# celestine

***Release 0.10.0***

**Joseph Fishlock**

**Nov 24, 2023**

# CONTENTS:

CHAPTER

# ONE

# BOUNDARY CONDITIONS

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.

`celestine.boundary_conditions.`**`dissolved_gas_BCs`**(*dissolved_gas_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`enthalpy_BCs`**(*enthalpy_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`gas_BCs`**(*gas_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`gas_fraction_BCs`**(*gas_fraction_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`liquid_fraction_BCs`**(*liquid_fraction_centers*, *cfg:* Config)

Add ghost cells to liquid fraction such that top and bottom boundaries take the same value as the top and bottom cell center

`celestine.boundary_conditions.`**`liquid_salinity_BCs`**(*liquid_salinity_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`pressure_BCs`**(*pressure_centers*, *cfg:* Config)

Add ghost cells to pressure so that W_l=0 at z=0 and p=0 at z=-1

`celestine.boundary_conditions.`**`salt_BCs`**(*salt_centers*, *cfg:* Config)

Add ghost cells with BCs to center quantity

`celestine.boundary_conditions.`**`temperature_BCs`**(*temperature_centers*, *time*, *cfg:* Config)

Add ghost cells with BCs to center quantity

Note this needs the current time as well as top temperature is forced.

CHAPTER

TWO

# DIMENSIONAL PARAMS

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** celestine.dimensional_params.**DimensionalParams**(*name: str*, *total_time_in_days: float = 365*, *savefreq_in_days: float = 1*, *data_path: str = 'data/'*, *lengthscale: float = 1*, *liquid_density: float = 1028*, *gas_density: float = 1*, *saturation_concentration: float = 1e-05*, *ocean_salinity: float = 34*, *eutectic_salinity: float = 270*, *eutectic_temperature: float = -21.1*, *latent_heat: float = 334000.0*, *specific_heat_capacity: float = 4184*, *thermal_conductivity: float = 0.598*, *salt_diffusivity: float = 0*, *gas_diffusivity: float = 0*, *frame_velocity_dimensional: float = 0*, *gravity: float = 9.81*, *liquid_viscosity: float = 0.00278*, *bubble_radius: float = 0.001*, *pore_radius: float = 0.001*, *pore_throat_scaling: float = 0.5*, *drag_exponent: float = 6.0*, *liquid_velocity_dimensional: float = 0.0*, *bubble_size_distribution_type: str = 'mono'*, *wall_drag_law_choice: str = 'power'*, *bubble_distribution_power: float = 1.5*, *minimum_bubble_radius: float = 1e-06*, *maximum_bubble_radius: float = 0.001*, *porosity_threshold: bool = False*, *porosity_threshold_value: float = 0.024*)

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 bubble_radius_scaled**

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B / R_0$$

**property concentration_ratio**

Calculate concentration ratio as

$$\mathcal{C} = S_i / \Delta S$$

**property expansion_coefficient**

calculate

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

**property frame_velocity**

calculate the frame velocity in non dimensional units

**get_config**(*boundary_conditions_config:* BoundaryConditionsConfig = *BoundaryConditionsConfig(initial_conditions_choice='uniform', far_gas_sat=1.0, far_temp=0.1, far_bulk_salinity=0)*, *forcing_config:* ForcingConfig = *ForcingConfig(temperature_forcing_choice='constant', constant_top_temperature=- 1.5, offset=- 1.0, amplitude=0.75, period=4.0)*, *numerical_params:* NumericalParams = *NumericalParams(I=50, timestep=0.0002, regularisation=1e-06, solver='LU')*)

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.

**get_darcy_law_params**()

return a DarcyLawParams object

**get_physical_params**()

return a PhysicalParams object

**get_scales**()

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

**property lewis_gas**

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

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

**property lewis_salt**

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

$$\text{Le}_S = \kappa / D_s$$

**property liquid_velocity**

convert given liquid velocity into non dimensional units

**classmethod load**(*path*)

load this object from a yaml configuration file.

