SpyIce

Sneha Iyer

Sep 03, 2024

CONTENTS

The SPyIce package is a software tool that enables 1D finite difference simulation for vertical transport equations. It specifically focuses on thermal diffusion with the influence of salinity and physical properties. The package utilizes the Thomas tridiagonal solver as the solver algorithm. With SPyIce, users can model and analyze the behavior of temperature, salinity, and other relevant variables in a vertical system. It provides a comprehensive framework for studying the thermal diffusion process and its interaction with salinity in various scenarios. Hydra is used to automate the simulation runs of the Sea-Ice Model. It is used to manage and run sea ice simulations, making it easier for users to explore different scenarios and optimize their models.

CHAPTER

ONE

SPYICE

The SPyIce package is a software tool that enables 1D finite difference simulation for vertical transport equations. It specifically focuses on thermal diffusion with the influence of salinity and physical properties. The package utilizes the Thomas tridiagonal solver as the solver algorithm. With SPyIce, users can model and analyze the behavior of temperature, salinity, and other relevant variables in a vertical system. It provides a comprehensive framework for studying the thermal diffusion process and its interaction with salinity in various scenarios. Hydra is used to automate the simulation runs of the Sea-Ice Model. It is used to manage and run sea ice simulations, making it easier for users to explore different scenarios and optimize their models.

1.1 Introduction

Multiphysics simulation has been growing in the past decades due to computational capabilities. It involves translating the physical partial differential equations to an efficient numerical code. Implementing good Scientific software development practices help in reducing computational power. Testing applicability of multiphysical computational models can prove to be challenging due to thermodynamic and numerical constraints. Therefore, improving the code reproducibility through automation and object oriented concepts is beneficial.

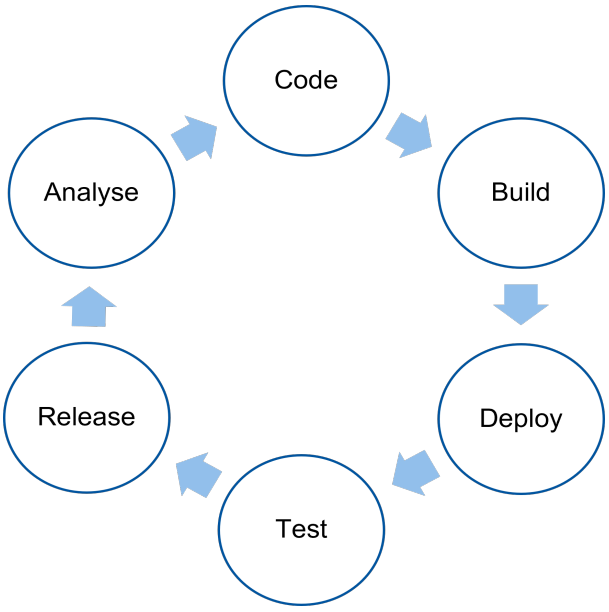


Fig. 1: Caption

This project aims to improve an existing 1D advection-diffusion sea ice simulation model using python to improve the benchmark testing for various model parameters. The automation of the model is performed using hydra python package. Hatch python project manager is used for testing, static analysis checks, and for creating reproducible build ecosystem. OOP concepts are leveraged and the mediator design pattern is implemented to improve sustainability of code.

1.2 Problem Statement

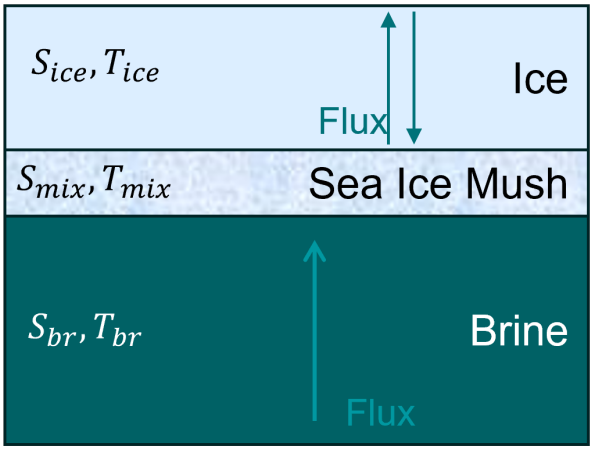


Fig. 2: caption

Modelling of Sea ice freezing involves ice-brine interface tracking which is modelling using the generalised Stefan problem. In this project, a 1D advective-diffusion model predominantly diffusive in nature is implemented using finite differences. The numerical equations models thermal and brine effects on the sea ice system with the following set of equations (**??**):

$$\begin{aligned}(\rho c)_{eff}\frac{\partial T}{\partial t} &= \frac{\partial}{\partial z}\left(k_{eff}\frac{\partial T}{\partial z}\right) - \rho_i L\frac{\partial \phi}{\partial t} \\ \phi\frac{\partial S_{br}}{\partial t} &= \frac{\partial}{\partial z}\left(D_{eff}\frac{\partial S_{br}}{\partial z}\right) - \frac{\rho_i}{\rho_{br}}P_S S_{br}\frac{\partial \phi}{\partial t}\end{aligned} \tag{1.1}$$

Brine in a sea ice system propagates along a vertical column and the flux across boundaries is considered to be constant (CHECK!!) as shown in Fig(). The Dirichlet boundary conditions (essential boundary condition) is implemented at the top layer of the vertical column and the system is assumed to be at melting temperature of sea ice as (**??**):.

$$T_m(S) = T_m - \Gamma S \tag{1.2}$$

The linear numerical model is solved using the *Thomas tri-diagonal solver* **(REF)** which is for a positive definite diagonally dominant matrix **(REF)**. The conditional stability of the implicit numerical difference system is verified using the *Fourier stability criteria* **(REF)**. The model parameters used in this project are number of iterations, sea ice freezing duration, and initial salinity.

Stefan Problem: So λ giving the minimum absolute result is determined and used to compute the location of the phase change.

$$-\frac{2sle^{-l^2}}{\sqrt{\pi}\,\mathrm{erf}(l)} - \frac{2se^{-2l^2}}{\pi\,\mathrm{erf}^2(l)} - 1 \tag{1.3}$$

1.3 Implementation

Hydra configuration tool is used to feed multiple input combinations to the Sea ice model and generate a structured output directory based on the given input combination. For example: all combinations of user defined input values for salinity, maximum iterations and time step size are sequentially run using hydra which improves the traceability of simulation along with logging files in their respective output directory. The inputs are initialised and processed by the Prepossessing class. Here the respective initial and boundary conditions are applied to the user-defined discrete finite difference mesh. The simulation model is executed for the given number of maximum iterations which allows to model for a time period of (time step size) * (max iterations). At a given time t, the numerical model is solved until it attains a convergence for field parameters temperature, salinity and volumetric liquid fraction whose pseudo code is given below in code_pseudo:

Listing 1: Python example

```
def run_model(self) -> None:
    """Runs the model using the provided configuration and output directory.
"""
    # apply boundary and initial conditions during the pre-processing stage
and get the pre-processed dataclass
    preprocess_data, userinput_data = PreProcess.get_variables(
        self.config, self.out_dir_final
    )
    # run the sea ice model and get the results dataclass
    results_data = SeaIceModelClass.get_results(preprocess_data, userinput_
data)
    # error analysis of results and get the analysis dataclass
    analysis_data = Analysis.get_error_results(
        t_k_diff=results_data.t_k_diff, t_stefan_diff=results_data.t_stefan_
diff
    )
    # plot the sea ice model using the user input, results, and analysis
dataclasses
    self.plot_model(userinput_data, results_data, analysis_data)
```

Once the field values are obtained, an error analysis is performed using Analytical class to verify discrepancies between numerical and analytical results. The analytical results are verified with the one-phase Stefan problem which keeps one of the two phases constant (liquid phase in this project) while modelling. The temperature field can be visualised using the Visualise-model class where the temperature fields can be compared at different spatial nodes points and their nodal time evolution in comparison to the analytical results.

CHAPTER

TWO

QUICKSTART

2.1 Installation Guide

2.1.1 Prerequisites

Before installing Spyice, make sure you have the following prerequisites installed:

- **Python 3.11 or above**: Spyice requires Python 3.11 or above to run. You can download the latest version of Python from the official website.
- **Hatch**: Hatch is a Python package manager that we'll use to create a virtual environment for Spyice. You can install Hatch by following the instructions on the official website.
- **Sphinx**: Sphinx is a documentation generator that we'll use to build the Spyice documentation. You can install Sphinx using the appropriate package manager for your system by following the instructions on the official website.

2.1.2 Installation Steps

Follow these steps to install Spyice and set up the environment using hatch:

1. Clone this project repository to your local machine.
2. (Optional) The required wheels can be built with the help of *hatch* without the worries of cross-compilation and native architecture support.

    ```
    ` hatch build `
    ```

This command will build the wheels for the project and store them under the name *dist/spyice-1.0.0.dev0-py3-none-any.whl*

3. Create a the new default python virtual environment

    ```
    ` hatch env create `
    ```

The project will be automatically installed in editable mode by *hatch* when the environment is created. Confirm the installation by running *pip show spyice* in the shell.

4. You're all set! You can now start using Spyice.

2.1.3 Viewing the documentation

Enter the project directory and run the following command in the terminal to view the pre-built static version of the sphinx documentation.

```
` python -m http.server --directory=docs/build/html/ `
```

This will start a local server at *http://localhost:8000/* where you can view the documentation by opening the link in your browser.

2.2 Sea-Ice Model Package (SPyIce)

The SPyIce package is a software tool that enables 1D finite difference simulation for vertical transport equations. It specifically focuses on thermal diffusion with the influence of salinity and physical properties. The package utilizes the Thomas tridiagonal solver as the solver algorithm. With SPyIce, users can model and analyze the behavior of temperature, salinity, and other relevant variables in a vertical system. It provides a comprehensive framework for studying the thermal diffusion process and its interaction with salinity in various scenarios. Hydra is used to automate the simulation runs of the Sea-Ice Model. It is used to manage and run sea ice simulations, making it easier for users to explore different scenarios and optimize their models.

2.2.1 Package Demo Example

Import Packages

[1]:
```
import os
from pathlib import Path

from omegaconf import OmegaConf

from spyice.main_process import create_output_directory
from spyice.postprocess import Analysis, VisualiseModel
from spyice.utils import ConfigSort
from spyice.models import SeaIceModel
from spyice.preprocess import PreProcess
```

Define Inputs and Project Output paths

[2]:
```
# creates a OmegaConf object from a dictionary
constants_dict = {"constants": "real", "dt": 47.0, "S_IC": "S34", "iter_max
↪": 1000}
config_raw = OmegaConf.create(constants_dict)
config = ConfigSort.getconfig_dataclass(config_raw, config_type="jupyter")

base_dir = Path.cwd()
output_base_dir = Path(base_dir, "output")
```

(continues on next page)

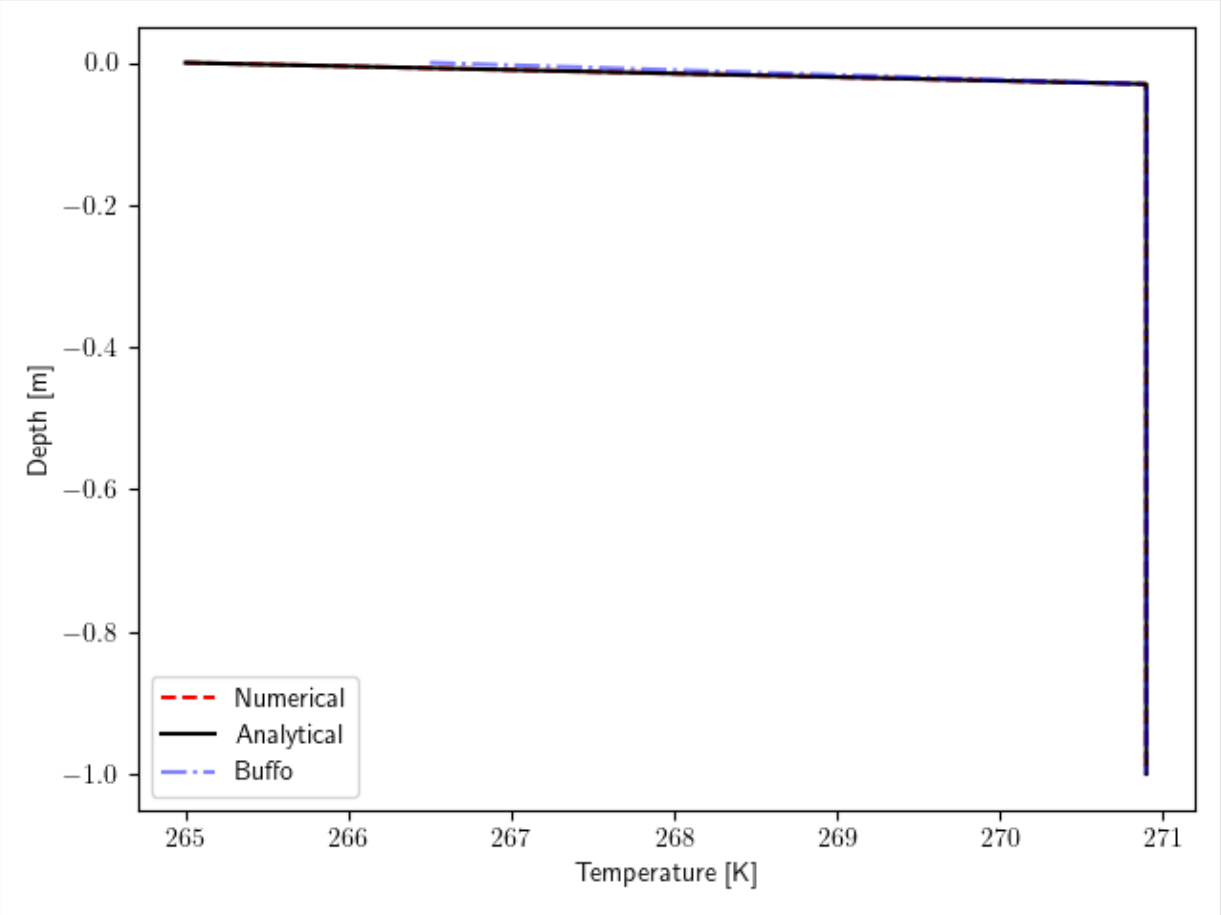
(continued from previous page)

```
wo_hydra_dir = Path(output_base_dir, "without_hydra")
out_dir_final = create_output_directory(wo_hydra_dir)
```

