
seaice3p

Release 0.15.0

Joseph Fishlock

Sep 18, 2024

CONTENTS:

1	seaice3p	1
1.1	seaice3p package	1
2	Indices and tables	43
	Python Module Index	45
	Index	47

SEAICE3P

1.1 seaice3p package

1.1.1 Subpackages

seaice3p.diagnostics package

Submodules

seaice3p.diagnostics.brine_drainage_parameterisation module

seaice3p.diagnostics.brine_drainage_parameterisation.**main**(*output_dir: Path*)

Module contents

seaice3p.enthalpy_method package

Submodules

seaice3p.enthalpy_method.common module

seaice3p.enthalpy_method.common.**calculate_common_enthalpy_method_vars**(*state: EQMState | DISEQState, cfg: Config, phase_masks*)
→ Tuple[ndarray[Any, dtype[_ScalarType_co]], ndarray[Any, dtype[_ScalarType_co]], ndarray[Any, dtype[_ScalarType_co]], ndarray[Any, dtype[_ScalarType_co]], ndarray[Any, dtype[_ScalarType_co]]]

seaice3p.enthalpy_method.enthalpy_method module

Module containing enthalpy method to calculate state variables from bulk enthalpy, bulk salinity and bulk gas.

seaice3p.enthalpy_method.enthalpy_method.**get_enthalpy_method**(*cfg*: [Config](#)) → Callable[[[EQMState](#) | [DISEQState](#)], [EQMStateFull](#) | [DISEQStateFull](#)]

seaice3p.enthalpy_method.gas module

seaice3p.enthalpy_method.gas.**calculate_DISEQ_dissolved_gas**(*state*: [DISEQState](#), *liquid_fraction*, *physical_params*: [EQMPhysicalParams](#) | [DISEQPhysicalParams](#), *phase_masks*) → ndarray[Any, dtype[_ScalarType_co]]

seaice3p.enthalpy_method.gas.**calculate_EQM_dissolved_gas**(*state*: [EQMState](#), *liquid_fraction*, *physical_params*: [EQMPhysicalParams](#) | [DISEQPhysicalParams](#)) → ndarray[Any, dtype[_ScalarType_co]]

seaice3p.enthalpy_method.gas.**calculate_EQM_gas_fraction**(*state*: [EQMState](#), *liquid_fraction*: ndarray[Any, dtype[_ScalarType_co]], *physical_params*: [EQMPhysicalParams](#) | [DISEQPhysicalParams](#)) → ndarray[Any, dtype[_ScalarType_co]]

seaice3p.enthalpy_method.phase_boundaries module

Module for calculating the phase boundaries needed for the enthalpy method.

calculates the phase boundaries neglecting the gas fraction so that

$$\phi_s + \phi_l = 1$$

seaice3p.enthalpy_method.phase_boundaries.**get_phase_masks**(*state*: [EQMState](#) | [DISEQState](#), *physical_params*: [EQMPhysicalParams](#) | [DISEQPhysicalParams](#))

Module contents

seaice3p.equations package

Subpackages

seaice3p.equations.RJW14 package

Submodules

seaice3p.equations.RJW14.brine_channel_sink_terms module

seaice3p.equations.RJW14.brine_channel_sink_terms.**get_brine_convection_sink**(*cfg*: [Config](#),
grids: [Grids](#)) →
 Callable[[[EQMStateBCs](#)
 | [DISEQStateBCs](#)],
 ndarray[Any,
 dtype[_ScalarType_co]]]

seaice3p.equations.RJW14.brine_drainage module

Module to calculate the Rees Jones and Worster 2014 parameterisation for brine convection velocity and the strenght of the sink term.

Exports the functions:

calculate_brine_convection_liquid_velocity To be used in velocities module when using brine convection parameterisation.

calculate_brine_channel_sink To be used to add sink terms to conservation equations when using brine convection parameterisation.

seaice3p.equations.RJW14.brine_drainage.**calculate_Rayleigh**(*cell_centers*, *edge_grid*, *liquid_salinity*,
liquid_fraction, *cfg*: [Config](#))

Calculate the local Rayleigh number for brine convection as

$$\text{Ra}(z) = \text{Ra}_S K(z)(z + h)\Theta_l$$

Parameters

- **cell_centers** (*Numpy Array shape (I,)*) – The vertical coordinates of cell centers.
- **edge_grid** (*Numpy Array (size I+1)*) – The vertical coordinate positions of the edge grid.
- **liquid_salinity** (*Numpy Array shape (I,)*) – liquid salinity on center grid
- **liquid_fraction** (*Numpy Array (size I)*) – liquid fraction on center grid
- **cfg** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Array of shape (I,) of Rayleigh number at cell centers

seaice3p.equations.RJW14.brine_drainage.**calculate_brine_channel_sink**(*liquid_fraction*,
liquid_salinity,
center_grid, *edge_grid*,
cfg: [Config](#))

Calculate the sink term due to brine channels.

$$\text{sink} = \mathcal{A}$$

in the convecting region. Zero elsewhere.

NOTE: If no ice is present or if no convecting region exists returns zero

Parameters

- **liquid_fraction** (*Numpy Array of shape (I,)*) – liquid fraction on center grid
- **liquid_salinity** (*Numpy Array of shape (I,)*) – liquid salinity on center grid
- **center_grid** (*Numpy Array of shape (I,)*) – vertical coordinate of center grid
- **edge_grid** (*Numpy Array of shape (I+1,)*) – Vertical coordinates of cell edges
- **cfg** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Strength of the sink term due to brine channels on the center grid.

seaice3p.equations.RJW14.brine_drainage.**calculate_brine_channel_strength**(*Rayleigh_number*,
ice_depth, *convecting_region_height*,
cfg: [Config](#))

Calculate the brine channel strength in the convecting region as

$$\mathcal{A} = \frac{\alpha \text{Ra}_e}{(h + z_c)^2}$$

the effective Rayleigh number multiplied by a tuning parameter (Rees Jones and Worster 2014) over the convecting region thickness squared.

Parameters

- **Rayleigh_number** (*Numpy Array of shape (I,)*) – local Rayleigh number on center grid
- **ice_depth** (*float*) – depth of ice (positive)
- **convecting_region_height** (*float*) – position of the convecting region boundary (negative)
- **cfg** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Brine channel strength parameter

seaice3p.equations.RJW14.brine_drainage.**calculate_brine_convection_liquid_velocity**(*liquid_fraction*,
liquid_salinity,
center_grid,
edge_grid,
cfg:
[Config](#))

Calculate the vertical liquid Darcy velocity from Rees Jones and Worster 2014

$$W_l = \mathcal{A}(z_c - z)$$

in the convecting region. The velocity is stagnant above the convecting region. The velocity is constant in the liquid region and continuous at the interface.

NOTE: If no ice is present or if no convecting region exists returns zero velocity

Parameters

- **liquid_fraction** (*Numpy Array of shape (I,)*) – liquid fraction on center grid
- **liquid_salinity** (*Numpy Array of shape (I,)*) – liquid salinity on center grid

- **center_grid** (*Numpy Array of shape (I,)*) – vertical coordinate of center grid
- **edge_grid** (*Numpy Array of shape (I+1,)*) – Vertical coordinates of cell edges
- **cfig** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Liquid darcy velocity on the edge grid.

`seaice3p.equations.RJW14.brine_drainage.calculate_integrated_mean_permeability(z, liquid_fraction, ice_depth, cell_centers, cfig: Config)`

Calculate the harmonic mean permeability from the base of the ice up to the cell containing the specified z value using the expression of ReesJones2014.

$$K(z) = \left(\frac{1}{h+z} \int_{-h}^z \frac{1}{\Pi(\phi_l(z'))} dz' \right)^{-1}$$

Parameters

- **z** (*float*) – height to integrate permeability up to
- **liquid_fraction** (*Numpy Array shape (I,)*) – liquid fraction on the center grid
- **ice_depth** (*float*) – positive depth position of ice ocean interface
- **cell_centers** (*Numpy Array of shape (I,)*) – cell center positions
- **cfig** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

permeability averaged from base of the ice up to given z value

`seaice3p.equations.RJW14.brine_drainage.calculate_permeability(liquid_fraction, cfig: Config)`

Calculate the absolute permeability as a function of liquid fraction

$$\Pi(\phi_l) = \phi_l^3$$

Alternatively if the porosity threshold flag is true

$$\Pi(\phi_l) = \phi_l^2(\phi_l - \phi_c)$$

Parameters

- **liquid_fraction** (*Numpy Array*) – liquid fraction
- **cfig** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

permeability on the same grid as liquid fraction

`seaice3p.equations.RJW14.brine_drainage.get_convecting_region_height(Rayleigh_number, edge_grid, cfig: Config)`

Calculate the height of the convecting region as the top edge of the highest cell in the domain for which the quantity

$$\text{Ra}(z) - \text{Ra}_c$$

is greater than or equal to zero.

NOTE: if no convecting region exists return np.NaN

Parameters

- **Rayleigh_number** (*Numpy Array of shape (I,)*) – local rayleigh number on center grid
- **edge_grid** (*Numpy Array (size I+1)*) – The vertical coordinate positions of the edge grid.
- **cfg** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Edge grid value at convecting boundary.

`seaice3p.equations.RJW14.brine_drainage.get_effective_Rayleigh_number(Rayleigh_number, cfg: Config)`

Calculate the effective Rayleigh Number as the maximum of

$$Ra(z) - Ra_c$$

in the convecting region.

NOTE: if no convecting region exists returns 0.

Parameters

- **Rayleigh_number** (*Numpy Array of shape (I,)*) – local rayleigh number on center grid
- **cfg** (*seaice3p.params.Config*) – Configuration object for the simulation.

Returns

Effective Rayleigh number.

Module contents

Module to calculate the sink terms for conservation equations when using the Rees Jones and Worster 2014 brine drainage parameterisation.

These terms represent loss through the brine channels and need to be added in the convecting region when using this parameterisation

seaice3p.equations.flux package

Submodules

seaice3p.equations.flux.bulk_dissolved_gas_flux module

calculate the flux terms for the dissolved gas equation in DISEQ model

`seaice3p.equations.flux.bulk_dissolved_gas_flux.calculate_bulk_dissolved_gas_flux(state_BCs, Wl, V, D_g, cfg)`

seaice3p.equations.flux.bulk_gas_flux module

`seaice3p.equations.flux.bulk_gas_flux.calculate_advective_dissolved_gas_flux`(*dissolved_gas*,
Wl, *cfg*)

`seaice3p.equations.flux.bulk_gas_flux.calculate_bubble_gas_flux`(*gas_fraction*, *Vg*)

`seaice3p.equations.flux.bulk_gas_flux.calculate_diffusive_gas_flux`(*dissolved_gas*,
liquid_fraction, *D_g*, *cfg*)

`seaice3p.equations.flux.bulk_gas_flux.calculate_frame_advection_gas_flux`(*gas*, *V*)

`seaice3p.equations.flux.bulk_gas_flux.calculate_gas_flux`(*state_BCs*, *Wl*, *V*, *Vg*, *D_g*, *cfg*)

seaice3p.equations.flux.gas_fraction_flux module

Calculate gas phase fluxes for disequilibrium model

`seaice3p.equations.flux.gas_fraction_flux.calculate_gas_fraction_flux`(*state_BCs*, *V*, *Vg*)

seaice3p.equations.flux.heat_flux module

`seaice3p.equations.flux.heat_flux.calculate_advective_heat_flux`(*temperature*, *Wl*)

`seaice3p.equations.flux.heat_flux.calculate_conductive_heat_flux`(*state_BCs*, *D_g*, *cfg*)

Calculate conductive heat flux as

$$-\frac{\partial \theta}{\partial z}$$

or alternatively if the `phase_average_conductivity` configuration parameter is set to `True` then we use the conductivity ratio as follows

$$-[(\phi_l + \lambda \phi_s) \frac{\partial \theta}{\partial z}]$$

Parameters

- **temperature** (*Numpy Array of size I+2*) – temperature including ghost cells
- **D_g** (*Numpy Array*) – difference matrix for ghost grid
- **cfg** (*seaice3p.params.Config*) – Simulation configuration

Returns

conductive heat flux

`seaice3p.equations.flux.heat_flux.calculate_frame_advection_heat_flux`(*enthalpy*, *V*)

`seaice3p.equations.flux.heat_flux.calculate_heat_flux`(*state_BCs*, *Wl*, *V*, *D_g*, *cfg*)

seaice3p.equations.flux.salt_flux module

seaice3p.equations.flux.salt_flux.calculate_advective_salt_flux(*liquid_salinity*, *Wl*, *cfg*)

seaice3p.equations.flux.salt_flux.calculate_diffusive_salt_flux(*liquid_salinity*, *liquid_fraction*,
D_g, *cfg*)

Take liquid salinity and liquid fraction on ghost grid and interpolate liquid fraction geometrically

seaice3p.equations.flux.salt_flux.calculate_frame_advection_salt_flux(*salt*, *V*)

seaice3p.equations.flux.salt_flux.calculate_salt_flux(*state_BCs*, *Wl*, *V*, *D_g*, *cfg*)

Module contents

Module for calculating the fluxes using upwind scheme

seaice3p.equations.flux.get_dz_fluxes(*cfg*: *Config*, *grids*: *Grids*) → Callable[[*EQMStateBCs* |
DISEQStateBCs, ndarray[Any, dtype[_ScalarType_co]],
ndarray[Any, dtype[_ScalarType_co]], ndarray[Any,
dtype[_ScalarType_co]], ndarray[Any, dtype[_ScalarType_co]]]

seaice3p.equations.velocities package

Submodules

seaice3p.equations.velocities.bubble_parameters module

seaice3p.equations.velocities.bubble_parameters.calculate_bubble_size_fraction(*bubble_radius_scaled*,
*liq-
uid_fraction*,
cfg: *Config*)

Takes bubble radius scaled and liquid fraction on edges and calculates the bubble size fraction as

$$\lambda = \Lambda / (\phi_l^q + \text{reg})$$

Returns the bubble size fraction on the edge grid.

seaice3p.equations.velocities.mono_distribution module

seaice3p.equations.velocities.mono_distribution.calculate_lag_function(*bubble_size_fraction*)

Calculate lag function from bubble size fraction on edge grid as

$$G(\lambda) = 1 - \lambda/2$$

for $0 < \lambda < 1$. Edge cases are given by $G(0)=1$ and $G(1) = 0.5$ for values outside this range.

seaice3p.equations.velocities.mono_distribution.calculate_mono_lag_factor(*liquid_fraction*, *cfg*:
Config)

Take liquid fraction on the ghost grid and calculate the lag factor for a mono bubble size distribution as

$$I_2 = G(\lambda)$$

returns lag factor on the edge grid

`seaice3p.equations.velocities.mono_distribution.calculate_mono_wall_drag_factor`(*liquid_fraction*,
cfg:
 Config)

Take liquid fraction on the ghost grid and calculate the wall drag factor for a mono bubble size distribution as

$$I_1 = \frac{\lambda^2}{K(\lambda)}$$

returns wall drag factor on the edge grid

`seaice3p.equations.velocities.mono_distribution.calculate_wall_drag_function`(*bubble_size_fraction*,
cfg: Config)

Calculate wall drag function from bubble size fraction on edge grid as

$$\frac{1}{K(\lambda)} = (1 - \lambda)^r$$

in the power law case or in the Haberman case from the paper

$$\frac{1}{K(\lambda)} = \frac{1 - 1.5\lambda + 1.5\lambda^5 - \lambda^6}{1 + 1.5\lambda^5}$$

for $0 < \lambda < 1$. Edge cases are given by $K(0)=1$ and $K(1) = 0$ for values outside this range.

seaice3p.equations.velocities.power_law_distribution module

`seaice3p.equations.velocities.power_law_distribution.calculate_lag_integral`(*bubble_size_fraction_min*:
 float, *bubble_size_fraction_max*:
 float, *cfg*:
 Config)

`seaice3p.equations.velocities.power_law_distribution.calculate_lag_integrand`(*bubble_size_fraction*:
 float, *cfg*:
 Config)

Scalar function to calculate lag integrand for polydispersive case.