**property maximum_bubble_radius_scaled**

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B/R_0$$

**property minimum_bubble_radius_scaled**

calculate the bubble radius divided by the pore scale

$$\Lambda = R_B/R_0$$

**property ocean_freezing_temperature**

calculate salinity dependent freezing temperature using liquidus for typical ocean salinity

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

**property salinity_difference**

calculate difference between eutectic salinity and typical ocean salinity

$$\Delta S = S_E - S_i$$

**save**()

save this object to a yaml file in the specified data path.

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

**property stefan_number**

calculate Stefan number

$$\mathrm{St} = L/c_p\Delta T$$

**property temperature_difference**

calculate

$$\Delta T = T_i - T_E$$

**property thermal_diffusivity**

Return thermal diffusivity in m2/s

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

**property total_time**

calculate the total time in non dimensional units for the simulation

**class** celestine.dimensional_params.**Scales**(*lengthscale: float*, *thermal_diffusivity: float*, *ocean_salinity: float*, *salinity_difference: float*, *ocean_freezing_temperature: float*, *temperature_difference: float*, *gas_density: float*, *saturation_concentration: float*)

**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_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

celestine.dimensional_params.**calculate_timescale_in_days**(*lengthscale*, *thermal_diffusivity*)
calculate timescale given domain height and thermal diffusivity.

**Parameters**

- **lengthscale** (*float*) – domain height in m
- **thermal_diffusivity** (*float*) – thermal diffusivity in m2/s

**Returns**
timescale in days

celestine.dimensional_params.**calculate_velocity_scale_in_m_day**(*lengthscale*, *thermal_diffusivity*)
calculate the velocity scale given domain height and thermal diffusivity

**Parameters**

- **lengthscale** (*float*) – domain height in m
- **thermal_diffusivity** (*float*) – thermal diffusivity in m2/s

**Returns**
velocity scale in m/day

CHAPTER

THREE

# ENTHALPY METHOD

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

**class** celestine.enthalpy_method.**EnthalpyMethod**(*physical_params:* PhysicalParams)

Template for an enthalpy method. To implement a new method overwrite the initializer to initialise the physical parameters and a suitable phase boundaries object. Then implement a calculate enthalpy method that takes a state and uses bulk enthalpy, salt and gas to return (temperature, liquid_fraction, gas_fraction, solid_fraction, liquid_salinity, dissolved_gas).

**class** celestine.enthalpy_method.**FullEnthalpyMethod**(*physical_params:* PhysicalParams)

**class** celestine.enthalpy_method.**ReducedEnthalpyMethod**(*physical_params:* PhysicalParams)

celestine.enthalpy_method.**get_enthalpy_method**(*cfg*)

Return the enthalpy method object required depending on solver choice

LU: Full RED: Reduced SCI: Reduced

**Parameters**

**cfg** – configuration for simulation

CHAPTER

# FOUR

# FLUX

Module for calculating the fluxes using upwind scheme

celestine.flux.**calculate_conductive_heat_flux**(*temperature*, *D_g*)

Calculate conductive heat flux as

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

**Parameters**

- **temperature** (*Numpy Array of size I+2*) – temperature including ghost cells
- **D_g** (*Numpy Array*) – difference matrix for ghost grid

**Returns**

conductive heat flux

celestine.flux.**calculate_diffusive_salt_flux**(*liquid_salinity*, *liquid_fraction*, *D_g*, *cfg*)

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

celestine.flux.**take_forward_euler_step**(*quantity*, *flux*, *timestep*, *D_e*)

Advance the given quantity one forward Euler step using the given flux

The quantity is given on cell centers and the flux on cell edges.

Discretise the conservation equation

$$\frac{\partial Q}{\partial t} = -\frac{\partial F}{\partial z}$$

as

$$Q^{n+1} = Q^n - \Delta t(\frac{\partial F}{\partial z})$$

CHAPTER

FIVE

# FORCING

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.

celestine.forcing.**barrow_temperature_forcing**(*time*, *cfg*)

Take non dimensional time and return non dimensional air temperature at the Barrow site in 2009.

For this to work you must have created the configuration cfg from dimensional parameters as it must have the conversion scales object.

celestine.forcing.**dimensional_barrow_temperature_forcing**(*time_in_days*, *cfg:* Config)

Take time in days and linearly interp 2009 Barrow air temperature data to get temperature in degrees Celsius.

