CelerisAi

Release 0.0.1

Willington Rentería

Feb 20, 2025

CONTENTS:

CelerisAi is Python-Taichi-based software for nearshore wave modeling. This solver offers high-performance simulations on various hardware platforms and seamlessly integrates with machine learning and artificial intelligence environments. The solver makes use of the flexibility of the Python language for customization and interoperability, while the Taichi implementation provides high-performance capabilities and facilitates integration into artificial intelligence environments such as PyTorch.

Check out the *Usage* section for further information,, including how to *install* the project.

Note

This project is under active development.

CHAPTER ONE

DOCUMENTATION FOR THE CODE

1.1 Introduction

1.1.1 CelerisAi: A Nearshore Wave Modeling Framework for Integrated AI Applications

CelerisAi is a Python and Taichi-based framework for nearshore wave modeling that seamlessly integrates with AI environments. It provides GPU and multi-CPU acceleration, flexible setup, and interactive visualization—enhancing coastal engineering applications such as depth inversion and complex nearshore simulations.

1.1.2 Methods

CelerisAi is built upon Celeris Base [TavakkolLynett2020], which solves the extended Boussinesq equations derived by [MadseSorensen1992] to account for nonlinear and dispersive effects in nearshore wave transformation, and follows the architecture presented in the CelerisWebGPU version (CelerisWebGPU).

1.1.3 Characteristics

The main characteristics of CelerisAi include:

- **Optimized Kernels:** Leverages custom Taichi-kernels for efficient computation.
- **OS Agnosticism:** Compatible across various operating systems.
- **High Performance:** Utilizes multiple CPUs, GPUs, or cloud services for accelerated simulations.
- **Modularity:** Structured around four main classes, facilitating easy expansion with additional physics or models.
- **AI Integration:** Incorporates PyTorch to send data directly from numerical models to neural networks without offloading from the GPU.
- **Visualization Options:** Supports both headless and interactive visualization.

1.1.4 Class Architecture and Workflow

The simulation setup can be performed directly within a Python script or through JSON files by specifying the domain geometry, initial conditions, and boundary conditions.

CelerisAi can solve the shallow water equations or the extended Boussinesq equations. In both cases, the scalars and vectors are the same, but the methods used to solve the numerical scheme are different.

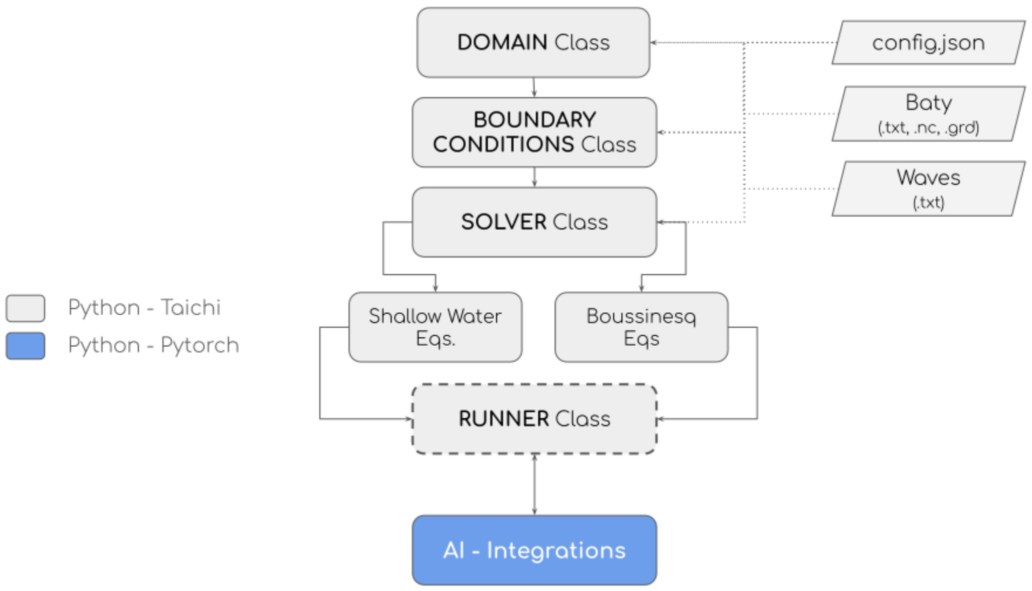

Fig. 1: CelerisAi classes and setup.

