GCM needs to make sure the MARBL instance has all the data it needs to compute interior tendencies correctly. Specifically it needs to to the following.

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.

```
call marbl_instances(iblock)%set_global_scalars('interior')
```

Note that at this point, MARBL 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).

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 (including `kmt`, the index of the level containing the ocean bottom)
2. Interior forcing data
3. Tracer values (for each level and each tracer)
4. Saved state

```
! --- set marbl_domain kmt and if partial bottom cells then also delta_z ---

marbl_instances(bid)%domain%kmt = KMT(i, c, bid)
if (partial_bottom_cells) then
   marbl_instances(bid)%domain%delta_z(:) = DZT(i, c, :, bid)
end if
```

(continues on next page)

(continued from previous page)

```
! --- set forcing fields ---

do n = 1, size(interior_forcing_fields)
  if (interior_forcing_fields(n)%rank == 2) then
    marbl_instances(bid)%interior_input_forcings(n)%field_0d(1) = &
         interior_forcing_fields(n)%field_0d(i,c,bid)
  else
    marbl_instances(bid)%interior_input_forcings(n)%field_1d(1,:) = &
         interior_forcing_fields(n)%field_1d(i,c,:,bid)
  end if
end do

! --- set column tracers, averaging 2 time levels into 1 ---

do n = 1, ecosys_tracer_cnt
   marbl_instances(bid)%column_tracers(n, :) = p5*(tracer_module_old(i, c, :, n) +␣
↪tracer_module_cur(i, c, :, n))
end do

! --- copy data from slab to column for marbl_saved_state ---
do n=1,size(saved_state_interior)
  marbl_instances(bid)%interior_saved_state%state(n)%field_3d(:,1) = &
    saved_state_interior(n)%field_3d(:,i,c,bid)
end do
```

What the GCM needs after MARBL returns

MARBL returns tracer tendencies on a per-column basis, and that needs to be stored in the GCM. Additionally, saved state needs to be saved so it is available in the next time step and any fields that are globally averaged also need to be stored.

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

do n = 1, ecosys_tracer_cnt
   dtracer_module(i, c, :, n) = marbl_instances(bid)%column_dtracers(n, :)
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(:)
```

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 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, `surface_forcings(:)` is the MARBL data provided through the interface, and `surface_forcing_fields(:)` is the copy into the POP datatype.

```
allocate(surface_forcing_fields(size(surface_forcings)))
do n=1,size(surface_forcing_fields)
  marbl_varname = surface_forcings(n)%metadata%varname
  units         = surface_forcings(n)%metadata%field_units
  select case (trim(surface_forcings(n)%metadata%varname))
    case ('surface_mask')
      mask_ind = n
      call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                           marbl_varname=marbl_varname, field_units=units,      &
                           driver_varname='SURFACE_MASK', id=n)

    case ('d13c')
      d13c_ind = n
      call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                           marbl_varname=marbl_varname, field_units=units,      &
                           driver_varname='D13C', id=n)

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

    case ('d14c_gloavg')
      d14c_glo_ind = n
      call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                           marbl_varname=marbl_varname, field_units=units,      &
                           driver_varname='D14C_GLOAVG', id=n)
```

(continues on next page)

(continued from previous page)

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

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

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

  case ('xco2')
    xco2_ind = n
    if (trim(atm_co2_opt).eq.'const') then
      call surface_forcing_fields(n)%add_forcing_field(field_source='const', &
                           marbl_varname=marbl_varname, field_units=units,   &
                           field_constant=atm_co2_const, id=n)
    else if (trim(atm_co2_opt).eq.'drv_prog') then
      call surface_forcing_fields(n)%add_forcing_field(field_source='named_field', &
                           marbl_varname=marbl_varname, field_units=units,         &
                           named_field='ATM_CO2_PROG', id=n)
    else if (trim(atm_co2_opt).eq.'drv_diag') then
      call surface_forcing_fields(n)%add_forcing_field(field_source='named_field', &
                           marbl_varname=marbl_varname, field_units=units,         &
                           named_field='ATM_CO2_DIAG', 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_forcing_fields(n)%add_forcing_field(field_source='const', &
                           marbl_varname=marbl_varname, field_units=units,   &
                           field_constant=atm_alt_co2_const, 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_forcing_fields(n)%add_forcing_field(field_source='internal', &
                           marbl_varname=marbl_varname, field_units=units,      &
```