CHAPTER

# SIX

# GRIDS

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

celestine.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).

celestine.grids.**average**(*ghosts*)

Returns arithmetic mean pairwise of first dimension of an array

This should get values on the ghost grid and returns the arithmetic average onto the edge grid

celestine.grids.**geometric**(*ghosts*)

Returns geometric mean of the first dimension of an array

CHAPTER

SEVEN

# INITIAL CONDITIONS

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

celestine.initial_conditions.**apply_value_in_ice_layer**(*depth_of_ice*, *ice_value*, *liquid_value*, *grid*)

assume that top part of domain contains mushy ice of given depth and lower part of domain is liquid. This function returns output on the given grid where the ice part of the domain takes one value and the liquid a different.

This is useful for initialising the barrow simulation where we have an initial ice layer.

celestine.initial_conditions.**get_barrow_initial_conditions**(*cfg:* Config)

initialise domain with an initial ice layer of given temperature and bulk salinity. These values are hard coded in from Moreau paper to match barrow study. They also assume that the initial ice layer has 1/5 the saturation amount in pure liquid of dissolved gas to account for previous gas loss.

Initialise with bulk gas being (1/5) in ice and saturation in liquid. Bulk salinity is 5.92 g/kg in ice and ocean value in liquid. Enthalpy is calculated by inverting temperature relation in ice and ocean. Ice temperature is given as -8.15 degC and ocean is the far value from boundary config.

celestine.initial_conditions.**get_uniform_initial_conditions**(*cfg*)

Generate uniform initial solution on the ghost grid

**Returns**

initial solution arrays on ghost grid (enthalpy, salt, gas, pressure)

CHAPTER

EIGHT

# LOGGING CONFIG

Module to create logger for simulation

CHAPTER

# NINE

# PARAMS

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** celestine.params.**BoundaryConditionsConfig**(*initial_conditions_choice: str = 'uniform'*, *far_gas_sat: float = 1.0*, *far_temp: float = 0.1*, *far_bulk_salinity: float = 0*)

values for bottom (ocean) boundary

**class** celestine.params.**Config**(*name: str*, *physical_params:* PhysicalParams = *PhysicalParams(expansion_coefficient=0.029, concentration_ratio=0.17, stefan_number=4.2, lewis_salt=inf, lewis_gas=inf, frame_velocity=0)*, *boundary_conditions_config:* BoundaryConditionsConfig = *BoundaryConditionsConfig(initial_conditions_choice='uniform', far_gas_sat=1.0, far_temp=0.1, far_bulk_salinity=0)*, *darcy_law_params:* DarcyLawParams = *DarcyLawParams(B=100, pore_throat_scaling=0.5, liquid_velocity=0.0, bubble_size_distribution_type='mono', wall_drag_law_choice='power', drag_exponent=6.0, bubble_radius_scaled=1.0, bubble_distribution_power=1.5, minimum_bubble_radius_scaled=0.001, maximum_bubble_radius_scaled=1, porosity_threshold=False, porosity_threshold_value=0.024)*, *forcing_config:* ForcingConfig = *ForcingConfig(temperature_forcing_choice='constant', constant_top_temperature=- 1.5, offset=- 1.0, amplitude=0.75, period=4.0)*, *numerical_params:* NumericalParams = *NumericalParams(I=50, timestep=0.0002, regularisation=1e-06, solver='LU')*, *scales: Optional[int] = None*, *total_time: float = 4.0*, *savefreq: float = 0.0005*, *data_path: str = 'data/'*)

contains all information needed to run a simulation and save output

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

**check_thermal_Courant_number**()

Check if courant number for thermal diffusion term is low enough for explicit method and if it isn't log a warning.

**class** celestine.params.**DarcyLawParams**(*B: float = 100*, *pore_throat_scaling: float = 0.5*, *liquid_velocity: float = 0.0*, *bubble_size_distribution_type: str = 'mono'*, *wall_drag_law_choice: str = 'power'*, *drag_exponent: float = 6.0*, *bubble_radius_scaled: float = 1.0*, *bubble_distribution_power: float = 1.5*, *minimum_bubble_radius_scaled: float = 0.001*, *maximum_bubble_radius_scaled: float = 1*, *porosity_threshold: bool = False*, *porosity_threshold_value: float = 0.024*)

non dimensional parameters for calculating liquid and gas darcy velocities

class celestine.params.**ForcingConfig**(*temperature_forcing_choice: str = 'constant'*, *constant_top_temperature: float = - 1.5*, *offset: float = - 1.0*, *amplitude: float = 0.75*, *period: float = 4.0*)

choice of top boundary (atmospheric) forcing and required parameters

**load_forcing_data**()

populate class attributes with barrow dimensional air temperature and time in days (with missing values filtered out).