Workflow to solve the Shallow water equations

Workflow to solve the extended Boussinesq equations

1.1.5 Performance

Ongoing efforts focus on replicating benchmark cases to verify accuracy—such as the solitary wave run-up on a sloping beach.

At run-time, simulation states can be transferred to PyTorch tensors, enabling data-driven updates or neural network–based parameter updating in real-time. This approach reduces the necessity of offline data generation, facilitating a new paradigm of concurrent modeling learning, and allows for advanced applications like nearshore bathymetry inversion.

1.1.6 References

1.2 Usage

1.2.1 Installation

The prerequisites for CelerisAi are standard Python libraries, with Taichi being the most important. You can install these libraries manually:

```
$ pip install imageio>=2.36.0 matplotlib>=3.7.2 numpy>=1.24.3 scipy>=1.9.0 taichi>=1.7.0
```

Alternatively, you can install all required dependencies automatically using the provided requirements file:

```
$ pip install -r requirements.txt
```

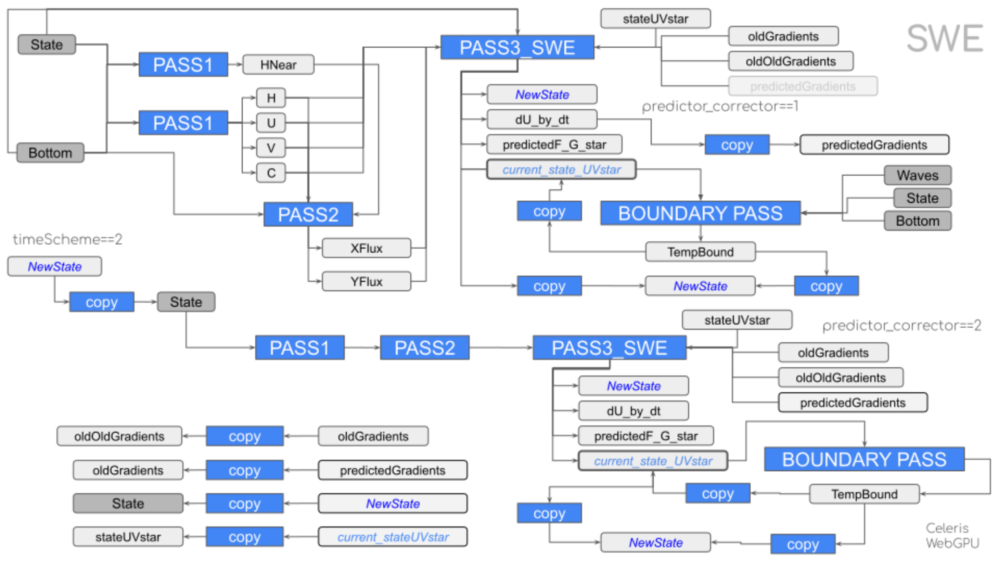

Fig. 2: SWE workflow

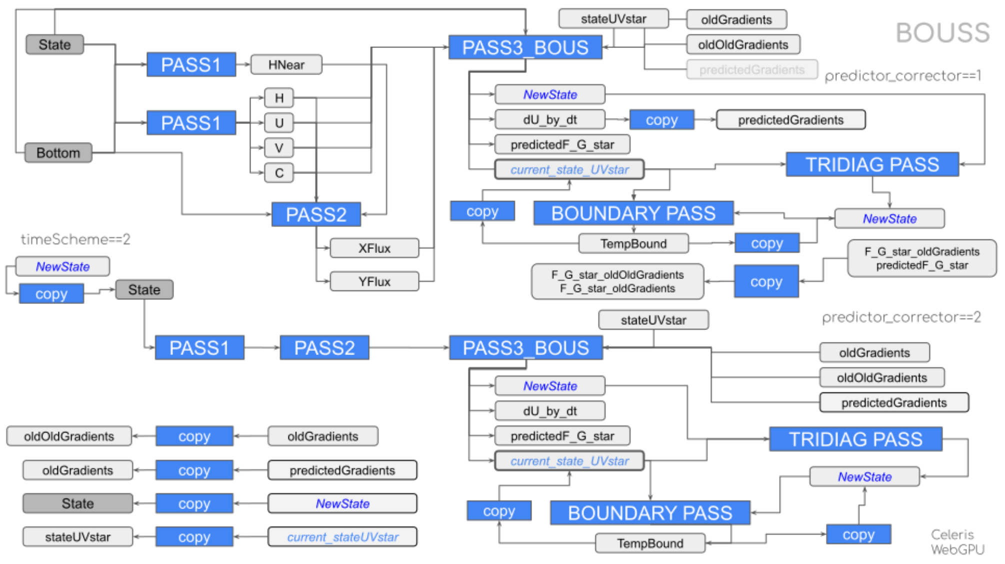

Fig. 3: Boussinesq workflow

1.2.2 Downloading the Source Code

Clone the repository from GitHub to download the source code:

```
$ git clone https://github.com/wrenteria/CelerisAi.git
```

1.2.3 Running the Examples

Note

Make sure to execute these examples from within the CelerisAi directory.

After installing the dependencies and downloading the source, you can verify the installation by running the provided examples.

For a 1D example, execute:

```
$ python setrun_1D.py
```

For a 2D example based on the configuration files created by CelerisWebGPU, execute:

```
$ python setrun_web.py
```

For more details on configuring CelerisWebGPU, please refer to its application at CelerisWebGPU.

1.3 Celeris Modules

1.3.1 Modules

Domain module

class `celeris.domain.`**`Topodata`**(*filename=None*, *datatype=None*, *path=None*)

