

---

# **seaice3p**

***Release 0.23.0***

**Joseph Fishlock**

**Oct 18, 2024**



**CONTENTS:**

<b>1</b>	<b>seaice3p</b>	<b>1</b>
1.1	seaice3p package . . . . .	1
<b>2</b>	<b>Indices and tables</b>	<b>49</b>
	<b>Python Module Index</b>	<b>51</b>
	<b>Index</b>	<b>53</b>



## 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`]]]

## 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
- **cfg** (*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, cfg: 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
- **cfg** (*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, cfg: 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
- **cfg** (*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, cfg: 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\_bubble\_flux(*gas\_fraction*,  
*liquid\_fraction*,  
*D\_g*, *cfg*: [Config](#))

seaice3p.equations.flux.bulk\_gas\_flux.calculate\_diffusive\_gas\_flux(*dissolved\_gas*,  
*liquid\_fraction*, *D\_g*, *cfg*:  
[Config](#))

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*, *D\_g*,  
*cfg*: [Config](#))

**seaice3p.equations.flux.heat\_flux module**

seaice3p.equations.flux.heat\_flux.calculate\_advective\_heat\_flux(*temperature*, *liquid\_fraction*,  
*Wl*)

seaice3p.equations.flux.heat\_flux.calculate\_conductive\_heat\_flux(*state\_BCs*, *D\_g*, *cfg*)

Calculate conductive heat flux as

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

**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\_conductivity(*cfg*: [Config](#), *solid\_fraction*: *ndarray*[*Any*,  
*dtype*[\_*ScalarType\_co*]] | *float*) →  
*ndarray*[*Any*, *dtype*[\_*ScalarType\_co*]] |  
*float*

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.heat_flux.pure_liquid_switch(liquid_fraction: ndarray[Any,
                                                    dtype[_ScalarType_co]] | float) →
                                                    ndarray[Any, dtype[_ScalarType_co]] | float
```

Take the liquid fraction and return a smoothed switch that is equal to 1 in a pure liquid region and goes to zero rapidly outside of this

## 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: Config)
```

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 polydisperse case.

Bubble size fraction is given as a scalar input to calculate

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

`seai3p.equations.velocities.power_law_distribution.calculate_power_law_lag_factor(liquid_fraction, cfg: Con-fig)`

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

Return on edge grid

`seai3p.equations.velocities.power_law_distribution.calculate_power_law_wall_drag_factor(liquid_fraction, cfg: Con-fig)`

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

Return on edge grid

`seai3p.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.

`seai3p.equations.velocities.power_law_distribution.calculate_wall_drag_integral(bubble_size_fraction_min: float, bubble_size_fraction_max: float, cfg: Config)`

`seai3p.equations.velocities.power_law_distribution.calculate_wall_drag_integrand(bubble_size_fraction: float, cfg: Config)`

Scalar function to calculate wall drag integrand for polydisperse 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.

## seai3p.equations.velocities module

`seai3p.equations.velocities.calculate_frame_velocity(cfg: Config)`

`seai3p.equations.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.

seaice3p.equations.radiative\_heating.run\_two\_stream\_model(*state\_bcs*: EQMStateBCs | DISEQStateBCs, *cfg*: Config, *grids*: Grids) → SpectralIrradiance

## 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.find_ghost_cell_temperature`(*state*: EQM-State-Full | DIS-E-QS-tate-Full, *cfg*: Con-fig) → float

Returns non dimensional ghost cell temperature such that surface heat flux is the sum of incoming LW, outgoing LW, sensible and latent heat fluxes. The SW heat flux is determined in the radiative heating term.

### 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_latent_heat_flux`(*cfg*: Con-fig, *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_sensible_heat_flux(cfg:
    Config,
    time:
    float,
    top_cell_is_ice:
    bool,
    surface_temp:
    float)
    →
    float
```

Calculate sensible heat flux from [2]

## 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.radiative_forcing.get_SW_penetration_fraction(state_bcs: EQMStateBCs |
    DISEQStateBCs, cfg: Config) →
    float
```

## 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(state: EQMStateFull |  
                                                                    DISEQStateFull, 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.DimensionalBaseBubbleParams(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.DimensionalMonoBubbleParams(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: *DimensionalBaseBubbleParams*

**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.DimensionalPowerLawBubbleParams(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

$$\Lambda = R_B/R_0$$

**seaice3p.params.dimensionalconvection module**

```
class seaice3p.params.dimensionalconvection.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,
    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

**reference\_permeability:** float = 1e-08

```
class seaice3p.params.dimensionalconvection.NoBrineConvection
```

Bases: object

No brine convection

**seaice3p.params.dimensionaldimensional 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 seai3p.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 |
    DimensionalRobinForcing |
    DimensionalERASForcing,
    ocean_forcing_config: DimensionalBRW09OceanForcing |
    DimensionalFixedTempOceanForcing |
    DimensionalFixedHeatFluxOceanForcing,
    initial_conditions_config:
    DimensionalOilInitialConditions
    | UniformInitialConditions |
    BRW09InitialConditions |
    PreviousSimulation,
    water_params:
    DimensionalWaterParams =
    DimensionalWaterParams(liquid_density=1028,
    ice_density=916,
    ocean_salinity=34,
    liquidus=LinearLiquidus(eutectic_temperature=-
    21.1, eutectic_salinity=270),
    latent_heat=334000.0, liquid_specific_heat_capacity=4184,
    solid_specific_heat_capacity=2009,
    liquid_thermal_conductivity=0.54,
    solid_thermal_conductivity=2.22,
    snow_thermal_conductivity=0.31,
    eddy_diffusivity=0,
    salt_diffusivity=0, haline_contraction_coefficient=0.00075,
    liquid_viscosity=0.00278),
    numerical_params:
    NumericalParams =
    NumericalParams(I=50,
    regularisation=1e-06,
    solver_choice='RK23'),
    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* | *DimensionalRobinForcing* | *DimensionalERA5Forcing*

#### 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* | *PreviousSimulation*

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, solver\_choice='RK23')

**ocean\_forcing\_config:** DimensionalBRW09OceanForcing | DimensionalFixedTempOceanForcing | DimensionalFixedHeatFluxOceanForcing

**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, ice\_density=916, ocean\_salinity=34, liquidus=LinearLiquidus(eutectic\_temperature=-21.1, eutectic\_salinity=270), latent\_heat=334000.0, liquid\_specific\_heat\_capacity=4184, solid\_specific\_heat\_capacity=2009, liquid\_thermal\_conductivity=0.54, solid\_thermal\_conductivity=2.22, snow\_thermal\_conductivity=0.31, eddy\_diffusivity=0, salt\_diffusivity=0, haline\_contraction\_coefficient=0.00075, 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, median\_oil\_droplet\_radius: float = 0.5, ice\_type: str = 'FYI', fast\_solve: bool = False, wavelength\_cutoff: float | None = 1200*)

Bases: object



```

fast_solve: bool = False

ice_type: str = 'FYI'

median_oil_droplet_radius: float = 0.5

oil_mass_ratio: float = 0

wavelength_cutoff: float | None = 1200

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_min_wavelength:
                                                                    float = 350,
                                                                    SW_max_wavelength:
                                                                    float = 3000,
                                                                    num_wavelength_samples:
                                                                    int = 7,
                                                                    SW_penetration_fraction:
                                                                    float = 0.4)

    Bases: object

    SW_irradiance: float = 280

    SW_max_wavelength: float = 3000

    SW_min_wavelength: float = 350

    SW_penetration_fraction: float = 0.4

    num_wavelength_samples: int = 7

```

```
class seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux(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

Parameters for calculating the turbulent surface sensible and latent heat fluxes

NOTE: If you are running a simulation with ERA5 reanalysis forcing you must set the ref\_height=2m as this is the appropriate value for the atmospheric reanalysis quantities

**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

```

class seaice3p.params.dimensional.forcing.DimensionalERA5Forcing(data_path: Path, start_date:
    str, use_snow_data: bool =
    False, SW_forcing: DimensionalConstantSWForcing =
    DimensionalConstantSWForcing(SW_irradiance=280,
    SW_min_wavelength=350,
    SW_max_wavelength=3000,
    num_wavelength_samples=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,
    median_oil_droplet_radius=0.5,
    ice_type='FYI',
    fast_solve=False,
    wavelength_cutoff=1200))

```

Bases: object

read ERA5 data from netCDF file located at data\_path.