Note the metadata explaining how to use the barrow temperature data is also in celestine/forcing_data. The indices corresponding to days and air temp are hard coded in as class variables.

class celestine.params.**NumericalParams**(*I: int = 50*, *timestep: float = 0.0002*, *regularisation: float = 1e-06*, *solver: str = 'LU'*)

parameters needed for discretisation and choice of numerical method

**property Courant**

This number must be <0.5 for stability of temperature diffusion terms

class celestine.params.**PhysicalParams**(*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*)

non dimensional numbers for the mushy layer

celestine.params.**filter_missing_values**(*air_temp*, *days*)

Filter out missing values are recorded as 9999

CHAPTER

TEN

# PHASE BOUNDARIES

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

**class** celestine.phase_boundaries.**FullPhaseBoundaries**(*physical_params:* PhysicalParams)

calculates the phase boundaries when we include gas fraction in bulk enthalpy and bulk salinity.

**class** celestine.phase_boundaries.**PhaseBoundaries**(*physical_params:* PhysicalParams)

Template for phase boundary calculation.

Concrete implementations should use the state containing enthalpy, salt and gas to calculate the liquidus, enthalpy, solidus and saturation boundaries and then return masks for each possible phase of the system.

**class** celestine.phase_boundaries.**ReducedPhaseBoundaries**(*physical_params:* PhysicalParams)

calculates the phase boundaries neglecting the gas fraction so that

$$\phi_s + \phi_l = 1$$

CHAPTER

# ELEVEN

# RUN SIMULATION

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

celestine.run_simulation.**run_batch**(*list_of_cfg*)

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[celestine.params.Config]*) – list of configurations

celestine.run_simulation.**solve**(*cfg:* Config)

Solve simulation choosing appropriate solver from the choice in the config.

CHAPTER

# TWELVE

# STATE

Classes to store solution variables

State: store variables on cell centers StateBCs: add boundary conditions in ghost cells to cell center variables Solution: store primary variables at each timestep we want to save data

**class** celestine.state.**Solution**(*cfg:* Config)

store solution at specified times on the center grid

**add_state**(*state:* State, *index: int*)

add state to stored solution at given time index

**class** celestine.state.**State**(*cfg:* Config, *time*, *enthalpy*, *salt*, *gas*, *pressure=None*)

Stores information needed for solution at one timestep on cell centers

**class** celestine.state.**StateBCs**(*state:* State)

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

Note must initialise once enthalpy method has already run on State.

CHAPTER

# THIRTEEN

# VELOCITIES

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.

`celestine.velocities.`**`calculate_bubble_size_fraction`**(*bubble_radius_scaled*, *liquid_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.

`celestine.velocities.`**`calculate_gas_interstitial_velocity`**(*liquid_fraction*, *liquid_interstitial_velocity*, *wall_drag_factor*, *lag_factor*, *cfg:* Config)

Calculate Vg from liquid fraction on the ghost frid and liquid interstitial velocity

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

Return Vg on edge grid

`celestine.velocities.`**`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.

`celestine.velocities.`**`calculate_lag_integrand`**(*bubble_size_fraction: float*, *cfg:* Config)

Scalar function to calculate lag integrand for polydispersive case.

Bubble size fraction is given as a scalar input to calculate

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

celestine.velocities.**calculate_liquid_darcy_velocity**(*liquid_fraction*, *cfg:* Config)

Calculate liquid Darcy velocity as

$$W_l = \frac{\phi_l U_0}{2}$$

This assumes that we are given the non dimensional maximum interstitial liquid velocity.