Manages all topography/bathymetry data formats for CelerisAi.

This class handles different input formats for bathymetric or topographic data, such as 2D (XYZ), 1D (XZ), and Celeris-native formats. It reads the corresponding files (or folders) and loads them into a NumPy array for further processing or analysis.

`filename`

Name of the file containing topographic/bathymetric data. If *path* is provided, the file is assumed to be located there.

Type

str, optional

`datatype`

Format of the data. Accepted values are: - "xyz": 2D data in three columns (x, y, z) - "xz": 1D data in two columns (x, z) - "celeris": 2D data located in a folder with a file named "bathy.txt"

Type

str, optional

`path`

Directory path where the file is located. If not provided, *filename* should be a complete path or in the current working directory.

Type
str, optional

Example

```
>>> from celeris.domain import Topodata
>>> baty = Topodata(datatype='celeris',path='./examples/DuckFRF_NC')
```

`z()`

Loads the bathymetry/topography data based on the specified datatype.

Depending on the *datatype*, this method reads the corresponding file(s) and returns the data as a NumPy array.

- For “xyz”: Expects a file with three columns (x, y, z).
- For “xz”: Expects a file with two columns (x, z).
- For “celeris”: Expects a folder containing a file named “bathy.txt”. The returned array values are multiplied by -1.

Returns

- A NumPy array if *datatype* is recognized and the file is successfully read.
- The string “No supported format” if *datatype* is not recognized.

Return type
numpy.ndarray or str

Raises
`OSError` – If the file (or directory for “celeris”) cannot be found or read.

`class celeris.domain.`**`BoundaryConditions`**(*celeris=True*, *precision=ti.f32*, *North=10*, *South=10*, *East=10*, *West=10*, *WaveType=-1*, *Amplitude=0.5*, *path='./scratch'*, *filename='waves.txt'*, *BoundaryWidth=20*, *init_eta=5*, *sine_wave=None*)

Manages boundary conditions for a rectangular domain in the CelerisAi model.