Simulation will take atmospheric forcings from the start date specified in the format YYYY-MM-DD

```

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

```

```

SW_forcing: DimensionalConstantSWForcing =
DimensionalConstantSWForcing(SW_irradiance=280, SW_min_wavelength=350,
SW_max_wavelength=3000, num_wavelength_samples=7, SW_penetration_fraction=0.4)

```

data\_path: Path

```

oil_heating: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating |
DimensionalNoHeating = DimensionalBackgroundOilHeating(oil_mass_ratio=0,
median_oil_droplet_radius=0.5, ice_type='FYI', fast_solve=False,
wavelength_cutoff=1200)

```

```
start_date: str

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)

use_snow_data: bool = False

seaice3p.params.dimensional.forcing.DimensionalLWForcing
    alias of DimensionalConstantLWForcing

class seaice3p.params.dimensional.forcing.DimensionalMobileOilHeating(ice_type: str = 'FYI',
                                                                    fast_solve: bool = False,
                                                                    wavelength_cutoff: float |
                                                                    None = 1200)

    Bases: object

    fast_solve: bool = False

    ice_type: str = 'FYI'

    wavelength_cutoff: float | None = 1200

class seaice3p.params.dimensional.forcing.DimensionalNoHeating

    Bases: object
```

```

class seaice3p.params.dimensional.forcing.DimensionaRadForcing(SW_forcing:
    seaice3p.params.dimensional.forcing.Dimensiona
    = DimensionalConstantSWForc-
    ing(SW_irradiance=280,
    SW_min_wavelength=350,
    SW_max_wavelength=3000,
    num_wavelength_samples=7,
    SW_penetration_fraction=0.4),
    LW_forcing:
    seaice3p.params.dimensional.forcing.Dimensiona
    = DimensionalConstantLWForc-
    ing(LW_irradiance=260,
    ice_emissivity=0.99,
    water_emissivity=0.97),
    turbulent_flux:
    seaice3p.params.dimensional.forcing.Dimensiona
    = DimensionalConstantTurbu-
    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.dimensional.forcing.Dimensiona
    |
    seaice3p.params.dimensional.forcing.Dimensiona
    |
    seaice3p.params.dimensional.forcing.Dimensiona
    = DimensionalBackgroundOil-
    Heating(oil_mass_ratio=0,
    median_oil_droplet_radius=0.5,
    ice_type='FYI',
    fast_solve=False,
    wavelength_cutoff=1200))

```

Bases: object

```

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

SW_forcing: DimensionalConstantSWForcing =
DimensionalConstantSWForcing(SW_irradiance=280, SW_min_wavelength=350,
SW_max_wavelength=3000, num_wavelength_samples=7, SW_penetration_fraction=0.4)

oil_heating: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating |
DimensionalNoHeating = DimensionalBackgroundOilHeating(oil_mass_ratio=0,
median_oil_droplet_radius=0.5, ice_type='FYI', fast_solve=False,
wavelength_cutoff=1200)

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 seai3p.params.dimensional.forcing.DimensionRobinForcing(heat_transfer_coefficient:
    float = 6.3,
    restoring_temperature: float =
    -30)
```

Bases: object

This forcing imposes a Robin boundary condition of the form  $\text{surface\_heat\_flux} = \text{heat\_transfer\_coefficient} * (\text{restoring\_temp} - \text{surface\_temp})$

**heat\_transfer\_coefficient:** float = 6.3

**restoring\_temperature:** float = -30

```
seai3p.params.dimensional.forcing.DimensionSWForcing
```

alias of *DimensionalConstantSWForcing*

```
seai3p.params.dimensional.forcing.DimensionTurbulentFlux
```

alias of *DimensionalConstantTurbulentFlux*

```
class seai3p.params.dimensional.forcing.DimensionYearlyForcing(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

## seai3p.params.dimensional.gas module

```
class seai3p.params.dimensional.gas.DimensionDISEQGasParams(gas_density: float = 1,
    saturation_concentration: float
    = 1e-05, ocean_saturation_state:
    float = 1.0, gas_diffusivity: float
    = 0, tolerance_super_saturation_fraction:
    float = 1, gas_viscosity: float =
    0, gas_bubble_eddy_diffusion:
    bool = False,
    nucleation_timescale: float =
    6869075)
```

Bases: *\_DimensionalGasParams*

**nucleation\_timescale:** float = 6869075

```
class seai3p.params.dimensional.gas.DimensionEQMGasParams(gas_density: float = 1,
    saturation_concentration: float =
    1e-05, ocean_saturation_state:
    float = 1.0, gas_diffusivity: float =
    0, tolerance_super_saturation_fraction:
    float = 1, gas_viscosity: float = 0,
    gas_bubble_eddy_diffusion: bool =
    False)
```

Bases: `_DimensionalGasParams`

### 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,
                                                                    ini-
                                                                    tial_oil_free_depth:
                                                                    float
                                                                    = 0)
```

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\_free\_depth:** `float = 0`

**initial\_oil\_volume\_fraction:** `float = 1e-07`

```
class seaice3p.params.dimensional.initial_conditions.PreviousSimulation(data_path:
                                                                    pathlib.Path)
```

Bases: `object`

**data\_path:** Path

**class** seaice3p.params.dimensionsal.initial\_conditions.UniformInitialConditions

Bases: object

values for bottom (ocean) boundary

### seaice3p.params.dimensionsal.numerical module

**class** seaice3p.params.dimensionsal.numerical.NumericalParams(*I: int = 50, regularisation: float = 1e-06, solver\_choice: str = 'RK23'*)

Bases: object

parameters needed for discretisation and choice of numerical method

**I:** int = 50

**regularisation:** float = 1e-06

**solver\_choice:** str = 'RK23'

property step

### seaice3p.params.dimensionsal.water module

**class** seaice3p.params.dimensionsal.water.CubicLiquidus(*eutectic\_temperature: float = -21.1, a0: float = -1.2, a1: float = -21.8, a2: float = -0.919, a3: float = -0.0178*)

Bases: object

Cubic fit to liquidus to give liquidus salinity in terms of temperature

$S = a_0 + a_1 T + a_2 T^2 + a_3 T^3$

defaults are taken from Notz PhD thesis for fit to Assur seawater data

**a0:** float = -1.2

**a1:** float = -21.8

**a2:** float = -0.919

**a3:** float = -0.0178

**eutectic\_temperature:** float = -21.1

**get\_liquidus\_salinity**(*temperature*)

**get\_liquidus\_temperature**(*salinity*)



```

class seaice3p.params.dimensionsal.water.DimensionaWaterParams(liquid_density: float = 1028,
                                                                ice_density: float = 916,
                                                                ocean_salinity: float = 34,
                                                                liquidus:
                                                                seaice3p.params.dimensionsal.water.LinearLiquidus
                                                                |
                                                                seaice3p.params.dimensionsal.water.CubicLiquidus
                                                                =
                                                                LinearLiquidus(eutectic_temperature=-
                                                                21.1, eutectic_salinity=270),
                                                                latent_heat: float = 334000.0,
                                                                liquid_specific_heat_capacity:
                                                                float = 4184,
                                                                solid_specific_heat_capacity:
                                                                float = 2009,
                                                                liquid_thermal_conductivity: float
                                                                = 0.54,
                                                                solid_thermal_conductivity: float
                                                                = 2.22,
                                                                snow_thermal_conductivity: float
                                                                = 0.31, eddy_diffusivity: float = 0,
                                                                salt_diffusivity: float = 0,
                                                                haline_contraction_coefficient:
                                                                float = 0.00075, 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

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

**eddy\_diffusivity: float = 0**

**property eddy\_diffusivity\_ratio**

Calculate the ratio of eddy diffusivity to thermal diffusivity in the liquid phase

$$\lambda = \frac{\kappa_{\text{turbulent}}}{\kappa_l}$$

**property eutectic\_salinity**

**property eutectic\_temperature**

**haline\_contraction\_coefficient: float = 0.00075**

**ice\_density: float = 916**

**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\_specific\_heat\_capacity:** float = 4184

**liquid\_thermal\_conductivity:** float = 0.54

**liquid\_viscosity:** float = 0.00278

**liquidus:** *LinearLiquidus* | *CubicLiquidus* =  
*LinearLiquidus*(eutectic\_temperature=-21.1, eutectic\_salinity=270)

**property ocean\_freezing\_temperature**

calculate salinity dependent freezing temperature using linear liquidus with ocean salinity

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

or using a cubic fit for the liquidus curve

**ocean\_salinity:** float = 34

**property salinity\_difference**

calculate difference between eutectic salinity and typical ocean salinity

$$\Delta S = S_E - S_i$$

**salt\_diffusivity:** float = 0

**property snow\_conductivity\_ratio**

Calculate the ratio of snow to liquid thermal conductivity

$$\lambda = \frac{k_{sn}}{k_l}$$

**snow\_thermal\_conductivity:** float = 0.31

**solid\_specific\_heat\_capacity:** float = 2009

**solid\_thermal\_conductivity:** float = 2.22

**property specific\_heat\_ratio**

Calculate the ratio of solid to liquid specific heat capacities

$$\lambda = \frac{c_{p,s}}{c_{p,l}}$$

**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 m<sup>2</sup>/s

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

**class** seaice3p.params.dimensional.water.**LinearLiquidus**(*eutectic\_temperature: float = -21.1,*  
*eutectic\_salinity: float = 270*)

Bases: object

**eutectic\_salinity:** float = 270

**eutectic\_temperature:** float = -21.1

## 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 seai3p.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
```

```
seai3p.params.bubble.get_dimensionless_bubble_params(dimensional_params: DimensionalParams)
                                                    → MonoBubbleParams |
                                                    PowerLawBubbleParams
```

## seai3p.params.convection module

```
class seai3p.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
```

```
seai3p.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, liquid_density:
                                     float, ice_density: float, saturation_concentration: float,
                                     pore_radius: float, haline_contraction_coefficient: 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
```

```
haline_contraction_coefficient: float
ice_density: float
lengthscale: float
liquid_density: float
liquid_thermal_conductivity: float
ocean_freezing_temperature: float
ocean_salinity: float
pore_radius: 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(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'
```

```
class seaice3p.params.forcing.ConstantForcing(constant_top_temperature: float = -1.5)
```

Bases: object

Constant temperature forcing

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

```

class seaice3p.params.forcing.ERA5Forcing(data_path: Path, start_date: str, timescale_in_days: float,
                                          use_snow_data: bool = False, SW_forcing:
                                          DimensionalConstantSWForcing =
                                          DimensionalConstantSWForcing(SW_irradiance=280,
                                          SW_min_wavelength=350, SW_max_wavelength=3000,
                                          num_wavelength_samples=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,
                                          median_oil_droplet_radius=0.5, ice_type='FYI',
                                          fast_solve=False, wavelength_cutoff=1200))

```

Bases: object

Forcing parameters for simulation forced with atmospheric variables from reanalysis data in netCDF file located at data\_path.