Preprocessing, Running and Analysis of Sea-Ice Model

[3]:
```
preprocess_data, userinput_data = PreProcess.get_variables(config, out_dir_
↪final)
results_data = SeaIceModel.get_results(preprocess_data, userinput_data)
analysis_data = Analysis.get_error_results(
    t_k_diff=results_data.t_k_diff, t_stefan_diff=results_data.t_stefan_diff
)
```

```
Preprocessing...
User Configuration Data Setup Complete...
Geometry Data Setup Complete...
Results Data Setup Complete...
Time step set to: 47.0s
Applied Initial & Boundary Conditions...
Preprocessing done.
Running model...
| (!) 999/1000 [100%] in 10.5s (95.78/s
Model run complete and Ready for Analysis.
Running error analysis...
Calculating errors...
```



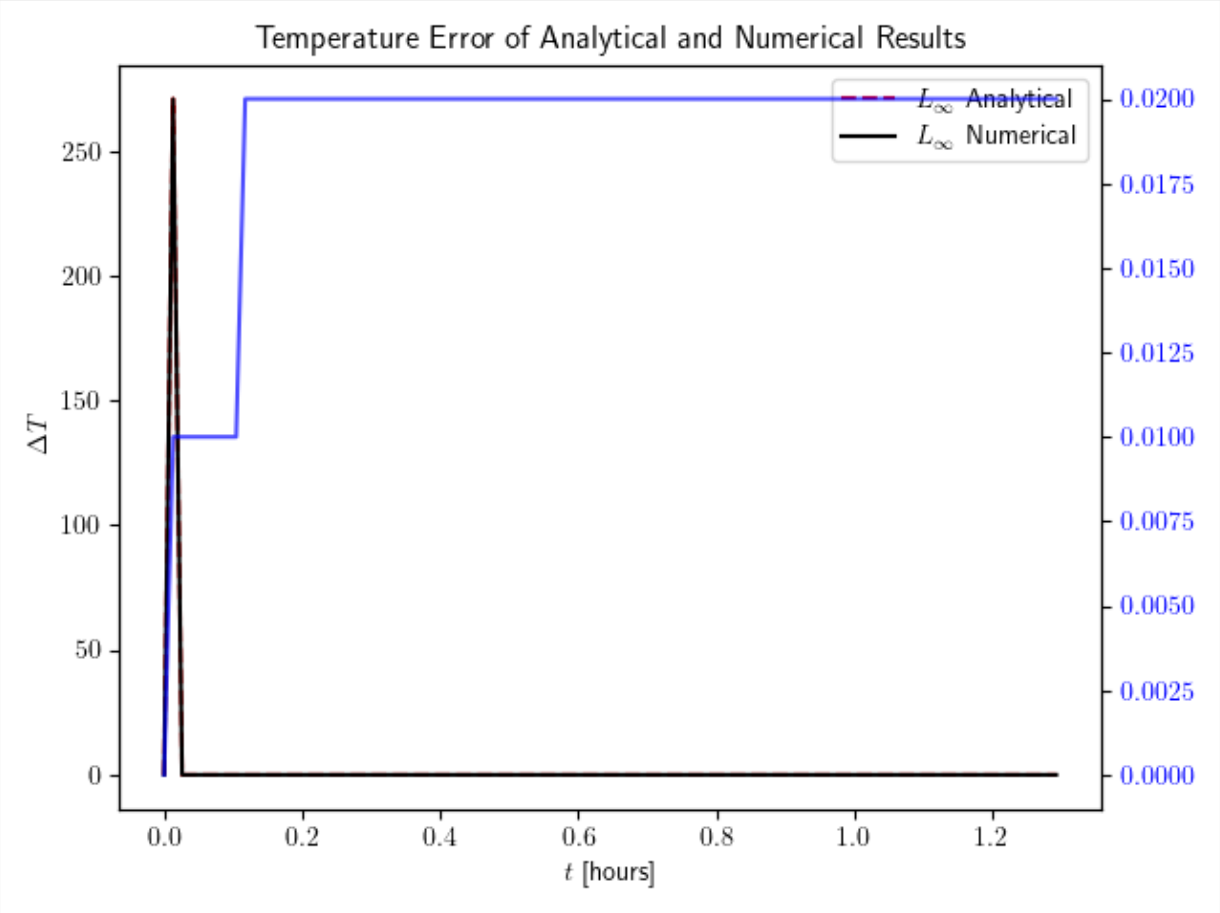
Visualization of Model with VisualiseModel

[4]:
```
model_visualization_object = VisualiseModel(
    user_input_dataclass=userinput_data,
    results_dataclass=results_data,
    error_analysis_dataclass=analysis_data,
)
```

```
Visualisation object created...
```

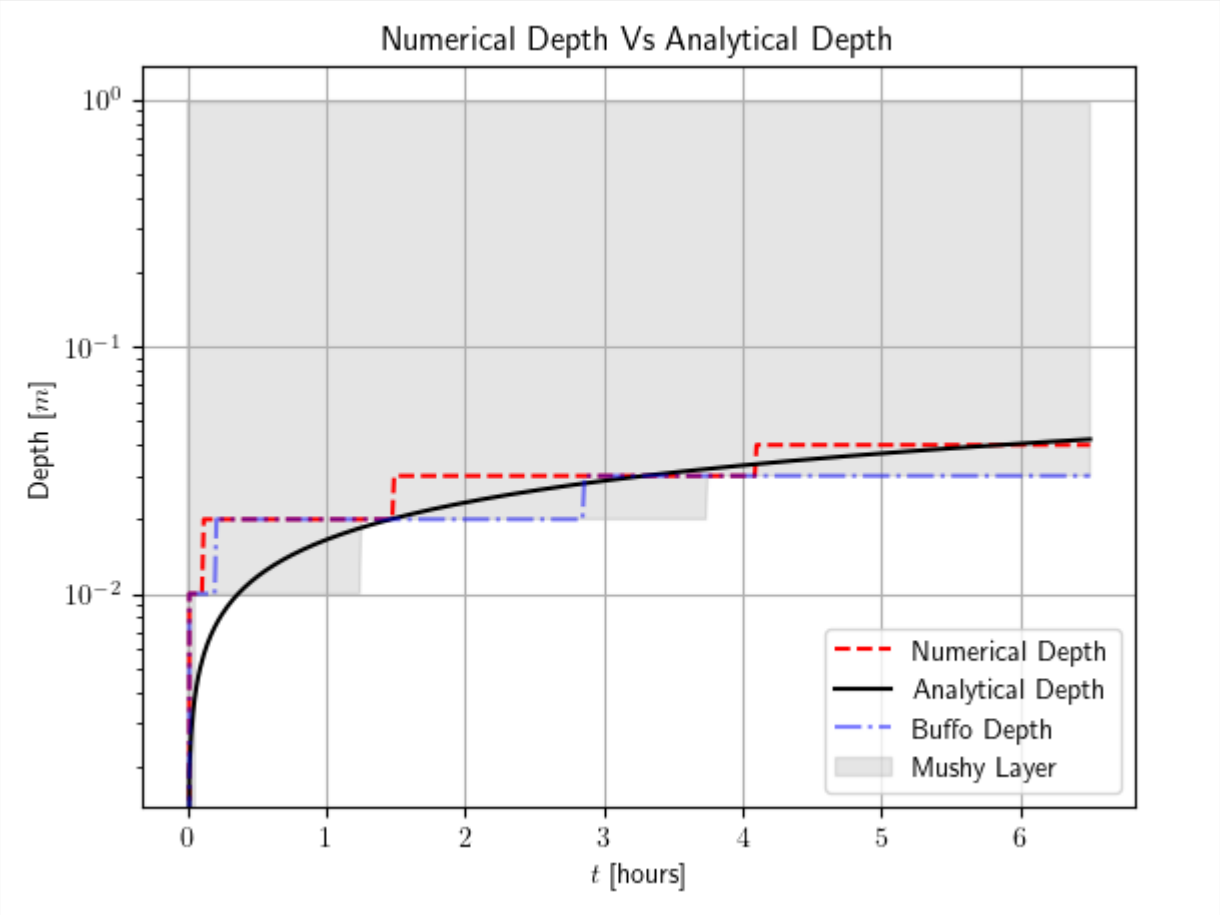
[5]:
```
# Plots the Temperature Difference between Analytical and Numerical
↪Solutions
model_visualization_object.plot_error_temp(100, norm="inf", savefig=False)
```

```
Plotting Temperature errors using inf norm...
```



[6]:
```
# Plots the interface tracking over time for Analytical and Numerical
↪Solutions
model_visualization_object.plot_depth_over_time(savefig=True)
```

```
Plotting Depth over time...
```



[7]:
```
import os

from hydra import (
    compose,
    initialize,
)
from omegaconf import OmegaConf

# import the main process class
from spyice.main_process import MainProcess
```

To run the Sea-Ice Model using Hydra and the `MainProcess` script, users simply need to initialize Hydra, load the configuration file, specify any desired overrides, and then create an instance of the `MainProcess` class. The `run_model()` method is then called to execute the simulation. This streamlined process makes it simple for users to run the model with different configurations and analyze the results.

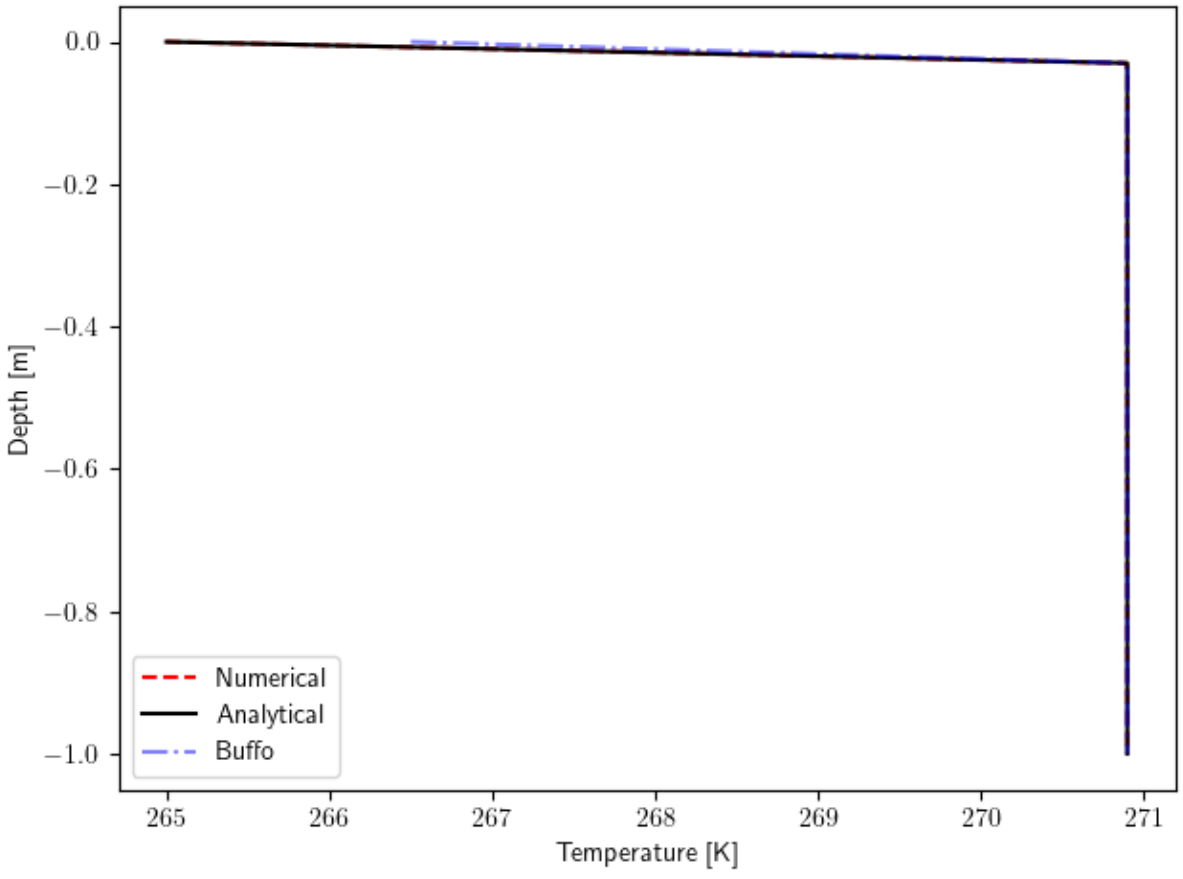
[8]:
```
with initialize(version_base=None, config_path="conf"):
    cfg = compose(
        config_name="config.yaml",
        overrides=["iter_max=iter_max1000", "dt=dt47", "S_IC=S34"],
    )
    out_hydra_dir = Path(output_base_dir, "with_hydra")
    main = MainProcess(cfg, out_hydra_dir)
```