Bubble size fraction is given as a scalar input to calculate

$$\lambda^{3-p}G(\lambda)$$

`seaice3p.equations.velocities.power_law_distribution.calculate_power_law_lag_factor`(*liquid_fraction*,
cfg:
 Config)

Take liquid fraction on the ghost grid and calculate the lag factor for power law bubble size distribution.

Return on edge grid

`seaice3p.equations.velocities.power_law_distribution.calculate_power_law_wall_drag_factor`(*liquid_fraction*,
cfg:
 Config)

Take liquid fraction on the ghost grid and calculate the wall drag factor for power law bubble size distribution.

Return on edge grid

`seaice3p.equations.velocities.power_law_distribution.calculate_volume_integrand`(*bubble_size_fraction*: float, *cfg*: Config)

Scalar function to calculate the integrand for volume under a power law bubble size distribution given as

$$\lambda^{3-p}$$

in terms of the bubble size fraction.

`seaice3p.equations.velocities.power_law_distribution.calculate_wall_drag_integral`(*bubble_size_fraction_min*: float, *bubble_size_fraction_max*: float, *cfg*: Config)

`seaice3p.equations.velocities.power_law_distribution.calculate_wall_drag_integrand`(*bubble_size_fraction*: float, *cfg*: Config)

Scalar function to calculate wall drag integrand for polydispersive case.

Bubble size fraction is given as a scalar input to calculate

$$\frac{\lambda^{5-p}}{K(\lambda)}$$

where the wall drag enhancement function K can be given by a power law fit or taken from the Haberman paper.

seaice3p.equations.velocities.velocities module

`seaice3p.equations.velocities.velocities.calculate_frame_velocity`(*cfg*: Config)

`seaice3p.equations.velocities.velocities.calculate_gas_interstitial_velocity`(*liquid_fraction*, *liquid_darcy_velocity*, *wall_drag_factor*, *lag_factor*, *cfg*: Config)

Calculate V_g from liquid fraction on the ghost grid and liquid interstitial velocity

$$V_g = \mathcal{B}(\phi_l^{2q} I_1) + U_0 I_2$$

Return V_g on edge grid

`seaice3p.equations.velocities.velocities.calculate_liquid_darcy_velocity`(*liquid_fraction*, *liquid_salinity*, *center_grid*, *edge_grid*, *cfg*: Config)

Calculate liquid Darcy velocity either using brine convection parameterisation or as stagnant

Parameters

- **liquid_fraction** (Numpy Array (size I+2)) – liquid fraction on ghost grid

- **liquid_salinity** (*Numpy Array (size I+2)*) – liquid salinity on ghost grid
- **center_grid** (*Numpy Array of shape (I,)*) – vertical coordinates of cell centers
- **edge_grid** (*Numpy Array (size I+1)*) – Vertical coordinates of cell edges
- **cfg** (*seaice3p.params.Config*) – simulation configuration object

Returns

liquid darcy velocity on edge grid

`seaice3p.equations.velocities.velocities.calculate_velocities(state_BCs, cfg: Config)`

Inputs on ghost grid, outputs on edge grid

needs the simulation config, liquid fraction, liquid salinity and grids

Module contents

Module to calculate Darcy velocities.

The liquid Darcy velocity must be parameterised.

The gas Darcy velocity is calculated as `gas_fraction` x interstitial bubble velocity

Interstitial bubble velocity is found by a steady state Stoke's flow calculation. We have implemented two cases mono: All bubbles nucleate and remain the same size `power_law`: A power law bubble size distribution with fixed max and min.

Submodules**seaice3p.equations.equations module**

`seaice3p.equations.equations.get_equations(cfg: Config, grids) → Callable[[EQMStateBCs | DISEQStateBCs], ndarray[Any, dtype[_ScalarType_co]]]`

seaice3p.equations.nucleation module

`seaice3p.equations.nucleation.get_nucleation(cfg: Config) → Callable[[EQMStateBCs | DISEQStateBCs], ndarray[Any, dtype[_ScalarType_co]]]`

seaice3p.equations.radiative_heating module

Calculate internal shortwave radiative heating due to oil droplets

`seaice3p.equations.radiative_heating.get_radiative_heating(cfg: Config, grids: Grids) → Callable[[EQMStateBCs | DISEQStateBCs], ndarray[Any, dtype[_ScalarType_co]]]`

Calculate internal shortwave heating source for enthalpy equation.

if the `RadForcing` object is given as the forcing config then calculates internal heating based on the object given in the configuration for `oil_heating`.

If another forcing is chosen then just returns a function to create an array of zeros as no internal heating is calculated.

Module contents

seaice3p.forcing package

Subpackages

seaice3p.forcing.surface_energy_balance package

Submodules

seaice3p.forcing.surface_energy_balance.surface_energy_balance module

Module to compute the surface heat flux from geophysical energy balance

following [1]

Refs: [1] P. D. Taylor and D. L. Feltham, ‘A model of melt pond evolution on sea ice’, J. Geophys. Res., vol. 109, no. C12, p. 2004JC002361, Dec. 2004, doi: 10.1029/2004JC002361.

```
seaice3p.forcing.surface_energy_balance.surface_energy_balance.calculate_emissivity(cfg:
    Con-
    fig,
    top_cell_is_ice:
    bool)
    →
    float
```

```
seaice3p.forcing.surface_energy_balance.surface_energy_balance.calculate_total_heat_flux(cfg:
    Con-
    fig,
    time:
    float,
    top_cell_is_ice:
    bool,
    sur-
    face_temp:
    float)
    →
    float
```

Takes non-dimensional surface temperature and returns non-dimensional heat flux

```
seaice3p.forcing.surface_energy_balance.surface_energy_balance.convert_surface_temperature_to_kelvin(cfg:
    Co-
    fig,
    non-
    floo
    →
    floo
```



```

seaice3p.forcing.surface_energy_balance.surface_energy_balance.find_ghost_cell_temperature(state:
EQM-
State-
Full
|
DIS-
E-
QS-
tate-
Full,
cfg:
Con-
fig)
→
float

```

```

seaice3p.forcing.surface_energy_balance.surface_energy_balance.solve_for_surface_temp(cfg:
Con-
fig,
time:
float,
top_cell_solid_fraction:
float,
top_cell_center_temp:
float,
second_cell_center_temp:
float)
→
float

```

Returns non dimensional surface temperature

```

seaice3p.forcing.surface_energy_balance.surface_energy_balance.surface_temp_gradient(cfg:
Con-
fig,
surface_temp:
float,
top_cell_center_temp:
float,
second_cell_center_temp:
float)
→
float

```

Approximate non dimensional temperature gradient using the unknown surface temperature value (top of edge grid) and the top two known temperature values on the center grid

```

seaice3p.forcing.surface_energy_balance.surface_energy_balance.top_cell_conductivity(cfg:
Con-
fig,
solid_fraction:
float)
→
float

```

seaice3p.forcing.surface_energy_balance.turbulent_heat_flux module

Module to compute the turbulent atmospheric sensible and latent heat fluxes

All temperatures are in Kelvin in this module

Refs: [1] P. D. Taylor and D. L. Feltham, 'A model of melt pond evolution on sea ice', J. Geophys. Res., vol. 109, no. C12, p. 2004JC002361, Dec. 2004, doi: 10.1029/2004JC002361.

[2] E. E. Ebert and J. A. Curry, 'An intermediate one-dimensional thermodynamic sea ice model for investigating ice-atmosphere interactions', Journal of Geophysical Research: Oceans, vol. 98, no. C6, pp. 10085–10109, 1993, doi: 10.1029/93JC00656.

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_bulk_transfer_coefficient(cfg:
    Config,
    top_cell_is_ice:
    bool,
    time:
    float,
    surface_temp:
    float)
    →
    float
```

Calculation of bulk transfer coeff from [2]

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_latent_heat_flux(cfg:
    Config,
    time:
    float,
    top_cell_is_ice:
    bool,
    surface_temp:
    float)
    →
    float
```

Calculate latent heat flux from [2]

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_ref_air_temp(cfg:
    Config,
    time:
    float)
    → float
```

return air temperature at reference level above the ice in Kelvin

in the configuration the air temperature is given in deg C

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_ref_atmospheric_pressure(cfg:
Con-
fig,
time:
float)
→
float
```

return atmospheric pressure at reference level above the ice

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_ref_specific_humidity(cfg:
Con-
fig,
time:
float)
→
float
```

return specific humidity at reference level above the ice

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_ref_windspeed(cfg:
Con-
fig,
time:
float)
→
float
```

return windspeed at reference level above the ice

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_sensible_heat_flux(cfg:
Con-
fig,
time:
float,
top_cell_is_ice:
bool,
sur-
face_temp:
float)
→
float
```

Calculate sensible heat flux from [2]

```
seaice3p.forcing.surface_energy_balance.turbulent_heat_flux.calculate_surface_specific_humidity(cfg:
Con-
fig,
time:
float,
sur-
face_temp:
float)
→
float
```

Following expression given in [1]

Module contents

Submodules

seaice3p.forcing.boundary_conditions module

Module to provide functions to add boundary conditions to each quantity on the centered grid that needs to be on the ghost grid for the upwind scheme.

```
seaice3p.forcing.boundary_conditions.get_boundary_conditions(cfg: Config) →  
    Callable[[EQMStateFull |  
             DISEQStateFull], EQMStateBCs |  
             DISEQStateBCs]
```

seaice3p.forcing.radiative_forcing module

Module for providing surface radiative forcing to simulation.

Currently only total surface shortwave irradiance (integrated over entire shortwave part of the spectrum) is provided and this is used to calculate internal radiative heating.

Unlike temperature forcing this provides dimensional forcing

```
seaice3p.forcing.radiative_forcing.get_LW_forcing(time: float, cfg: Config) → float
```

```
seaice3p.forcing.radiative_forcing.get_SW_forcing(time, cfg: Config)
```

seaice3p.forcing.temperature_forcing module

Module for providing surface temperature forcing to simulation.

Note that the barrow temperature data is read in from a file if needed by the simulation configuration.

```
seaice3p.forcing.temperature_forcing.get_bottom_temperature_forcing(time, cfg: Config)
```

```
seaice3p.forcing.temperature_forcing.get_temperature_forcing(state: EQMStateFull |  
                                                             DISEQStateFull, cfg: Config)
```

Module contents

seaice3p.params package

Subpackages

seaice3p.params.dimensional package

Submodules

seaice3p.params.dimensional.bubble module

```
class seaice3p.params.dimensional.bubble.DimensionaBaseBubbleParams(pore_radius: float =
    0.001,
    pore_throat_scaling: float
    = 0.5, porosity_threshold:
    bool = False,
    porosity_threshold_value:
    float = 0.024,
    escape_ice_surface: bool
    = True)
```

Bases: object

escape_ice_surface: bool = True

pore_radius: float = 0.001

pore_throat_scaling: float = 0.5

porosity_threshold: bool = False

porosity_threshold_value: float = 0.024

```
class seaice3p.params.dimensional.bubble.DimensionaMonoBubbleParams(pore_radius: float =
    0.001,
    pore_throat_scaling: float
    = 0.5, porosity_threshold:
    bool = False,
    porosity_threshold_value:
    float = 0.024,
    escape_ice_surface: bool
    = True, bubble_radius:
    float = 0.001)
```

Bases: *DimensionaBaseBubbleParams*

bubble_radius: float = 0.001

property bubble_radius_scaled

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B/R_0$$

```
class seaice3p.params.dimensional.bubble.DimensionaPowerLawBubbleParams(pore_radius: float =
    0.001,
    pore_throat_scaling:
    float = 0.5,
    porosity_threshold:
    bool = False, poros-
    ity_threshold_value:
    float = 0.024,
    escape_ice_surface:
    bool = True, bub-
    ble_distribution_power:
    float = 1.5, mini-
    mum_bubble_radius:
    float = 1e-06, maxi-
    mum_bubble_radius:
    float = 0.001)
```

Bases: *DimensionalBaseBubbleParams*

bubble_distribution_power: float = 1.5

maximum_bubble_radius: float = 0.001

property maximum_bubble_radius_scaled

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B/R_0$$

minimum_bubble_radius: float = 1e-06

property minimum_bubble_radius_scaled

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B/R_0$$

seai3p.params.dimensional.convection module

```
class seai3p.params.dimensional.convection.DimensionalRJW14Params(couple_bubble_to_horizontal_flow:
    bool = False, couple_bubble_to_vertical_flow:
    bool = False,
    Rayleigh_critical: float =
    2.9, convection_strength:
    float = 0.13, haline_contraction_coefficient:
    float = 0.00075,
    reference_permeability:
    float = 1e-08)
```

Bases: object

Rayleigh_critical: float = 2.9

convection_strength: float = 0.13

couple_bubble_to_horizontal_flow: bool = False

couple_bubble_to_vertical_flow: bool = False

haline_contraction_coefficient: float = 0.00075

reference_permeability: float = 1e-08

```
class seai3p.params.dimensional.convection.NoBrineConvection
```

Bases: object

No brine convection

seaice3p.params.dimensionals.dimensionals module

Dimensional parameters required to run a simulation and convert output to dimensional variables.

The DimensionalParams class contains all the dimensional parameters needed to produce a simulation configuration.

The Scales class contains all the dimensional parameters required to convert simulation output between physical and non-dimensional variables.