Never create this object directly but instead initialise from a dimensional simulation configuration as we must pass it the simulation timescale to correctly read the atmospheric variables from the netCDF file.

```

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

```

```

NEGLIGIBLE_SNOW_DEPTH: ClassVar[float] = 0.05

```

```

SW_forcing: DimensionalConstantSWForcing =
DimensionalConstantSWForcing(SW_irradiance=280, SW_min_wavelength=350,
SW_max_wavelength=3000, num_wavelength_samples=7, SW_penetration_fraction=0.4)

```

```

data_path: Path

```

```

oil_heating: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating |
DimensionalNoHeating = DimensionalBackgroundOilHeating(oil_mass_ratio=0,
median_oil_droplet_radius=0.5, ice_type='FYI', fast_solve=False,
wavelength_cutoff=1200)

```

```

start_date: str

```

```

timescale_in_days: float

```

```

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)

```

```

use_snow_data: bool = False

```

```

class seai3p.params.forcing.RadForcing(SW_forcing: DimensionalConstantSWForcing =
    DimensionalConstantSWForcing(SW_irradiance=280,
    SW_min_wavelength=350, SW_max_wavelength=3000,
    num_wavelength_samples=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,
    median_oil_droplet_radius=0.5, ice_type='FYI',
    fast_solve=False, wavelength_cutoff=1200))

```

Bases: object

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_min_wavelength=350,
SW_max_wavelength=3000, num_wavelength_samples=7, SW_penetration_fraction=0.4)

oil_heating: DimensionalBackgroundOilHeating | DimensionalMobileOilHeating |
DimensionalNoHeating = DimensionalBackgroundOilHeating(oil_mass_ratio=0,
median_oil_droplet_radius=0.5, ice_type='FYI', fast_solve=False,
wavelength_cutoff=1200)

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 seai3p.params.forcing.RobinForcing(biot: float = 12, restoring_temperature: float = -1.3)

```

Bases: object

Dimensionless forcing parameters for Robin boundary condition

**biot:** float = 12

**restoring\_temperature:** float = -1.3

```

class seai3p.params.forcing.YearlyForcing(offset: float = -1.0, amplitude: float = 0.75, period: float =
4.0)

```

Bases: object

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 | RobinForcing | ERA5Forcing

## 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, *initial\_oil\_free\_depth*: float =  
 0)

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\_free\_depth:** float = 0

**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 |  
 PreviousSim-  
 ulation

## 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 | RobinForcing | ERA5Forcing, ocean_forcing_config:
    FixedTempOceanForcing | FixedHeatFluxOceanForcing |
    BRW09OceanForcing, initial_conditions_config:
    UniformInitialConditions | BRW09InitialConditions |
    OilInitialConditions | PreviousSimulation, numerical_params:
    NumericalParams = NumericalParams(I=50, regularisation=1e-06,
    solver_choice='RK23'), 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](#) | [RobinForcing](#) | [ERA5Forcing](#)

**initial\_conditions\_config:** [UniformInitialConditions](#) | [BRW09InitialConditions](#) | [OilInitialConditions](#) | [PreviousSimulation](#)

**classmethod** load(path)

**name:** str

**numerical\_params:** [NumericalParams](#) = NumericalParams(I=50, regularisation=1e-06, solver\_choice='RK23')

**ocean\_forcing\_config:** [FixedTempOceanForcing](#) | [FixedHeatFluxOceanForcing](#) | [BRW09OceanForcing](#)

**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,
specific_heat_ratio: float = 0.5, conductivity_ratio:
float = 4.11, eddy_diffusivity_ratio: float = 0,
snow_conductivity_ratio: float = 0.574,
tolerable_super_saturation_fraction: float = 1,
gas_viscosity_ratio: float = 0,
gas_bubble_eddy_diffusion: bool = False,
get_liquidus_temperature: Callable | None = None,
get_liquidus_salinity: Callable | None = None)
```

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
eddy_diffusivity_ratio:  float = 0
expansion_coefficient:  float = 0.029
frame_velocity:  float = 0
gas_bubble_eddy_diffusion:  bool = False
gas_viscosity_ratio:  float = 0
get_liquidus_salinity:  Callable | None = None
get_liquidus_temperature:  Callable | None = None
lewis_gas:  float = inf
lewis_salt:  float = inf
snow_conductivity_ratio:  float = 0.574
specific_heat_ratio:  float = 0.5
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,
                                                    specific_heat_ratio: float = 0.5,
                                                    conductivity_ratio: float = 4.11,
                                                    eddy_diffusivity_ratio: float = 0,
                                                    snow_conductivity_ratio: float = 0.574,
                                                    tolerable_super_saturation_fraction: float = 1,
                                                    gas_viscosity_ratio: float = 0,
                                                    gas_bubble_eddy_diffusion: bool = False,
                                                    get_liquidus_temperature: Callable | None =
                                                    None, get_liquidus_salinity: Callable | None =
                                                    None, 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, specific_heat_ratio:
                                                    float = 0.5, conductivity_ratio: float = 4.11,
                                                    eddy_diffusivity_ratio: float = 0,
                                                    snow_conductivity_ratio: float = 0.574,
                                                    tolerable_super_saturation_fraction: float = 1,
                                                    gas_viscosity_ratio: float = 0,
                                                    gas_bubble_eddy_diffusion: bool = False,
                                                    get_liquidus_temperature: Callable | None = None,
                                                    get_liquidus_salinity: Callable | None = None)
```

Bases: [BasePhysicalParams](#)

non dimensional numbers for the mushy layer

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

## 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 seaice3p.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

Initialiase 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.average(points: ndarray[Any, dtype[_ScalarType_co]]) → ndarray[Any, dtype[_ScalarType_co]]`

Returns arithmetic mean of adjacent points in an array

takes ghosts -> edges -> centers

`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

```
class seaice3p.load.DISEQResults(cfg: seaice3p.params.params.Config, dcfg: None |
                                seaice3p.params.dimensionals.DimensionalsParams, times:
                                numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]], enthalpy:
                                numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]], salt:
                                numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]],
                                bulk_dissolved_gas: numpy.ndarray[typing.Any,
                                numpy.dtype[+_ScalarType_co]], gas_fraction:
                                numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]])
```

Bases: `_BaseResults`

**bulk\_dissolved\_gas:** `ndarray[Any, dtype[_ScalarType_co]]`

**property bulk\_gas:** `ndarray[Any, dtype[_ScalarType_co]]`

Dimensionless bulk gas the same as the EQM model

**gas\_fraction:** ndarray[Any, dtype[\_ScalarType\_co]]

```
class seaice3p.load.EQMResults(cfg: seaice3p.params.params.Config, dcfg: None |
    seaice3p.params.dimensionsal.dimensionsal.DimensionalParams, times:
    numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]], enthalpy:
    numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]], salt:
    numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]], bulk_gas:
    numpy.ndarray[typing.Any, numpy.dtype[+_ScalarType_co]])
```

Bases: `_BaseResults`

**bulk\_gas:** ndarray[Any, dtype[\_ScalarType\_co]]

**property gas\_fraction:** ndarray[Any, dtype[\_ScalarType\_co]]