(continues on next page)

(continued from previous page)

```
                          driver_varname='ICE Fraction', id=n)
    else if (trim(gas_flux_forcing_opt).eq.'file') then
      file_details => fice_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, 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
      call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                          marbl_varname=marbl_varname, field_units=units,      &
                          driver_varname='AP_FILE_INPUT', id=n)
    else if (trim(gas_flux_forcing_opt).eq.'file') then
      file_details => ap_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, 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_ind = n
    if (trim(dust_flux_source).eq.'driver') then
      call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                          marbl_varname=marbl_varname, field_units=units,      &
                          driver_varname='DUST_FLUX', id=n)
    else if (trim(dust_flux_source).eq.'monthly-calendar') then
      file_details => dust_flux_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, 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
```

(continues on next page)

(continued from previous page)

```
        bc_ind = n
        call surface_forcing_fields(n)%add_forcing_field(field_source='internal', &
                             marbl_varname=marbl_varname, field_units=units,      &
                             driver_varname='BLACK_CARBON_FLUX', id=n)
      else if (trim(iron_flux_source).eq.'monthly-calendar') then
        Fe_ind = n
        file_details => iron_flux_file_loc
        call init_monthly_surface_forcing_metadata(file_details)
        call surface_forcing_fields(n)%add_forcing_field(                    &
                             field_source='POP monthly calendar',            &
                             marbl_varname=marbl_varname, field_units=units, &
                             forcing_calendar_name=file_details, id=n)
      else
        write(err_msg, "(A,1X,A)") trim(iron_flux_source),                &
             'is not a valid option for iron_flux_source'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
      end if

    case ('NOx Flux')
      nox_ind = n
      if (trim(ndep_data_type).eq.'shr_stream') then
        call surface_forcing_fields(n)%add_forcing_field(field_source='shr_stream', &
                             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, id=n)
      else if (trim(ndep_data_type).eq.'monthly-calendar') then
        file_details => nox_flux_monthly_file_loc
        call init_monthly_surface_forcing_metadata(file_details)
        call surface_forcing_fields(n)%add_forcing_field(                    &
                             field_source='POP monthly calendar',            &
                             marbl_varname=marbl_varname, field_units=units, &
                             forcing_calendar_name=file_details, 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')
      nhy_ind = n
      if (trim(ndep_data_type).eq.'shr_stream') then
        call surface_forcing_fields(n)%add_forcing_field(field_source='shr_stream', &
                             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,           &
                             year_align = ndep_shr_stream_year_align,         &
                             filename = ndep_shr_stream_file, id=n)
      else if (trim(ndep_data_type).eq.'monthly-calendar') then
        file_details => nhy_flux_monthly_file_loc
```

(continues on next page)

(continued from previous page)

```
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                           field_source='POP monthly calendar',            &
                           marbl_varname=marbl_varname, field_units=units, &
                           forcing_calendar_name=file_details, 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 ('DIN River Flux')
    file_details => din_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
                         marbl_varname=marbl_varname, field_units=units,  &
                         forcing_calendar_name=file_details, id=n)

  case ('DIP River Flux')
    file_details => dip_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
                         marbl_varname=marbl_varname, field_units=units,  &
                         forcing_calendar_name=file_details, id=n)

  case ('DON River Flux')
    file_details => don_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
                         marbl_varname=marbl_varname, field_units=units,  &
                         forcing_calendar_name=file_details, id=n)

  case ('DOP River Flux')
    file_details => dop_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
                         marbl_varname=marbl_varname, field_units=units,  &
                         forcing_calendar_name=file_details, id=n)

  case ('DSi River Flux')
    file_details => dsi_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
                         marbl_varname=marbl_varname, field_units=units,  &
                         forcing_calendar_name=file_details, id=n)

  case ('DFe River Flux')
    file_details => dfe_riv_flux_file_loc
    call init_monthly_surface_forcing_metadata(file_details)
    call surface_forcing_fields(n)%add_forcing_field(                     &
                         field_source='POP monthly calendar',             &
```