The domain has four faces: north, south, east, and west. Each face can be configured with different boundary types (e.g., sponge layer, solid wall, incoming wave). Boundary conditions can be set in one of two ways:

1) **Celeris format** (*celeris=True*): Reads from a *config.json* file.
2) **Manual** (*celeris=False*): Uses manually supplied values.

If incoming waves (boundary type = 2) are defined, the wave type can be specified via *WaveType*. If *WaveType* is -1, the wave parameters are read from a file (e.g., *“waves.txt”*). Otherwise, a sine wave is assumed and set by *sine_wave*.

This class also includes utility methods for handling wave data input and conversion from NumPy arrays to Taichi tensors.

`precision`

The precision used by Taichi (e.g. *ti.f32*, *ti.f64*).

Type
taichi.types.primitive_types

`North`

Boundary type for the north face. Valid types are:

- 0: Solid wall
- 1: Sponge layer
- 2: Incoming wave

Type
int

`South`

Boundary type for the south face.

Type
int

`East`

Boundary type for the east face.

Type
int

`West`

Boundary type for the west face.

Type
int

`WaveType`

Wave type indicator. Valid values are:

- -1: Wave parameters read from a file (*waves.txt*)
- Any other integer: Use the *sine_wave* array for wave parameters

Type
int

`Amplitude`

Wave amplitude, used if not reading from a file.

Type
float

`path`

Path to the directory containing *waves.txt* and/or *config.json*.

Type
str

`filename`

Name of the file that stores wave parameters (e.g. *waves.txt*).

Type
str

BoundaryWidth

Width of the sponge or boundary zone.

Type
int

sine_wave

Parameters defining a sine wave if *WaveType* is not -1.

Type
list of float

celeris

If True, boundary conditions are read from *config.json*.

Type
bool

configfile

Loaded JSON configuration when *celeris=True*.

Type
dict, optional

data

Placeholder array for wave data (size depends on *N_data*).

Type
numpy.ndarray

N_data

Number of wave entries to read from file.

Type
int

W_data

Unused placeholder (could store wave data in some contexts).

Type
None

Example

```
>>> from celeris.domain import BoundaryConditions
>>> bc = BoundaryConditions(celeris=True, path='./examples/DuckFRF_NC',␣
↪precision=precision)
```

SineWave()

Provides the sine wave parameters as a NumPy array of the specified precision.

Returns
An array of length 4 containing the sine wave parameters.

Return type
numpy.ndarray

Sponge(*width=None*)

Returns the sponge (boundary) width to be used in the model.

Parameters
`width` (*int, optional*) – Overrides the class-level sponge width. Defaults to None.

Returns
The sponge width (either the class attribute or the passed-in argument).

Return type
int

`get_data()`

Ensures wave data is loaded if reading from a file, then returns a Taichi field.

If *WaveType* is -1, calls *load_data()* to populate *self.data* from the file. Converts the resulting NumPy array into a Taichi field and returns it.

Returns
A Taichi field of shape *(N_data, 4)* containing wave parameters.

Return type
ti.field

`load_data()`

Loads wave parameters from *waves.txt* if *WaveType* is -1.

If a valid file is found, it reads the wave parameters:

- The first three lines are header-like info (only the ‘NumberOfWaves’ line is used).
- After skipping three lines, wave parameters are loaded. The shape of *self.data* is *(N_data, 4)*, where *N_data* is determined by reading the ‘NumberOfWaves’ line in the file.

`tseries()`

Checks if a wave parameter file is specified.

Returns
True if *self.filename* is not None, indicating that time series wave data can be read from a file. Otherwise, False.