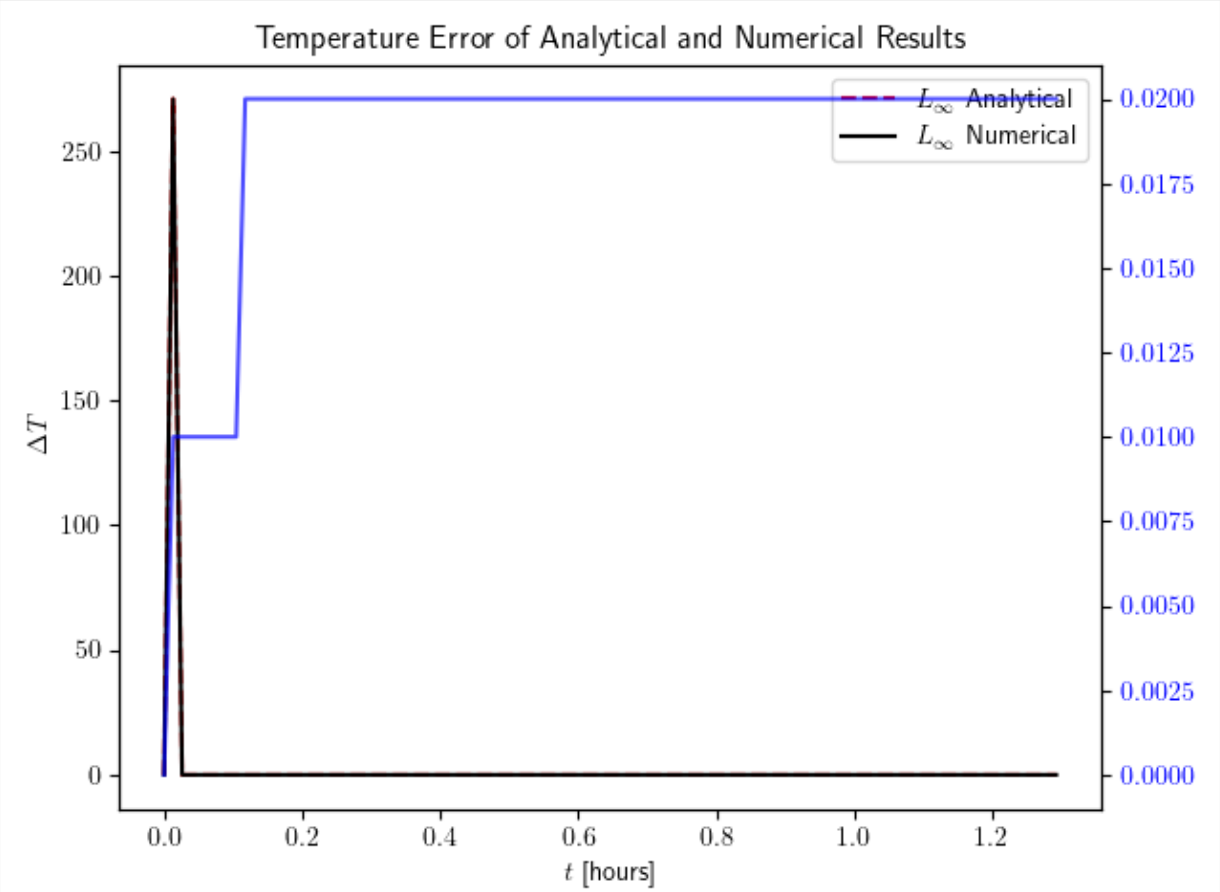
(continues on next page)

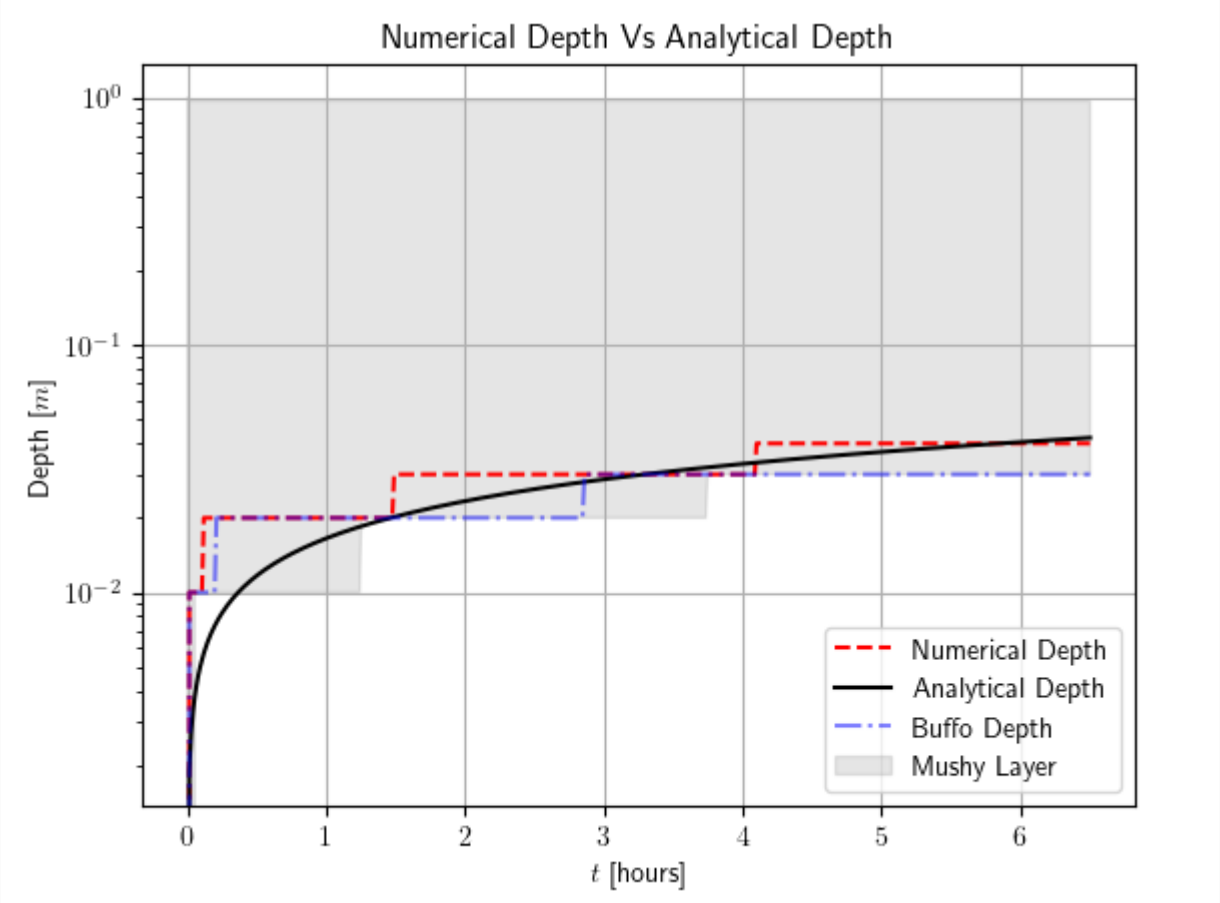
(continued from previous page)

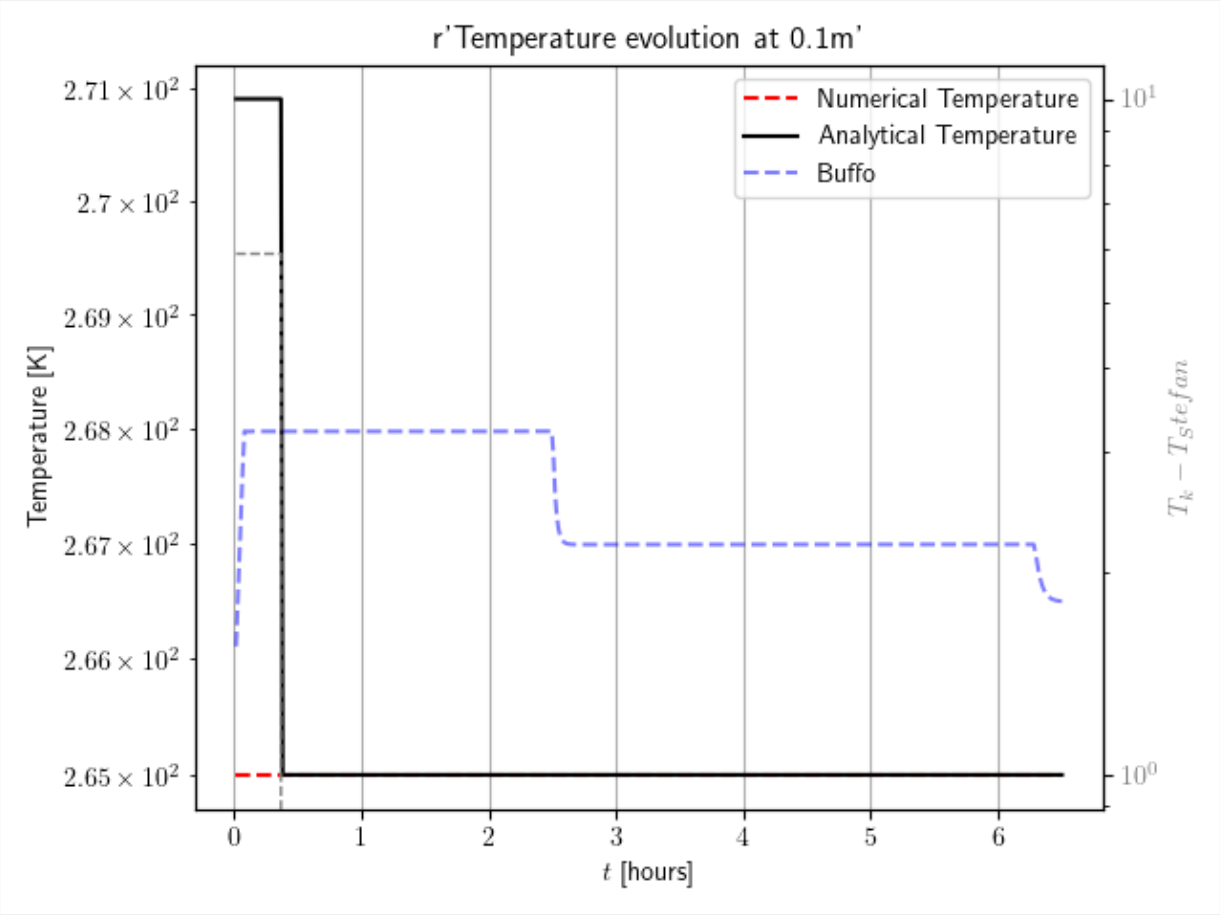
```
    main.run_model()
```

```
Preprocessing...
User Configuration Data Setup Complete...
Geometry Data Setup Complete...
Results Data Setup Complete...
Time step set to: 47.0s
Applied Initial & Boundary Conditions...
Preprocessing done.
Running model...
| (!) 999/1000 [100%] in 10.1s (98.63/s
Model run complete and Ready for Analysis.
Running error analysis...
Calculating errors...
Postprocessing...
Visualisation object created...
Plotting Temperature errors using inf norm...
Plotting Depth over time...
Plotting Temperature evolution at 0.1m...
Postprocessing done.
```




Temperature Error of Analytical and Numerical Results
L_∞ Analytical
L_∞ Numerical
ΔT
t [hours]
0.0
0.2
0.4
0.6
0.8
1.0
1.2
0
50
100
150
200
250
0.0000
0.0025
0.0050
0.0075
0.0100
0.0125
0.0150
0.0175
0.0200


Numerical Depth Vs Analytical Depth
Depth [m]
t [hours]
10^0
10^{-1}
10^{-2}
0
1
2
3
4
5
6
Numerical Depth
Analytical Depth
Buffo Depth
Mushy Layer


r'Temperature evolution at 0.1m'
Numerical Temperature
Analytical Temperature
Buffo
Temperature [K]
2.71×10^2
2.7×10^2
2.69×10^2
2.68×10^2
2.67×10^2
2.66×10^2
2.65×10^2
10^1
10^0
$T_k - T_{stefan}$
0
1
2
3
4
5
6
t [hours]

CHAPTER

THREE

SPYICE

3.1 spyice package

3.1.1 Subpackages

spyice.models package

Submodules

class spyice.models.advection_diffusion.**AdvectionDiffusion**

Bases: object

Class representing an advection-diffusion model.

Parameters

- **argument** (*str*) – The argument for the advection-diffusion equation. Must be either “temperature” or “salinity”.
- **X** (*float*) – The value of X.
- **source** (*float*) – The source value.
- **X_initial** (*float*) – The initial value of X.
- **W** (*float*) – The value of W.
- **W_initial** (*float*) – The initial value of W.
- **w** (*float*) – The value of w.
- **dt** (*float*) – The time step size.
- **dz** (*float*) – The spatial step size.
- **nz** (*int*) – The number of spatial steps.
- **t_passed** (*float*) – The time passed.
- **S_IC** (*float*) – The initial condition for S.
- **Stefan** (*bool, optional*) – Whether to use Stefan condition. Defaults to False.
- **Buffo** (*bool, optional*) – Whether to use Buffo condition. Defaults to False.

- **bc_neumann** (*float, optional*) – The Neumann boundary condition. Defaults to None.

Raises

- **AssertionError** – If the argument is not “temperature” or “salinity”.
- **AssertionError** – If both Stefan and Buffo conditions are set to True.

Buffosolver(*a, b, c, f*)

Solves a tridiagonal linear system using the Thomas algorithm. :param a: Lower diagonal of the tridiagonal matrix. :type a: np.ndarray :param b: Upper diagonal of the tridiagonal matrix. :type b: np.ndarray :param c: Main diagonal of the tridiagonal matrix. :type c: np.ndarray :param f: Right-hand side vector. :type f: np.ndarray

Returns
Solution vector of the linear system.

Return type
np.ndarray

TDMAsolver(*a, b, c, d*)

Solves a tridiagonal matrix system using the TDMA (Thomas algorithm). :param a: Coefficients of the sub-diagonal elements. :type a: array-like :param b: Coefficients of the diagonal elements. :type b: array-like :param c: Coefficients of the super-diagonal elements. :type c: array-like :param d: Right-hand side vector. :type d: array-like

Returns
Solution vector.

Return type
array-like

References

- Tridiagonal matrix algorithm: http://en.wikipedia.org/wiki/Tridiagonal_matrix_algorithm
- TDMA (Thomas algorithm): http://www.cfd-online.com/Wiki/Tridiagonal_matrix_algorithm_-_TDMA_(Thomas_algorithm)

__init__(*argument, X, source, X_initial, W, W_initial, w, dt, dz, nz, t_passed, S_IC, Stefan=False, Buffo=False, bc_neumann=None*)

Parameters

- **argument** (*str*) – The argument for the advection-diffusion equation. Must be either “temperature” or “salinity”.
- **X** (*float*) – The value of X.
- **source** (*float*) – The source value.
- **X_initial** (*float*) – The initial value of X.
- **W** (*float*) – The value of W.

- **W_initial** (*float*) – The initial value of W.
- **w** (*float*) – The value of w.
- **dt** (*float*) – The time step size.
- **dz** (*float*) – The spatial step size.
- **nz** (*int*) – The number of spatial steps.
- **t_passed** (*float*) – The time passed.
- **S_IC** (*float*) – The initial condition for S.
- **Stefan** (*bool, optional*) – Whether to use Stefan condition. Defaults to False.
- **Buffo** (*bool, optional*) – Whether to use Buffo condition. Defaults to False.
- **bc_neumann** (*float, optional*) – The Neumann boundary condition. Defaults to None.