```
class seaice3p.params.dimensionals.dimensionals.DimensionalParams(name: str, total_time_in_days:
    float, savefreq_in_days: float,
    lengthscale: float, gas_params:
    DimensionalEQMGasParams |
    DimensionalDISEQGasParams,
    bubble_params:
    DimensionalMonoBubbleParams
    | DimensionalPowerLawBubbleParams,
    brine_convection_params:
    DimensionalRJW14Params |
    NoBrineConvection,
    forcing_config:
    DimensionalRadForcing |
    DimensionalBRW09Forcing |
    DimensionalConstantForcing |
    DimensionalYearlyForcing,
    initial_conditions_config:
    DimensionalOilInitialConditions
    | UniformInitialConditions |
    BRW09InitialConditions,
    water_params:
    DimensionalWaterParams =
    DimensionalWater-
    Params(liquid_density=1028,
    ocean_salinity=34,
    eutectic_salinity=270,
    eutectic_temperature=-21.1,
    ocean_temperature=-0.81,
    latent_heat=334000.0,
    specific_heat_capacity=4184,
    phase_average_conductivity=False,
    liq-
    uid_thermal_conductivity=0.54,
    solid_thermal_conductivity=2.22,
    salt_diffusivity=0,
    liquid_viscosity=0.00278),
    numerical_params:
    NumericalParams =
    NumericalParams(I=50,
    regularisation=1e-06),
    frame_velocity_dimensional:
    float = 0, gravity: float = 9.81)
```

Bases: object

Contains all dimensional parameters needed to calculate non dimensional numbers.

To see the units each input should have look at the comment next to the default value.

property B

calculate the non dimensional scale for buoyant rise of gas bubbles as

$$\mathcal{B} = \frac{\rho_l g R_0^2 h}{3\mu\kappa}$$

property Rayleigh_salt

Calculate the haline Rayleigh number as

$$\text{Ra}_S = \frac{\rho_l g \beta \Delta S H K_0}{\kappa \mu}$$

brine_convection_params: *DimensionalRJW14Params* | *NoBrineConvection*

bubble_params: *DimensionalMonoBubbleParams* | *DimensionalPowerLawBubbleParams*

property damkohler_number

Return damkohler number as ratio of thermal timescale to nucleation timescale

property expansion_coefficient

calculate

$$\chi = \rho_l \xi_{\text{sat}} / \rho_g$$

forcing_config: *DimensionalRadForcing* | *DimensionalBRW09Forcing* | *DimensionalConstantForcing* | *DimensionalYearlyForcing*

property frame_velocity

calculate the frame velocity in non dimensional units

frame_velocity_dimensional: float = 0

gas_params: *DimensionalEQMGasParams* | *DimensionalDISEQGasParams*

gravity: float = 9.81

initial_conditions_config: *DimensionalOilInitialConditions* | *UniformInitialConditions* | *BRW09InitialConditions*

lengthscale: float

property lewis_gas

Calculate the lewis number for dissolved gas, return np.inf if there is no dissolved gas diffusion.

$$\text{Le}_\xi = \kappa / D_\xi$$

classmethod load(path)

load this object from a yaml configuration file.

name: str

numerical_params: *NumericalParams* = NumericalParams(I=50, regularisation=1e-06)

save(*directory: Path*)

save this object to a yaml file in the specified directory.

The name will be the name given with `_dimensional` appended to distinguish it from a saved non-dimensional configuration.

property savefreq

calculate the save frequency in non dimensional time

savefreq_in_days: float

property scales

return a Scales object used for converting between dimensional and non dimensional variables.

property total_time

calculate the total time in non dimensional units for the simulation

total_time_in_days: float

water_params: *DimensionalWaterParams* = DimensionalWaterParams(liquid_density=1028, ocean_salinity=34, eutectic_salinity=270, eutectic_temperature=-21.1, ocean_temperature=-0.81, latent_heat=334000.0, specific_heat_capacity=4184, phase_average_conductivity=False, liquid_thermal_conductivity=0.54, solid_thermal_conductivity=2.22, salt_diffusivity=0, liquid_viscosity=0.00278)

seaice3p.params.dimensional.forcing module

class seaice3p.params.dimensional.forcing.DimensionalBRW09Forcing(*Barrow_top_temperature_data_choice: str = 'air'*)

Bases: object

Barrow_top_temperature_data_choice: str = 'air'

class seaice3p.params.dimensional.forcing.DimensionalBackgroundOilHeating(*oil_mass_ratio: float = 0, ice_type: str = 'FYI'*)

Bases: object

ice_type: str = 'FYI'

oil_mass_ratio: float = 0

class seaice3p.params.dimensional.forcing.DimensionalConstantForcing(*constant_top_temperature: float = -30.32*)

Bases: object

constant_top_temperature: float = -30.32

class seaice3p.params.dimensional.forcing.DimensionalConstantLWForcing(*LW_irradiance: float = 260, ice_emissivity: float = 0.99, water_emissivity: float = 0.97*)

Bases: object

LW_irradiance: float = 260

ice_emissivity: float = 0.99

water_emissivity: float = 0.97

class seaice3p.params.dimensional.forcing.**DimensionalConstantSWForcing**(*SW_irradiance: float = 280, SW_albedo: float = 0.7, SW_penetration_fraction: float = 0.4*)

Bases: object

SW_albedo: float = 0.7

SW_irradiance: float = 280

SW_penetration_fraction: float = 0.4

class seaice3p.params.dimensional.forcing.**DimensionalConstantTurbulentFlux**(*ref_height: float = 10, windspeed: float = 5, air_temp: float = 0, specific_humidity: float = 0.0036, atm_pressure: float = 101.325, air_density: float = 1.275, air_heat_capacity: float = 1005, air_latent_heat_of_vaporisation: float = 2501000.0*)

Bases: object

air_density: float = 1.275

air_heat_capacity: float = 1005

air_latent_heat_of_vaporisation: float = 2501000.0

air_temp: float = 0

atm_pressure: float = 101.325

ref_height: float = 10

specific_humidity: float = 0.0036

windspeed: float = 5

seaice3p.params.dimensional.forcing.**DimensionalLWForcing**

alias of *DimensionalConstantLWForcing*

class seaice3p.params.dimensional.forcing.**DimensionalMobileOilHeating**(*ice_type: str = 'FYI'*)

Bases: object

```

    ice_type: str = 'FYI'

class seaice3p.params.dimensional.forcing.DimensionaNoHeating
    Bases: object

class seaice3p.params.dimensional.forcing.DimensionaRadForcing(SW_forcing:
    seaice3p.params.dimensionaForcing.Dimensiona
    = DimensionaConstantSWForc-
    ing(SW_irradiance=280,
    SW_albedo=0.7,
    SW_penetration_fraction=0.4),
    LW_forcing:
    seaice3p.params.dimensionaForcing.Dimensiona
    = DimensionaConstantLWForc-
    ing(LW_irradiance=260,
    ice_emissivity=0.99,
    water_emissivity=0.97),
    turbulent_flux:
    seaice3p.params.dimensionaForcing.Dimensiona
    = DimensionaConstantTurbu-
    lentFlux(ref_height=10,
    windspeed=5, air_temp=0,
    specific_humidity=0.0036,
    atm_pressure=101.325,
    air_density=1.275,
    air_heat_capacity=1005,
    air_latent_heat_of_vaporisation=2501000.0),
    oil_heating:
    seaice3p.params.dimensionaForcing.Dimensiona
    |
    seaice3p.params.dimensionaForcing.Dimensiona
    |
    seaice3p.params.dimensionaForcing.Dimensiona
    = DimensionaBackgroundOil-
    Heating(oil_mass_ratio=0,
    ice_type='FYI'))

    Bases: object

    LW_forcing: DimensionaConstantLWForcing =
    DimensionaConstantLWForcing(LW_irradiance=260, ice_emissivity=0.99,
    water_emissivity=0.97)

    SW_forcing: DimensionaConstantSWForcing =
    DimensionaConstantSWForcing(SW_irradiance=280, SW_albedo=0.7,
    SW_penetration_fraction=0.4)

    oil_heating: DimensionaBackgroundOilHeating | DimensionaMobileOilHeating |
    DimensionaNoHeating = DimensionaBackgroundOilHeating(oil_mass_ratio=0,
    ice_type='FYI')

    turbulent_flux: DimensionaConstantTurbulentFlux =
    DimensionaConstantTurbulentFlux(ref_height=10, windspeed=5, air_temp=0,
    specific_humidity=0.0036, atm_pressure=101.325, air_density=1.275,
    air_heat_capacity=1005, air_latent_heat_of_vaporisation=2501000.0)

```

seaice3p.params.dimensional.forcing.**DimensionalSWForcing**

alias of *DimensionalConstantSWForcing*

seaice3p.params.dimensional.forcing.**DimensionalTurbulentFlux**

alias of *DimensionalConstantTurbulentFlux*

class seaice3p.params.dimensional.forcing.**DimensionalYearlyForcing**(*offset: float = -1.0,*
amplitude: float = 0.75,
period: float = 4.0)

Bases: object

amplitude: float = 0.75

offset: float = -1.0

period: float = 4.0

seaice3p.params.dimensional.gas module

class seaice3p.params.dimensional.gas.**DimensionalDISEQGasParams**(*gas_density: float = 1,*
saturation_concentration: float = 1e-05, ocean_saturation_state: float = 1.0, gas_diffusivity: float = 0, tolerable_super_saturation_fraction: float = 1, nucleation_timescale: float = 6869075)

Bases: *DimensionalEQMGasParams*

nucleation_timescale: float = 6869075

class seaice3p.params.dimensional.gas.**DimensionalEQMGasParams**(*gas_density: float = 1,*
saturation_concentration: float = 1e-05, ocean_saturation_state: float = 1.0, gas_diffusivity: float = 0, tolerable_super_saturation_fraction: float = 1)

Bases: object

gas_density: float = 1

gas_diffusivity: float = 0

ocean_saturation_state: float = 1.0

saturation_concentration: float = 1e-05

tolerable_super_saturation_fraction: float = 1

seaice3p.params.dimensional.initial_conditions module

```
class seaice3p.params.dimensional.initial_conditions.BRW09InitialConditions(Barrow_initial_bulk_gas_in_ice:  
                                                                           float = 0.2)
```

Bases: object

values for bottom (ocean) boundary

Barrow_initial_bulk_gas_in_ice: float = 0.2

```
class seaice3p.params.dimensional.initial_conditions.DimensionalOilInitialConditions(initial_ice_depth:  
                                                                                   float  
                                                                                   =  
                                                                                   1, ini-  
                                                                                   tial_ocean_temperature  
                                                                                   float  
                                                                                   = -2,  
                                                                                   ini-  
                                                                                   tial_ice_temperature:  
                                                                                   float  
                                                                                   = -4,  
                                                                                   ini-  
                                                                                   tial_oil_volume_fraction  
                                                                                   float  
                                                                                   = 1e-  
                                                                                   07,  
                                                                                   ini-  
                                                                                   tial_ice_bulk_salinity:  
                                                                                   float  
                                                                                   =  
                                                                                   5.92)
```

Bases: object

initial_ice_bulk_salinity: float = 5.92

initial_ice_depth: float = 1

initial_ice_temperature: float = -4

initial_ocean_temperature: float = -2

initial_oil_volume_fraction: float = 1e-07

```
class seaice3p.params.dimensional.initial_conditions.UniformInitialConditions
```

Bases: object

values for bottom (ocean) boundary

seaice3p.params.dimension.numerical module

```
class seaice3p.params.dimension.numerical.NumericalParams(I: int = 50, regularisation: float = 1e-06)
```

Bases: object

parameters needed for discretisation and choice of numerical method

I: int = 50

regularisation: float = 1e-06

property step

seaice3p.params.dimension.water module

```
class seaice3p.params.dimension.water.DimensionWaterParams(liquid_density: float = 1028,
    ocean_salinity: float = 34,
    eutectic_salinity: float = 270,
    eutectic_temperature: float = -21.1, ocean_temperature: float = -0.81, latent_heat: float = 334000.0, specific_heat_capacity: float = 4184,
    phase_average_conductivity: bool = False,
    liquid_thermal_conductivity: float = 0.54,
    solid_thermal_conductivity: float = 2.22, salt_diffusivity: float = 0,
    liquid_viscosity: float = 0.00278)
```

Bases: object

property concentration_ratio

Calculate concentration ratio as

$$C = S_i / \Delta S$$

property conductivity_ratio

Calculate the ratio of solid to liquid thermal conductivity

$$\lambda = \frac{k_s}{k_l}$$

eutectic_salinity: float = 270

eutectic_temperature: float = -21.1

latent_heat: float = 334000.0

property lewis_salt

Calculate the lewis number for salt, return np.inf if there is no salt diffusion.

$$Le_S = \kappa / D_s$$

liquid_density: float = 1028

liquid_thermal_conductivity: float = 0.54

liquid_viscosity: float = 0.00278

property ocean_freezing_temperature

calculate salinity dependent freezing temperature using liquidus for typical ocean salinity

$$T_i = T_L(S_i) = T_E S_i / S_E$$

ocean_salinity: float = 34

ocean_temperature: float = -0.81

phase_average_conductivity: bool = False

property salinity_difference

calculate difference between eutectic salinity and typical ocean salinity

$$\Delta S = S_E - S_i$$

salt_diffusivity: float = 0

solid_thermal_conductivity: float = 2.22

specific_heat_capacity: float = 4184

property stefan_number

calculate Stefan number

$$St = L / c_p \Delta T$$

property temperature_difference

calculate

$$\Delta T = T_i - T_E$$

property thermal_diffusivity

Return thermal diffusivity in m²/s

$$\kappa = \frac{k}{\rho_l c_p}$$

Module contents

Submodules

seaice3p.params.bubble module

```
class seaice3p.params.bubble.BaseBubbleParams(B: float = 100, pore_throat_scaling: float = 0.46,  
                                              porosity_threshold: bool = False,  
                                              porosity_threshold_value: float = 0.024,  
                                              escape_ice_surface: bool = True)
```

Bases: `object`

Not to be used directly but provides parameters for bubble model in sea ice common to other bubble parameter objects.

B: float = 100

escape_ice_surface: bool = True

pore_throat_scaling: float = 0.46

porosity_threshold: bool = False

porosity_threshold_value: float = 0.024