Return type
bool

`class celeris.domain.Domain`(*precision=ti.f32*, *x1=0.0*, *x2=0.0*, *y1=0.0*, *y2=0.0*, *Nx=1*, *Ny=1*, *topodata=None*, *north_sl=0.0*, *south_sl=0.0*, *east_sl=0.0*, *west_sl=0.0*, *Courant=0.2*, *isManning=0*, *friction=0.001*, *base_depth=None*, *BoundaryShift=4*)

Defines the numerical domain for wave propagation solver in CelerisAi.

This class sets up the domain geometry (x1, x2, y1, y2) and resolution (Nx, Ny), handles bathymetric/topographic data (via an instance of a *Topodata* class), configures boundary sea levels on each face (north, south, east, west), and stores critical parameters such as Courant number (*Courant*), friction, and base depth. Two main branches are supported for configuration:

1. **Celeris format**: Reads from a *config.json* file (if *topodata.datatype == “celeris”*).
2. **Manual**: Uses provided arguments directly.

The class also defines utility methods to:

- Create a meshgrid for the domain (*grid*).
- Load and interpolate topographic/bathymetric data (*topofield*, *bottom*).
- Compute maximum depth (*maxdepth*) and highest topography (*maxtopo*).

- Compute the time step (*dt*) based on Courant criteria.
- Provide reflection indices for solid boundary conditions (*reflect_x*, *reflect_y*).
- Create Taichi field templates for solver states (*states*, *states_one*).

`precision`

Taichi precision (e.g., *ti.f32*, *ti.f64*).

Type
ti.types.primitive_types

`x1`

Minimum x-coordinate of the domain.

Type
float

`x2`

Maximum x-coordinate of the domain.

Type
float

`y1`

Minimum y-coordinate of the domain.

Type
float

`y2`

Maximum y-coordinate of the domain.

Type
float

`Nx`

Number of grid cells in the x-direction.

Type
int

`Ny`

Number of grid cells in the y-direction.

Type
int

`topodata`

Instance that handles bathymetry/topography data.

Type
Topodata

`north_sl`

Sea level at the north boundary.

Type
float

`south_sl`
Sea level at the south boundary.

Type
float

`east_sl`
Sea level at the east boundary.

Type
float

`west_sl`
Sea level at the west boundary.

Type
float

`Courant`
Courant number for numerical stability.

Type
float

`isManning`
Switch indicating whether Manning friction is used.

Type
int

`friction`
Friction (e.g., Manning n) value.

Type
float

`base_depth_`
Reference base depth of the domain. If None, it is inferred from the topography.

Type
float or None

`Boundary_shift`
Shift parameter used for boundary indexing or conditions.

Type
int

`pixels`
Taichi field for potential 2D visualization or debugging (shape = [Nx, Ny]).

Type
ti.field

`g`
Gravitational constant (9.80665).

Type
float

configfile

Loaded JSON dictionary if using Celeris format.

Type
dict or None

seaLevel

Reference sea level (set to 0.0 here).

Type
float

Example

```
>>> from celeris.domain import Domain,Topodata
>>> topo = Topodata(filename='bathy.txt', datatype='xyz')
>>> domain = Domain(x1=0, x2=100, y1=0, y2=100, Nx=100, Ny=100, topodata=topo)
>>> xgrid, ygrid, bathy = domain.grid()
>>> dt = domain.dt()
>>> print(f"Time step: {dt}")
```

bottom()

Creates a 3D NumPy array of shape (4, Nx, Ny) to store bottom elevation (inverted sign), plus any other auxiliary fields (e.g., near-dry flags).

Index mapping:

- [2, :, :] => Stores the bathymetry/topography (with a -1 factor).
- [3, :, :] => A placeholder used for near-dry or similar state flags.

Returns
A Taichi field of shape [4, Nx, Ny] with the bottom information.

Return type
ti.field

dt()

Computes the time step based on the Courant criterion:

dt = Courant * dx / sqrt(g * maxdepth)

Returns
The computed time step.

Return type
float

grid()

Returns the meshgrid of domain coordinates and corresponding bathymetry/topography.