```
seaice3p.load.load_simulation(sim_config_path: Path, sim_data_path: Path, is_dimensional: bool = True)
    → EQMResults | DISEQResults
```

### 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_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 = 5.92,
    initial_oil_free_ice_depth: float = 0,
    SW_min_wavelength=350,
    SW_max_wavelength=3000,
    num_wavelength_samples=7,
    solver_choice='RK23', eddy_diffusivity=0,
    brine_convection_params:
    DimensionalRJW14Params |
    NoBrineConvection = Dimensional-
    RJW14Params(couple_bubble_to_horizontal_flow=False,
    couple_bubble_to_vertical_flow=False,
    Rayleigh_critical=2.9,
    convection_strength=0.13,
    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](#), 48  
[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](#), 12  
[seaice3p.equations.radiative\\_heating](#), 12  
[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](#), 9  
[seaice3p.equations.velocities.power\\_law\\_distribution](#), 9  
[seaice3p.equations.velocities.velocities](#), 10  
[seaice3p.example](#), 45  
[seaice3p.forcing](#), 15  
[seaice3p.forcing.boundary\\_conditions](#), 14  
[seaice3p.forcing.radiative\\_forcing](#), 14  
[seaice3p.forcing.surface\\_energy\\_balance](#), 14  
[seaice3p.forcing.surface\\_energy\\_balance.surface\\_energy\\_balance](#), 12  
[seaice3p.forcing.surface\\_energy\\_balance.turbulent\\_heat\\_flux](#), 13  
[seaice3p.forcing.temperature\\_forcing](#), 15  
[seaice3p.grids](#), 45  
[seaice3p.initial\\_conditions](#), 46  
[seaice3p.load](#), 46  
[seaice3p.oil\\_simulation](#), 47  
[seaice3p.params](#), 40  
[seaice3p.params.bubble](#), 31  
[seaice3p.params.convection](#), 32  
[seaice3p.params.convert](#), 33  
[seaice3p.params.dimensionality](#), 31  
[seaice3p.params.dimensionality.bubble](#), 15  
[seaice3p.params.dimensionality.convection](#), 17  
[seaice3p.params.dimensionality.dimensionality](#), 17  
[seaice3p.params.dimensionality.forcing](#), 20  
[seaice3p.params.dimensionality.gas](#), 26  
[seaice3p.params.dimensionality.initial\\_conditions](#), 27  
[seaice3p.params.dimensionality.numerical](#), 28  
[seaice3p.params.dimensionality.water](#), 28  
[seaice3p.params.forcing](#), 34  
[seaice3p.params.initial\\_conditions](#), 37  
[seaice3p.params.params](#), 38  
[seaice3p.params.physical](#), 39  
[seaice3p.printing](#), 48  
[seaice3p.run\\_simulation](#), 48  
[seaice3p.state](#), 44  
[seaice3p.state.disequilibrium\\_state](#), 40  
[seaice3p.state.equilibrium\\_state](#), 43





## INDEX

### A

(*seaice3p.params.dimensionals.water.CubicLiquidus*  
*attribute*), 28  
 (*seaice3p.params.dimensionals.water.CubicLiquidus*  
*attribute*), 28  
 (*seaice3p.params.dimensionals.water.CubicLiquidus*  
*attribute*), 28  
 (*seaice3p.params.dimensionals.water.CubicLiquidus*  
*attribute*), 28  
 (in module *seaice3p.grids*), 45  
 (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
*attribute*), 22  
 (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
*attribute*), 22  
  
*(seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
*attribute*), 22  
 (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
*attribute*), 22  
 (*seaice3p.params.dimensionals.forcing.DimensionalsYearlyForcing*  
*attribute*), 26  
 (*seaice3p.params.forcing.YearlyForcing* *at-*  
*tribute*), 36  
 (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
*attribute*), 22  
 (in module *seaice3p.grids*), 45

### B

**B** (*seaice3p.params.bubble.BaseBubbleParams* *attribute*),  
 31  
**B** (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
*property*), 19  
**Barrow\_initial\_bulk\_gas\_in\_ice**  
*(seaice3p.params.dimensionals.initial\_conditions.BRW09InitialConditions*  
*attribute*), 27  
**Barrow\_top\_temperature\_data\_choice**  
*(seaice3p.params.dimensionals.forcing.DimensionalsBRW09Forcing*  
*attribute*), 20  
**Barrow\_top\_temperature\_data\_choice**  
*(seaice3p.params.forcing.BRW09Forcing*  
*attribute*), 34

**BaseBubbleParams** (*class* in *seaice3p.params.bubble*),  
 31  
**BasePhysicalParams** (*class* in  
*seaice3p.params.physical*), 39  
**biot** (*seaice3p.params.forcing.RobinForcing* *attribute*),  
 36  
**brine\_convection\_params**  
*(seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
*attribute*), 19  
**brine\_convection\_params**  
*(seaice3p.params.params.Config* *attribute*), 38  
**BRW09Forcing** (*class* in *seaice3p.params.forcing*), 34  
**BRW09InitialConditions** (*class* in  
*seaice3p.params.dimensionals.initial\_conditions*),  
 27  
**bubble\_distribution\_power**  
*(seaice3p.params.bubble.PowerLawBubbleParams*  
*attribute*), 32  
**bubble\_distribution\_power**  
*(seaice3p.params.dimensionals.bubble.DimensionalsPowerLawBubbleParams*  
*attribute*), 16  
**bubble\_params** (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
*attribute*), 19  
**bubble\_params** (*seaice3p.params.params.Config*  
*attribute*), 38  
**bubble\_radius** (*seaice3p.params.dimensionals.bubble.DimensionalsMonoBubbleParams*  
*attribute*), 16  
**bubble\_radius\_scaled**  
*(seaice3p.params.bubble.MonoBubbleParams*  
*attribute*), 31  
**bubble\_radius\_scaled**  
*(seaice3p.params.dimensionals.bubble.DimensionalsMonoBubbleParams*  
*property*), 16  
**bulk\_dissolved\_gas** (*seaice3p.load.DISEQResults* *at-*  
*tribute*), 46  
**bulk\_dissolved\_gas** (*seaice3p.state.disequilibrium\_state.DISEQState*  
*attribute*), 41  
**bulk\_dissolved\_gas** (*seaice3p.state.disequilibrium\_state.DISEQStateBC*  
*attribute*), 41  
**bulk\_dissolved\_gas** (*seaice3p.state.disequilibrium\_state.DISEQStateFu*  
*attribute*), 42  
**bulk\_gas** (*seaice3p.load.DISEQResults* *property*), 46

bulk\_gas (*seai3p.load.EQMResults* attribute), 47

## C

calculate\_advective\_dissolved\_gas\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_advective\_heat\_flux() (in module *seai3p.equations.flux.heat\_flux*), 7

calculate\_advective\_salt\_flux() (in module *seai3p.equations.flux.salt\_flux*), 8

calculate\_brine\_channel\_sink() (in module *seai3p.equations.RJW14.brine\_drainage*), 3

calculate\_brine\_channel\_strength() (in module *seai3p.equations.RJW14.brine\_drainage*), 4

calculate\_brine\_convection\_liquid\_velocity() (in module *seai3p.equations.RJW14.brine\_drainage*), 4

calculate\_bubble\_gas\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_bubble\_size\_fraction() (in module *seai3p.equations.velocities.bubble\_parameters*), 8

calculate\_bulk\_dissolved\_gas\_flux() (in module *seai3p.equations.flux.bulk\_dissolved\_gas\_flux*), 6

calculate\_common\_enthalpy\_method\_vars() (in module *seai3p.enthalpy\_method.common*), 1

calculate\_conductive\_heat\_flux() (in module *seai3p.equations.flux.heat\_flux*), 7

calculate\_conductivity() (in module *seai3p.equations.flux.heat\_flux*), 7

calculate\_diffusive\_gas\_bubble\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_diffusive\_gas\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_diffusive\_salt\_flux() (in module *seai3p.equations.flux.salt\_flux*), 8

calculate\_DISEQ\_dissolved\_gas() (in module *seai3p.enthalpy\_method.gas*), 2

calculate\_EQM\_dissolved\_gas() (in module *seai3p.enthalpy\_method.gas*), 2

calculate\_EQM\_gas\_fraction() (in module *seai3p.enthalpy\_method.gas*), 2

calculate\_frame\_advection\_gas\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_frame\_advection\_heat\_flux() (in module *seai3p.equations.flux.heat\_flux*), 7

calculate\_frame\_advection\_salt\_flux() (in module *seai3p.equations.flux.salt\_flux*), 8

calculate\_frame\_velocity() (in module *seai3p.equations.velocities.velocities*), 10

calculate\_gas\_flux() (in module *seai3p.equations.flux.bulk\_gas\_flux*), 7

calculate\_gas\_fraction\_flux() (in module *seai3p.equations.flux.gas\_fraction\_flux*), 7

calculate\_gas\_interstitial\_velocity() (in module *seai3p.equations.velocities.velocities*), 10

calculate\_heat\_flux() (in module *seai3p.equations.flux.heat\_flux*), 7

calculate\_ice\_ocean\_boundary\_depth() (in module *seai3p.grids*), 46

calculate\_integrated\_mean\_permeability() (in module *seai3p.equations.RJW14.brine\_drainage*), 5

calculate\_lag\_function() (in module *seai3p.equations.velocities.mono\_distribution*), 9

calculate\_lag\_integral() (in module *seai3p.equations.velocities.power\_law\_distribution*), 9