```
class seaice3p.params.bubble.MonoBubbleParams(B: float = 100, pore_throat_scaling: float = 0.46,  
                                              porosity_threshold: bool = False,  
                                              porosity_threshold_value: float = 0.024,  
                                              escape_ice_surface: bool = True,  
                                              bubble_radius_scaled: float = 1.0)
```

Bases: [*BaseBubbleParams*](#)

Parameters for population of identical spherical bubbles.

bubble_radius_scaled: float = 1.0

```
class seaice3p.params.bubble.PowerLawBubbleParams(B: float = 100, pore_throat_scaling: float = 0.46,  
                                                  porosity_threshold: bool = False,  
                                                  porosity_threshold_value: float = 0.024,  
                                                  escape_ice_surface: bool = True,  
                                                  bubble_distribution_power: float = 1.5,  
                                                  minimum_bubble_radius_scaled: float = 0.001,  
                                                  maximum_bubble_radius_scaled: float = 1)
```

Bases: [*BaseBubbleParams*](#)

Parameters for population of bubbles following a power law size distribution between a minimum and maximum radius.

bubble_distribution_power: float = 1.5

maximum_bubble_radius_scaled: float = 1

minimum_bubble_radius_scaled: float = 0.001

```
seaice3p.params.bubble.get_dimensionless_bubble_params(dimensional_params: DimensionalParams)  
→ MonoBubbleParams | PowerLawBubbleParams
```


seaice3p.params.convection module

```
class seaice3p.params.convection.RJW14Params(Rayleigh_salt: float = 44105, Rayleigh_critical: float = 2.9, convection_strength: float = 0.13, couple_bubble_to_horizontal_flow: bool = False, couple_bubble_to_vertical_flow: bool = False)
```

Bases: object

Parameters for the RJW14 parameterisation of brine convection

Rayleigh_critical: float = 2.9

Rayleigh_salt: float = 44105

convection_strength: float = 0.13

couple_bubble_to_horizontal_flow: bool = False

couple_bubble_to_vertical_flow: bool = False

```
seaice3p.params.convection.get_dimensionless_brine_convection_params(dimensional_params: DimensionalParams) → RJW14Params | NoBrineConvection
```

seaice3p.params.convert module

```
class seaice3p.params.convert.Scales(lengthscale: float, thermal_diffusivity: float, liquid_thermal_conductivity: float, ocean_salinity: float, salinity_difference: float, ocean_freezing_temperature: float, temperature_difference: float, gas_density: float, saturation_concentration: float)
```

Bases: object

convert_dimensional_bulk_air_to_argon_content(*dimensional_bulk_gas*)

Convert kg/m3 of air to micromole of Argon per Liter of ice

convert_from_dimensional_bulk_gas(*dimensional_bulk_gas*)

Non dimensionalise bulk gas content in kg/m3

convert_from_dimensional_bulk_salinity(*dimensional_bulk_salinity*)

Non dimensionalise bulk salinity in g/kg

convert_from_dimensional_dissolved_gas(*dimensional_dissolved_gas*)

convert from dissolved gas in kg(gas)/kg(liquid) to dimensionless

convert_from_dimensional_grid(*dimensional_grid*)

Non dimensionalise domain depths in meters

convert_from_dimensional_heat_flux(*dimensional_heat_flux*)

convert from heat flux in W/m2 to dimensionless units

convert_from_dimensional_heating(*dimensional_heating*)

convert from heating rate in W/m3 to dimensionless units

convert_from_dimensional_temperature(*dimensional_temperature*)

Non dimensionalise temperature in deg C

convert_from_dimensional_time(*dimensional_time*)

Non dimensionalise time in days

convert_to_dimensional_bulk_gas(*bulk_gas*)

Convert dimensionless bulk gas content to kg/m³

convert_to_dimensional_bulk_salinity(*bulk_salinity*)

Convert non dimensional bulk salinity to g/kg

convert_to_dimensional_dissolved_gas(*dissolved_gas*)

convert from non dimensional dissolved gas to dimensional dissolved gas in kg(gas)/kg(liquid)

convert_to_dimensional_grid(*grid*)

Get domain depths in meters from non dimensional values

convert_to_dimensional_temperature(*temperature*)

get temperature in deg C from non dimensional temperature

convert_to_dimensional_time(*time*)

Convert non dimensional time into time in days since start of simulation

gas_density: float

lengthscale: float

liquid_thermal_conductivity: float

ocean_freezing_temperature: float

ocean_salinity: float

salinity_difference: float

saturation_concentration: float

temperature_difference: float

thermal_diffusivity: float

property time_scale

in days

property velocity_scale

in m /day

seaice3p.params.forcing module

```
class seaice3p.params.forcing.BRW09Forcing(ocean_bulk_salinity: float = 0, ocean_gas_sat: float = 1.0,
                                             Barrow_top_temperature_data_choice: str = 'air')
```

Bases: object

Surface and ocean temperature data loaded from thermistor temperature record during the Barrow 2009 field study.

```
Barrow_top_temperature_data_choice: str = 'air'
```

```
ocean_bulk_salinity: float = 0
```

```
ocean_gas_sat: float = 1.0
```

```
class seaice3p.params.forcing.BaseOceanForcing(ocean_temp: float = 0.1, ocean_bulk_salinity: float =
0, ocean_gas_sat: float = 1.0)
```

Bases: `object`

Not to be used directly but provides parameters for fixed ocean properties: gas saturation, temperature and bulk salinity to other forcing configuration classes

```
ocean_bulk_salinity: float = 0
```

```
ocean_gas_sat: float = 1.0
```

```
ocean_temp: float = 0.1
```

```
class seaice3p.params.forcing.ConstantForcing(ocean_temp: float = 0.1, ocean_bulk_salinity: float = 0,
ocean_gas_sat: float = 1.0, constant_top_temperature:
float = -1.5)
```

Bases: `BaseOceanForcing`

Constant temperature forcing

```
constant_top_temperature: float = -1.5
```

```
class seaice3p.params.forcing.RadForcing(ocean_temp: float = 0.1, ocean_bulk_salinity: float = 0,
ocean_gas_sat: float = 1.0, SW_forcing:
DimensionalConstantSWForcing =
DimensionalConstantSWForcing(SW_irradiance=280,
SW_albedo=0.7, SW_penetration_fraction=0.4), LW_forcing:
DimensionalConstantLWForcing =
DimensionalConstantLWForcing(LW_irradiance=260,
ice_emissivity=0.99, water_emissivity=0.97), turbulent_flux:
DimensionalConstantTurbulentFlux =
DimensionalConstantTurbulentFlux(ref_height=10,
windspeed=5, air_temp=0, specific_humidity=0.0036,
atm_pressure=101.325, air_density=1.275,
air_heat_capacity=1005,
air_latent_heat_of_vaporisation=2501000.0), oil_heating:
DimensionalBackgroundOilHeating |
DimensionalMobileOilHeating | DimensionalNoHeating =
DimensionalBackgroundOilHeating(oil_mass_ratio=0,
ice_type='FYI'))
```

Bases: `BaseOceanForcing`

Forcing parameters for radiative transfer simulation with oil drops

we have not implemented the non-dimensionalisation for these parameters yet and so we just pass the dimensional values directly to the simulation

```
LW_forcing: DimensionalConstantLWForcing =
DimensionalConstantLWForcing(LW_irradiance=260, ice_emissivity=0.99,
water_emissivity=0.97)
```

```
SW_forcing: DimensionalConstantSWForcing =  
DimensionalConstantSWForcing(SW_irradiance=280, SW_albedo=0.7,  
SW_penetration_fraction=0.4)  
  
oil_heating: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating |  
DimensionalNoHeating = DimensionalBackgroundOilHeating(oil_mass_ratio=0,  
ice_type='FYI')  
  
turbulent_flux: DimensionalConstantTurbulentFlux =  
DimensionalConstantTurbulentFlux(ref_height=10, windspeed=5, air_temp=0,  
specific_humidity=0.0036, atm_pressure=101.325, air_density=1.275,  
air_heat_capacity=1005, air_latent_heat_of_vaporisation=2501000.0)
```

```
class seaice3p.params.forcing.YearlyForcing(ocean_temp: float = 0.1, ocean_bulk_salinity: float = 0,  
                                             ocean_gas_sat: float = 1.0, offset: float = -1.0, amplitude:  
                                             float = 0.75, period: float = 4.0)
```

Bases: *BaseOceanForcing*

Yearly sinusoidal temperature forcing

amplitude: float = 0.75

offset: float = -1.0

period: float = 4.0

```
seaice3p.params.forcing.get_dimensionless_forcing_config(dimensional_params:  
                                                         DimensionalParams) → ConstantForcing  
                                                         | YearlyForcing | BRW09Forcing |  
                                                         RadForcing
```

seaice3p.params.initial_conditions module

```
class seaice3p.params.initial_conditions.OilInitialConditions(initial_ice_depth: float = 0.5,  
                                                              initial_ocean_temperature: float =  
                                                              -0.05, initial_ice_temperature: float =  
                                                              -0.1, initial_oil_volume_fraction:  
                                                              float = 1e-07,  
                                                              initial_ice_bulk_salinity: float =  
                                                              -0.1)
```

Bases: object

values for bottom (ocean) boundary

initial_ice_bulk_salinity: float = -0.1

initial_ice_depth: float = 0.5

initial_ice_temperature: float = -0.1

initial_ocean_temperature: float = -0.05

initial_oil_volume_fraction: float = 1e-07

```
seaice3p.params.initial_conditions.get_dimensionless_initial_conditions_config(dimensional_params:
    Dimensional-
    Params) →
    UniformIni-
    tialCondi-
    tions |
    BRW09InitialConditions
    | OilInitial-
    Conditions
```

seaice3p.params.params module

Classes containing parameters required to run a simulation

The config class contains all the parameters needed to run a simulation as well as methods to save and load this configuration to a yaml file.

```
class seaice3p.params.params.Config(name: str, total_time: float, savefreq: float, physical_params:
    EQMPhysicalParams | DISEQPhysicalParams, bubble_params:
    MonoBubbleParams | PowerLawBubbleParams,
    brine_convection_params: RJW14Params | NoBrineConvection,
    forcing_config: ConstantForcing | YearlyForcing | BRW09Forcing |
    RadForcing, initial_conditions_config: UniformInitialConditions |
    BRW09InitialConditions | OilInitialConditions, numerical_params:
    NumericalParams = NumericalParams(I=50, regularisation=1e-06),
    scales: Scales | None = None)
```

Bases: object

contains all information needed to run a simulation and save output

this config object can be saved and loaded to a yaml file.

brine_convection_params: *RJW14Params* | *NoBrineConvection*

bubble_params: *MonoBubbleParams* | *PowerLawBubbleParams*

forcing_config: *ConstantForcing* | *YearlyForcing* | *BRW09Forcing* | *RadForcing*

initial_conditions_config: *UniformInitialConditions* | *BRW09InitialConditions* | *OilInitialConditions*

classmethod load(*path*)

name: str

numerical_params: *NumericalParams* = NumericalParams(I=50, regularisation=1e-06)

physical_params: *EQMPhysicalParams* | *DISEQPhysicalParams*

save(*directory*: Path)

savefreq: float

scales: *Scales* | None = None

total_time: float

`seaice3p.params.params.get_config(dimensional_params: DimensionalParams) → Config`

Return a Config object for the simulation.

physical parameters and Darcy law parameters are calculated from the dimensional input. You can modify the numerical parameters and boundary conditions and forcing provided for the simulation.

seaice3p.params.physical module

```
class seaice3p.params.physical.BasePhysicalParams(expansion_coefficient: float = 0.029,  
                                                    concentration_ratio: float = 0.17, stefan_number:  
                                                    float = 4.2, lewis_salt: float = inf, lewis_gas: float  
                                                    = inf, frame_velocity: float = 0,  
                                                    phase_average_conductivity: bool = False,  
                                                    conductivity_ratio: float = 4.11,  
                                                    tolerable_super_saturation_fraction: float = 1)
```

Bases: object

Not to be used directly but provides the common parameters for physical params objects

```
concentration_ratio: float = 0.17  
conductivity_ratio: float = 4.11  
expansion_coefficient: float = 0.029  
frame_velocity: float = 0  
lewis_gas: float = inf  
lewis_salt: float = inf  
phase_average_conductivity: bool = False  
stefan_number: float = 4.2  
tolerable_super_saturation_fraction: float = 1
```

```
class seaice3p.params.physical.DISEQPhysicalParams(expansion_coefficient: float = 0.029,  
                                                    concentration_ratio: float = 0.17, stefan_number:  
                                                    float = 4.2, lewis_salt: float = inf, lewis_gas:  
                                                    float = inf, frame_velocity: float = 0,  
                                                    phase_average_conductivity: bool = False,  
                                                    conductivity_ratio: float = 4.11,  
                                                    tolerable_super_saturation_fraction: float = 1,  
                                                    damkohler_number: float = 1)
```

Bases: [BasePhysicalParams](#)

non dimensional numbers for the mushy layer

```
damkohler_number: float = 1
```

```
class seaice3p.params.physical.EQMPhysicalParams(expansion_coefficient: float = 0.029,  
                                                    concentration_ratio: float = 0.17, stefan_number:  
                                                    float = 4.2, lewis_salt: float = inf, lewis_gas: float =  
                                                    inf, frame_velocity: float = 0,  
                                                    phase_average_conductivity: bool = False,  
                                                    conductivity_ratio: float = 4.11,  
                                                    tolerable_super_saturation_fraction: float = 1)
```

Bases: *BasePhysicalParams*

non dimensional numbers for the mushy layer

```
seaice3p.params.physical.get_dimensionless_physical_params(dimensional_params:
    DimensionalParams) →
    EQMPhysicalParams |
    DISEQPhysicalParams
```

return a PhysicalParams object

Module contents

seaice3p.state package

Submodules

seaice3p.state.disequilibrium_state module