Raises

- **AssertionError** – If the argument is not "temperature" or "salinity".
- **AssertionError** – If both Stefan and Buffo conditions are set to True.

factor_1(*argument, a, c, dt, dz, nz*)

Factor 1 and avoid zero divison error

Parameters

- **argument** (*str*) – The argument value.
- **a** (*numpy.ndarray*) – The array containing values for 'a'.
- **c** (*numpy.ndarray*) – The array containing values for 'c'.
- **dt** (*float*) – The value of 'dt'.
- **dz** (*float*) – The value of 'dz'.
- **nz** (*int*) – The value of 'nz'.

Returns

The calculated factor(s) based on the given argument.
If the argument is not "salinity1", returns a numpy.ndarray. If the argument is "salinity1", returns a list containing two numpy.ndarrays.

Return type
numpy.ndarray or list

factor_2(*a, b, dt, dz, nz*)

Calculates the factor2 array for advection-diffusion model. :param a: Array containing values for 'a'. :type a: numpy.ndarray :param b: Array containing values for 'b'. :type b: numpy.ndarray :param dt: Time step. :type dt: float :param dz: Spatial step. :type dz: float :param nz: Number of elements in the array. :type nz: int

Returns
Array containing the calculated factor2 values.

Return type
numpy.ndarray

factor_3(*a*, *d*, *nz*)

Factor 3 and avoid zero division error.

Parameters

- **a** (*numpy.ndarray*) – Array containing values.
- **d** (*numpy.ndarray*) – Array containing values.
- **nz** (*int*) – Number of elements.

Returns
Array containing factor 3 values.

Return type
numpy.ndarray

set_up_tridiagonal()

Set up the tridiagonal matrix for Salinity and Temperature. This method sets up the tridiagonal matrix for Salinity and Temperature calculations. It initializes the main diagonal (main_A), upper diagonal (upper_A), and lower diagonal (lower_A) of the matrix based on the given arguments. :param self: The object instance.

Returns
None

unknowns_matrix()

Calculates the unknowns matrix for the advection-diffusion model. :returns:

A tuple containing the following elements:

- X_new (ndarray): The solution vector for the unknowns.
- X_wind (ndarray): The solution vector corrected for brine movement.
- dt (float): The time step used in the calculation.

Return type
tuple

spyice.models.advection_diffusion.**top_temp = 'Stefan'**

a * (dU/dt) + b * (dU/dz) + d/dz(c * dU/dz) + d * (dW/dt) = 0

Type
Solves for X in Advection- Diffusion- Equation of the form

class spyice.models.sea_ice_model.**SeaIceModel**

Bases: object

SeaIceModelClass represents a class that models the behavior of sea ice.

Parameters

- **dataclass** (PreprocessData (page **??**)) – The preprocessed data for the model.
- **user_dataclass** (UserInput (page **??**)) – The user input data for the model.

__init__(*dataclass*, *user_dataclass*)

Parameters

- **dataclass** (PreprocessData (page **??**)) – The preprocessed data for the model.
- **user_dataclass** (UserInput (page **??**)) – The user input data for the model.

bc_neumann(*phi_k*, *nz*, *bc_condition=None*)

Apply Neumann boundary condition to the sea ice model.

Parameters

- **phi_k** (*float*) – The value of phi at the k-th layer.
- **nz** (*int*) – The number of layers in the sea ice model.
- **bc_condition** (*str, optional*) – The type of boundary condition to apply. Defaults to None.

Returns

None

choose_phase_type_iteration(*t*)

Choose the phase type iteration based on the one-phase and two-phase generalised Stefan Probem.

Parameters

t (*int*) – The time index.

Returns

A tuple containing the following values:

- t_stefan (float): The Stefan temperature.
- error_depth_t (float): The error in depth.
- depth_stefan_t (float): The depth at time t.

Return type

tuple

Raises

InvalidPhaseError (page **??**) – If the phase type is invalid (not 1 or 2).

convergence_loop_iteration(*t*, *t_km1*, *s_km1*, *phi_km1*, *buffo=False*, *stefan=False*, *temp_grad=None*)

Performs a single iteration of the convergence loop.

Parameters

- **t** (*float*) – Current temperature.

- **t_km1** (*float*) – Temperature at the previous time step.
- **s_km1** (*float*) – Salinity at the previous time step.
- **phi_km1** (*float*) – Porosity at the previous time step.
- **buffo** (*bool, optional*) – Flag indicating whether to use the buffo method. Defaults to False.
- **stefan** (*bool, optional*) – Flag indicating whether to use the Stefan method. Defaults to False.
- **temp_grad** (*float, optional*) – Temperature gradient. Defaults to None.

Returns

A tuple containing the following values:

- t_k (float): Current temperature.
- t_prev (float): Temperature at the previous time step.
- s_k (float): Current salinity.
- s_prev (float): Salinity at the previous time step.
- phi_k (float): Current porosity.
- phi_prev (float): Porosity at the previous time step.
- h_k (float): Current heat flux.
- h_solid (float): Heat flux at the solid-liquid interface.
- thickness (float): Current thickness.
- thickness_index (int): Index of the thickness.
- t_km1 (float): Temperature at the previous time step.
- s_km1 (float): Salinity at the previous time step.
- phi_km1 (float): Porosity at the previous time step.

Return type
tuple

classmethod get_results(*dataclass*, *user_dataclass*)

Runs the sea ice model and returns the results.

Parameters

- **cls** (*class*) – The class object.
- **dataclass** (`PreprocessData` (page **??**)) – The dataclass containing preprocessed data.
- **user_dataclass** (`UserInput` (page **??**)) – The dataclass containing user input.

Returns
The results dataclass object generated by running the sea ice model.

Return type
Results

initialize_state_variables(*t*, *t_km1*, *s_km1*, *phi_km1*)

Initializes the state variables for the sea ice model.

Parameters

- **t** (*int*) – The current time step.
- **t_km1** (*float*) – The temperature at the previous time step.
- **s_km1** (*float*) – The salinity at the previous time step.
- **phi_km1** (*float*) – The liquid fraction at the previous time step.

Returns

A tuple containing the initialized state variables:

- t_initial (float): The initial temperature.
- t_km1 (float): The temperature at the previous time step.
- s_km1 (float): The salinity at the previous time step.
- phi_initial (float): The initial liquid fraction.
- phi_km1 (float): The liquid fraction at the previous time step.
- temp_grad (float): The temperature gradient.

Return type
tuple

phi_all_mush_list(*phi_k_*, *phi_all_mush_list*)

Calculates the number of elements in **phi_k_** that fall within the mush_lowerbound and mush_upperbound range.

Parameters

- **phi_k** (*numpy.ndarray*) – The input array containing the values to be checked.
- **phi_all_mush_list** (*list*) – The list to which the count of elements within the range will be appended.

Returns
The updated phi_all_mush_list with the count of elements within the range appended.

Return type
list

record_iteration_data()

Records the iteration data for temperature and phi values. This method appends the temperature and phi values from the current iteration to the respective arrays. The arrays are used to store the iteration data for further analysis.

Parameters
None

Returns
None

reset_iteration_parameters(*t*, *tkm1*, *s_km1*, *phi_km1*)

Reset the iteration parameters for the sea ice model.

Parameters

- **t** (*float*) – Current temperature.
- **tkm1** (*float*) – Temperature at the previous time step.
- **s_km1** (*float*) – Salinity at the previous time step.
- **phi_km1** (*float*) – Liquid fraction at the previous time step.

Returns

A tuple containing the following iteration parameters:

- t_km1 (float): Temperature at the previous time step.
- s_km1 (float): Salinity at the previous time step.
- phi_km1 (float): Liquid fraction at the previous time step.
- temp_grad (float): Temperature gradient.
- t_err (float): Temperature error.
- s_err (float): Salinity error.
- phi_err (float): Liquid fraction error.
- t_initial (float): Initial temperature.
- phi_initial (float): Initial liquid fraction.
- t_source (ndarray): Array of temperature sources.
- counter (int): Iteration counter.

Return type
tuple

run_sea_ice_model()

Runs the sea ice model.

This function iterates over a specified number of time steps and performs calculations to simulate the behavior of sea ice. It updates the results and saves a temperature profile plot at the end.

Parameters
None

Returns
None

run_while_convergence_iteration(*t*, *t_km1*, *s_km1*, *phi_km1*, *buffo*, *stefan*, *t_err*, *s_err*, *phi_err*, *t_initial*, *phi_initial*, *t_source*, *counter*)

Runs the convergence loop until convergence is reached.

Parameters

- **t** – Time step
- **t_km1** – Previous temperature array
- **s_km1** – Previous salinity array
- **phi_km1** – Previous phi array
- **buffo** – Flag for Buffo
- **stefan** – Flag for Stefan
- **t_err** – Temperature error
- **s_err** – Salinity errorhatch
- **phi_err** – Phi error
- **t_initial** – Initial temperature
- **phi_initial** – Initial phi
- **t_source** – Temperature source
- **counter** – Iteration counter

Returns
Tuple of updated arrays and indices

set_boundary_condition_type(*critical_depth*, *bc_type='Neumann'*)

Sets the boundary condition type for the model. This method sets the boundary condition type for the model. It calculates the temperature gradient based on the given critical depth and boundary condition type. If the boundary condition type is "Neumann", the temperature gradient is calculated using the formula: temp_grad = dz * (temperature_melt - boundary_top_temperature) / critical_depth If the boundary condition type is not "Neumann", the temperature gradient is set to None.

Parameters

- **critical_depth** (*float*) – The critical depth value.
- **bc_type** (*str, optional*) – The type of boundary condition. Defaults to "Neumann".

Example

model.set_boundary_condition_type(10.0, "Neumann")

set_dataclass(*_dataclass*)

Sets the dataclass attributes of the object.

Parameters
_dataclass – An instance of the dataclass.

Returns
None

t_running(*fig*, *ax1*, *t_stefan*, *t_k*, *t_k_buffo=None*, *count=0*)

Plot the temperature profile against depth.

Parameters

- **fig** (*matplotlib.figure.Figure*) – The figure object to plot on.
- **ax1** (*matplotlib.axes.Axes*) – The axes object to plot on.
- **t_stefan** (*numpy.ndarray*) – The temperature profile obtained analytically.
- **t_k** (*numpy.ndarray*) – The temperature profile obtained numerically.
- **t_k_buffo** (*numpy.ndarray, optional*) – The temperature profile obtained using Buffo method. Defaults to None.
- **count** (*int, optional*) – The count value. Defaults to 0.

Returns
None

track_mush_for_parameter(*phi_k_*, *param*, *param_iterlist*)

Track the mush for a given parameter.

Parameters

- **phi_k** – numpy array representing the values of phi_k
- **param** – numpy array representing the parameter values
- **param_iterlist** – list to store the tracked mush values for the parameter
- **parameter** (*Updated list with the tracked mush values for the*)

spyice.models.sea_ice_model.**locate_ice_ocean_interface**(*phi*, *dz*, *nz*, ***kwargs*)

Locate ice ocean interface, based on liquid fraction equivalent ice thickness

Parameters

- **phi** (*array-like*) – Liquid fraction [-]
- **dz** (*float*) – Spatial discretization [m]
- **nz** (*int*) – Number of computational nodes
- ****kwargs** – Additional keyword arguments Stefan (bool): Validation with Stefan problem (default: True)

Returns

A tuple containing:

- if_depth (float): Location of the ice-water interface/sea ice total thickness [m]
- if_depth_index (int): Index of the 'transition cell' from ice to ocean (freezing) or water to ice (melting)

Return type
tuple

class spyice.models.stefan_problem.**StefanProblem**

Bases: object

A class to solve the Stefan problem.

static calculate_temperature_profile(*depth_stefan*, *t*, *dz*, *nz*, *ui*)

Calculates the temperature profile for the Stefan problem.

Parameters

- **depth_stefan** (*float*) – The depth of the Stefan problem.
- **t** (*float*) – The time.
- **dz** (*float*) – The step size in the z-direction.
- **nz** (*int*) – The number of grid points in the z-direction.
- **ui** (UserInput (page **??**)) – An instance of the UserInput class containing the required constants and parameters.

Returns

An array containing the temperature profile for the Stefan problem.

Return type

numpy.ndarray

static calculate_temperature_twophase_profiles(*depth_stefan*, *t*, *dz*, *nz*, *ui*)

Calculate the temperature and salinity profiles for the Stefan problem in a two-phase system.

Parameters

- **depth_stefan** (*float*) – The depth of the Stefan problem.
- **t** (*float*) – The time.
- **dz** (*float*) – The grid spacing.
- **nz** (*int*) – The number of grid points.
- **ui** (UserInput (page **??**)) – An instance of the UserInput class containing the input parameters.

Returns

A tuple containing the temperature and salinity profiles as numpy arrays.

Return type

tuple

static stefan_problem(*t*, *ui*)

Calculates the Stefan problem solution for a given time and user input.

Parameters

- **t** (*float*) – The time value.
- **ui** (UserInput (page **??**)) – An instance of the UserInput class containing the necessary input parameters.

Returns
The calculated Stefan problem solution.

Return type
float

Raises
None –

static stefan_problem_twophase(*t*, *ui*)

Solves the Stefan problem for a two-phase system.

Parameters

- **t** (*float*) – Time parameter.
- **ui** (UserInput (page **??**)) – User input object containing boundary conditions and constants.

Returns
The result of the Stefan problem calculation.

Return type
float

Module contents

spyice.parameters package

Submodules

class spyice.parameters.constants.**Constants**

Bases: Enum

Enumeration class for constants.

DEBUG = DebugConstants(phi_ini=1, phi_ini_Stefan=1, beta=1, kappa=1, mu=1, Ra_c=0, alpha=1, k_i=1, k_br=1, k_w=1, D_s=0, c_br=1, c_i=1, c_w=1, L=1, rho_i=1, rho_br=1, rho_w=1, m=1, g=0, phi_c=1, P_s=1, a_phi=1, b_phi=1)

REAL = RealConstants(phi_ini=1, phi_ini_Stefan=1, beta=0.0005836, kappa=1.37e-07, mu=0.00188, Ra_c=1.01, alpha=0.0015600000000000002, k_i=2.0, k_br=0.6, k_w=0.6, D_s=2e-09, c_i=2000, c_br=3985, c_w=4200, L=334774, rho_br=1028, rho_i=917, rho_w=1000, m=2, g=9.8, phi_c=0.06823, P_s=0.01, a_phi=5.9e-06, b_phi=169491.5254237288)

class spyice.parameters.debug_constants.**DebugConstants**

Bases: object

Class representing debug constants used in the model.