calculate\_lag\_integrand() (in module *seai3p.equations.velocities.power\_law\_distribution*), 9

calculate\_latent\_heat\_flux() (in module *seai3p.forcing.surface\_energy\_balance.turbulent\_heat\_flux*), 13

calculate\_liquid\_darcy\_velocity() (in module *seai3p.equations.velocities.velocities*), 11

calculate\_mono\_lag\_factor() (in module *seai3p.equations.velocities.mono\_distribution*), 9

calculate\_mono\_wall\_drag\_factor() (in module *seai3p.equations.velocities.mono\_distribution*), 9

calculate\_permeability() (in module *seai3p.equations.RJW14.brine\_drainage*), 5

calculate\_power\_law\_lag\_factor() (in module *seai3p.equations.velocities.power\_law\_distribution*), 9

calculate\_power\_law\_wall\_drag\_factor() (in module *seai3p.equations.velocities.power\_law\_distribution*), 10

calculate\_Rayleigh() (in module *seai3p.equations.RJW14.brine\_drainage*), 3

calculate\_salt\_flux() (in module *seai3p.equations.flux.salt\_flux*), 8

calculate\_sensible\_heat\_flux() (in module *seai3p.forcing.surface\_energy\_balance.turbulent\_heat\_flux*), 13

calculate\_velocities() (in module *seai3p.equations.velocities.velocities*), 11

calculate\_volume\_integrand() (in module *seai3p.equations.velocities.power\_law\_distribution*), 10

calculate\_wall\_drag\_function() (in module *seai3p.equations.velocities.mono\_distribution*),

9  
 calculate\_wall\_drag\_integral() (in module (seaice3p.params.convert.Scales method),  
 seaice3p.equations.velocities.power\_law\_distribution), 33  
 10  
 calculate\_wall\_drag\_integrand() (in module (seaice3p.params.convert.Scales method),  
 seaice3p.equations.velocities.power\_law\_distribution), 33  
 10  
 centers (seaice3p.grids.Grids property), 45  
 concentration\_ratio (seaice3p.params.dimensionals.water.DimensionalWaterParams  
 property), 29  
 concentration\_ratio (seaice3p.params.physical.BasePhysicalParams  
 attribute), 39  
 conductivity\_ratio (seaice3p.params.dimensionals.water.DimensionalWaterParams  
 property), 29  
 conductivity\_ratio (seaice3p.params.physical.BasePhysicalParams  
 attribute), 39  
 Config (class in seaice3p.params.params), 38  
 constant\_top\_temperature (seaice3p.params.dimensionals.forcing.DimensionalConstantForcing  
 attribute), 21  
 constant\_top\_temperature (seaice3p.params.forcing.ConstantForcing  
 attribute), 34  
 ConstantForcing (class in seaice3p.params.forcing), 34  
 convection\_strength (seaice3p.params.convection.RJW14Params  
 attribute), 32  
 convection\_strength (seaice3p.params.dimensionals.convection.DimensionalRJW14Params  
 attribute), 17  
 convert\_dimensional\_bulk\_air\_to\_argon\_content() (in module  
 (seaice3p.params.convert.Scales method), 33  
 convert\_from\_dimensional\_bulk\_gas() (seaice3p.params.convert.Scales method),  
 33  
 convert\_from\_dimensional\_bulk\_salinity() (seaice3p.params.convert.Scales method), 33  
 convert\_from\_dimensional\_dissolved\_gas() (seaice3p.params.convert.Scales method), 33  
 convert\_from\_dimensional\_grid() (seaice3p.params.convert.Scales method),  
 33  
 convert\_from\_dimensional\_heat\_flux() (seaice3p.params.convert.Scales method),  
 33  
 convert\_from\_dimensional\_heating() (seaice3p.params.convert.Scales method),  
 33  
 convert\_from\_dimensional\_temperature() (seaice3p.params.convert.Scales method),  
 33  
 convert\_from\_dimensional\_time() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_bulk\_gas() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_bulk\_salinity() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_dissolved\_gas() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_grid() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_temperature() (seaice3p.params.convert.Scales method),  
 33  
 convert\_to\_dimensional\_time() (seaice3p.params.convert.Scales method),  
 33  
 couple\_bubble\_to\_horizontal\_flow (seaice3p.params.convection.RJW14Params  
 attribute), 32  
 couple\_bubble\_to\_horizontal\_flow (seaice3p.params.dimensionals.convection.DimensionalRJW14Params  
 attribute), 17  
 couple\_bubble\_to\_vertical\_flow (seaice3p.params.convection.RJW14Params  
 attribute), 32  
 couple\_bubble\_to\_vertical\_flow (seaice3p.params.dimensionals.convection.DimensionalRJW14Params  
 attribute), 17  
 create\_and\_save\_config() (in module  
 seaice3p.example), 45  
 CubicLiquidus (class in  
 seaice3p.params.dimensionals.water), 28  
**D**  
 D\_e (seaice3p.grids.Grids property), 45  
 D\_g (seaice3p.grids.Grids property), 45  
 damkohler\_number (seaice3p.params.dimensionals.dimensional.Dimensionals  
 property), 19  
 damkohler\_number (seaice3p.params.physical.DISEQPhysicalParams  
 attribute), 40  
 data\_path (seaice3p.params.dimensionals.forcing.DimensionalERA5Forcing  
 attribute), 23  
 data\_path (seaice3p.params.dimensionals.initial\_conditions.PreviousSimulation  
 attribute), 27  
 data\_path (seaice3p.params.forcing.ERA5Forcing attribute), 35  
 DimensionalBackgroundOilHeating (class in  
 seaice3p.params.dimensionals.forcing), 20

DimensionalBaseBubbleParams	(class in seaice3p.params.dimensional.bubble), 15	in seaice3p.state.disequilibrium_state), 41
DimensionalBRW09Forcing	(class in seaice3p.params.dimensional.forcing), 20	DISEQStateFull (class in seaice3p.state.disequilibrium_state), 42
DimensionalConstantForcing	(class in seaice3p.params.dimensional.forcing), 21	dissolved_gas (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 41
DimensionalConstantLWForcing	(class in seaice3p.params.dimensional.forcing), 21	dissolved_gas (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 42
DimensionalConstantSWForcing	(class in seaice3p.params.dimensional.forcing), 21	dissolved_gas (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 43
DimensionalConstantTurbulentFlux	(class in seaice3p.params.dimensional.forcing), 21	dissolved_gas (seaice3p.state.equilibrium_state.EQMStateFull attribute), 44
DimensionalDISEQGasParams	(class in seaice3p.params.dimensional.gas), 26	<b>E</b>
DimensionalEQMGasParams	(class in seaice3p.params.dimensional.gas), 26	eddy_diffusivity (seaice3p.params.dimensional.water.DimensionalWater attribute), 29
DimensionalERA5Forcing	(class in seaice3p.params.dimensional.forcing), 22	eddy_diffusivity_ratio (seaice3p.params.dimensional.water.DimensionalWaterParams property), 29
DimensionalLWForcing	(in module seaice3p.params.dimensional.forcing), 24	eddy_diffusivity_ratio (seaice3p.params.physical.BasePhysicalParams attribute), 39
DimensionalMobileOilHeating	(class in seaice3p.params.dimensional.forcing), 24	edges (seaice3p.grids.Grids property), 45
DimensionalMonoBubbleParams	(class in seaice3p.params.dimensional.bubble), 15	enthalpy (seaice3p.state.disequilibrium_state.DISEQState attribute), 41
DimensionalNoHeating	(class in seaice3p.params.dimensional.forcing), 24	enthalpy (seaice3p.state.disequilibrium_state.DISEQStateBCs attribute), 41
DimensionalOilInitialConditions	(class in seaice3p.params.dimensional.initial_conditions), 27	enthalpy (seaice3p.state.disequilibrium_state.DISEQStateFull attribute), 42
DimensionalParams	(class in seaice3p.params.dimensional.dimensionals), 17	enthalpy (seaice3p.state.equilibrium_state.EQMState attribute), 43
DimensionalPowerLawBubbleParams	(class in seaice3p.params.dimensional.bubble), 16	enthalpy (seaice3p.state.equilibrium_state.EQMStateBCs attribute), 43
DimensionalRadForcing	(class in seaice3p.params.dimensional.forcing), 24	enthalpy (seaice3p.state.equilibrium_state.EQMStateFull attribute), 44
DimensionalRJW14Params	(class in seaice3p.params.dimensional.convection), 17	EQMPhysicalParams (class in seaice3p.params.physical), 40
DimensionalRobinForcing	(class in seaice3p.params.dimensional.forcing), 26	EQMResults (class in seaice3p.load), 47
DimensionalSWForcing	(in module seaice3p.params.dimensional.forcing), 26	EQMState (class in seaice3p.state.equilibrium_state), 43
DimensionalTurbulentFlux	(in module seaice3p.params.dimensional.forcing), 26	EQMStateBCs (class in seaice3p.state.equilibrium_state), 43
DimensionalWaterParams	(class in seaice3p.params.dimensional.water), 28	EQMStateFull (class in seaice3p.state.equilibrium_state), 44
DimensionalYearlyForcing	(class in seaice3p.params.dimensional.forcing), 26	ERA5Forcing (class in seaice3p.params.forcing), 34
DISEQPhysicalParams	(class in seaice3p.params.physical), 39	escape_ice_surface (seaice3p.params.bubble.BaseBubbleParams attribute), 31