```
class seaice3p.state.disequilibrium_state.DISEQState(time: float, enthalpy: ndarray[Any,
    dtype[_ScalarType_co]], salt: ndarray[Any,
    dtype[_ScalarType_co]], bulk_dissolved_gas:
    ndarray[Any, dtype[_ScalarType_co]],
    gas_fraction: ndarray[Any,
    dtype[_ScalarType_co]])
```

Bases: object

Contains the principal variables for solution with non-equilibrium gas phase. The total bulk gas is partitioned between dissolved gas and free phase gas with a finite nucleation rate (non dimensional damkohler number).

principal solution components: bulk enthalpy bulk salinity bulk dissolved gas gas fraction

all on the center grid.

Note: Define bulk dissolved gas for the system as

$\text{expansion_coefficient} * \text{liquid_fraction} * \text{dissolved_gas}$

so that this is different from the dissolved gas concentration and

$\text{bulk_gas} = \text{bulk_dissolved_gas} + \text{gas_fraction}$

in non-dimensional units.

bulk_dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

enthalpy: ndarray[Any, dtype[_ScalarType_co]]

property gas: ndarray[Any, dtype[_ScalarType_co]]

Calculate bulk gas content and use same attribute name as EQMState

gas_fraction: ndarray[Any, dtype[_ScalarType_co]]

salt: ndarray[Any, dtype[_ScalarType_co]]

time: float

```
class seai3p.state.disequilibrium_state.DISEQStateBCs(time: float, enthalpy: ndarray[Any,
                                                    dtype[_ScalarType_co]], salt:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    temperature: ndarray[Any,
                                                    dtype[_ScalarType_co]], liquid_salinity:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    dissolved_gas: ndarray[Any,
                                                    dtype[_ScalarType_co]], liquid_fraction:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    bulk_dissolved_gas: ndarray[Any,
                                                    dtype[_ScalarType_co]], gas_fraction:
                                                    ndarray[Any, dtype[_ScalarType_co]])
```

Bases: object

Stores information needed for solution at one timestep with BCs on ghost cells as well

Initialise the prime variables for the solver: enthalpy, bulk salinity and bulk air

bulk_dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

enthalpy: ndarray[Any, dtype[_ScalarType_co]]

gas_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_salinity: ndarray[Any, dtype[_ScalarType_co]]

salt: ndarray[Any, dtype[_ScalarType_co]]

temperature: ndarray[Any, dtype[_ScalarType_co]]

time: float

```
class seai3p.state.disequilibrium_state.DISEQStateFull(time: float, enthalpy: ndarray[Any,
                                                    dtype[_ScalarType_co]], salt:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    bulk_dissolved_gas: ndarray[Any,
                                                    dtype[_ScalarType_co]], gas_fraction:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    temperature: ndarray[Any,
                                                    dtype[_ScalarType_co]], liquid_fraction:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    solid_fraction: ndarray[Any,
                                                    dtype[_ScalarType_co]], liquid_salinity:
                                                    ndarray[Any, dtype[_ScalarType_co]],
                                                    dissolved_gas: ndarray[Any,
                                                    dtype[_ScalarType_co]])
```

Bases: object

Contains all variables variables for solution with non-equilibrium gas phase after running the enthalpy method on DISEQState. The total bulk gas is partitioned between dissolved gas and free phase gas with a finite nucleation rate (non dimensional damkohler number).

principal solution components: bulk enthalpy bulk salinity bulk dissolved gas gas fraction

enthalpy method variables: temperature liquid_fraction solid_fraction liquid_salinity dissolved_gas

all on the center grid.

Note: Define bulk dissolved gas for the system as

$\text{expansion_coefficient} * \text{liquid_fraction} * \text{dissolved_gas}$

so that this is different from the dissolved gas concentration and

$\text{bulk_gas} = \text{bulk_dissolved_gas} + \text{gas_fraction}$

in non-dimensional units.

bulk_dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

enthalpy: ndarray[Any, dtype[_ScalarType_co]]

property gas: ndarray[Any, dtype[_ScalarType_co]]

Calculate bulk gas content and use same attribute name as EQMState

gas_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_salinity: ndarray[Any, dtype[_ScalarType_co]]

salt: ndarray[Any, dtype[_ScalarType_co]]

solid_fraction: ndarray[Any, dtype[_ScalarType_co]]

temperature: ndarray[Any, dtype[_ScalarType_co]]

time: float

seaice3p.state.equilibrium_state module

```
class seaice3p.state.equilibrium_state.EQMState(time: float, enthalpy: ndarray[Any,
                                              dtype[_ScalarType_co]], salt: ndarray[Any,
                                              dtype[_ScalarType_co]], gas: ndarray[Any,
                                              dtype[_ScalarType_co]])
```

Bases: object

Contains the principal variables for solution with equilibrium gas phase:

bulk enthalpy bulk salinity bulk gas

all on the center grid.

enthalpy: ndarray[Any, dtype[_ScalarType_co]]

gas: ndarray[Any, dtype[_ScalarType_co]]

salt: ndarray[Any, dtype[_ScalarType_co]]

time: float

```
class seaice3p.state.equilibrium_state.EQMStateBCs(time: float, enthalpy: ndarray[Any,  
                                                dtype[_ScalarType_co]], salt: ndarray[Any,  
                                                dtype[_ScalarType_co]], gas: ndarray[Any,  
                                                dtype[_ScalarType_co]], temperature:  
                                                ndarray[Any, dtype[_ScalarType_co]],  
                                                liquid_salinity: ndarray[Any,  
                                                dtype[_ScalarType_co]], dissolved_gas:  
                                                ndarray[Any, dtype[_ScalarType_co]],  
                                                gas_fraction: ndarray[Any,  
                                                dtype[_ScalarType_co]], liquid_fraction:  
                                                ndarray[Any, dtype[_ScalarType_co]])
```

Bases: object

Stores information needed for solution at one timestep with BCs on ghost cells as well

Initialiase the prime variables for the solver: enthalpy, bulk salinity and bulk air

dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]

enthalpy: ndarray[Any, dtype[_ScalarType_co]]

gas: ndarray[Any, dtype[_ScalarType_co]]

gas_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_fraction: ndarray[Any, dtype[_ScalarType_co]]

liquid_salinity: ndarray[Any, dtype[_ScalarType_co]]

salt: ndarray[Any, dtype[_ScalarType_co]]

temperature: ndarray[Any, dtype[_ScalarType_co]]

time: float

```
class seaice3p.state.equilibrium_state.EQMStateFull(time: float, enthalpy: ndarray[Any,  
                                                  dtype[_ScalarType_co]], salt: ndarray[Any,  
                                                  dtype[_ScalarType_co]], gas: ndarray[Any,  
                                                  dtype[_ScalarType_co]], temperature:  
                                                  ndarray[Any, dtype[_ScalarType_co]],  
                                                  liquid_fraction: ndarray[Any,  
                                                  dtype[_ScalarType_co]], solid_fraction:  
                                                  ndarray[Any, dtype[_ScalarType_co]],  
                                                  liquid_salinity: ndarray[Any,  
                                                  dtype[_ScalarType_co]], dissolved_gas:  
                                                  ndarray[Any, dtype[_ScalarType_co]],  
                                                  gas_fraction: ndarray[Any,  
                                                  dtype[_ScalarType_co]])
```

Bases: object

Contains all variables variables for solution with equilibrium gas phase after running the enthalpy method on EQMSate.

principal solution components: bulk enthalpy bulk salinity bulk gas

enthalpy method variables: temperature liquid_fraction solid_fraction liquid_salinity dissolved_gas gas_fraction
all on the center grid.