Parameters

- **phi_ini** (*int*) – Freezing constant.
- **phi_ini_Stefan** (*int*) – Freezing constant for Stefan condition.

- **beta** (*int*) – Coefficient for density dependence on salinity.
- **kappa** (*int*) – Thermal diffusivity.
- **mu** (*int*) – Viscosity.
- **Ra_c** (*int*) – Critical Rayleigh number.
- **alpha** (*int*) – Linear coefficient for Rayleigh number driven advection.
- **k_i** (*int*) – Thermal conductivity of ice [W/m/K].
- **k_br** (*int*) – Thermal conductivity of brine [W/m/K].
- **k_w** (*int*) – Thermal conductivity of water.
- **D_s** (*int*) – Diffusivity for salt.
- **c_br** (*int*) – Specific heat of seawater (J/kg/K).
- **c_i** (*int*) – Specific heat of ice.
- **c_w** (*int*) – Specific heat of water.
- **L** (*int*) – Latent heat of fusion ice<->water (J/Kg).
- **rho_i** (*int*) – Density of ice (Kg/m^3).
- **rho_br** (*int*) – Density of ocean used in volume averaging.
- **rho_w** (*int*) – Density of water.
- **m** (*int*) – Cementation exponent for Archie's equation.
- **g** (*int*) – Gravity constant.
- **phi_c** (*int*) – Constant for phi.
- **P_s** (*int*) – Constant for P_s.
- **a_phi** (*int*) – Constant for a_phi.
- **b_phi** (*int*) – Constant for b_phi.

__init__(*phi_ini=1, phi_ini_Stefan=1, beta=1, kappa=1, mu=1, Ra_c=0, alpha=1, k_i=1, k_br=1, k_w=1, D_s=0, c_br=1, c_i=1, c_w=1, L=1, rho_i=1, rho_br=1, rho_w=1, m=1, g=0, phi_c=1, P_s=1, a_phi=1, b_phi=1*)

Parameters

- **phi_ini** (*int*)
- **phi_ini_Stefan** (*int*)
- **beta** (*int*)
- **kappa** (*int*)
- **mu** (*int*)
- **Ra_c** (*int*)
- **alpha** (*int*)
- **k_i** (*int*)
- **k_br** (*int*)

- **k_w** (*int*)
- **D_s** (*int*)
- **c_br** (*int*)
- **c_i** (*int*)
- **c_w** (*int*)
- **L** (*int*)
- **rho_i** (*int*)
- **rho_br** (*int*)
- **rho_w** (*int*)
- **m** (*int*)
- **g** (*int*)
- **phi_c** (*int*)
- **P_s** (*int*)
- **a_phi** (*int*)
- **b_phi** (*int*)

Return type
None

D_s: int

L: int

P_s: int

Ra_c: int

a_phi: int

alpha: int

b_phi: int

beta: int

c_br: int

c_i: int

c_w: int

g: int

k_br: int

k_i: int

k_w: int

kappa: int

m: int

mu: int

phi_c: int

phi_ini: int

phi_ini_Stefan: int

rho_br: int

rho_i: int

rho_w: int

class spyice.parameters.real_constants.**RealConstants**

Bases: object

Class representing real-valued constants used in the model.

Parameters

- **phi_ini** (*int*) – Initial freezing value.
- **phi_ini_Stefan** (*int*) – Initial freezing value according to Stefan condition.
- **beta** (*float*) – Coefficient for density dependence on salinity.
- **kappa** (*float*) – Thermal diffusivity.
- **mu** (*float*) – Viscosity.
- **Ra_c** (*float*) – Critical Rayleigh number.
- **alpha** (*float*) – Linear coefficient for Rayleigh number driven advection.
- **k_i** (*float*) – Thermal conductivity of ice.
- **k_br** (*float*) – Thermal conductivity of brine.
- **k_w** (*float*) – Thermal conductivity of water.
- **D_s** (*float*) – Diffusivity for salt.
- **c_i** (*int*) – Specific heat of ice.
- **c_br** (*int*) – Specific heat of seawater.
- **c_w** (*int*) – Specific heat of water.
- **L** (*int*) – Latent heat of fusion between ice and water.
- **rho_br** (*int*) – Density of ocean.
- **rho_i** (*int*) – Density of ice.
- **rho_w** (*int*) – Density of water.
- **m** (*int*) – Cementation exponent for Archie's equation.

- **g** (*float*) – Earth gravity.
- **phi_c** (*float*) – Critical porosity.
- **P_s** (*float*) – Partition coefficient.
- **a_phi** (*float*) – Coefficient a for porosity calculation.
- **b_phi** (*float*) – Coefficient b for porosity calculation.
- **critical_depth** (*float*) – Critical depth value.

__init__(*phi_ini=1*, *phi_ini_Stefan=1*, *beta=0.0005836*, *kappa=1.37e-07*, *mu=0.00188*, *Ra_c=1.01*, *alpha=0.0015600000000000002*, *k_i=2.0*, *k_br=0.6*, *k_w=0.6*, *D_s=2e-09*, *c_i=2000*, *c_br=3985*, *c_w=4200*, *L=334774*, *rho_br=1028*, *rho_i=917*, *rho_w=1000*, *m=2*, *g=9.8*, *phi_c=0.06823*, *P_s=0.01*, *a_phi=5.9e-06*, *b_phi=169491.5254237288*)

Parameters

- **phi_ini** (*int*)
- **phi_ini_Stefan** (*int*)
- **beta** (*float*)
- **kappa** (*float*)
- **mu** (*float*)
- **Ra_c** (*float*)
- **alpha** (*float*)
- **k_i** (*float*)
- **k_br** (*float*)
- **k_w** (*float*)
- **D_s** (*float*)
- **c_i** (*int*)
- **c_br** (*int*)
- **c_w** (*int*)
- **L** (*int*)
- **rho_br** (*int*)
- **rho_i** (*int*)
- **rho_w** (*int*)
- **m** (*int*)
- **g** (*float*)
- **phi_c** (*float*)
- **P_s** (*float*)
- **a_phi** (*float*)

- **b_phi** (*float*)

Return type
None

D_s: float

L: int

P_s: float

Ra_c: float

a_phi: float

alpha: float

b_phi: float

beta: float

c_br: int

c_i: int

c_w: int

critical_depth = 0.01

g: float

k_br: float

k_i: float

k_w: float

kappa: float

m: int

mu: float

phi_c: float

phi_ini: int

phi_ini_Stefan: int

rho_br: int

rho_i: int

rho_w: int

class spyice.parameters.results_params.**ResultsParams**

Bases: object

Class to store the results of the simulation.

Parameters

- **iter_max** (*int*) – The maximum number of iterations.
- **nz** (*int*) – The number of depth levels.

Variables

- **error_temperature** (*ndarray*) – An array of size nz to store the temperature errors.
- **error_temperature_sum** (*ndarray*) – An array of size iter_max to store the sum of temperature errors.
- **error_temperature_sum_weighted** (*ndarray*) – An array of size iter_max to store the weighted sum of temperature errors.
- **temp_grad** (*None*) – A placeholder for the temperature gradient.
- **thickness_index_total** (*ndarray*) – An array of size iter_max to store the total thickness index.
- **depth_stefan_all** (*ndarray*) – An array of size iter_max to store the depth Stefan values.
- **t_stefan_prev** (*ndarray*) – An array of size nz to store the previous Stefan temperature values.
- **t_k_prev** (*ndarray*) – An array of size nz to store the previous temperature values.
- **t_stefan_diff** (*ndarray*) – A 2D array of size (iter_max, nz) to store the differences in Stefan temperature values.
- **t_k_diff** (*ndarray*) – A 2D array of size (iter_max, nz) to store the differences in temperature values.
- **t_stefan_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the Stefan temperature values.
- **t_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the temperature values.
- **t_k_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered temperature values.
- **thickness_list** (*ndarray*) – An array of size iter_max to store the thickness values.
- **thickness_list_buffo** (*ndarray*) – An array of size iter_max to store the buffered thickness values.
- **phi_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the phi values.
- **phi_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered phi values.

- **s_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the s values.
- **s_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered s values.
- **h_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the h values.
- **h_solid_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the solid h values.
- **temp_interface** (*ndarray*) – An array of size iter_max to store the interface temperatures.
- **t_k_iter** (*list*) – A list to store the temperature values at each iteration.
- **phi_k_iter** (*list*) – A list to store the phi values at each iteration.
- **all_phi_iter** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **t_k_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all temperature values at each iteration.
- **phi_k_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **all_phi_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **all_t** (*ndarray*) – A 2D array of size (iter_max, nz) to store all temperature values.
- **all_s** (*ndarray*) – A 2D array of size (iter_max, nz) to store all s values.
- **all_phi** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values.
- **all_h** (*ndarray*) – A 2D array of size (iter_max, nz) to store all h values.
- **all_h_solid** (*ndarray*) – A 2D array of size (iter_max, nz) to store all solid h values.
- **all_w** (*ndarray*) – A 2D array of size (iter_max, nz) to store all w values.
- **all_thick** (*ndarray*) – An array of size iter_max to store all thickness values.
- **all_t_passed** (*ndarray*) – An array of size iter_max to store all passed temperature values.

__init__(*iter_max*, *nz*)

Parameters

- **iter_max** (*int*) – The maximum number of iterations.

- **nz** (*int*) – The number of depth levels.

Variables

- **error_temperature** (*ndarray*) – An array of size nz to store the temperature errors.
- **error_temperature_sum** (*ndarray*) – An array of size iter_max to store the sum of temperature errors.
- **error_temperature_sum_weighted** (*ndarray*) – An array of size iter_max to store the weighted sum of temperature errors.
- **temp_grad** (*None*) – A placeholder for the temperature gradient.
- **thickness_index_total** (*ndarray*) – An array of size iter_max to store the total thickness index.
- **depth_stefan_all** (*ndarray*) – An array of size iter_max to store the depth Stefan values.
- **t_stefan_prev** (*ndarray*) – An array of size nz to store the previous Stefan temperature values.
- **t_k_prev** (*ndarray*) – An array of size nz to store the previous temperature values.
- **t_stefan_diff** (*ndarray*) – A 2D array of size (iter_max, nz) to store the differences in Stefan temperature values.
- **t_k_diff** (*ndarray*) – A 2D array of size (iter_max, nz) to store the differences in temperature values.
- **t_stefan_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the Stefan temperature values.
- **t_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the temperature values.
- **t_k_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered temperature values.
- **thickness_list** (*ndarray*) – An array of size iter_max to store the thickness values.
- **thickness_list_buffo** (*ndarray*) – An array of size iter_max to store the buffered thickness values.
- **phi_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the phi values.
- **phi_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered phi values.
- **s_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the s values.
- **s_buffo_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the buffered s values.
- **h_k_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the h values.

- **h_solid_list** (*ndarray*) – A 2D array of size (iter_max, nz) to store the solid h values.
- **temp_interface** (*ndarray*) – An array of size iter_max to store the interface temperatures.
- **t_k_iter** (*list*) – A list to store the temperature values at each iteration.
- **phi_k_iter** (*list*) – A list to store the phi values at each iteration.
- **all_phi_iter** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **t_k_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all temperature values at each iteration.
- **phi_k_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **all_phi_iter_all** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values at each iteration.
- **all_t** (*ndarray*) – A 2D array of size (iter_max, nz) to store all temperature values.
- **all_s** (*ndarray*) – A 2D array of size (iter_max, nz) to store all s values.
- **all_phi** (*ndarray*) – A 2D array of size (iter_max, nz) to store all phi values.
- **all_h** (*ndarray*) – A 2D array of size (iter_max, nz) to store all h values.
- **all_h_solid** (*ndarray*) – A 2D array of size (iter_max, nz) to store all solid h values.
- **all_w** (*ndarray*) – A 2D array of size (iter_max, nz) to store all w values.
- **all_thick** (*ndarray*) – An array of size iter_max to store all thickness values.
- **all_t_passed** (*ndarray*) – An array of size iter_max to store all passed temperature values.

static store_results(*results_dataclass*, *temp*, *s_sw*, *phi*, *h*, *h_solid*, *w*, *thickness*, *t_passed*, *t*)

Stores the results of the simulation in the given results_dataclass.

Parameters

- **results_dataclass** – An instance of the ResultsDataClass where the results will be stored.
- **temp** – The temperature values to be stored.
- **s_sw** – The s_sw values to be stored.
- **phi** – The phi values to be stored.

- **h** – The h values to be stored.
- **h_solid** – The h_solid values to be stored.
- **w** – The w values to be stored.
- **thickness** – The thickness value to be stored.
- **t_passed** – The t_passed value to be stored.
- **t** – The t value to be stored.

Returns
The updated results_dataclass with the stored results.

static store_results_for_iter_t(*results_dataclass*, *t*, *thickness_index*, *t_k*, *t_stefan*, *s_k*, *s_k_buffo*, *phi_k*, *phi_k_buffo*, *h_k*, *h_solid*, *thickness*, *thickness_buffo*, *thickness_stefan*, *t_k_buffo*, *buffo=False*)

Stores the results for a given iteration 't' in the 'results_dataclass'.

Parameters

- **results_dataclass** – An instance of the results dataclass.
- **t** – The iteration number.
- **thickness_index** – The index of the thickness.
- **t_k** – The temperature values.
- **t_stefan** – The Stefan temperature values.
- **s_k** – The entropy values.
- **s_k_buffo** – The entropy values for buffo.
- **phi_k** – The phase fraction values.
- **phi_k_buffo** – The phase fraction values for buffo.
- **h_k** – The enthalpy values.
- **h_solid** – The solid enthalpy values.
- **thickness** – The thickness values.
- **thickness_buffo** – The thickness values for buffo.
- **thickness_stefan** – The Stefan thickness values.
- **t_k_buffo** – The temperature values for buffo.
- **buffo** – A boolean indicating if buffo is enabled (default is False).