DISEQResults	(class in seaice3p.load), 46	escape_ice_surface (seaice3p.params.dimensional.bubble.Dimensional attribute), 15
DISEQState	(class in seaice3p.state.disequilibrium_state), 40	eutectic_salinity (seaice3p.params.dimensional.water.DimensionalWater property), 29
DISEQStateBCs	(class in seaice3p.state.disequilibrium_state), 40	eutectic_salinity (seaice3p.params.dimensional.water.LinearLiquidus attribute), 31
		eutectic_temperature (seaice3p.params.dimensional.water.CubicLiquidus attribute), 28



eutectic\_temperature (seaice3p.params.dimensional.water.DimensionaWaterParameter), 29  
 eutectic\_temperature (seaice3p.params.dimensional.water.LinearLiquidus attribute), 31  
 expansion\_coefficient (seaice3p.params.dimensional.dimensiona.DimensionaParameter), 19  
 expansion\_coefficient (seaice3p.params.physical.BasePhysicalParams attribute), 39  
**F**  
 fast\_solve (seaice3p.params.dimensiona.forcing.DimensionaBackgroundHeating attribute), 20  
 fast\_solve (seaice3p.params.dimensiona.forcing.DimensionaMobileOilHeating attribute), 24  
 find\_ghost\_cell\_temperature() (in module seaice3p.forcing.surface\_energy\_balance.surface\_energy\_balance), 12  
 forcing\_config (seaice3p.params.dimensiona.dimensiona.DimensionaParameter), 19  
 forcing\_config (seaice3p.params.params.Config attribute), 38  
 frame\_velocity (seaice3p.params.dimensiona.dimensiona.DimensionaParameter), 19  
 frame\_velocity (seaice3p.params.physical.BasePhysicalParams attribute), 39  
 frame\_velocity\_dimensional (seaice3p.params.dimensiona.dimensiona.DimensionaParameter), 19  
**G**  
 gas (seaice3p.state.disequilibrium\_state.DISEQState property), 41  
 gas (seaice3p.state.disequilibrium\_state.DISEQStateFull property), 42  
 gas (seaice3p.state.equilibrium\_state.EQMState attribute), 43  
 gas (seaice3p.state.equilibrium\_state.EQMStateBCs attribute), 43  
 gas (seaice3p.state.equilibrium\_state.EQMStateFull attribute), 44  
 gas\_bubble\_eddy\_diffusion (seaice3p.params.physical.BasePhysicalParams attribute), 39  
 gas\_density (seaice3p.params.convert.Scales attribute), 33  
 gas\_fraction (seaice3p.load.DISEQResults attribute), 46  
 gas\_fraction (seaice3p.load.EQMResults property), 47  
 gas\_fraction (seaice3p.state.disequilibrium\_state.DISEQState attribute), 41  
 gas\_fraction (seaice3p.state.disequilibrium\_state.DISEQStateBCs attribute), 41  
 gas\_fraction (seaice3p.state.disequilibrium\_state.DISEQStateFull attribute), 42  
 gas\_fraction (seaice3p.state.equilibrium\_state.EQMStateBCs attribute), 43  
 gas\_fraction (seaice3p.state.equilibrium\_state.EQMStateFull attribute), 44  
 gas\_params (seaice3p.params.dimensiona.dimensiona.DimensionaParameter), 19  
 gas\_viscosity\_ratio (seaice3p.params.physical.BasePhysicalParams attribute), 39  
 generate\_oil\_simulation\_config() (in module seaice3p.oil\_simulation), 47  
 geometric() (in module seaice3p.grids), 46  
 get\_bottom\_temperature\_forcing() (in module seaice3p.forcing.temperature\_forcing), 15  
 get\_boundary\_conditions() (in module seaice3p.forcing.boundary\_conditions), 14  
 get\_brine\_convection\_sink() (in module seaice3p.equations.RJW14.brine\_channel\_sink\_terms), 3  
 get\_config() (in module seaice3p.params.params), 38  
 get\_convecting\_region\_height() (in module seaice3p.equations.RJW14.brine\_drainage), 5  
 get\_difference\_matrix() (in module seaice3p.grids), 46  
 get\_dimensionless\_brine\_convection\_params() (in module seaice3p.params.convection), 32  
 get\_dimensionless\_bubble\_params() (in module seaice3p.params.bubble), 32  
 get\_dimensionless\_forcing\_config() (in module seaice3p.params.forcing), 37  
 get\_dimensionless\_initial\_conditions\_config() (in module seaice3p.params.initial\_conditions), 37  
 get\_dimensionless\_physical\_params() (in module seaice3p.params.physical), 40  
 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.initial\_conditions), 46  
 get\_liquidus\_salinity (seaice3p.params.physical.BasePhysicalParams

attribute), 39  
 get\_liquidus\_salinity() (seaice3p.params.dimensionsal.water.CubicLiquidus method), 28  
 get\_liquidus\_temperature (seaice3p.params.physical.BasePhysicalParams attribute), 39  
 get\_liquidus\_temperature() (seaice3p.params.dimensionsal.water.CubicLiquidus method), 28  
 get\_LW\_forcing() (in module seaice3p.forcing.radiative\_forcing), 14  
 get\_nucleation() (in module seaice3p.equations.nucleation), 12  
 get\_phase\_masks() (in module seaice3p.enthalpy\_method.phase\_boundaries), 2  
 get\_printer() (in module seaice3p.printing), 48  
 get\_radiative\_heating() (in module seaice3p.equations.radiative\_heating), 12  
 get\_SW\_forcing() (in module seaice3p.forcing.radiative\_forcing), 14  
 get\_SW\_penetration\_fraction() (in module seaice3p.forcing.radiative\_forcing), 14  
 get\_temperature\_forcing() (in module seaice3p.forcing.temperature\_forcing), 15  
 get\_unpacker() (in module seaice3p.state), 44  
 ghosts (seaice3p.grids.Grids property), 45  
 gravity (seaice3p.params.dimensionsal.dimensionsal.DimensionalParams attribute), 19  
 Grids (class in seaice3p.grids), 45  
**H**  
 haline\_contraction\_coefficient (seaice3p.params.convert.Scales attribute), 33  
 haline\_contraction\_coefficient (seaice3p.params.dimensionsal.water.DimensionalWaterParams attribute), 29  
 heat\_transfer\_coefficient (seaice3p.params.dimensionsal.forcing.DimensionalRobinForcing attribute), 26  
**I**  
 I (seaice3p.params.dimensionsal.numerical.NumericalParams attribute), 28  
 ice\_density (seaice3p.params.convert.Scales attribute), 34  
 ice\_density (seaice3p.params.dimensionsal.water.DimensionalWaterParams attribute), 29  
 ice\_emissivity (seaice3p.params.dimensionsal.forcing.DimensionalRobinForcing attribute), 21  
 ice\_type (seaice3p.params.dimensionsal.forcing.DimensionalBackgroundOilHeating attribute), 21  
 ice\_type (seaice3p.params.dimensionsal.forcing.DimensionalMobileOilHeating attribute), 24  
 initial\_conditions\_config (seaice3p.params.dimensionsal.dimensionsal.DimensionalParams attribute), 19  
 initial\_conditions\_config (seaice3p.params.params.Config attribute), 38  
 initial\_ice\_bulk\_salinity (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_ice\_bulk\_salinity (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
 initial\_ice\_depth (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_ice\_depth (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
 initial\_ice\_temperature (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_ice\_temperature (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
 initial\_ocean\_temperature (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_ocean\_temperature (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
 initial\_oil\_free\_depth (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_oil\_free\_depth (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
 initial\_oil\_volume\_fraction (seaice3p.params.dimensionsal.initial\_conditions.DimensionalOilInitialConditions attribute), 27  
 initial\_oil\_volume\_fraction (seaice3p.params.initial\_conditions.OilInitialConditions attribute), 37  
**L**  
 latent\_heat (seaice3p.params.dimensionsal.water.DimensionalWaterParams attribute), 29  
 lengthscale (seaice3p.params.convert.Scales attribute), 34  
 lengthscale (seaice3p.params.dimensionsal.dimensionsal.DimensionalParams attribute), 19  
 lewis\_gas (seaice3p.params.dimensionsal.dimensionsal.DimensionalParams attribute), 19  
 lewis\_gas (seaice3p.params.physical.BasePhysicalParams attribute), 39  
 lewis\_gas (seaice3p.params.physical.BasePhysicalParams attribute), 39

lewis\_salt (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *property*), 30  
 lewis\_salt (*seaice3p.params.physical.BasePhysicalParams*  
     *attribute*), 39  
 LinearLiquidus (class in *seaice3p.params.dimension.water*), 31  
 liquid\_density (*seaice3p.params.convert.Scales* *at-*  
     *tribute*), 34  
 liquid\_density (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *attribute*), 30  
 liquid\_fraction (*seaice3p.state.disequilibrium\_state.DISEQStateBCs*  
     *attribute*), 41  
 liquid\_fraction (*seaice3p.state.disequilibrium\_state.DISEQStateFull*  
     *attribute*), 42  
 liquid\_fraction (*seaice3p.state.equilibrium\_state.EQMStateBCs*  
     *attribute*), 43  
 liquid\_fraction (*seaice3p.state.equilibrium\_state.EQMStateFull*  