Returns

- **xx** (numpy.ndarray): x-coordinates (shape: Nx x Ny if 2D; Nx if 1D).
- **yy** (numpy.ndarray): y-coordinates (shape: Nx x Ny if 2D; or topography for 1D).
- **zz** (numpy.ndarray): Bathymetry/topography data in 2D case; None or irrelevant in 1D case (depending on interpretation).

Return type
tuple

`maxdepth()`

Computes the maximum depth in the domain. If `base_depth_` is specified, returns that. Otherwise:

- For 1D ('xz'): returns the maximum of the interpolated topofield array.
- For 2D ('xyz', 'celeris'): returns the maximum of topofield array values.

Returns
Maximum depth (`base_depth_` if set, else maximum from topofield).

Return type
float

`maxtopo()`

Computes the highest topographic elevation in the domain. For 1D ('xz'), returns the minimum of the array (assuming negative values represent depth). For 2D, returns the minimum of the topofield array for a similar reason.

Returns
The highest elevation (or least negative) in the domain.

Return type
float

`reflect_x()`

Computes an x-reflection index for enforcing solid boundary conditions.

Returns
The x-reflection index (2*(Nx-3)).

Return type
int

`reflect_y()`

Computes a y-reflection index for enforcing solid boundary conditions.

Returns
The y-reflection index (2*(Ny-3)).

Return type
int

`states()`

Creates a Taichi Vector field of shape [Nx, Ny], each containing 4 components (e.g., water depth, momentum in x, momentum in y, and an scalaritional parameter).

Returns
A 4-component vector field in Taichi.

Return type
ti.types.vector.field

`states_one()`

Creates a Taichi Vector field of shape [Nx, Ny], each containing 1 component.

Returns
A 1-component vector field in Taichi.

Return type
ti.types.vector.field

`topofield()`

Loads and/or interpolates topographic/bathymetric data into a NumPy meshgrid and returns it in a format suitable for use in the solver.

Returns

- **x_out** (numpy.ndarray): Mesh of x-coordinates for the domain.
- **y_out** (numpy.ndarray): Mesh of y-coordinates for the domain (or 1D array if *datatype='xz'*).
- **z_out** (numpy.ndarray): Corresponding bathymetry/topography values.

Return type
tuple

Raises
`ValueError` – If *topodata.datatype* is not one of 'celeris', 'xyz', or 'xz'.

`celeris.domain.checjson`(*variable*, *data*)

Checks if a key exists in a JSON config file (i.e., CelerisWebGPU configuration file).

Parameters

- `key` (*str*) – The key to check in the JSON config file.
- `config` (*dict*) – The JSON-loaded dictionary.

Returns
1 if the key is found in the JSON dictionary, 0 otherwise.

Return type
int

`celeris.domain.ti2np`(*ti_type*)

Converts a Taichi precision type to a NumPy dtype.

Parameters
`precision` (*ti.types.primitive_types*) – Taichi precision type (e.g., ti.f32, ti.f64).

Returns
Corresponding NumPy dtype (e.g., np.float32, np.float64).

Return type
numpy.dtype

Solver module

`class celeris.solver.Solver`(*domain=None*, *boundary_conditions=None*, *dissipation_threshold=0.3*, *theta=2.0*, *timeScheme=2*, *pred_or_corrector=1*, *show_window=True*, *maxsteps=1000*, *Bcoef=0.06666666666666667*, *outdir=None*, *model='SWE'*, *useBreakingModel=False*, *whiteWaterDecayRate=0.01*, *whiteWaterDispersion=0.1*, *useSedTransModel=False*, *sediment=<celeris.solver.SedClass object>*, *infiltrationRate=0.001*, *clearCon=1*, *showBreaking=0*, *delta_breaking=2.0*, *T_star_coef=5.0*, *dzdt_I_coef=0.5*, *dzdt_F_coef=0.15*)

Main numerical solver class for the CelerisAi model.