**Parameters**

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

**Returns**

liquid darcy velocity on edge grid

celestine.velocities.**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

celestine.velocities.**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

celestine.velocities.**calculate_power_law_lag_factor**(*liquid_fraction*, *cfg:* Config)

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

Return on edge grid

celestine.velocities.**calculate_power_law_wall_drag_factor**(*liquid_fraction*, *cfg:* Config)

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

Return on edge grid

celestine.velocities.**calculate_velocities**(*state_BCs*, *cfg:* Config)

Inputs on ghost grid, outputs on edge grid

celestine.velocities.**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.

celestine.velocities.**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.

celestine.velocities.**calculate_wall_drag_integrand**(*bubble_size_fraction: float*, *cfg:* Config)

Scalar function to calculate wall drag integrand for polydispersive case.

Bubble size fraction is given as a scalar input to calculate

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

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

CHAPTER

# FOURTEEN

# TEMPLATE

Template for a solver concrete solvers should inherit and overwrite required methods

**class** celestine.solvers.template.**SolverTemplate**(*cfg:* Config)

**generate_initial_solution**()

Generate initial solution on the ghost grid

**Returns**
initial solution arrays on ghost grid (enthalpy, salt, gas, pressure)

**pre_solve_checks**()

Optionally implement this method if you want to check anything before running the solver.

For example to check the timestep and grid step satisfy some constraint.

**abstract take_timestep**(*state:* State) → *State*

advance enthalpy, salt, gas and pressure to the next timestep.

Note as of 2023-05-17 removed ability to have adaptive timestepping for simplicity.

**Parameters**
**state** (celestine.solvers.template.State) – object containing current time, enthalpy, salt, gas, pressure and surface temperature.

**Returns**
state of system (new enthalpy, salt, gas and pressure) after one timestep.

CHAPTER

# FIFTEEN

# REDUCED MODEL SOLVER

**class** celestine.solvers.reduced_solver.**ReducedSolver**(*cfg:* Config)

Take timestep using forward Euler upwind scheme using reduced model.

**pre_solve_checks**()

Optionally implement this method if you want to check anything before running the solver.

For example to check the timestep and grid step satisfy some constraint.

**take_timestep**(*state:* State)

advance enthalpy, salt, gas and pressure to the next timestep.

Note as of 2023-05-17 removed ability to have adaptive timestepping for simplicity.

**Parameters**

**state** (celestine.solvers.template.State) – object containing current time, enthalpy, salt, gas, pressure and surface temperature.

**Returns**

state of system (new enthalpy, salt, gas and pressure) after one timestep.

celestine.solvers.reduced_solver.**prevent_gas_rise_into_saturated_cell**(*Vg*, *state_BCs:* StateBCs)

Modify the gas interstitial velocity to prevent bubble rise into a cell which is already theoretically saturated with gas.

From the state with boundary conditions calculate the gas and solid fraction in the cells (except at lower ghost cell). If any of these are such that there is more gas fraction than pore space available then set gas insterstitial velocity to zero on the edge below. Make sure the very top boundary velocity is not changed as we want to always alow flux to the atmosphere regardless of the boundary conditions imposed.

**Parameters**

- **Vg** (*Numpy array (size I+1)*) – gas insterstitial velocity on cell edges
- **state_BCs** (celestine.state.StateBCs) – state of system with boundary conditions

**Returns**

filtered gas interstitial velocities on edges to prevent gas rise into a fully gas saturated cell

CHAPTER

# SIXTEEN

# SCIPY SOLVER

**class** celestine.solvers.scipy.**ScipySolver**(*cfg:* Config)

Solve reduced model using scipy solve_ivp using RK23 solver. This is the "SCI" solver option.

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.

The interface of this class is a little different as we overwrite the solve method from the template and must provide a function to calculate the ode forcing for solve_ivp. However the solve function still saves the data in the same format using the *celestine.state.Solution* class.

**take_timestep**(*state:* State)

advance enthalpy, salt, gas and pressure to the next timestep.

Note as of 2023-05-17 removed ability to have adaptive timestepping for simplicity.

**Parameters**

**state** (celestine.solvers.template.State) – object containing current time, enthalpy, salt, gas, pressure and surface temperature.

**Returns**

state of system (new enthalpy, salt, gas and pressure) after one timestep.

CHAPTER

SEVENTEEN

# INDICES AND TABLES

- genindex
- modindex
- search

# PYTHON MODULE INDEX

## C

# INDEX

## D



## E



## F



## G



## L