```

dissolved_gas: ndarray[Any, dtype[_ScalarType_co]]
enthalpy: ndarray[Any, dtype[_ScalarType_co]]
gas: ndarray[Any, dtype[_ScalarType_co]]
gas_fraction: ndarray[Any, dtype[_ScalarType_co]]
liquid_fraction: ndarray[Any, dtype[_ScalarType_co]]
liquid_salinity: ndarray[Any, dtype[_ScalarType_co]]
salt: ndarray[Any, dtype[_ScalarType_co]]
solid_fraction: ndarray[Any, dtype[_ScalarType_co]]
temperature: ndarray[Any, dtype[_ScalarType_co]]
time: float

```

Module contents

`seaice3p.state.get_unpacker(cfg: Config) → Callable[[float, ndarray[Any, dtype[_ScalarType_co]]], EQMState | DISEQState]`

1.1.2 Submodules

1.1.3 seaice3p.example module

Script to run a simulation starting with dimensional parameters and plot output

`seaice3p.example.create_and_save_config(data_directory: Path, simulation_dimensional_params: DimensionalParams)`

`seaice3p.example.main(data_directory: Path, frames_directory: Path, simulation_dimensional_params: DimensionalParams)`

Generate non dimensional simulation config and save along with dimensional config then run simulation and save data.

1.1.4 seaice3p.grids module

Module providing functions to initialise the different grids and interpolate quantities between them.

class `seaice3p.grids.Grids(number_of_cells: int)`

Bases: `object`

Class initialised from number of grid cells to contain:

grid cell width, center, edge and ghost grids and difference matrices

property `D_e`: `ndarray[Any, dtype[_ScalarType_co]]`

Difference matrix to differentiate edge grid quantities to the center grid

property `D_g`: `ndarray[Any, dtype[_ScalarType_co]]`

Difference matrix to differentiate ghost grid quantities to the edge grid

property centers: ndarray[Any, dtype[_ScalarType_co]]

Center grid

property edges: ndarray[Any, dtype[_ScalarType_co]]

Edge grid

property ghosts: ndarray[Any, dtype[_ScalarType_co]]

Ghost grid

number_of_cells: int

property step: float

Grid cell width

`seaice3p.grids.add_ghost_cells(centers, bottom, top)`

Add specified bottom and top value to center grid

Parameters

- **centers** (*Numpy array*) – numpy array on centered grid (size I).
- **bottom** (*float*) – bottom value placed at index 0.
- **top** (*float*) – top value placed at index -1.

Returns

numpy array on ghost grid (size I+2).

`seaice3p.grids.calculate_ice_ocean_boundary_depth(liquid_fraction, edge_grid)`

Calculate the depth of the ice ocean boundary as the edge position of the first cell from the bottom to be not completely liquid. I.e the first time the liquid fraction goes below 1.

If the ice has made it to the bottom of the domain raise an error.

If the domain is completely liquid set h=0.

NOTE: depth is a positive quantity and our grid coordinate increases from -1 at the bottom of the domain to 0 at the top.

Parameters

- **liquid_fraction** (*Numpy Array (size I)*) – liquid fraction on center grid
- **edge_grid** (*Numpy Array (size I+1)*) – The vertical coordinate positions of the edge grid.

Returns

positive depth value of ice ocean interface

`seaice3p.grids.geometric(ghosts)`

Returns geometric mean of the first dimension of an array

`seaice3p.grids.get_difference_matrix(size, step)`

`seaice3p.grids.upwind(ghosts, velocity)`

1.1.5 seaice3p.initial_conditions module

Module to provide initial state of bulk enthalpy, bulk salinity and bulk gas for the simulation.

`seaice3p.initial_conditions.get_initial_conditions(cfg: Config)`

1.1.6 seaice3p.load module

`seaice3p.load.get_array_data(attr: str, cfg, times, data)`

`seaice3p.load.get_state(non_dimensional_time, times, data, cfg)`

`seaice3p.load.load_data(sim_name: str, data_directory: Path, sim_config_name=None, is_dimensional=False, config_extension='yaml')`

1.1.7 seaice3p.oil_simulation module

`seaice3p.oil_simulation.generate_oil_simulation_config(name: str, total_time_in_days: float, lengthscale: float, initial_oil_mass_ratio: float, oil_density: float, oil_droplet_radius: float, SW_irradiance: float, SW_albedo: float, SW_penetration_fraction: float, LW_irradiance: float, air_temp: float, windspeed: float, ref_height: float, oil_heating_params: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating | DimensionalNoHeating, initial_ice_depth: float, initial_ice_temperature: float, initial_ocean_temperature: float, initial_ice_bulk_salinity: float = 34, brine_convection_params: DimensionalRJW14Params | NoBrineConvection = DimensionalRJW14Params(couple_bubble_to_horizontal_flow=False, couple_bubble_to_vertical_flow=False, Rayleigh_critical=2.9, convection_strength=0.13, haline_contraction_coefficient=0.00075, reference_permeability=1e-08), I=50, savefreq_in_days=1.0, config_directory=PosixPath('.')) → None`

Parameters to generate a simulation config for melting of an initially uniform layer of ice in an ocean under SW, LW radiative fluxes and sensible heat flux.

The latent heat flux is disabled by setting the latent heat of vaporisation to 0.

The initially uniform mass concentration of oil in the domain is set in ng/g.

1.1.8 seaice3p.printing module

`seaice3p.printing.get_printer(verbosity_level: int) → Callable[[str], None]`

1.1.9 seaice3p.run_simulation module

Module to run the simulation on the given configuration with the appropriate solver.

Solve reduced model using scipy solve_ivp using RK23 solver.

Impose a maximum timestep constraint using courant number for thermal diffusion as this is an explicit method.

This solver uses adaptive timestepping which makes it a good choice for running simulations with large buoyancy driven gas bubble velocities and we save the output at intervals given by the savefreq parameter in configuration.

`seaice3p.run_simulation.run_batch(list_of_cfg: List[Config], directory: Path, verbosity_level=0) → None`

Run a batch of simulations from a list of configurations.

Each simulation name is logged, as well as if it successfully runs or crashes. Output from each simulation is saved in a .npz file.

Parameters

`list_of_cfg` (`List[seaice3p.params.Config]`) – list of configurations

`seaice3p.run_simulation.solve(cfg: Config, directory: Path, verbosity_level=0) → Literal[0]`

1.1.10 Module contents

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

S

[seaice3p](#), 42
[seaice3p.diagnostics](#), 1
[seaice3p.diagnostics.brine_drainage_parameterisation](#), 1
[seaice3p.enthalpy_method](#), 2
[seaice3p.enthalpy_method.common](#), 1
[seaice3p.enthalpy_method.enthalpy_method](#), 2
[seaice3p.enthalpy_method.gas](#), 2
[seaice3p.enthalpy_method.phase_boundaries](#), 2
[seaice3p.equations](#), 12
[seaice3p.equations.equations](#), 11
[seaice3p.equations.flux](#), 8
[seaice3p.equations.flux.bulk_dissolved_gas_flux](#), 6
[seaice3p.equations.flux.bulk_gas_flux](#), 7
[seaice3p.equations.flux.gas_fraction_flux](#), 7
[seaice3p.equations.flux.heat_flux](#), 7
[seaice3p.equations.flux.salt_flux](#), 8
[seaice3p.equations.nucleation](#), 11
[seaice3p.equations.radiative_heating](#), 11
[seaice3p.equations.RJW14](#), 6
[seaice3p.equations.RJW14.brine_channel_sink_terms](#), 3
[seaice3p.equations.RJW14.brine_drainage](#), 3
[seaice3p.equations.velocities](#), 11
[seaice3p.equations.velocities.bubble_parameters](#), 8
[seaice3p.equations.velocities.mono_distribution](#), 8
[seaice3p.equations.velocities.power_law_distribution](#), 9
[seaice3p.equations.velocities.velocities](#), 10
[seaice3p.example](#), 39
[seaice3p.forcing](#), 16
[seaice3p.forcing.boundary_conditions](#), 16
[seaice3p.forcing.radiative_forcing](#), 16
[seaice3p.forcing.surface_energy_balance](#), 16
[seaice3p.forcing.surface_energy_balance.surface_energy_balance](#), 12
[seaice3p.forcing.surface_energy_balance.turbulent_heat_flux](#), 14
[seaice3p.forcing.temperature_forcing](#), 16
[seaice3p.grids](#), 39
[seaice3p.initial_conditions](#), 41
[seaice3p.load](#), 41
[seaice3p.oil_simulation](#), 41
[seaice3p.params](#), 35
[seaice3p.params.bubble](#), 27
[seaice3p.params.convection](#), 29
[seaice3p.params.convert](#), 29
[seaice3p.params.dimensionality](#), 27
[seaice3p.params.dimensionality.bubble](#), 16
[seaice3p.params.dimensionality.convection](#), 18
[seaice3p.params.dimensionality.dimensionality](#), 19
[seaice3p.params.dimensionality.forcing](#), 21
[seaice3p.params.dimensionality.gas](#), 24
[seaice3p.params.dimensionality.initial_conditions](#), 25
[seaice3p.params.dimensionality.numerical](#), 26
[seaice3p.params.dimensionality.water](#), 26
[seaice3p.params.forcing](#), 30
[seaice3p.params.initial_conditions](#), 32
[seaice3p.params.params](#), 33
[seaice3p.params.physical](#), 34
[seaice3p.printing](#), 42
[seaice3p.run_simulation](#), 42
[seaice3p.state](#), 39
[seaice3p.state.disequilibrium_state](#), 35
[seaice3p.state.equilibrium_state](#), 37

A

add_ghost_cells() (in module *seaice3p.grids*), 40
air_density (*seaice3p.params.dimensionnal.forcing.DimensionnalConstantTurbulentFlux* attribute), 22
air_heat_capacity (*seaice3p.params.dimensionnal.forcing.DimensionnalConstantTurbulentFlux* attribute), 22
air_latent_heat_of_vaporisation (*seaice3p.params.dimensionnal.forcing.DimensionnalConstantTurbulentFlux* attribute), 22
air_temp (*seaice3p.params.dimensionnal.forcing.DimensionnalConstantTurbulentFlux* attribute), 22
amplitude (*seaice3p.params.dimensionnal.forcing.DimensionnalYearlyForcing* attribute), 24
amplitude (*seaice3p.params.forcing.YearlyForcing* attribute), 32
atm_pressure (*seaice3p.params.dimensionnal.forcing.DimensionnalConstantTurbulentFlux* attribute), 22

B

B (*seaice3p.params.bubble.BaseBubbleParams* attribute), 28
B (*seaice3p.params.dimensionnal.dimensionnal.DimensionnalParams* property), 20
Barrow_initial_bulk_gas_in_ice (*seaice3p.params.dimensionnal.initial_conditions.BRW09InitialConditions* attribute), 25
Barrow_top_temperature_data_choice (*seaice3p.params.dimensionnal.forcing.DimensionnalBRW09Forcing* attribute), 21
Barrow_top_temperature_data_choice (*seaice3p.params.forcing.BRW09Forcing* attribute), 30
BaseBubbleParams (class in *seaice3p.params.bubble*), 27
BaseOceanForcing (class in *seaice3p.params.forcing*), 31
BasePhysicalParams (class in *seaice3p.params.physical*), 34
brine_convection_params (*seaice3p.params.dimensionnal.dimensionnal.DimensionnalParams* attribute), 20
brine_convection_params (*seaice3p.params.params.Config* attribute), 33
BRW09Forcing (class in *seaice3p.params.forcing*), 30
BRW09InitialConditions (class in *seaice3p.params.dimensionnal.initial_conditions*), 25
bubble_distribution_power (*seaice3p.params.bubble.PowerLawBubbleParams* attribute), 28
bubble_distribution_power (*seaice3p.params.dimensionnal.bubble.DimensionnalPowerLawBubbleParams* attribute), 18
bubble_params (*seaice3p.params.dimensionnal.dimensionnal.DimensionnalParams* attribute), 20
bubble_params (*seaice3p.params.params.Config* attribute), 33
bubble_radius (*seaice3p.params.dimensionnal.bubble.DimensionnalMonoBubbleParams* attribute), 17
bubble_radius_scaled (*seaice3p.params.bubble.MonoBubbleParams* attribute), 28
bubble_radius_scaled (*seaice3p.params.dimensionnal.bubble.DimensionnalMonoBubbleParams* property), 17
bulk_dissolved_gas (*seaice3p.state.disequilibrium_state.DISEQState* attribute), 35
bulk_dissolved_gas (*seaice3p.state.disequilibrium_state.DISEQStateBC* attribute), 36
bulk_dissolved_gas (*seaice3p.state.disequilibrium_state.DISEQStateFu* attribute), 37

C

calculate_advective_dissolved_gas_flux() (in module *seaice3p.equations.flux.bulk_gas_flux*), 7
calculate_advective_heat_flux() (in module *seaice3p.equations.flux.heat_flux*), 7
calculate_advective_salt_flux() (in module *seaice3p.equations.flux.salt_flux*), 8
calculate_brine_channel_sink() (in module *seaice3p.equations.RJW14.brine_drainage*), 3
calculate_brine_channel_strength() (in module

<code>seai3p.equations.RJW14.brine_drainage)</code> , 4	<code>calculate_lag_function()</code> (in module <code>seai3p.equations.velocities.mono_distribution</code>), 8
<code>calculate_brine_convection_liquid_velocity()</code> (in module <code>seai3p.equations.RJW14.brine_drainage</code>), 4	<code>calculate_lag_integral()</code> (in module <code>seai3p.equations.velocities.power_law_distribution</code>), 9
<code>calculate_bubble_gas_flux()</code> (in module <code>seai3p.equations.flux.bulk_gas_flux</code>), 7	<code>calculate_lag_integrand()</code> (in module <code>seai3p.equations.velocities.power_law_distribution</code>), 9
<code>calculate_bubble_size_fraction()</code> (in module <code>seai3p.equations.velocities.bubble_parameters</code>), 8	<code>calculate_latent_heat_flux()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 14
<code>calculate_bulk_dissolved_gas_flux()</code> (in module <code>seai3p.equations.flux.bulk_dissolved_gas_flux</code>), 6	<code>calculate_liquid_darcy_velocity()</code> (in module <code>seai3p.equations.velocities.velocities</code>), 10
<code>calculate_bulk_transfer_coefficient()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 14	<code>calculate_mono_lag_factor()</code> (in module <code>seai3p.equations.velocities.mono_distribution</code>), 8
<code>calculate_common_enthalpy_method_vars()</code> (in module <code>seai3p.enthalpy_method.common</code>), 1	<code>calculate_mono_wall_drag_factor()</code> (in module <code>seai3p.equations.velocities.mono_distribution</code>), 8
<code>calculate_conductive_heat_flux()</code> (in module <code>seai3p.equations.flux.heat_flux</code>), 7	<code>calculate_permeability()</code> (in module <code>seai3p.equations.RJW14.brine_drainage</code>), 5
<code>calculate_diffusive_gas_flux()</code> (in module <code>seai3p.equations.flux.bulk_gas_flux</code>), 7	<code>calculate_power_law_lag_factor()</code> (in module <code>seai3p.equations.velocities.power_law_distribution</code>), 9
<code>calculate_diffusive_salt_flux()</code> (in module <code>seai3p.equations.flux.salt_flux</code>), 8	<code>calculate_power_law_wall_drag_factor()</code> (in module <code>seai3p.equations.velocities.power_law_distribution</code>), 9
<code>calculate_DISEQ_dissolved_gas()</code> (in module <code>seai3p.enthalpy_method.gas</code>), 2	<code>calculate_Rayleigh()</code> (in module <code>seai3p.equations.RJW14.brine_drainage</code>), 3
<code>calculate_emissivity()</code> (in module <code>seai3p.forcing.surface_energy_balance.surface_energy_balance</code>), 12	<code>calculate_ref_air_temp()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 14
<code>calculate_EQM_dissolved_gas()</code> (in module <code>seai3p.enthalpy_method.gas</code>), 2	<code>calculate_ref_atmospheric_pressure()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 14
<code>calculate_EQM_gas_fraction()</code> (in module <code>seai3p.enthalpy_method.gas</code>), 2	<code>calculate_ref_specific_humidity()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 15
<code>calculate_frame_advection_gas_flux()</code> (in module <code>seai3p.equations.flux.bulk_gas_flux</code>), 7	<code>calculate_ref_windspeed()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 15
<code>calculate_frame_advection_heat_flux()</code> (in module <code>seai3p.equations.flux.heat_flux</code>), 7	<code>calculate_salt_flux()</code> (in module <code>seai3p.equations.flux.salt_flux</code>), 8