(continues on next page)

(continued from previous page)

```
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, id=n)

    case ('DIC River Flux')
      file_details => dic_riv_flux_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, id=n)

    case ('ALK River Flux')
      file_details => alk_riv_flux_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, id=n)

    case ('DOC River Flux')
      file_details => doc_riv_flux_file_loc
      call init_monthly_surface_forcing_metadata(file_details)
      call surface_forcing_fields(n)%add_forcing_field(                    &
                          field_source='POP monthly calendar',             &
                          marbl_varname=marbl_varname, field_units=units, &
                          forcing_calendar_name=file_details, id=n)

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

  ! All surface forcing fields are 0d; if a 1d field is introduced later,
  ! move this allocate into the select case
  allocate(surface_forcing_fields(n)%field_0d(nx_block, ny_block, nblocks_clinic))

  ! Zero out forcing field. If a 1d field is introduced later, check to see
  ! which of field_0d and field_1d is allocated.
  surface_forcing_fields(n)%field_0d = c0
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, `interior_forcings` is provided through the MARBL interface and `interior_forcing_fields` is a POP construct.

```
allocate(interior_forcing_fields(size(interior_forcings)))

do n=1,size(interior_forcing_fields)
  marbl_varname = interior_forcings(n)%metadata%varname
  units = interior_forcings(n)%metadata%field_units

  var_processed = .false.
  ! Check to see if this forcing field is tracer restoring
```

(continues on next page)

(continued from previous page)

```
  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(restoreable_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_forcing_fields(n)%add_forcing_field(                 &
                   field_source='file_time_invariant',                     &
                   marbl_varname=marbl_varname, field_units=units,         &
                   filename=restore_data_filenames(m),                     &
                   file_varname=restore_data_file_varnames(m),             &
                   id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
        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
    tracer_name = trim(marbl_varname(1:scan(marbl_varname,' ')))
    select case (trim(restore_inv_tau_opt))
      case('const')
        call interior_forcing_fields(n)%add_forcing_field(                 &
                   field_source='const',                                   &
                   marbl_varname=marbl_varname, field_units=units,         &
                   field_constant = restore_inv_tau_const,                 &
                   id=n)
      ! case('shr_stream')
      ! NOT SUPPORTED YET
      ! will require additional namelist variables, and we can consider
      ! reading in one file per tracer instead of using the same mask
      ! for all restoring fields
      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)
```

(continues on next page)

(continued from previous page)

```
    end select
    allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
    var_processed = .true.
  end if

  if (.not.var_processed) then
    select case (trim(interior_forcings(n)%metadata%varname))
      case ('Dust Flux')
        dustflux_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='dust_flux', id=n)
        allocate(interior_forcing_fields(n)%field_0d(nx_block, ny_block, nblocks_
↪clinic))
      case ('PAR Column Fraction')
        PAR_col_frac_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='PAR_col_frac', id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, mcog_nbins,␣
↪nblocks_clinic))
      case ('Surface Shortwave')
        surf_shortwave_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='surf_shortwave', id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, mcog_nbins,␣
↪nblocks_clinic))
      case ('Temperature')
        temperature_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='temperature', id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
      case ('Salinity')
        salinity_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='salinity', id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
      case ('Pressure')
        pressure_ind = n
        call interior_forcing_fields(n)%add_forcing_field(field_source='internal', &
                    marbl_varname=marbl_varname, field_units=units,              &
                    driver_varname='pressure', id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
      case ('Iron Sediment Flux')
        fesedflux_ind = n
        call interior_forcing_fields(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,            &
```

(continues on next page)

(continued from previous page)