Returns
The updated 'results_dataclass' with the stored results.

class spyice.parameters.user_input.**UserInput**

Bases: object

Represents the user input parameters for the model.

Variables

- **constants** (RealConstants (page **??**) | DebugConstants (page **??**)) – The type of constants to use.
- **max_iterations** (*int*) – The maximum number of iterations.
- **is_stefan** (*bool*) – Flag indicating whether Stefan condition is applied.
- **is_buffo** (*bool*) – Flag indicating whether Buffo condition is applied.
- **liquidus_relation_type** (*str*) – The type of liquidus relation to use.
- **grid_resolution_dz** (*float*) – The grid resolution in the z-direction.
- **boundary_condition_type** (*str*) – The type of boundary condition to use.
- **temperature_tolerance** (*float*) – The temperature tolerance.
- **salinity_tolerance** (*float*) – The salinity tolerance.
- **liquid_fraction_tolerance** (*float*) – The liquid fraction tolerance.
- **initial_temperature** (*str*) – The initial temperature profile.
- **initial_salinity** (*str*) – The initial salinity profile.
- **initial_liquid_fraction** (*str*) – The initial liquid fraction profile.
- **output_suffix** (*str*) – The suffix to be added to the output files.
- **temperature_top_type** (*str*) – The type of temperature condition at the top boundary.
- **phase_type** (*int*) – The type of phase to consider.
- **grid_timestep_dt** (*float*) – The grid timestep.
- **dir_output_name** (*str*) – The name of the output directory.
- **critical_liquid_fraction** (*float*) – The critical liquid fraction.
- **boundary_salinity** (*float*) – The boundary salinity (automatically calculated).
- **temperature_melt** (*float*) – The temperature at which the material melts (automatically calculated).
- **boundary_top_temperature** (*float*) – The temperature at the top boundary (automatically calculated).
- **geometry_type** (*int*) – The type of geometry.
- **counter_limit** (*int*) – The counter limit.

__post_init__()

Performs post-initialization tasks.

__init__(*constants=RealConstants(phi_ini=1, phi_ini_Stefan=1, beta=0.0005836, kappa=1.37e-07, mu=0.00188, Ra_c=1.01, alpha=0.0015600000000000002, k_i=2.0, k_br=0.6, k_w=0.6, D_s=2e-09, c_i=2000, c_br=3985, c_w=4200, L=334774, rho_br=1028, rho_i=917, rho_w=1000, m=2, g=9.8, phi_c=0.06823, P_s=0.01, a_phi=5.9e-06, b_phi=169491.5254237288), max_iterations=500, is_stefan=True, is_buffo=True, liquidus_relation_type='Normal', grid_resolution_dz=0.01, boundary_condition_type='Dirichlet', temperature_tolerance=0.01, salinity_tolerance=0.01, liquid_fraction_tolerance=0.001, initial_temperature='T(S)', initial_salinity='S34', initial_liquid_fraction='P1', output_suffix='const_dens-mushfix', temperature_top_type='Stefan', phase_type=1, grid_timestep_dt=47.0, dir_output_name='Temperature_{S_IC}_{bc_condition}_{dz}_{dt}_{iter_max}_{cap_dens}', critical_liquid_fraction=0.1, counter_limit=100000*)

Parameters

- **constants** (`RealConstants` (page **??**) | `DebugConstants` (page **??**))
- **max_iterations** (*int*)
- **is_stefan** (*bool*)
- **is_buffo** (*bool*)
- **liquidus_relation_type** (*str*)
- **grid_resolution_dz** (*float*)
- **boundary_condition_type** (*str*)
- **temperature_tolerance** (*float*)
- **salinity_tolerance** (*float*)
- **liquid_fraction_tolerance** (*float*)
- **initial_temperature** (*str*)
- **initial_salinity** (*str*)
- **initial_liquid_fraction** (*str*)
- **output_suffix** (*str*)
- **temperature_top_type** (*str*)
- **phase_type** (*int*)
- **grid_timestep_dt** (*float*)
- **dir_output_name** (*str*)
- **critical_liquid_fraction** (*float*)
- **counter_limit** (*int*)

Return type
None

boundary_condition_type: str = 'Dirichlet'

boundary_salinity: float

boundary_top_temperature: float

constants: *RealConstants* (page ??) | *DebugConstants* (page ??) = RealConstants(phi_ini=1, phi_ini_Stefan=1, beta=0.0005836, kappa=1.37e-07, mu=0.00188, Ra_c=1.01, alpha=0.0015600000000000002, k_i=2.0, k_br=0.6, k_w=0.6, D_s=2e-09, c_i=2000, c_br=3985, c_w=4200, L=334774, rho_br=1028, rho_i=917, rho_w=1000, m=2, g=9.8, phi_c=0.06823, P_s=0.01, a_phi=5.9e-06, b_phi=169491.5254237288)

counter_limit: int = 100000

critical_liquid_fraction: float = 0.1

dir_output_name: str = 'Temperature_{S_IC}_{bc_condition}_{dz}_{dt}_{iter_max}_{cap_dens}'

geometry_type: int

grid_resolution_dz: float = 0.01

grid_timestep_dt: float = 47.0

initial_liquid_fraction: str = 'P1'

initial_salinity: str = 'S34'

initial_temperature: str = 'T(S)'

is_buffo: bool = True

is_stefan: bool = True

liquid_fraction_tolerance: float = 0.001

liquidus_relation_type: str = 'Normal'

max_iterations: int = 500

output_suffix: str = 'const_dens-mushfix'

phase_type: int = 1

salinity_tolerance: float = 0.01

temperature_melt: float

temperature_tolerance: float = 0.01

temperature_top_type: str = 'Stefan'

spyice.parameters.user_input.**fourier_number_timestep**(*constants*)

Calculates the Fourier number for the given timestep.

Parameters

constants (*object*) – An object containing the required constants.

Returns
The calculated Fourier number.

Return type
float

Module contents

spyice.postprocess package

Submodules

class spyice.postprocess.analysis.**Analysis**

Bases: object

Represents an analysis object that performs error analysis on temperature differences.

Parameters

- **t_k_diff** (*float*) – The temperature difference in Kelvin.
- **t_stefan_diff** (*float*) – The temperature difference in Stefan-Boltzmann units.

__init__(*t_k_diff*, *t_stefan_diff*)

Parameters

- **t_k_diff** (*float*) – The temperature difference in Kelvin.
- **t_stefan_diff** (*float*) – The temperature difference in Stefan-Boltzmann units.

calculate_errors(*field_array*, *error_norms_object*)

Calculates the errors of a given field array using the provided error norms object.

Parameters

- **field_array** (*numpy.ndarray*) – The field array to calculate the errors for.
- **error_norms_object** (ErrorNorms (page **??**)) – The error norms object used to calculate the errors.

Returns
A tuple containing the one-norm error, infinity-norm error, and two-norm error.

Return type
tuple

error_analytical_numerical()

Calculates the errors of Numerical and Analytical using error norms one, two and infinity.

Parameters
None

Returns

A tuple containing the errors calculated using different error norms:

- T_k_Stefan_diff_L1norm (float): L1 norm of the difference between T_k_list and T_Stefan_list.
- T_k_Stefan_diff_infnorm (float): Infinity norm of the difference between T_k_list and T_Stefan_list.
- T_k_Stefan_diff_L2norm (float): L2 norm of the difference between T_k_list and T_Stefan_list.
- T_Stefan_diff_L1norm (float): L1 norm of the difference between consecutive T_Stefan values.
- T_Stefan_diff_infnorm (float): Infinity norm of the difference between consecutive T_Stefan values.
- T_Stefan_diff_L2norm (float): L2 norm of the difference between consecutive T_Stefan values.
- T_k_diff_infnorm (float): Infinity norm of the difference between consecutive T_k values.
- T_k_diff_L2norm (float): L2 norm of the difference between consecutive T_k values.
- T_k_diff_L1norm (float): L1 norm of the difference between consecutive T_k values.

Return type
tuple

classmethod get_error_results(*t_k_diff*, *t_stefan_diff*)

Runs error analysis on the given temperature differences.

Parameters

- **cls** – The class object.
- **t_k_diff** (*array*) – The temperature difference for k.
- **t_stefan_diff** (*array*) – The temperature difference for Stefan.

Returns
An instance of the AnalysisData class containing the error analysis results.

Return type
AnalysisData (page **??**)

set_analysis()

Sets up the analysis for the current object.

This method initializes the *error_norms_object* attribute with an instance of the *ErrorNorms* class, passing the *t_k_diff* and *t_stefan_diff* attributes as arguments. It then calculates the numerical analytical difference using the *numerical_analytical_diff* method of the *error_norms_object*.

Parameters
None

Returns
None

static set_dataclass(*data_to_be_converted*, *dataclass*)

Sets the values of a dataclass object using a dictionary.

Parameters

- **data_to_be_converted** (*dict*) – A dictionary containing the values to be set.
- **dataclass** (*dataclass*) – The dataclass object to be updated.

Returns
The updated dataclass object.

Return type
dataclass

store_field_errors(*T_k_Stefan_diff_L1norm*, *T_k_Stefan_diff_infnorm*, *T_k_Stefan_diff_L2norm*, *T_Stefan_diff_infnorm*, *T_Stefan_diff_L2norm*, *T_Stefan_diff_L1norm*, *T_k_diff_infnorm*, *T_k_diff_L2norm*, *T_k_diff_L1norm*)

Stores the field errors.

Parameters

- **T_k_Stefan_diff_L1norm** (*float*) – The L1 norm of the temperature and concentration difference.
- **T_k_Stefan_diff_infnorm** (*float*) – The infinity norm of the temperature and concentration difference.
- **T_k_Stefan_diff_L2norm** (*float*) – The L2 norm of the temperature and concentration difference.
- **T_Stefan_diff_infnorm** (*float*) – The infinity norm of the temperature difference.
- **T_Stefan_diff_L2norm** (*float*) – The L2 norm of the temperature difference.
- **T_Stefan_diff_L1norm** (*float*) – The L1 norm of the temperature difference.
- **T_k_diff_infnorm** (*float*) – The infinity norm of the temperature and concentration difference.
- **T_k_diff_L2norm** (*float*) – The L2 norm of the temperature and concentration difference.
- **T_k_diff_L1norm** (*float*) – The L1 norm of the temperature and concentration difference.

class spyice.postprocess.analysis.**AnalysisData**

Bases: object

Represents the analysis data.

Variables
all_variables (*dict*) – A dictionary containing all the variables.

__init__(*all_variables*)

Parameters
all_variables (*dict*)

Return type
None

all_variables: dict

class spyice.postprocess.visualise_model.**VisualiseModel**

Bases: object

A class for visualizing sea ice model results.

Parameters

- **user_input_dataclass** (UserInput (page **??**)) – An instance of the UserInput class containing user input data.
- **results_dataclass** (ResultsParams (page **??**)) – An instance of the ResultsParams class containing results data.
- **error_analysis_dataclass** (Analysis (page **??**)) – An instance of the Analysis class containing error analysis data.

__init__(*user_input_dataclass*, *results_dataclass*, *error_analysis_dataclass*)

Parameters

- **user_input_dataclass** (UserInput (page **??**)) – An instance of the UserInput class containing user input data.
- **results_dataclass** (ResultsParams (page **??**)) – An instance of the ResultsParams class containing results data.
- **error_analysis_dataclass** (Analysis (page **??**)) – An instance of the Analysis class containing error analysis data.

Return type
None

phi_slope(*iteration*)

Calculates the indices of the mushy regions based on the phi values.

Parameters
iteration (*int*) – The iteration number.

Returns
The indices of the mushy regions.

Return type
numpy.ndarray

plot_depth_over_time(*savefig=False*)

Plots the depth over time.

Parameters

savefig (*bool, optional*) – Whether to save the figure. Defaults to False.

plot_error_temp(*zoom_x, norm='inf', savefig=True*)

Plots the temperature errors using the specified norm.

Parameters

- **zoom_x** (*int*) – The maximum value for the x-axis.
- **norm** (*str, optional*) – The norm to be used for plotting. Defaults to "inf".
- **savefig** (*bool, optional*) – Whether to save the figure. Defaults to True.

plot_error_temp_diff(*zoom_x, savefig='True'*)

Plots the temperature differences between two consecutive iterations.

Parameters

- **zoom_x** (*int*) – The maximum value for the x-axis.
- **savefig** (*str, optional*) – Indicates whether to save the figure or not. Defaults to "True".

plot_temperature(*z_depth, savefig=True, Buffo_matlab=False*)

Plots the temperature evolution at a given depth.

Parameters

- **z_depth** (*float*) – The depth at which to plot the temperature evolution.
- **savefig** (*bool, optional*) – Whether to save the figure. Defaults to True.
- **Buffo_matlab** (*bool, optional*) – Whether to include Buffo-matlab data in the plot. Defaults to False.

Module contents

spyice.preprocess package

Submodules

class spyice.preprocess.geometry_settings.**GeometrySettings**

Bases: object

Set up model geometry with two parameters, an integer geom and a float dz.

Parameters

- **geom** (*int*) – 1 (is a test case scenario) or 2 (is according to W3 in Buffo et al. 2018)

- **dz** (*float*) – The parameter *dz* appears to be a float type. It is likely used to represent a specific value related to geometry or spatial calculations. If you need further assistance or have any specific questions about how to use this parameter in your code, feel free to ask!

__init__(*geom*, *dz*)

Parameters

- **geom** (*int*) – 1 (is a test case scenario) or 2 (is according to W3 in Buffo et al. 2018)
- **dz** (*float*) – The parameter *dz* appears to be a float type. It is likely used to represent a specific value related to geometry or spatial calculations. If you need further assistance or have any specific questions about how to use this parameter in your code, feel free to ask!

Return type
None

exception
spyice.preprocess.initial_boundary_conditions.**SalinityUnavailableError**

Bases: Exception

Exception raised when the S_IC option is not available in initial conditions.

spyice.preprocess.initial_boundary_conditions.**boundary_condition**(*argument*, *t_passed*, *initial_salinity*, ***kwargs*)

Calculates the boundary conditions for temperature or salinity.

Parameters

- **argument** (*str*) – The argument specifying whether to calculate temperature or salinity.
- **t_passed** (*float*) – The time passed.
- **initial_salinity** (*float*) – The initial salinity value.
- ****kwargs** – Additional keyword arguments.