<code>calculate_frame_advection_salt_flux()</code> (in module <code>seai3p.equations.flux.salt_flux</code>), 8	<code>calculate_sensible_heat_flux()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 15
<code>calculate_frame_velocity()</code> (in module <code>seai3p.equations.velocities.velocities</code>), 10	<code>calculate_surface_specific_humidity()</code> (in module <code>seai3p.forcing.surface_energy_balance.turbulent_heat_flux</code>), 15
<code>calculate_gas_flux()</code> (in module <code>seai3p.equations.flux.bulk_gas_flux</code>), 7	<code>calculate_total_heat_flux()</code> (in module <code>seai3p.forcing.surface_energy_balance.surface_energy_balance</code>), 12
<code>calculate_gas_fraction_flux()</code> (in module <code>seai3p.equations.flux.gas_fraction_flux</code>), 7	<code>calculate_velocities()</code> (in module <code>seai3p.equations.velocities.velocities</code>), 10
<code>calculate_gas_interstitial_velocity()</code> (in module <code>seai3p.equations.velocities.velocities</code>), 10	
<code>calculate_heat_flux()</code> (in module <code>seai3p.equations.flux.heat_flux</code>), 7	
<code>calculate_ice_ocean_boundary_depth()</code> (in module <code>seai3p.grids</code>), 40	
<code>calculate_integrated_mean_permeability()</code> (in module <code>seai3p.equations.RJW14.brine_drainage</code>), 5	

seaice3p.equations.velocities.velocities), 29
 11
 calculate_volume_integrand() (in module (seaice3p.params.convert.Scales method),
 seaice3p.equations.velocities.power_law_distribution), 29
 9
 calculate_wall_drag_function() (in module (seaice3p.params.convert.Scales method),
 seaice3p.equations.velocities.mono_distribution), 29
 9
 calculate_wall_drag_integral() (in module (seaice3p.params.convert.Scales method),
 seaice3p.equations.velocities.power_law_distribution), 30
 10
 calculate_wall_drag_integrand() (in module module seaice3p.forcing.surface_energy_balance.surface_energy.
 seaice3p.equations.velocities.power_law_distribution), 12
 10
 centers (seaice3p.grids.Grids property), 39
 concentration_ratio (seaice3p.params.dimensionals.water.DimensionalsWaterParams property), 26
 concentration_ratio (seaice3p.params.physical.BasePhysicalParams attribute), 34
 conductivity_ratio (seaice3p.params.dimensionals.water.DimensionalsWaterParams property), 26
 conductivity_ratio (seaice3p.params.physical.BasePhysicalParams attribute), 34
 Config (class in seaice3p.params.params), 33
 constant_top_temperature (seaice3p.params.dimensionals.forcing.DimensionalsConstantForcing attribute), 21
 constant_top_temperature (seaice3p.params.forcing.ConstantForcing attribute), 31
 ConstantForcing (class in seaice3p.params.forcing), 31
 convection_strength (seaice3p.params.convection.RJW14Params attribute), 29
 convection_strength (seaice3p.params.dimensionals.convection.DimensionalsRJW14Params attribute), 18
 convert_dimensional_bulk_air_to_argon_content() (seaice3p.params.convert.Scales method), 29
 convert_from_dimensional_bulk_gas() (seaice3p.params.convert.Scales method), 29
 convert_from_dimensional_bulk_salinity() (seaice3p.params.convert.Scales method), 29
 convert_from_dimensional_dissolved_gas() (seaice3p.params.convert.Scales method), 29
 convert_from_dimensional_grid() (seaice3p.params.convert.Scales method), 29
 convert_from_dimensional_heat_flux() (seaice3p.params.convert.Scales method),
 29
 convert_from_dimensional_heating() (seaice3p.params.convert.Scales method),
 29
 convert_from_dimensional_temperature() (seaice3p.params.convert.Scales method),
 29
 convert_from_dimensional_time() (seaice3p.params.convert.Scales method),
 30
 convert_surface_temperature_to_kelvin() (in
 module seaice3p.forcing.surface_energy_balance.surface_energy.
 12
 convert_to_dimensional_bulk_gas() (seaice3p.params.convert.Scales method),
 30
 convert_to_dimensional_bulk_salinity() (seaice3p.params.convert.Scales method),
 30
 convert_to_dimensional_dissolved_gas() (seaice3p.params.convert.Scales method),
 30
 convert_to_dimensional_grid() (seaice3p.params.convert.Scales method),
 30
 convert_to_dimensional_temperature() (seaice3p.params.convert.Scales method),
 30
 convert_to_dimensional_time() (seaice3p.params.convert.Scales method),
 30
 couple_bubble_to_horizontal_flow (seaice3p.params.convection.RJW14Params
 attribute), 29
 couple_bubble_to_horizontal_flow (seaice3p.params.dimensionals.convection.DimensionalsRJW14Params
 attribute), 18
 couple_bubble_to_vertical_flow (seaice3p.params.convection.RJW14Params
 attribute), 29
 couple_bubble_to_vertical_flow (seaice3p.params.dimensionals.convection.DimensionalsRJW14Params
 attribute), 18
 create_and_save_config() (in module
 seaice3p.example), 39

D

D_e (seaice3p.grids.Grids property), 39
 D_g (seaice3p.grids.Grids property), 39
 damkohler_number (seaice3p.params.dimensionals.dimensionals.Dimensionals property), 20
 damkohler_number (seaice3p.params.physical.DISEQPhysicalParams attribute), 34

DimensionalBackgroundOilHeating	(class seaice3p.params.dimensional.forcing), 21	in	dissolved_gas (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 36
DimensionalBaseBubbleParams	(class seaice3p.params.dimensional.bubble), 16	in	dissolved_gas (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 37
DimensionalBRW09Forcing	(class seaice3p.params.dimensional.forcing), 21	in	dissolved_gas (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 38
DimensionalConstantForcing	(class seaice3p.params.dimensional.forcing), 21	in	dissolved_gas (seaice3p.state.equilibrium_state.EQMStateFull attribute), 38
DimensionalConstantLWForcing	(class seaice3p.params.dimensional.forcing), 21	in	
DimensionalConstantSWForcing	(class seaice3p.params.dimensional.forcing), 22	in	E
DimensionalConstantTurbulentFlux	(class seaice3p.params.dimensional.forcing), 22	in	edges (seaice3p.grids.Grids property), 40
DimensionalDISEQGasParams	(class seaice3p.params.dimensional.gas), 24	in	enthalpy (seaice3p.state.disequilibrium_state.DISEQState attribute), 35
DimensionalEQMGasParams	(class seaice3p.params.dimensional.gas), 24	in	enthalpy (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 36
DimensionalLWForcing	(in module seaice3p.params.dimensional.forcing), 22	in	enthalpy (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 37
DimensionalMobileOilHeating	(class seaice3p.params.dimensional.forcing), 22	in	enthalpy (seaice3p.state.equilibrium_state.EQMState attribute), 37
DimensionalMonoBubbleParams	(class seaice3p.params.dimensional.bubble), 17	in	enthalpy (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 38
DimensionalNoHeating	(class seaice3p.params.dimensional.forcing), 23	in	enthalpy (seaice3p.state.equilibrium_state.EQMStateFull attribute), 39
DimensionalOilInitialConditions	(class seaice3p.params.dimensional.initial_conditions), 25	in	EQMPhysicalParams (class in seaice3p.params.physical), 34
DimensionalParams	(class seaice3p.params.dimensional.dimensionals), 19	in	EQMState (class in seaice3p.state.equilibrium_state), 37
DimensionalPowerLawBubbleParams	(class seaice3p.params.dimensional.bubble), 17	in	EQMStateBCs (class in seaice3p.state.equilibrium_state), 37
DimensionalRadForcing	(class seaice3p.params.dimensional.forcing), 23	in	EQMStateFull (class in seaice3p.state.equilibrium_state), 38
DimensionalRJW14Params	(class seaice3p.params.dimensional.convection), 18	in	escape_ice_surface (seaice3p.params.bubble.BaseBubbleParams attribute), 28
DimensionalSWForcing	(in module seaice3p.params.dimensional.forcing), 23	in	escape_ice_surface (seaice3p.params.dimensional.bubble.Dimensional attribute), 17
DimensionalTurbulentFlux	(in module seaice3p.params.dimensional.forcing), 24	in	eutectic_salinity (seaice3p.params.dimensional.water.DimensionalWa attribute), 26
DimensionalWaterParams	(class seaice3p.params.dimensional.water), 26	in	eutectic_temperature (seaice3p.params.dimensional.water.DimensionalWaterParams attribute), 26
DimensionalYearlyForcing	(class seaice3p.params.dimensional.forcing), 24	in	expansion_coefficient (seaice3p.params.dimensionals.DimensionalParams property), 20
DISEQPhysicalParams	(class seaice3p.params.physical), 34	in	expansion_coefficient (seaice3p.params.physical.BasePhysicalParams attribute), 34
DISEQState	(class in seaice3p.state.disequilibrium_state), 35	in	F
DISEQStateBCs	(class seaice3p.state.disequilibrium_state), 35	in	find_ghost_cell_temperature() (in module seaice3p.forcing.surface_energy_balance.surface_energy_balanc 12
DISEQStateFull	(class seaice3p.state.disequilibrium_state), 36	in	forcing_config (seaice3p.params.dimensionals.Dimensional attribute), 20
		in	forcing_config (seaice3p.params.params.Config at- tribute), 33

frame_velocity (seaice3p.params.dimensionals.dimensionless.DimensionlessPhysical property), 20
 frame_velocity (seaice3p.params.physical.BasePhysical property), 34
 frame_velocity_dimensional (seaice3p.params.dimensionals.dimensionals.DimensionalsPhysical property), 20
G
 gas (seaice3p.state.disequilibrium_state.DISEQState property), 35
 gas (seaice3p.state.disequilibrium_state.DISEQStateFull property), 37
 gas (seaice3p.state.equilibrium_state.EQMState property), 37
 gas (seaice3p.state.equilibrium_state.EQMStateBCs property), 38
 gas (seaice3p.state.equilibrium_state.EQMStateFull property), 39
 gas_density (seaice3p.params.convert.Scales property), 30
 gas_density (seaice3p.params.dimensionals.gas.DimensionalsPhysical property), 24
 gas_diffusivity (seaice3p.params.dimensionals.gas.DimensionalsPhysical property), 24
 gas_fraction (seaice3p.state.disequilibrium_state.DISEQState property), 35
 gas_fraction (seaice3p.state.disequilibrium_state.DISEQStateBCs property), 36
 gas_fraction (seaice3p.state.disequilibrium_state.DISEQStateFull property), 37
 gas_fraction (seaice3p.state.equilibrium_state.EQMStateBCs property), 38
 gas_fraction (seaice3p.state.equilibrium_state.EQMStateFull property), 39
 gas_params (seaice3p.params.dimensionals.dimensionals.DimensionalsPhysical property), 20
 generate_oil_simulation_config() (in module seaice3p.oil_simulation), 41
 geometric() (in module seaice3p.grids), 40
 get_array_data() (in module seaice3p.load), 41
 get_bottom_temperature_forcing() (in module seaice3p.forcing.temperature_forcing), 16
 get_boundary_conditions() (in module seaice3p.forcing.boundary_conditions), 16
 get_brine_convection_sink() (in module seaice3p.equations.RJW14.brine_channel_sink_terms), 3
 get_config() (in module seaice3p.params.params), 33
 get_convecting_region_height() (in module seaice3p.equations.RJW14.brine_drainage), 5
 get_difference_matrix() (in module seaice3p.grids), 40
 get_dimensionless_bubble_params() (in module seaice3p.params.bubble), 28
 get_dimensionless_forcing_config() (in module seaice3p.params.forcing), 32
 get_dimensionless_initial_conditions_config() (in module seaice3p.params.initial_conditions), 32
 get_dimensionless_physical_params() (in module seaice3p.params.physical), 35
 get_dz_fluxes() (in module seaice3p.equations.flux), 8
 get_effective_Rayleigh_number() (in module seaice3p.equations.RJW14.brine_drainage), 6
 get_enthalpy_method() (in module seaice3p.enthalpy_method.enthalpy_method), 2
 get_equations() (in module seaice3p.equations.equations), 11
 get_initial_conditions() (in module seaice3p.params.initial_conditions), 41
 get_LW_forcing() (in module seaice3p.forcing.radiative_forcing), 16
 get_nucleation() (in module seaice3p.equations.nucleation), 11
 get_phase_masks() (in module seaice3p.enthalpy_method.phase_boundaries), 2
 get_printer() (in module seaice3p.printing), 42
 get_radiative_heating() (in module seaice3p.equations.radiative_heating), 11
 get_state() (in module seaice3p.load), 41
 get_SW_forcing() (in module seaice3p.forcing.radiative_forcing), 16
 get_temperature_forcing() (in module seaice3p.forcing.temperature_forcing), 16
 get_unpacker() (in module seaice3p.state), 39
 ghosts (seaice3p.grids.Grids property), 40
 gravity (seaice3p.params.dimensionals.dimensionals.DimensionalsPhysical property), 20
 Grids (class in seaice3p.grids), 39
H
 haline_contraction_coefficient (seaice3p.params.dimensionals.convection.DimensionalsPhysical property), 18
I
 I (seaice3p.params.dimensionals.numerical.NumericalParams property), 26
 ice_emissivity (seaice3p.params.dimensionals.forcing.DimensionalsPhysical property), 21

ice_type (seaice3p.params.dimensionals.forcing.DimensionalsIceType attribute), 21

ice_type (seaice3p.params.dimensionals.forcing.DimensionalsIceType attribute), 22

initial_conditions_config (seaice3p.params.dimensionals.dimensionals.DimensionalsParameters attribute), 20

initial_conditions_config (seaice3p.params.params.Config attribute), 33

initial_ice_bulk_salinity (seaice3p.params.dimensionals.initial_conditions.DimensionalsIceBulkSalinity attribute), 25

initial_ice_bulk_salinity (seaice3p.params.initial_conditions.OilInitialConditions attribute), 32

initial_ice_depth (seaice3p.params.dimensionals.initial_conditions.DimensionalsIceDepth attribute), 25

initial_ice_depth (seaice3p.params.initial_conditions.OilInitialConditions attribute), 32

initial_ice_temperature (seaice3p.params.dimensionals.initial_conditions.DimensionalsIceTemperature attribute), 25

initial_ice_temperature (seaice3p.params.initial_conditions.OilInitialConditions attribute), 32

initial_ocean_temperature (seaice3p.params.dimensionals.initial_conditions.DimensionalsOceanTemperature attribute), 25

initial_ocean_temperature (seaice3p.params.initial_conditions.OilInitialConditions attribute), 32

initial_oil_volume_fraction (seaice3p.params.dimensionals.initial_conditions.DimensionalsOilVolumeFraction attribute), 25

initial_oil_volume_fraction (seaice3p.params.initial_conditions.OilInitialConditions attribute), 32

liquid_bulk_density (seaice3p.params.dimensionals.water.DimensionalsWaterBulkDensity attribute), 26

liquid_bulk_density (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 36

liquid_bulk_density (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 38

liquid_bulk_density (seaice3p.state.equilibrium_state.EQMStateFull attribute), 39

liquid_bulk_salinity (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 36

liquid_bulk_salinity (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 37

liquid_bulk_salinity (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 38

liquid_bulk_salinity (seaice3p.state.equilibrium_state.EQMStateFull attribute), 39

liquid_salinity (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 37

liquid_salinity (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 38

liquid_salinity (seaice3p.state.equilibrium_state.EQMStateFull attribute), 39

liquid_thermal_conductivity (seaice3p.params.convert.Scales attribute), 20

liquid_thermal_conductivity (seaice3p.params.dimensionals.water.DimensionalsWaterParameters attribute), 27

liquid_viscosity (seaice3p.params.dimensionals.water.DimensionalsWaterViscosity attribute), 27

load() (seaice3p.params.params.Config class method), 33

load_data() (in module seaice3p.load), 41

LW_forcing (seaice3p.params.dimensionals.forcing.DimensionalsRadForcing attribute), 31

LW_forcing (seaice3p.params.forcing.RadForcing attribute), 31

LW_irradiance (seaice3p.params.dimensionals.forcing.DimensionalsConstantLW attribute), 21

L

latent_heat (seaice3p.params.dimensionals.water.DimensionalsLatentHeat attribute), 26

lengthscale (seaice3p.params.convert.Scales attribute), 30