```
                        id=n)
        allocate(interior_forcing_fields(n)%field_1d(nx_block, ny_block, km, nblocks_
↪clinic))
      case DEFAULT
        write(err_msg, "(A,1X,A)") trim(interior_forcings(n)%metadata%varname), &
                        'is not a valid interior forcing field name.'
        call document(subname, err_msg)
        call exit_POP(sigAbort, 'Stopping in ' // subname)
    end select
  end if

  ! Zero out field
  if (allocated(interior_forcing_fields(n)%field_0d)) then
    interior_forcing_fields(n)%field_0d = c0
  else
    interior_forcing_fields(n)%field_1d = c0
  end if
end do
```

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, $\mathbf{x}$.

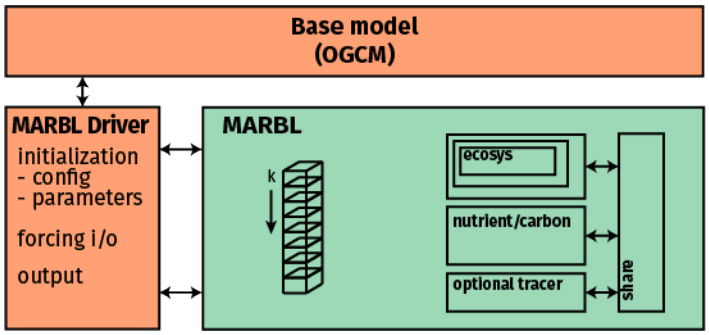
The OGCM computes the lefthand-side of (4.1) (time tendency, advection, diffusion); MARBL's role is to compute the righthand-side ($B_\chi(\mathbf{x})$). MARBL returns this tendency to the OGCM, which steps (4.1) forward in time. MARBL also computes air-sea fluxes for constituents like CO_2; the OGCM 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(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
```

(continues on next page)

(continued from previous page)

```
  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_set_interior_forcing()` (part of `marbl_mod.F90`). The timer calls look like this:

```
subroutine marbl_set_interior_forcing( &
.
.
.
  type    (marbl_internal_timers_type)        , intent(inout) :: marbl_timers
  .
  .
  .
  call marbl_timers%start(marbl_timer_indices%carbonate_chem_id,            &
                          marbl_status_log)
  call marbl_compute_carbonate_chemistry(domain, temperature, pressure,     &
       salinity, tracer_local(:, :), marbl_tracer_indices, carbonate(:),    &
       ph_prev_col(:), ph_prev_alt_co2_col(:), zsat_calcite(:),             &
       zsat_aragonite(:), 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

  .
  .
  .

end subroutine start_timer
```

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

   ! 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_saved_
↪state             ! input/output
   type(marbl_saved_state_type)              , public               :: interior_saved_
↪state             ! input/output
   type(marbl_surface_saved_state_indexing_type), public            :: surf_state_ind
   type(marbl_interior_saved_state_indexing_type), public           :: interior_state_
↪ind
   type(marbl_timers_type)                   , public               :: timer_summary

   ! public data - interior forcing
   real (r8)                                 , public, allocatable  :: column_
↪tracers(:,:)     ! input  *
   real (r8)                                 , public, allocatable  :: column_
↪dtracers(:,:)    ! output *
   type(marbl_interior_forcing_indexing_type), public               :: interior_
↪forcing_ind         !
   type(marbl_forcing_fields_type)           , public, allocatable  :: interior_input_
↪forcings(:)
```

(continues on next page)