     *attribute*), 44  
 liquid\_salinity (*seaice3p.state.disequilibrium\_state.DISEQStateBCs*  
     *attribute*), 41  
 liquid\_salinity (*seaice3p.state.disequilibrium\_state.DISEQStateFull*  
     *attribute*), 42  
 liquid\_salinity (*seaice3p.state.equilibrium\_state.EQMStateBCs*  
     *attribute*), 43  
 liquid\_salinity (*seaice3p.state.equilibrium\_state.EQMStateFull*  
     *attribute*), 44  
 liquid\_specific\_heat\_capacity  
     (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *attribute*), 30  
 liquid\_thermal\_conductivity  
     (*seaice3p.params.convert.Scales* *attribute*),  
     34  
 liquid\_thermal\_conductivity  
     (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *attribute*), 30  
 liquid\_viscosity (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *attribute*), 30  
 liquidus (*seaice3p.params.dimension.water.DimensionWaterParams*  
     *attribute*), 30  
 load() (*seaice3p.params.dimension.dimension.DimensionParams*  
     class method), 19  
 load() (*seaice3p.params.params.Config* class method),  
     38  
 load\_simulation() (in module *seaice3p.load*), 47  
 LW\_forcing (*seaice3p.params.dimension.forcing.DimensionERA5Forcing*  
     *attribute*), 23  
 LW\_forcing (*seaice3p.params.dimension.forcing.DimensionRadForcing*  
     *attribute*), 25  
 LW\_forcing (*seaice3p.params.forcing.ERA5Forcing* *at-*  
     *tribute*), 35  
 LW\_forcing (*seaice3p.params.forcing.RadForcing* *at-*  
     *tribute*), 36  
 LW\_irradiance (*seaice3p.params.dimension.forcing.DimensionConstantLWForcing*  
     *attribute*), 21

seaice3p.equations.velocities, 11  
 seaice3p.equations.velocities.bubble\_parameters, 8  
 seaice3p.equations.velocities.mono\_distribution, 9  
 seaice3p.equations.velocities.power\_law\_distribution, 9  
 seaice3p.equations.velocities.velocities, 10  
 seaice3p.example, 45  
 seaice3p.forcing, 15  
 seaice3p.forcing.boundary\_conditions, 14  
 seaice3p.forcing.radiative\_forcing, 14  
 seaice3p.forcing.surface\_energy\_balance, 14  
 seaice3p.forcing.surface\_energy\_balance.surface\_energy\_balance, 12  
 seaice3p.forcing.surface\_energy\_balance.thermal\_heat\_flux, 13  
 seaice3p.forcing.temperature\_forcing, 15  
 seaice3p.grids, 45  
 seaice3p.initial\_conditions, 46  
 seaice3p.load, 46  
 seaice3p.oil\_simulation, 47  
 seaice3p.params, 40  
 seaice3p.params.bubble, 31  
 seaice3p.params.convection, 32  
 seaice3p.params.convert, 33  
 seaice3p.params.dimension, 31  
 seaice3p.params.dimension.bubble, 15  
 seaice3p.params.dimension.convection, 17  
 seaice3p.params.dimension.dimension, 17  
 seaice3p.params.dimension.forcing, 20  
 seaice3p.params.dimension.gas, 26  
 seaice3p.params.dimension.initial\_conditions, 27  
 seaice3p.params.dimension.numerical, 28  
 seaice3p.params.dimension.water, 28  
 seaice3p.params.forcing, 34  
 seaice3p.params.initial\_conditions, 37  
 seaice3p.params.params, 38  
 seaice3p.params.physical, 39  
 seaice3p.printing, 48  
 seaice3p.run\_simulation, 48  
 seaice3p.state, 44  
 seaice3p.state.disequilibrium\_state, 40  
 seaice3p.state.equilibrium\_state, 43  
 MonoBubbleParams (class in seaice3p.params.bubble), 31  
 name (seaice3p.params.dimension.dimension.DimensionParams attribute), 19  
 name (seaice3p.params.params.Config attribute), 38  
 NEGLIGIBLE\_SNOW\_DEPTH  
 (seaice3p.params.forcing.ERA5Forcing attribute), 35  
 NonBrineConvection (class in seaice3p.params.dimension.convection), 17  
 nucleation\_timescale (seaice3p.params.dimension.gas.DimensionDISEQGasParams attribute), 26  
 num\_wavelength\_samples (seaice3p.params.dimension.forcing.DimensionConstantSWF attribute), 21  
 number\_of\_cells (seaice3p.grids.Grids attribute), 45  
 surface\_energy\_balance (seaice3p.params.dimension.dimension.DimensionParams attribute), 20  
 thermal\_heat\_flux (seaice3p.params.params.Config attribute), 38  
 NumericalParams (class in seaice3p.params.dimension.numerical), 28  
**O**  
 ocean\_forcing\_config (seaice3p.params.dimension.dimension.DimensionParams attribute), 20  
 ocean\_forcing\_config (seaice3p.params.params.Config attribute), 38  
 ocean\_freezing\_temperature (seaice3p.params.convert.Scales attribute), 34  
 ocean\_freezing\_temperature (seaice3p.params.dimension.water.DimensionWaterParams property), 30  
 ocean\_salinity (seaice3p.params.convert.Scales attribute), 34  
 ocean\_salinity (seaice3p.params.dimension.water.DimensionWaterParams attribute), 30  
 offset (seaice3p.params.dimension.forcing.DimensionYearlyForcing attribute), 26  
 offset (seaice3p.params.forcing.YearlyForcing attribute), 37  
 oil\_heating (seaice3p.params.dimension.forcing.DimensionERA5Forcing attribute), 23  
 oil\_heating (seaice3p.params.dimension.forcing.DimensionRadForcing attribute), 25  
 oil\_heating (seaice3p.params.forcing.ERA5Forcing attribute), 35  
 oil\_heating (seaice3p.params.forcing.RadForcing attribute), 36  
 oil\_mass\_ratio (seaice3p.params.dimension.forcing.DimensionBack attribute), 21



OilInitialConditions (class in `seaice3p.params.initial_conditions`), 37

**P**

period (`seaice3p.params.dimensionsal.forcing.DimensionallyForcing` attribute), 26

period (`seaice3p.params.forcing.YearlyForcing` attribute), 37

physical\_params (`seaice3p.params.params.Config` attribute), 38

pore\_radius (`seaice3p.params.convert.Scales` attribute), 34

pore\_radius (`seaice3p.params.dimensionsal.bubble.DimensionallyForcing` attribute), 15

pore\_throat\_scaling (`seaice3p.params.bubble.BaseBubbleParams` attribute), 31

pore\_throat\_scaling (`seaice3p.params.dimensionsal.bubble.DimensionallyForcing` attribute), 15

porosity\_threshold (`seaice3p.params.bubble.BaseBubbleParams` property), 30

porosity\_threshold (`seaice3p.params.dimensionsal.bubble.DimensionallyForcing` attribute), 15

porosity\_threshold\_value (`seaice3p.params.bubble.BaseBubbleParams` attribute), 31

porosity\_threshold\_value (`seaice3p.params.dimensionsal.bubble.DimensionallyForcing` attribute), 15

PowerLawBubbleParams (class in `seaice3p.params.bubble`), 31

PreviousSimulation (class in `seaice3p.params.dimensionsal.initial_conditions`), 27

pure\_liquid\_switch() (in module `seaice3p.equations.flux.heat_flux`), 7

**R**

RadForcing (class in `seaice3p.params.forcing`), 35

Rayleigh\_critical (`seaice3p.params.convection.RJW14Params` attribute), 32

Rayleigh\_critical (`seaice3p.params.dimensionsal.convection.DimensionallyForcing` attribute), 17

Rayleigh\_salt (`seaice3p.params.convection.RJW14Params` attribute), 32

Rayleigh\_salt (`seaice3p.params.dimensionsal.dimensionsal.DimensionallyForcing` property), 19

ref\_height (`seaice3p.params.dimensionsal.forcing.DimensionallyForcing` attribute), 22

reference\_permeability (`seaice3p.params.dimensionsal.convection.DimensionallyForcing` attribute), 17

regularisation (`seaice3p.params.dimensionsal.numerical.NumericalParams` attribute), 28

restoring\_temperature (`seaice3p.params.dimensionsal.forcing.DimensionallyForcing` attribute), 26

restoring\_temperature (`seaice3p.params.forcing.RobinForcing` attribute), 36

RJW14Params (class in `seaice3p.params.convection`), 32

RobinForcing (class in `seaice3p.params.forcing`), 36

run\_batch() (in module `seaice3p.run_simulation`), 48

run\_two\_stream\_model() (in module `seaice3p.equations.radiative_heating`), 12

**S**

salinity\_difference (`seaice3p.params.convert.Scales` attribute), 34

salinity\_difference (`seaice3p.params.dimensionsal.water.DimensionallyForcing` attribute), 30

salt (`seaice3p.state.disequilibrium_state.DISEQState` attribute), 41

salt (`seaice3p.state.disequilibrium_state.DISEQStateBCs` attribute), 41

salt (`seaice3p.state.disequilibrium_state.DISEQStateFull` attribute), 42

salt (`seaice3p.state.equilibrium_state.EQMState` attribute), 43

salt (`seaice3p.state.equilibrium_state.EQMStateBCs` attribute), 43

salt (`seaice3p.state.equilibrium_state.EQMStateFull` attribute), 44

salt\_diffusivity (`seaice3p.params.dimensionsal.water.DimensionallyForcing` attribute), 30

saturation\_concentration (`seaice3p.params.convert.Scales` attribute), 34

save() (`seaice3p.params.dimensionsal.dimensionsal.DimensionallyForcing` method), 20

save() (`seaice3p.params.params.Config` method), 38

savefreq (`seaice3p.params.dimensionsal.dimensionsal.DimensionallyForcing` attribute), 20

savefreq (`seaice3p.params.params.Config` attribute), 38

