celestine

Release 0.7.0

Joseph Fishlock

May 19, 2023

CONTENTS:

CHAPTER

ONE

BOUNDARY CONDITIONS

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.**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

ENTHALPY METHOD

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)

CHAPTER

THREE

FLUX

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

FOUR

FORCING

CHAPTER

FIVE

GRIDS

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

SIX

LOGGING CONFIG

CHAPTER

SEVEN

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**(*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(far_gas_sat=1.0, far_temp=0.1, far_bulk_salinity=0)*, *darcy_law_params:* DarcyLawParams = *DarcyLawParams(B=100, bubble_radius_scaled=1.0, pore_throat_scaling=0.5, drag_exponent=6.0, liquid_velocity=0.0)*, *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')*, *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.

class celestine.params.**DarcyLawParams**(*B: float = 100*, *bubble_radius_scaled: float = 1.0*, *pore_throat_scaling: float = 0.5*, *drag_exponent: float = 6.0*, *liquid_velocity: float = 0.0*)

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

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

CHAPTER

EIGHT

PHASE BOUNDARIES

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

NINE

RUN SIMULATION

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

CHAPTER

TEN

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

ELEVEN

VELOCITIES

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

Takes liquid fraction on edges and returns bubble radius parameter on edges

celestine.velocities.**calculate_gas_interstitial_velocity**(*liquid_fraction*, *pressure*, *D_g*, *cfg:* Config)

Calculate Vg from liquid fraction and pressure on ghost grid

Return Vg on edge grid

celestine.velocities.**calculate_liquid_darcy_velocity**(*liquid_fraction*, *pressure*, *D_g*)

Calculate liquid Darcy velocity as

$$W_l = -\Pi(\phi_l)\frac{\partial p}{\partial z}$$

Parameters

- **liquid_fraction** (*Numpy Array (size I+2)*) – liquid fraction on ghost grid
- **pressure** (*Numpy Array (size I+2)*) – pressure on ghost grid
- **D_g** (*Numpy Array (size I+2)*) – difference matrix for ghost grid

Returns

liquid darcy velocity on edge grid

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

Inputs on ghost grid, outputs on edge grid

celestine.velocities.**solve_pressure_equation**(*state_BCs*, *new_state_BCs*, *timestep*, *D_e*, *D_g*, *cfg:* Config)

Calculate pressure on ghost grid from current and new state on ghost grid

Return new pressure on centers but easy to add boundary conditions

CHAPTER

TWELVE

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

THIRTEEN

LAGGED SOLVER

class celestine.solvers.lagged_solver.**LaggedUpwindSolver**(*cfg:* Config)

Take timestep using upwind scheme with liquid velocity calculation lagged.

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.

CHAPTER

FOURTEEN

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

FIFTEEN

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

SIXTEEN

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

C

INDEX

L

M

N

P

R

S

T