lengthscale (seaice3p.params.dimensionals.dimensionals.DimensionalsParameters attribute), 20

lewis_gas (seaice3p.params.dimensionals.dimensionals.DimensionalsLewisGas property), 20

lewis_gas (seaice3p.params.physical.BasePhysicalParameters attribute), 34

lewis_salt (seaice3p.params.dimensionals.water.DimensionalsWaterLewisSalt property), 26

lewis_salt (seaice3p.params.physical.BasePhysicalParameters attribute), 34

M

main() (in module seaice3p.diagnostics.brine_drainage_parameterisation), 1

main() (in module seaice3p.example), 39

maximum_bubble_radius (seaice3p.params.dimensionals.bubble.DimensionalsPowerLawBubbleParameters attribute), 18

maximum_bubble_radius_scaled (seaice3p.params.bubble.PowerLawBubbleParameters attribute), 28

maximum_bubble_radius_scaled (seaice3p.params.dimensionals.bubble.DimensionalsPowerLawBubbleParameters property), 18

minimum_bubble_radius (seaice3p.params.dimensionals.bubble.DimensionalsPowerLawBubbleParameters attribute), 18

minimum_bubble_radius_scaled
 (*seai3p.params.bubble.PowerLawBubbleParams*
 attribute), 28
 minimum_bubble_radius_scaled
 (*seai3p.params.dimensionals.bubble.DimensionalPowerLawBubbleParams*
 property), 18
 module
 seai3p, 42
 seai3p.diagnostics, 1
 seai3p.diagnostics.brine_drainage_parameterization, 1
 seai3p.enthalpy_method, 2
 seai3p.enthalpy_method.common, 1
 seai3p.enthalpy_method.enthalpy_method, 2
 seai3p.enthalpy_method.gas, 2
 seai3p.enthalpy_method.phase_boundaries, 2
 seai3p.equations, 12
 seai3p.equations.equations, 11
 seai3p.equations.flux, 8
 seai3p.equations.flux.bulk_dissolved_gas_flux, 6
 seai3p.equations.flux.bulk_gas_flux, 7
 seai3p.equations.flux.gas_fraction_flux, 7
 seai3p.equations.flux.heat_flux, 7
 seai3p.equations.flux.salt_flux, 8
 seai3p.equations.nucleation, 11
 seai3p.equations.radiative_heating, 11
 seai3p.equations.RJW14, 6
 seai3p.equations.RJW14.brine_channel_sink_terms, 3
 seai3p.equations.RJW14.brine_drainage, 3
 seai3p.equations.velocities, 11
 seai3p.equations.velocities.bubble_parameters, 8
 seai3p.equations.velocities.mono_distribution, 8
 seai3p.equations.velocities.power_law_distribution, 9
 seai3p.equations.velocities.velocities, 10
 seai3p.example, 39
 seai3p.forcing, 16
 seai3p.forcing.boundary_conditions, 16
 seai3p.forcing.radiative_forcing, 16
 seai3p.forcing.surface_energy_balance, 16
 seai3p.forcing.surface_energy_balance.surface_energy_balance, 12
 seai3p.forcing.surface_energy_balance.turbulent_heat_flux, 14
 seai3p.forcing.temperature_forcing, 16
 seai3p.grids, 39
 seai3p.initial_conditions, 41
 seai3p.load, 41
 seai3p.run_simulation, 41
 seai3p.params, 35
 seai3p.params.bubble, 27
 seai3p.params.convection, 29
 seai3p.params.convert, 29
 seai3p.params.dimensionals, 27
 seai3p.params.dimensionals.bubble, 16
 seai3p.params.dimensionals.convection, 18
 seai3p.params.dimensionals.dimensionals, 19
 seai3p.params.dimensionals.forcing, 21
 seai3p.params.dimensionals.gas, 24
 seai3p.params.dimensionals.initial_conditions, 25
 seai3p.params.dimensionals.numerical, 26
 seai3p.params.dimensionals.water, 26
 seai3p.params.forcing, 30
 seai3p.params.initial_conditions, 32
 seai3p.params.params, 33
 seai3p.params.physical, 34
 seai3p.printing, 42
 seai3p.run_simulation, 42
 seai3p.state, 39
 seai3p.state.disequilibrium_state, 35
 seai3p.state.equilibrium_state, 37
 MonoBubbleParams (*class in seai3p.params.bubble*),
 N
 name (*seai3p.params.dimensionals.dimensionals.DimensionalParams*
 attribute), 20
 name (*seai3p.params.params.Config attribute*), 33
 NoBrineConvection (*class in*
 seai3p.params.dimensionals.convection),
 18
 distribution, timescale
 (*seai3p.params.dimensionals.gas.DimensionalDISEQGasParam*
 attribute), 24
 number_of_cells (*seai3p.grids.Grids attribute*), 40
 numerical_params (*seai3p.params.dimensionals.dimensionals.DimensionalParams*
 attribute), 20
 numerical_params (*seai3p.params.params.Config*
 attribute), 33
 NumericalParams (*class in*
 seai3p.params.dimensionals.numerical),
 18
 ocean_bulk_salinity

(*seai3p.params.forcing.BaseOceanForcing* attribute), 31

ocean_bulk_salinity (*seai3p.params.forcing.BRW09Forcing* attribute), 31

ocean_freezing_temperature (*seai3p.params.convert.Scales* attribute), 30

ocean_freezing_temperature (*seai3p.params.dimensionals.water.DimensionalsWaterParams* attribute), 27

ocean_gas_sat (*seai3p.params.forcing.BaseOceanForcing* attribute), 31

ocean_gas_sat (*seai3p.params.forcing.BRW09Forcing* attribute), 31

ocean_salinity (*seai3p.params.convert.Scales* attribute), 30

ocean_salinity (*seai3p.params.dimensionals.water.DimensionalsWaterParams* attribute), 27

ocean_saturation_state (*seai3p.params.dimensionals.gas.DimensionalsEQMGasParams* attribute), 24

ocean_temp (*seai3p.params.forcing.BaseOceanForcing* attribute), 31

ocean_temperature (*seai3p.params.dimensionals.water.DimensionalsWaterParams* attribute), 27

offset (*seai3p.params.dimensionals.forcing.DimensionalsYearlyForcing* attribute), 24

offset (*seai3p.params.forcing.YearlyForcing* attribute), 32

oil_heating (*seai3p.params.dimensionals.forcing.DimensionalsRadForcing* attribute), 23

oil_heating (*seai3p.params.forcing.RadForcing* attribute), 32

oil_mass_ratio (*seai3p.params.dimensionals.forcing.DimensionalsBackgroundOilHeating* attribute), 21

OilInitialConditions (class in *seai3p.params.initial_conditions*), 32

P

period (*seai3p.params.dimensionals.forcing.DimensionalsYearlyForcing* attribute), 24

period (*seai3p.params.forcing.YearlyForcing* attribute), 32

phase_average_conductivity (*seai3p.params.dimensionals.water.DimensionalsWaterParams* attribute), 27

phase_average_conductivity (*seai3p.params.physical.BasePhysicalParams* attribute), 34

physical_params (*seai3p.params.params.Config* attribute), 33

pore_radius (*seai3p.params.dimensionals.bubble.DimensionalsBaseBubbleParams* attribute), 17

pore_throat_scaling (*seai3p.params.bubble.BaseBubbleParams* attribute), 28

pore_throat_scaling (*seai3p.params.dimensionals.bubble.DimensionalsBaseBubbleParams* attribute), 17

porosity_threshold (*seai3p.params.bubble.BaseBubbleParams* attribute), 28

porosity_threshold (*seai3p.params.dimensionals.bubble.DimensionalsBaseBubbleParams* attribute), 17

porosity_threshold_value (*seai3p.params.bubble.BaseBubbleParams* attribute), 28

porosity_threshold_value (*seai3p.params.dimensionals.bubble.DimensionalsBaseBubbleParams* attribute), 17

PowerLawBubbleParams (class in *seai3p.params.bubble*), 28

R

RadForcing (class in *seai3p.params.forcing*), 31

Rayleigh_critical (*seai3p.params.convection.RJW14Params* attribute), 29

Rayleigh_critical (*seai3p.params.dimensionals.convection.DimensionalsRJW14Params* attribute), 29

Rayleigh_salt (*seai3p.params.convection.RJW14Params* attribute), 29

Rayleigh_salt (*seai3p.params.dimensionals.dimensionals.DimensionalsRJW14Params* attribute), 20

ref_height (*seai3p.params.dimensionals.forcing.DimensionalsConstantT* attribute), 22

reference_permeability (*seai3p.params.dimensionals.convection.DimensionalsRJW14Params* attribute), 18

regularization (*seai3p.params.dimensionals.numerical.NumericalParams* attribute), 26

RJW14Params (class in *seai3p.params.convection*), 29

run_batch() (in module *seai3p.run_simulation*), 42

S

salinity_difference (*seai3p.params.convert.Scales* attribute), 30

salinity_difference (*seai3p.params.dimensionals.water.DimensionalsWaterParams* attribute), 27

salt (*seai3p.state.disequilibrium_state.DISEQState* attribute), 35

salt (*seai3p.state.disequilibrium_state.DISEQStateBCs* attribute), 36

salt (*seai3p.state.disequilibrium_state.DISEQStateFull* attribute), 37

salt (*seai3p.state.disequilibrium_state.EQMState* attribute), 37

salt (*seaice3p.state.equilibrium_state.EQMStateBCs attribute*), 38
 salt (*seaice3p.state.equilibrium_state.EQMStateFull attribute*), 39
 salt_diffusivity (*seaice3p.params.dimensionsal.water.Dimensionality attribute*), 27
 saturation_concentration (*seaice3p.params.convert.Scales attribute*), 30
 saturation_concentration (*seaice3p.params.dimensionsal.gas.Dimensionality attribute*), 24
 save() (*seaice3p.params.dimensionsal.dimensionsal.Dimensionality method*), 20
 save() (*seaice3p.params.params.Config method*), 33
 savefreq (*seaice3p.params.dimensionsal.dimensionsal.Dimensionality property*), 21
 savefreq (*seaice3p.params.params.Config attribute*), 33
 savefreq_in_days (*seaice3p.params.dimensionsal.dimensionsal.Dimensionality attribute*), 21
 Scales (*class in seaice3p.params.convert*), 29
 scales (*seaice3p.params.dimensionsal.dimensionsal.Dimensionality property*), 21
 scales (*seaice3p.params.params.Config attribute*), 33
 seaice3p
 module, 42
 seaice3p.diagnostics
 module, 1
 seaice3p.diagnostics.brine_drainage_parameterisation
 module, 1
 seaice3p.enthalpy_method
 module, 2
 seaice3p.enthalpy_method.common
 module, 1
 seaice3p.enthalpy_method.enthalpy_method
 module, 2
 seaice3p.enthalpy_method.gas
 module, 2
 seaice3p.enthalpy_method.phase_boundaries
 module, 2
 seaice3p.equations
 module, 12
 seaice3p.equations.equations
 module, 11
 seaice3p.equations.flux
 module, 8
 seaice3p.equations.flux.bulk_dissolved_gas_flux
 module, 6
 seaice3p.equations.flux.bulk_gas_flux
 module, 7
 seaice3p.equations.flux.gas_fraction_flux
 module, 7
 seaice3p.equations.flux.heat_flux
 module, 7
 seaice3p.equations.flux.salt_flux
 module, 8
 seaice3p.equations.nucleation
 module, 11
 seaice3p.equations.radiative_heating
 module, 11
 seaice3p.equations.RJW14
 module, 6
 seaice3p.equations.RJW14.brine_channel_sink_terms
 module, 3
 seaice3p.equations.RJW14.brine_drainage
 module, 3
 seaice3p.equations.velocities
 module, 11
 seaice3p.equations.velocities.bubble_parameters
 module, 8
 seaice3p.equations.velocities.mono_distribution
 module, 8
 seaice3p.equations.velocities.power_law_distribution
 module, 9
 seaice3p.equations.velocities.velocities
 module, 10
 seaice3p.example
 module, 39
 seaice3p.forcing
 module, 16
 seaice3p.forcing.boundary_conditions
 module, 16
 seaice3p.forcing.radiative_forcing
 module, 16
 seaice3p.forcing.surface_energy_balance
 module, 16
 seaice3p.forcing.surface_energy_balance.surface_energy_balance
 module, 12
 seaice3p.forcing.surface_energy_balance.turbulent_heat_flux
 module, 14
 seaice3p.forcing.temperature_forcing
 module, 16
 seaice3p.grids
 module, 39
 seaice3p.initial_conditions
 module, 41
 seaice3p.load
 module, 41
 seaice3p.oil_simulation
 module, 41
 seaice3p.params
 module, 35
 seaice3p.params.bubble
 module, 27
 seaice3p.params.convection
 module, 29
 seaice3p.params.convert
 module, 29

seaice3p.params.dimensionals
 module, 27
 seaice3p.params.dimensionals.bubble
 module, 16
 seaice3p.params.dimensionals.convection
 module, 18
 seaice3p.params.dimensionals.dimensionals
 module, 19
 seaice3p.params.dimensionals.forcing
 module, 21
 seaice3p.params.dimensionals.gas
 module, 24
 seaice3p.params.dimensionals.initial_conditions
 module, 25
 seaice3p.params.dimensionals.numerical
 module, 26
 seaice3p.params.dimensionals.water
 module, 26
 seaice3p.params.forcing
 module, 30
 seaice3p.params.initial_conditions
 module, 32
 seaice3p.params.params
 module, 33
 seaice3p.params.physical
 module, 34
 seaice3p.printing
 module, 42
 seaice3p.run_simulation
 module, 42
 seaice3p.state
 module, 39
 seaice3p.state.disequilibrium_state
 module, 35
 seaice3p.state.equilibrium_state
 module, 37
 solid_fraction (seaice3p.state.disequilibrium_state.DISEQStateFull
 attribute), 37
 solid_fraction (seaice3p.state.equilibrium_state.EQMStateFull
 attribute), 39
 solid_thermal_conductivity
 (seaice3p.params.dimensionals.water.DimensionalsWaterParams
 attribute), 27
 solve() (in module seaice3p.run_simulation), 42
 solve_for_surface_temp() (in module
 seaice3p.forcing.surface_energy_balance.surface_energy_balance)
 13
 specific_heat_capacity
 (seaice3p.params.dimensionals.water.DimensionalsWaterParams
 attribute), 27
 specific_humidity (seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux
 attribute), 22
 stefan_number (seaice3p.params.dimensionals.water.DimensionalsWaterParams
 property), 27
 stefan_number (seaice3p.params.physical.BasePhysicalParams
 attribute), 34
 step (seaice3p.grids.Grids property), 40
 step (seaice3p.params.dimensionals.numerical.NumericalParams
 property), 26
 surface_temp_gradient() (in module
 seaice3p.forcing.surface_energy_balance.surface_energy_balance)
 13
 SW_albedo (seaice3p.params.dimensionals.forcing.DimensionalsConstantSW
 attribute), 22
 SW_forcing (seaice3p.params.dimensionals.forcing.DimensionalsRadForcing
 attribute), 23
 SW_forcing (seaice3p.params.forcing.RadForcing at-
 tribute), 31
 SW_irradiance (seaice3p.params.dimensionals.forcing.DimensionalsConstantSW
 attribute), 22
 SW_penetration_fraction
 (seaice3p.params.dimensionals.forcing.DimensionalsConstantSWF
 attribute), 22
 T
 temperature (seaice3p.state.disequilibrium_state.DISEQStateBCs
 attribute), 36
 temperature (seaice3p.state.disequilibrium_state.DISEQStateFull
 attribute), 37
 temperature (seaice3p.state.equilibrium_state.EQMStateBCs
 attribute), 38
 temperature (seaice3p.state.equilibrium_state.EQMStateFull
 attribute), 39
 temperature_difference
 (seaice3p.params.convert.Scales attribute),
 30
 temperature_difference
 (seaice3p.params.dimensionals.water.DimensionalsWaterParams
 property), 27
 thermal_diffusivity
 (seaice3p.params.convert.Scales attribute),
 30
 thermal_diffusivity
 (seaice3p.params.dimensionals.water.DimensionalsWaterParams
 property), 27
 time (seaice3p.state.disequilibrium_state.DISEQState
 attribute), 35
 time (seaice3p.state.disequilibrium_state.DISEQStateBCs
 attribute), 36
 time (seaice3p.state.disequilibrium_state.DISEQStateFull
 attribute), 37
 time (seaice3p.state.equilibrium_state.EQMState at-
 tribute), 37
 time (seaice3p.state.equilibrium_state.EQMStateBCs at-
 tribute), 38
 time (seaice3p.state.equilibrium_state.EQMStateFull at-
 tribute), 39

`time_scale` (*seaice3p.params.convert.Scales* property),
 30
`tolerable_super_saturation_fraction`
 (*seaice3p.params.dimensionals.gas.DimensionaleQMGasParams*
attribute), 24
`tolerable_super_saturation_fraction`
 (*seaice3p.params.physical.BasePhysicalParams*
attribute), 34
`top_cell_conductivity()` (in module
seaice3p.forcing.surface_energy_balance.surface_energy_balance),
 13
`total_time` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*
property), 21
`total_time` (*seaice3p.params.params.Config* attribute),
 33
`total_time_in_days` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*
attribute), 21
`turbulent_flux` (*seaice3p.params.dimensionals.forcing.DimensionalsRadForcing*
attribute), 23
`turbulent_flux` (*seaice3p.params.forcing.RadForcing*
attribute), 32

U

`UniformInitialConditions` (class in
seaice3p.params.dimensionals.initial_conditions),
 25
`upwind()` (in module *seaice3p.grids*), 40

V

`velocity_scale` (*seaice3p.params.convert.Scales* prop-
erty), 30

W

`water_emissivity` (*seaice3p.params.dimensionals.forcing.DimensionalsConstantLWForcing*
attribute), 22
`water_params` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*
attribute), 21
`windspeed` (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*
attribute), 22

Y

`YearlyForcing` (class in *seaice3p.params.forcing*), 32