(continued from previous page)

```
  type(marbl_diagnostics_type)              , public                :: interior_
↪forcing_diags  ! output

  ! public data surface forcing
  real (r8)                                 , public, allocatable   :: surface_vals(:,
↪:)           ! input  *
  type(marbl_surface_forcing_indexing_type) , public                :: surface_
↪forcing_ind         !
  type(marbl_forcing_fields_type)           , public, allocatable   :: surface_input_
↪forcings(:) ! input  *
  real (r8)                                 , public, allocatable   :: surface_tracer_
↪fluxes(:,:)  ! output *
  type(marbl_surface_forcing_output_type)   , public                :: surface_
↪forcing_output      ! output
  type(marbl_diagnostics_type)              , public                :: surface_
↪forcing_diags       ! output

  ! public data - global averages
  real (r8)                                 , public, allocatable   :: glo_avg_fields_
↪interior(:)   ! output (nfields)
  real (r8)                                 , public, allocatable   :: glo_avg_
↪averages_interior(:) ! input (nfields)
  real (r8)                                 , public, allocatable   :: glo_avg_fields_
↪surface(:,:)  ! output (num_elements,nfields)
  real (r8)                                 , public, allocatable   :: glo_avg_
↪averages_surface(:)  ! 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(:)
  real (r8)                                 , public, allocatable   :: glo_scalar_
↪surface(:)

  type(marbl_running_mean_0d_type)          , public, allocatable   :: glo_avg_rmean_
↪interior(:)
  type(marbl_running_mean_0d_type)          , public, allocatable   :: glo_avg_rmean_
↪surface(:)
  type(marbl_running_mean_0d_type)          , public, allocatable   :: glo_scalar_
↪rmean_interior(:)
  type(marbl_running_mean_0d_type)          , public, allocatable   :: glo_scalar_
↪rmean_surface(:)

  ! private data
  type(marbl_PAR_type)                      , private               :: PAR
  type(marbl_particulate_share_type)        , private               :: particulate_
↪share
  type(marbl_surface_forcing_share_type)    , private               :: surface_
↪forcing_share
  type(marbl_surface_forcing_internal_type) , private               :: surface_
↪forcing_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
```

(continues on next page)

(continued from previous page)

```
   procedure, public  :: init
   procedure, public  :: reset_timers
   procedure, public  :: extract_timing
   procedure, private :: glo_vars_init
   procedure, public  :: get_tracer_index
   procedure, public  :: set_interior_forcing
   procedure, public  :: set_surface_forcing
   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
   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, all of which are made in `marbl_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_forcing_diagnostics_indexing_type` and `marbl_interior_forcing_diagnostics_indexing_type`. `insitu_temp` is an interior forcing diagnostic.

```
type, private :: marbl_interior_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_surface_forcing_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 log_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_forcing_diags%diags, &
     ind   => marbl_interior_diag_ind &
     )
```

(continues on next page)

(continued from previous page)

```
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_set_interior_forcing()`. 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_set_surface_forcing()`.)

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.

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

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 :: &
     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)
     parm_kappa_nitrif_per_day,  & ! nitrification inverse time constant (1/day)
     .
     .
     .
```

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:

```
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_define_parameters(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
  parm_kappa_nitrif_per_day     = 0.06_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
# 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
```

(continues on next page)

(continued from previous page)

```
#              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                      !
!-----------------------------------------------------------------------------

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
.
.
```

(continues on next page)

(continued from previous page)

```
.
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, autotrophs, &
           zooplankton, 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_mod::marbl_ciso_init_tracer_metadata`.)

```
private :: marbl_init_non_autotroph_tracer_metadata
private :: marbl_init_non_autotroph_tracers_metadata
private :: marbl_init_zooplankton_tracer_metadata
private :: 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_mod::marbl_set_surface_forcing`:

```
subroutine marbl_set_surface_forcing( &
.
.
.
  associate(                                                                    ⌐
↪             &
  .
  .
  .
       stf                    => surface_tracer_fluxes(:,:),                    ⌐
↪             &
       .
       .
       .
       o2_ind             => marbl_tracer_indices%o2_ind,                      ⌐
↪          &
       .
       .
       .
       )

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

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

(continues on next page)

(continued from previous page)

```
  !-----------------------------------------------------------------------

  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(:) - surface_vals(:, o2_ind))
     stf(:, o2_ind) = stf(:, o2_ind) + flux_o2_loc(:)
```

Note: This subroutine will be renamed `marbl_compute_surface_fluxes` to better reflect what the code is doing.