Returns
A tuple containing the boundary conditions for temperature or salinity.

Return type
tuple

Raises
None –

spyice.preprocess.initial_boundary_conditions.**calculate_boundary_salinity**(*initial_salinity*)

Calculates the boundary salinity values based on the initial salinity.

Parameters
initial_salinity (*str*) – The initial salinity value.

Returns
A tuple containing the bottom and top salinity values.

Return type
tuple

Raises
SalinityException – If the initial salinity value is not valid.

spyice.preprocess.initial_boundary_conditions.**calculate_boundary_temperature**(*t_passed*, *initial_salinity*, *kwargs*)

Calculates the boundary temperature based on the given parameters.

Parameters

- **t_passed** (*float*) – The time passed.
- **initial_salinity** (*str*) – The initial salinity value.
- **kwargs** (*dict*) – Additional keyword arguments.

Returns
A tuple containing the bottom temperature and the top temperature.

Return type
tuple

spyice.preprocess.initial_boundary_conditions.**compute_melting_temperature_from_salinity**(*i*

Computes the melting temperature from the given initial salinity.

Parameters
initial_salinity (*float*) – The initial salinity value.

Returns
The computed melting temperature.

Return type
float

spyice.preprocess.initial_boundary_conditions.**raise_salinity_exception**(*salinity_value*)

Raises a custom exception if the salinity value is empty.

Parameters
salinity_value (*str*) – The salinity value.

Raises
SalinityUnavailableError (page **??**) – If the salinity value is empty.

spyice.preprocess.initial_boundary_conditions.**set_boundary_temperature**(*t_passed*, *temperature_bottom*, ***kwargs*)

Sets the boundary temperature based on the given parameters.

Parameters

- **t_passed** (*float*) – The time passed.
- **temperature_bottom** (*float*) – The bottom temperature.

- ****kwargs** – Additional keyword arguments.

Returns
A tuple containing the top temperature and the bottom temperature.

Return type
tuple

spyice.preprocess.initial_boundary_conditions.**set_inital_salinity**(*initial_salinity*, *nz*, *boundary_salinity*)

Sets the initial salinity values for each layer in the model.

Parameters

- **initial_salinity** (*str*) – The type of initial salinity distribution.
- **nz** (*int*) – The number of layers in the model.
- **boundary_salinity** (*float*) – The salinity value at the boundary.

Returns
An array of initial salinity values for each layer.

Return type
numpy.ndarray

Raises
SalinityException – If the initial_salinity value is not recognized.

spyice.preprocess.initial_boundary_conditions.**set_inital_temperature**(*initial_temperature*, *nz*, *boundary_salinity*, *boundary_top_temperature*)

Sets the initial temperature profile based on the given parameters.

Parameters

- **initial_temperature** (*str*) – The type of initial temperature profile to set.
- **nz** (*int*) – The number of vertical grid points.
- **boundary_salinity** (*float*) – The salinity at the boundary.
- **boundary_top_temperature** (*float*) – The temperature at the top boundary.

Returns
The initial temperature profile as a 1D numpy array.

Return type
numpy.ndarray

Raises
None –

spyice.preprocess.initial_boundary_conditions.**set_initial_conditions**(*nz*, *boundary_salinity*, *initial_temperature='T0'*, *initial_salinity='S1'*, *initial_liquid_fraction='P1'*, *boundary_top_temperature=265*)

Sets the initial conditions for the simulation.

Parameters

- **nz** (*int*) – Number of vertical grid points.
- **boundary_salinity** (*float*) – Salinity value at the boundary.
- **initial_temperature** (*str, optional*) – Initial temperature profile. Defaults to "T0".
- **initial_salinity** (*str, optional*) – Initial salinity profile. Defaults to "S1".
- **initial_liquid_fraction** (*str, optional*) – Initial liquid fraction profile. Defaults to "P1".
- **boundary_top_temperature** (*float, optional*) – Temperature value at the top boundary. Defaults to 265.0.

Returns

A tuple containing the temperature, salinity, liquid fraction, and upwind velocity arrays.

Return type

tuple

spyice.preprocess.initial_boundary_conditions.**set_initial_liquidfraction**(*initial_liquid_fraction*, *nz*)

Sets the initial liquid fraction based on the given input.

Parameters

- **initial_liquid_fraction** (*str*) – The initial liquid fraction type.
- **nz** (*int*) – The number of grid points.

Returns

The array representing the initial liquid fraction.

Return type

numpy.ndarray

Raises

None –

Examples

```
>>> set_initial_liquidfraction("P1", 10)
array([1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
>>> set_initial_liquidfraction("P_Stefan", 5)
array([0., 0., 0., 0., 0.])
>>> set_initial_liquidfraction("P0", 8)
array([0., 0., 0., 0., 0., 0., 0., 0.])
```

spyice.preprocess.initial_boundary_conditions.**t_w3**(*dt*)

Calculates the top boundary temperature and freeze date.

Parameters
dt (*float*) – The time step.

Returns
A tuple containing the top boundary temperature array and the freeze date.

Return type
tuple

spyice.preprocess.initial_boundary_conditions.**temperature_gradient**(*phi*, *nz*)

Calculates the temperature gradient based on the given potential temperature profile and number of vertical levels.

Parameters

- **phi** (*list*) – The potential temperature profile.
- **nz** (*int*) – The number of vertical levels.

Returns
The calculated temperature gradient.

Return type
float

Raises
None –

class spyice.preprocess.modify_initial_boundary.**ModifyInitialBoundary**

Bases: object

Defines functions for managing initial boundary conditions in a sea ice model.

__init__()

Return type
None

bc_neumann(*phi_k*, *nz*, *bc_condition=None*)

Applies Neumann boundary condition to the given field.

Parameters

- **phi_k** (*numpy.ndarray*) – The field to which the boundary condition is applied.

- **nz** (*int*) – The number of grid points in the vertical direction.
- **bc_condition** (*str, optional*) – The type of boundary condition. Defaults to None.

Returns
None

set_boundary_condition_type(*critical_depth*, *bc_type='Neumann'*)

Sets the boundary condition type for the given critical depth.

Parameters

- **critical_depth** (*float*) – The critical depth value.
- **bc_type** (*str, optional*) – The type of boundary condition. Defaults to "Neumann".

Raises
None –

Returns
None

class spyice.preprocess.pre_process.**PreProcess**

Bases: *UserInput* (page **??**), *GeometrySettings* (page **??**), *ResultsParams* (page **??**)

Class for preprocessing data before modeling.

Initialize the PreProcess class. :param config_data: The configuration data object. :type config_data: ConfigData :param output_dir: The output directory. :type output_dir: str

Raises
None –

Returns
None

__init__(*config_data*, *output_dir*)

Initialize the PreProcess class. :param config_data: The configuration data object. :type config_data: ConfigData :param output_dir: The output directory. :type output_dir: str

Raises
None –

Returns
None

get_userinput()

Returns a UserInput object with the following attributes:

Parameters
self (PreProcess (page **??**)) – The PreProcess instance.

Returns

A UserInput object with the following attributes:

- constants: The constants attribute.

- grid_timestep_dt: The grid_timestep_dt attribute.
- initial_salinity: The initial_salinity attribute.
- dir_output_name: The dir_output_name attribute.
- max_iterations: The max_iterations attribute.

Return type
UserInput (page **??**)

classmethod get_variables(*config*, *out_dir_final*)

Retrieves variables and user input data after preprocessing.

Parameters

- **cls** – The class object.
- **config** (*dataclass*) – The configuration object.
- **out_dir_final** (*str*) – The output directory.

Returns
A tuple containing the filtered variables and user input data.

Return type
tuple[PreprocessData (page **??**), UserInput (page **??**)]

preprocess()

Preprocesses the data before running the simulation. This method sets up the initial conditions and boundary conditions for the simulation. It calculates the solid enthalpy and updates the enthalpy based on temperature, salinity, and liquid fraction. Finally, it initializes the ice thickness and prints a message indicating that the initial and boundary conditions have been applied.

Args:
None

static set_dataclass(*data_to_be_converted*, *dataclass*)

Parameters

- **data_to_be_converted** (*dict*)
- **dataclass** (*dataclass*)

set_preprocess(*config_data*, *output_dir*)

class spyice.preprocess.pre_process.**PreprocessData**

Bases: object

Class representing the preprocessing of data.

Variables
is_preprocessing (*bool*) – Flag indicating if preprocessing is enabled.

__init__(*is_preprocessing=True*)

Parameters
is_preprocessing (*bool*)

Return type
None

is_preprocessing: bool = True

spyice.preprocess.pre_process.**set_up_iter**(*iter_max*, *grid_timestep_dt*)

Sets up the iteration parameters for the simulation. :param iter_max: The maximum number of iterations. :type iter_max: int :param grid_timestep_dt: The time step for the grid. :type grid_timestep_dt: float

Returns
Always returns 0.

Return type
int

Module contents

spyice.utils package

Submodules

class spyice.utils.config_sort.**ConfigData**

Bases: object

Class representing configuration data.

Parameters
setup (*bool*) – Indicates if the setup is enabled or not.

__init__()

Return type
None

setup = True

class spyice.utils.config_sort.**ConfigSort**

Bases: object

A class that provides methods for retrieving configuration parameters.

Parameters
config (*DictConfig*) – The configuration dictionary.

Variables
config (*DictConfig*) – The configuration dictionary.

get_config_params(config

DictConfig): Retrieves configuration parameters using the 'consts', 'dt', 'S_IC', and 'iter_max' keys.

get_ownconfig_params(*config*)

Retrieves configuration parameters using the 'constants', 'dt', 'S_IC', and 'iter_max' keys.

getconfig_dataclass(*config*, *config_type='default'*)

Retrieves configuration parameters based on the specified config_type.

Parameters

- **config** (*dataclass*)
- **config_type** (*str*)

Return type
dataclass

__init__(*config*)

A class that provides methods for retrieving configuration parameters.

Parameters
config (*DictConfig*) – The configuration dictionary.

Variables
config (*DictConfig*) – The configuration dictionary.

get_config_params(config

DictConfig): Retrieves configuration parameters using the 'consts', 'dt', 'S_IC', and 'iter_max' keys.

get_ownconfig_params(*config*)

Retrieves configuration parameters using the 'constants', 'dt', 'S_IC', and 'iter_max' keys.

getconfig_dataclass(*config*, *config_type='default'*)

Retrieves configuration parameters based on the specified config_type.

Parameters
- **config** (*dataclass*)
- **config_type** (*str*)

Return type
dataclass

get_config_params(*config*)

Get the configuration parameters from the given *config* dictionary.

Parameters
config (*DictConfig*) – The configuration dictionary.

Returns
None

Raises
None –

get_ownconfig_params(*config*)

Retrieves the parameters from the given configuration dictionary.

Parameters
config (*dict*) – The configuration dictionary.

Returns
None

classmethod getconfig_dataclass(*config*, *config_type='default'*)

Retrieves configuration parameters based on the specified config_type.

Parameters

- **config** (*dataclass*) – The configuration dictionary.
- **config_type** (*str*) – The type of configuration "default" or "jupyter". "jupyter" is used for Jupyter notebook configurations.

Returns

An instance of the ConfigData class.

Return type

ConfigData (page **??**)

class spyice.utils.error_norms.**ErrorNorms**

Bases: object

Defines functions for calculating error norms between numerical and analytical values.

Parameters

- **numerical_values** (*list*) – A list of numerical values.
- **analytical_values** (*list*) – A list of analytical values.

Variables

- **numerical_values** (*list*) – A list of numerical values.
- **analytical_values** (*list*) – A list of analytical values.
- **iteration_count** (*int*) – The count of iterations.

__init__(*numerical_values*, *analytical_values*)

Parameters

- **numerical_values** (*list*) – A list of numerical values.
- **analytical_values** (*list*) – A list of analytical values.

Variables

- **numerical_values** (*list*) – A list of numerical values.
- **analytical_values** (*list*) – A list of analytical values.
- **iteration_count** (*int*) – The count of iterations.

Return type

None

infinity_norm(*numerical_analytical_diff*)

Calculates the infinity norm of the given numerical-analytical difference.

Parameters

numerical_analytical_diff (*array-like*) – The numerical-analytical difference.

Returns
The infinity norm of the difference. If the analytical values are 1-dimensional, a single float value is returned. Otherwise, a numpy array containing the infinity norm for each iteration is returned.

Return type
float or numpy.ndarray

numerical_analytical_diff()

Calculates the absolute difference between the analytical values and the numerical values.

Parameters
None

Returns
The array containing the absolute differences between the analytical values and the numerical values.

Return type
numpy.ndarray

one_norm(*numerical_analytical_diff*)

Calculates the one norm of the numerical-analytical difference.

Parameters
numerical_analytical_diff (*ndarray*) – The numerical-analytical difference.

Returns
The one norm of the numerical-analytical difference. If the analytical values are 1D, a float is returned. Otherwise, an ndarray is returned with the one norm for each iteration.

Return type
ndarray or float

Raises
None –

two_norm(*numerical_analytical_diff*)

Calculates the two-norm error of the numerical-analytical difference.

Parameters
numerical_analytical_diff (*ndarray*) – The numerical-analytical difference.

Returns
The two-norm error. If the analytical values are 1D, a float is returned. If the analytical values are 2D, an ndarray is returned with the two-norm error for each iteration.

Return type
ndarray or float

spyice.utils.helpers.**set_dataclass**(*data_to_be_converted*, *dataclass*)

Sets the values of a dataclass object using a dictionary.

Parameters

- **data_to_be_converted** (*dict*) – The dictionary containing the data to be converted.
- **dataclass** (*dataclass*) – The dataclass object to be updated.

Returns
The updated dataclass object.

Return type
dataclass

spyice.utils.helpers.**t_total**(*t_passed*, *dt*)

Computes total time passed based on current time step dt and, total time of previous time step.

Parameters

- **t_passed** (*float*) – The time that has already passed.
- **dt** (*float*) – The time increment.

Returns
The total time.

Return type
float

exception spyice.utils.spyice_exceptions.**ConvergenceError**

Bases: Exception

Exception raised when convergence is not reached.