savefreq\_in\_days (`seaice3p.params.dimensionsal.dimensionsal.DimensionallyForcing` attribute), 20

scales (class in `seaice3p.params.convert`), 33

scales (`seaice3p.params.dimensionsal.dimensionsal.DimensionallyForcing` attribute), 20

scales (`seaice3p.params.params.Config` attribute), 38

seaice3p module, 48

seaice3p module, 1

seaice3p.diagnostics.brine_drainage_parameterisation	seaice3p.forcing.radiative_forcing
module, 1	module, 14
seaice3p.enthalpy_method	seaice3p.forcing.surface_energy_balance
module, 2	module, 14
seaice3p.enthalpy_method.common	seaice3p.forcing.surface_energy_balance.surface_energy_balance
module, 1	module, 12
seaice3p.enthalpy_method.enthalpy_method	seaice3p.forcing.surface_energy_balance.turbulent_heat_flux
module, 2	module, 13
seaice3p.enthalpy_method.gas	seaice3p.forcing.temperature_forcing
module, 2	module, 15
seaice3p.enthalpy_method.phase_boundaries	seaice3p.grids
module, 2	module, 45
seaice3p.equations	seaice3p.initial_conditions
module, 12	module, 46
seaice3p.equations.equations	seaice3p.load
module, 11	module, 46
seaice3p.equations.flux	seaice3p.oil_simulation
module, 8	module, 47
seaice3p.equations.flux.bulk_dissolved_gas_flux	seaice3p.params
module, 6	module, 40
seaice3p.equations.flux.bulk_gas_flux	seaice3p.params.bubble
module, 7	module, 31
seaice3p.equations.flux.gas_fraction_flux	seaice3p.params.convection
module, 7	module, 32
seaice3p.equations.flux.heat_flux	seaice3p.params.convert
module, 7	module, 33
seaice3p.equations.flux.salt_flux	seaice3p.params.dimensionality
module, 8	module, 31
seaice3p.equations.nucleation	seaice3p.params.dimensionality.bubble
module, 12	module, 15
seaice3p.equations.radiative_heating	seaice3p.params.dimensionality.convection
module, 12	module, 17
seaice3p.equations.RJW14	seaice3p.params.dimensionality.dimensionality
module, 6	module, 17
seaice3p.equations.RJW14.brine_channel_sink_term	seaice3p.params.dimensionality.forcing
module, 3	module, 20
seaice3p.equations.RJW14.brine_drainage	seaice3p.params.dimensionality.gas
module, 3	module, 26
seaice3p.equations.velocities	seaice3p.params.dimensionality.initial_conditions
module, 11	module, 27
seaice3p.equations.velocities.bubble_parameters	seaice3p.params.dimensionality.numerical
module, 8	module, 28
seaice3p.equations.velocities.mono_distribution	seaice3p.params.dimensionality.water
module, 9	module, 28
seaice3p.equations.velocities.power_law_distribution	seaice3p.params.forcing
module, 9	module, 34
seaice3p.equations.velocities.velocities	seaice3p.params.initial_conditions
module, 10	module, 37
seaice3p.example	seaice3p.params.params
module, 45	module, 38
seaice3p.forcing	seaice3p.params.physical
module, 15	module, 39
seaice3p.forcing.boundary_conditions	seaice3p.printing
module, 14	module, 48

seaice3p.run\_simulation  
     module, 48  
 seaice3p.state  
     module, 44  
 seaice3p.state.disequilibrium\_state  
     module, 40  
 seaice3p.state.equilibrium\_state  
     module, 43  
 snow\_conductivity\_ratio  
     (seaice3p.params.dimensionsal.water.Dimensiona  
     property), 30  
 snow\_conductivity\_ratio  
     (seaice3p.params.physical.BasePhysicalParams  
     attribute), 39  
 snow\_thermal\_conductivity  
     (seaice3p.params.dimensionsal.water.Dimensiona  
     attribute), 30  
 solid\_fraction(seaice3p.state.disequilibrium\_state.DISEQStateFull  
     attribute), 42  
 solid\_fraction(seaice3p.state.equilibrium\_state.EQMStateFull  
     attribute), 44  
 solid\_specific\_heat\_capacity  
     (seaice3p.params.dimensionsal.water.Dimensiona  
     attribute), 30  
 solid\_thermal\_conductivity  
     (seaice3p.params.dimensionsal.water.Dimensiona  
     attribute), 30  
 solve() (in module seaice3p.run\_simulation), 48  
 solver\_choice(seaice3p.params.dimensionsal.numerical.NumericalParams  
     attribute), 28  
 specific\_heat\_ratio  
     (seaice3p.params.dimensionsal.water.Dimensiona  
     property), 30  
 specific\_heat\_ratio  
     (seaice3p.params.physical.BasePhysicalParams  
     attribute), 39  
 specific\_humidity(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 22  
 start\_date(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 24  
 start\_date(seaice3p.params.forcing.ERA5Forcing at-  
     tribute), 35  
 stefan\_number(seaice3p.params.dimensionsal.water.Dimensiona  
     property), 30  
 stefan\_number(seaice3p.params.physical.BasePhysicalParams  
     attribute), 39  
 step(seaice3p.grids.Grids property), 45  
 step(seaice3p.params.dimensionsal.numerical.NumericalParams  
     property), 28  
 SW\_forcing(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 23  
 SW\_forcing(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 25  
 SW\_forcing(seaice3p.params.forcing.ERA5Forcing at-  
     tribute), 35  
 SW\_forcing(seaice3p.params.forcing.RadForcing at-  
     tribute), 36  
 SW\_irradiance(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 21  
 SW\_max\_wavelength(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 21  
 SW\_min\_wavelength(seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 21  
 SW\_super\_saturation\_fraction  
     (seaice3p.params.dimensionsal.forcing.Dimensiona  
     attribute), 21  
 T  
     temperature(seaice3p.state.disequilibrium\_state.DISEQStateBCs  
     attribute), 42  
     temperature(seaice3p.state.disequilibrium\_state.DISEQStateFull  
     attribute), 42  
     temperature(seaice3p.state.equilibrium\_state.EQMStateBCs  
     attribute), 43  
     temperature(seaice3p.state.equilibrium\_state.EQMStateFull  
     attribute), 44  
     temperature\_difference  
         (seaice3p.params.convert.Scales attribute),  
         34  
     temperature\_difference  
         (seaice3p.params.dimensionsal.water.Dimensiona  
         property), 30  
     thermal\_diffusivity  
         (seaice3p.params.convert.Scales attribute),  
         34  
     thermal\_diffusivity  
         (seaice3p.params.dimensionsal.water.Dimensiona  
         property), 31  
     time(seaice3p.state.disequilibrium\_state.DISEQState  
     attribute), 41  
     time(seaice3p.state.disequilibrium\_state.DISEQStateBCs  
     attribute), 42  
     time(seaice3p.state.disequilibrium\_state.DISEQStateFull  
     attribute), 43  
     time(seaice3p.state.equilibrium\_state.EQMState at-  
     tribute), 43  
     time(seaice3p.state.equilibrium\_state.EQMStateBCs at-  
     tribute), 43  
     time(seaice3p.state.equilibrium\_state.EQMStateFull at-  
     tribute), 44  
     time\_scale(seaice3p.params.convert.Scales property),  
     34  
     timescale\_in\_days(seaice3p.params.forcing.ERA5Forcing  
     attribute), 35  
     tolerable\_super\_saturation\_fraction  
         (seaice3p.params.physical.BasePhysicalParams  
         attribute), 39

`total_time` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
    *property*), 20

`total_time` (*seaice3p.params.params.Config* *attribute*),  
    38

`total_time_in_days` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
    *attribute*), 20

`turbulent_flux` (*seaice3p.params.dimensionals.forcing.DimensionalsERA5Forcing*  
    *attribute*), 24

`turbulent_flux` (*seaice3p.params.dimensionals.forcing.DimensionalsRadForcing*  
    *attribute*), 25

`turbulent_flux` (*seaice3p.params.forcing.ERA5Forcing*  
    *attribute*), 35

`turbulent_flux` (*seaice3p.params.forcing.RadForcing*  
    *attribute*), 36

## U

`UniformInitialConditions` (*class* *in*  
    *seaice3p.params.dimensionals.initial\_conditions*),  
    28

`upwind()` (*in module seaice3p.grids*), 46

`use_snow_data` (*seaice3p.params.dimensionals.forcing.DimensionalsERA5Forcing*  
    *attribute*), 24

`use_snow_data` (*seaice3p.params.forcing.ERA5Forcing*  
    *attribute*), 35

## V

`velocity_scale` (*seaice3p.params.convert.Scales* *prop-*  
    *erty*), 34

## W

`water_emissivity` (*seaice3p.params.dimensionals.forcing.DimensionalsConstantLWForcing*  
    *attribute*), 21

`water_params` (*seaice3p.params.dimensionals.dimensionals.DimensionalsParams*  
    *attribute*), 20

`wavelength_cutoff` (*seaice3p.params.dimensionals.forcing.DimensionalsBackgroundOilHeating*  
    *attribute*), 21

`wavelength_cutoff` (*seaice3p.params.dimensionals.forcing.DimensionalsMobileOilHeating*  
    *attribute*), 24

`windspeed` (*seaice3p.params.dimensionals.forcing.DimensionalsConstantTurbulentFlux*  
    *attribute*), 22

## Y

`YearlyForcing` (*class in seaice3p.params.forcing*), 36