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_mod::set_interior_forcing`:

```
subroutine marbl_set_interior_forcing( &
.
.
.
  call marbl_compute_PAR(domain, interior_forcings, interior_forcing_indices, &
                         autotroph_cnt, totalChl_local, PAR)

  do k = 1, km
  .
  .
  .
     call marbl_compute_autotroph_uptake(autotroph_cnt, autotrophs,  &
          tracer_local(:, k), marbl_tracer_indices,                  &
          autotroph_secondary_species(:, k))
  .
  .
  .
     call marbl_compute_denitrif(tracer_local(o2_ind, k), tracer_local(no3_ind, k), &
          dissolved_organic_matter(k)%DOC_remin, &
          dissolved_organic_matter(k)%DOCr_remin, &
          POC%remin(k), other_remin(k), sed_denitrif(k), denitrif(k))

     call marbl_compute_dtracer_local (autotroph_cnt, zooplankton_cnt,      &
          autotrophs, zooplankton,           &
          autotroph_secondary_species(:, k), &
          zooplankton_secondary_species(:, k), &
          dissolved_organic_matter(k), &
          nitrif(k), denitrif(k), sed_denitrif(k), &
          Fe_scavenge(k), Lig_prod(k), Lig_loss(k), &
```

(continues on next page)

(continued from previous page)

```
          P_iron%remin(k), POC%remin(k), POP%remin(k), &
          P_SiO2%remin(k), P_CaCO3%remin(k), P_CaCO3_ALT_CO2%remin(k), &
          other_remin(k), PON_remin(k), &
          interior_restore(:, k), &
          tracer_local(o2_ind, k), &
          o2_production(k), o2_consumption(k), &
          dtracers(:, k), marbl_tracer_indices )
  .
  .
  .
  end do
```

The tendencies are combined in `marbl_compute_dtracer_local` while subroutines like `marbl_compute_PAR`, `marbl_compute_autotroph_uptake`, and `marbl_compute_denitrif` are the per-process computations. So you will need to update `marbl_compute_dtracer_local` to compute the tracer tendency for your new tracer correctly:

```
subroutine marbl_compute_dtracer_local (auto_cnt, zoo_cnt, autotrophs,       &
           zooplankton, autotroph_secondary_species,                  &
           zooplankton_secondary_species, 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, interior_restore,   &
           O2_loc, o2_production, o2_consumption, dtracers, marbl_tracer_indices)
.
.
.
  associate(                                                                  &
  .
  .
  .
       o2_ind            => marbl_tracer_indices%o2_ind,          &
       .
       .
       .
  )
  .
  .
  .
  o2_consumption = (O2_loc - parm_o2_min) / parm_o2_min_delta
  o2_consumption = min(max(o2_consumption, c0), c1)
  o2_consumption = o2_consumption * ( (POC_remin * (c1 - POCremin_refract) + DOC_
↪remin &
       + DOCr_remin - (sed_denitrif * denitrif_C_N) - other_remin + sum(zoo_loss_
↪dic(:)) &
       + sum(zoo_graze_dic(:)) + sum(auto_loss_dic(:)) + sum(auto_graze_dic(:)) ) &
       / parm_Remin_D_C_O2 + (c2 * nitrif))

  dtracers(o2_ind) = o2_production - o2_consumption
```

Note:

1. This subroutine will be renamed `marbl_compute_interior_tendencies` to better reflect what the code is doing.
2. The `k` loop in the example may be removed in favor of doing per-process computations on an entire column at

once.

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.

4.4 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]$ 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

```
$ 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]$ 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/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
--------
```

(continues on next page)

(continued from previous page)

```
~~~~~~~~~~
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.