__init__(*message='Convergence not reached.'*)

exception spyice.utils.spyice_exceptions.**InvalidPhaseError**

Bases: Exception

Custom exception class for SeaIceModel errors.

class spyice.utils.spyice_logger.**SpyiceLogger**

Bases: StringIO

A class that replaces the print function with a logger object.

Parameters

- **logger** – The logger object used for logging.
- ***args** – Additional positional arguments.
- ****kwargs** – Additional keyword arguments.

__init__(*logger*, **args*, ***kwargs*)

Parameters

- **logger** – The logger object used for logging.
- ***args** – Additional positional arguments.
- ****kwargs** – Additional keyword arguments.

flush()

Flush write buffers, if applicable.

This is not implemented for read-only and non-blocking streams.

write(*message*)

Writes the given message to the logger if it is not empty.

Parameters
message (*str*) – The message to be written to the logger.

Returns
None

Module contents

3.1.2 Submodules

spyice.coefficients.**update_coefficients**(*argument*, *x_initial*, *w*, *phi*, *nz*, *salinity_initial*)

Updates of coefficients required to solve the Advection Reaction Diffusion Equation for each time step for temperature or salinity

Parameters

- **argument** (*str*) – Either 'temperature' or 'salinity'
- **x_initial** (*float*) – Initial value for salinity or temperature [ppt] or [K]
- **w** (*float*) – Brine velocity [ms-1]
- **phi** (*float*) – Liquid fraction [-]
- **nz** (*int*) – Number of computational nodes

Returns

A tuple containing the following coefficients:

- a (numpy.ndarray): 'temperature': heat capacity | 'salinity': liquid fraction
- b (numpy.ndarray): Brine velocity [ms-1]
- c (numpy.ndarray): 'temperature': thermal conductivity | 'salinity': salt diffusivity
- d (numpy.ndarray): 'temperature': latent heat | 'salinity': factor to determine salinity increase due to liquid fraction decrease

Return type
tuple

class spyice.main_process.**MainProcess**

Bases: object

Main class to run the model.

Parameters

- **config** – The configuration object.
- **hyd_output_dir** (*Path | str*) – The directory path for the hydraulic output. Defaults to the 'outputs' directory in the current working directory.
- **project_path** (*Path | str*) – The project path. Defaults to the current working directory.

__init__(*config*, *hyd_output_dir=WindowsPath('C:/Users/sneha/Documents/MBDHiWi/SusEng/Project/spyicedir*, *project_path='C:\\Users\\sneha\\Documents\\MBDHiWi\\SusEng\\Project\\spyicedir\\spyi*

Parameters

- **config** – The configuration object.
- **hyd_output_dir** (*Path | str*) – The directory path for the hydraulic output. Defaults to the 'outputs' directory in the current working directory.
- **project_path** (*Path | str*) – The project path. Defaults to the current working directory.

plot_model(*userinput_data*, *results_data*, *analysis_data*)

Plots various visualizations of the model.

Parameters

- **userinput_data** (*UserInputData*) – The user input data.
- **results_data** (*ResultsData*) – The results data.
- **analysis_data** (AnalysisData (page **??**)) – The error analysis data.

Returns
None

Raises
None –

run_model()

Runs the model using the provided configuration and output directory.

Parameters
None

Returns
None

Return type
None

spyice.main_process.**create_output_directory**(*hyd_dir*)

Creates an output directory for storing temperature data.

Parameters
hyd_dir (*str*) – The directory path where the output directory will be created.

Returns
The path of the created output directory.

Return type
str

Raises
None –

spyice.rhs.**apply_boundary_condition**(*argument*, *x_initial*, *source*, *factor1*, *factor3*, *a*, *delta_upwind*, *w*, *nz*, *t_passed*, *salinity_initial*, *_temperature_top*, *is_stefan*, *is_buffo=False*, *bc_neumann=None*)

Creates the right hand side of the matrix equation considering source terms.

Parameters

- **argument** (*str*) – Either 'salinity' for salt equation or 'temperature' for temperature equation.
- **x_initial** (*float*) – Value of X at the last time step.
- **source** (*float*) – Source term.
- **factor1** (*float*) – Factor 1.
- **factor3** (*float*) – Factor 3.
- **a** (*float*) – A parameter.
- **delta_upwind** (*float*) – Difference of ice volume fraction between this and the last time step.
- **w** (*float*) – W parameter.
- **nz** (*int*) – Number of computational nodes.
- **t_passed** (*float*) – Time passed in seconds.
- **salinity_initial** (*float*) – Initial salinity value.
- **_temperature_top** (*float*) – Top temperature value.
- **is_stefan** (*bool*) – Indicates if Stefan condition is used.
- **is_buffo** (*bool, optional*) – Indicates if Buffo condition is used. Defaults to False.
- **bc_neumann** (*float, optional*) – Neumann boundary condition. Defaults to None.
- **float** – The right hand side of the equation.

spyice.rhs.**correct_for_brine_movement**(*argument*, *x_initial*, *w*, *t_passed*, *nz*, *salinity_initial*, *top_temp*)

Corrects for brine movement based on the given arguments.

Parameters

- **argument** (*str*) – The argument for correction, either "salinity" or "temperature".
- **x_initial** (*numpy.ndarray*) – The initial values of x.

- **w** (*numpy.ndarray*) – The values of w.
- **t_passed** (*float*) – The time passed.
- **nz** (*int*) – The number of elements.
- **salinity_initial** (*float*) – The initial salinity value.
- **top_temp** (*float*) – The top temperature value.

Returns
The corrected values of x.

Return type
numpy.ndarray

Raises
None –

spyice.statevariables.**compute_error_for_convergence**(*temperature_calculated*, *temperature_previous*, *salinity_calculated*, *salinity_previous*, *liquid_fraction_calculated*, *liquid_fraction_previous*)

Computes the errors for convergence between the calculated and previous values of temperature, salinity, and liquid fraction.

Parameters

- **temperature_calculated** (*numpy.ndarray*) – Array of calculated temperature values.
- **temperature_previous** (*numpy.ndarray*) – Array of previous temperature values.
- **salinity_calculated** (*numpy.ndarray*) – Array of calculated salinity values.
- **salinity_previous** (*numpy.ndarray*) – Array of previous salinity values.
- **liquid_fraction_calculated** (*numpy.ndarray*) – Array of calculated liquid fraction values.
- **liquid_fraction_previous** (*numpy.ndarray*) – Array of previous liquid fraction values.

Returns

A tuple containing the following error values:

- temperature_error_max (float): Maximum temperature error for convergence check.
- temperature_error_all (numpy.ndarray): Full temperature error for convergence check.
- salinity_error_max (float): Maximum salinity error for convergence check.

- salinity_err_all (numpy.ndarray): Full salinity error for convergence check.
- liquid_fraction_error_max (float): Maximum liquid fraction error for convergence check.
- liquid_fraction_error_all (numpy.ndarray): Full liquid fraction error for convergence check.

Return type
tuple

spyice.statevariables.**define_previous_statevariable**(*temperature_calculated*, *salinity_calculated*, *liquid_fraction_calculated*)

Defines the previous state variables for iteration.

Parameters

- **temperature_calculated** (*float*) – The calculated temperature.
- **salinity_calculated** (*float*) – The calculated salinity.
- **liquid_fraction_calculated** (*float*) – The calculated liquid fraction.

Returns
A tuple containing the previous temperature, salinity, and liquid fraction.

Return type
tuple

spyice.statevariables.**initialize_statevariables**(*temperature*, *salinity*, *liquid_fraction*)

Initializes the state variables for the given temperature, salinity, and liquid fraction.

Parameters

- **temperature** (*float*) – The initial temperature.
- **salinity** (*float*) – The initial salinity.
- **liquid_fraction** (*float*) – The initial liquid fraction.

Returns

A tuple containing the initialized state variables:

- temperature_initial (float): The initial temperature.
- temperature_new (float): The current temperature for iteration.
- temperature_previous (float): The previous temperature for initialization.
- salinity_initial (float): The initial salinity.
- salinity_new (float): The current salinity for iteration.
- salinity_previous (float): The previous salinity for initialization.
- phi_initial (float): The initial liquid fraction.

- liquid_fraction_new (float): The current liquid fraction for iteration.
- phi_prev (float): The previous liquid fraction for initialization.

Return type
tuple

spyice.statevariables.**overwrite_statevariables**(*temperature_calculated*, *salinity_calculated*, *liquid_fraction_calculated*)

Overwrites the state variables with the calculated values. :param temperature_calculated: The calculated temperature. :type temperature_calculated: float :param salinity_calculated: The calculated salinity. :type salinity_calculated: float :param liquid_fraction_calculated: The calculated liquid fraction. :type liquid_fraction_calculated: float

Returns
A tuple containing the updated temperature, salinity, and liquid fraction.

Return type
tuple

spyice.statevariables.**reset_error_for_while_loop**(*temperature_tolerance*, *salinity_tolerance*, *liquid_fraction_tolerance*)

Resets the error values for a while loop based on the given tolerances.

Parameters

- **temperature_tolerance** (*float*) – The tolerance for temperature error.
- **salinity_tolerance** (*float*) – The tolerance for salinity error.
- **liquid_fraction_tolerance** (*float*) – The tolerance for liquid fraction error.

Returns
A tuple containing the reset error values for temperature, salinity, and liquid fraction.

Return type
tuple

spyice.statevariables.**set_statevariables**(*temperature_calculated*, *salinity_calculated*, *liquid_fraction_calculated*)

Set the state variables for temperature, salinity, and liquid fraction.

Parameters

- **temperature_calculated** (*float*) – The calculated temperature.
- **salinity_calculated** (*float*) – The calculated salinity.
- **liquid_fraction_calculated** (*float*) – The calculated liquid fraction.

Returns
A tuple containing the initial and previous values of temperature, salinity, and liquid fraction.

Return type
Tuple[float, float, float, float, float, float]

spyice.update_physical_values.**H_function**(*_self*, *_temperature*, *_enthalpy_s1*)

Calculates the value of H function.

Parameters

- **_self** (*float*) – The value of self.
- **_temperature** (*float*) – The temperature.
- **_enthalpy_s1** (*float*) – The enthalpy.

Returns
The calculated value of H function.

Return type
float

spyice.update_physical_values.**H_function_derivate**(*_x*, *_enthalpy_s1*)

Calculates the derivative of the H function.

Parameters

- **_x** (*float*) – The value of x.
- **_enthalpy_s1** (*float*) – The value of enthalpy_s1.

Returns
The derivative of the H function.

Return type
float

spyice.update_physical_values.**H_newton_iteration**(*_temperature*, *_enthalpy_s1*)

Performs Newton iteration to find the root of the H_function.

Parameters

- **_temperature** (*float*) – The temperature value.
- **_enthalpy_s1** (*float*) – The enthalpy value.

Returns
The root of the H_function.

Return type
float

Raises
None –

spyice.update_physical_values.**calculate_melting_temperature_from_salinity**(*_salinity*, *_temperature_melt=270.899*, *_liquid_relation='Norr*

Calculates the melting temperature of seawater based on salinity.

Parameters

- **_salinity** (*numpy.ndarray*) – Array of salinity values.
- **_temperature_melt** (*float, optional*) – Melting temperature. Defaults to _temperature_melt.
- **_liquid_relation** (*str, optional*) – Liquid relation type. Must be either "Normal" or "Frezchem". Defaults to "Normal".

Returns
Array of melting temperature values.

Return type
numpy.ndarray

Raises
TypeError – If _liquid_relation is not "Normal" or "Frezchem".

spyice.update_physical_values.**phi_control_for_infinite_values**(*_phi*)

Calculates the control values for infinite phi values.

Parameters
_phi (*numpy.ndarray*) – The input array of phi values.

Returns
The control values for the given phi values.

Return type
numpy.ndarray

spyice.update_physical_values.**phi_func**(*_enthalpy_k1*, *_enthalpy_s1*)

Calculates the phi value based on the given enthalpy values.

Parameters

- **_enthalpy_k1** (*float*) – The enthalpy value for k1.
- **_enthalpy_s1** (*float*) – The enthalpy value for s1.

Returns
The calculated phi value.

Return type
float

spyice.update_physical_values.**update_enthalpy**(*_temperature*, *_salinity*, *_liquid_fraction*, *_nz*, *_method='likebuffo'*)

Updates the enthalpy based on the given parameters.

Parameters

- **_temperature** (*float*) – The temperature value.
- **_salinity** (*float*) – The salinity value.
- **_liquid_fraction** (*float*) – The liquid fraction value.
- **_nz** (*int*) – The nz value.

- **_method** (*str, optional*) – The method used for calculating enthalpy. Defaults to “likebuffo”.

Returns
The updated enthalpy value.

Return type
float

Raises
TypeError – If the given method is not available.

spyice.update_physical_values.**update_enthalpy_solid_state**(*_salinity*, *_nz*, *_liq_rel='Normal'*, *_temperature_melt=270.8999285714286*)

Updates the enthalpy in the solid state based on the given parameters. :param _salinity: The salinity value. :type _salinity: float :param _nz: The nz value. :type _nz: int :param _liq_rel: The liquid relation. Defaults to “Normal”. :type _liq_rel: str, optional :param _temperature_melt: The melting temperature. Defaults to _temperature_melt. :type _temperature_melt: float, optional

Returns
The updated enthalpy in the solid state.

Return type
float

spyice.update_physical_values.**update_liquid_fraction**(*_temperature*, *_salinity*, *_liquid_fraction*, *_enthalpy*, *_enthalpy_solid*, *_nz*, *_is_stefan=False*, *_method='likebuffo'*)

Updates the liquid fraction based on temperature, salinity, enthalpy, and other parameters.

Parameters

- **_temperature** (*float*) – The temperature value.
- **_salinity** (*float*) – The salinity value.
- **_liquid_fraction** (*float*) – The liquid fraction value.
- **_enthalpy** (*float*) – The enthalpy value.
- **_enthalpy_solid** (*float*) – The solid enthalpy value.
- **_nz** (*int*) – The number of vertical grid points.
- **_is_stefan** (*bool, optional*) – Whether to use Stefan condition. Defaults to False.
- **_method** (*str, optional*) – The method to use. Defaults to “likebuffo”.

Returns
A tuple containing the updated liquid fraction and the temperature.

Return type
tuple

Raises
AssertionError – If the liquid fraction has a non-physical value.

3.1.3 Module contents

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

S