This class manages the entire simulation process, including:

- Initializing solution states and bottom fields from the *Domain* class.
- Controlling boundary conditions from the *BoundaryConditions* class.
- Executing the time-stepping scheme (Euler, Adams-Bashforth variants) and 1D/2D flow updates (SWE or Boussinesq).
- Incorporating breaking models, sediment transport, and the kernels for reconstruction (Pass1), flux computations (Pass2), and final updates (Pass3).

`domain`

Instance of the Domain class containing grid info and topography.

Type
Domain

`bc`

Instance managing boundary condition types and wave settings.

Type
BoundaryConditions

`dissipation_threshold`

Used for visualization (mark cells above certain foam/dissipation).

Type
float

`theta`

Midmod limiter parameter (1.0 more dissipative, 2.0 less dissipative).

Type
float

`timeScheme`

Time integration scheme: - 0 => Euler - 1 => 3rd-order predictor - 2 => 4th-order predictor/corrector

Type
int

`pred_or_corrector`

Indicates stage in solver loop (1 => predictor, 2 => corrector).

Type
int

`Bcoef`

Dispersion parameter for Boussinesq model; default 1/15.

Type
float

`model`

Type of model, ‘SWE’ or ‘Bouss’.

Type
str

`useBreakingModel`

True if wave-breaking model is included.

Type
bool

`whiteWaterDecayRate`

Turbulence decay rate for foam (visualization).

Type
float

`whiteWaterDispersion`

Turbulence dispersion factor.

Type
float

`useSedTransModel`

True if sediment transport is included.

Type
bool

`sediment`

Default or custom sediment parameters.

Type
object

`infiltrationRate`

For modeling infiltration on dry beaches.

Type
float

`clearCon`

If 1, concentration channel is cleared for visualization.

Type
int

`showBreaking`

If > 0, shows wave breaking foam areas.

Type
int

`delta_breaking`

Eddy viscosity coefficient in breaking zones.

Type
float

`T_star_coef`

Timescale factor for fully developed breaking.

Type
float

`dzdt_I_coef`

Start-breaking parameter threshold.

Type
float

`dzdt_F_coef`

End-breaking parameter threshold.

Type
float

The class initializes a large set of Taichi vector fields (state vectors, flux arrays, intermediate arrays for Boussinesq tridiagonal solves, sediment transport arrays, etc.) to manage the numerical solution.

Example

```
>>> solver = Solver(domain=dom, boundary_conditions=bc, model='SWE')
```

`BoundSineWaves`(*NumWaves*, *Waves*, *x*, *y*, *t*, *d_here*, *grav*)

Computes boundary conditions for incoming sine waves at a domain boundary.

Parameters

- **NumWaves** (*int*) – Number of wave components in *Waves*.
- **Waves** (*ti.field*) – Wave parameter array [numWaves, 4].
- **x** (*float*) – x-coordinate at boundary cell.
- **y** (*float*) – y-coordinate at boundary cell.
- **t** (*float*) – Current time.
- **d_here** (*float*) – Local water depth if positive; 0 if dry.
- **grav** (*float*) – Gravity constant.

Returns
[eta, hu, hv] aggregated from all wave components.

Return type
ti.types.vector(3, float)

`BoundaryPass`(*time: ti.f32*, *txState: DummyTemplate*)

Updates boundary cells with the appropriate boundary conditions:

- Sponge layers (type=1)
- Solid walls (type=0)
- Incoming waves (type=2) including sine wave or solitary wave

This kernel is also responsible for handling near-dry logic and simple wetting/drying checks at the domain edges.

Parameters

- **time** (*float*) – Current time.
- **txState** (*ti.field*) – A Taichi 2D vector field for the state to be updated.

`InitStates`()

Initializes the solver states (State, stateUVstar) to zeros at the start of the simulation.

`Pass1`()

Reconstruction step (Pass1):

- Builds left/right (or N/E/S/W) interface values of eta, momentum, and scalar concentration using a generalized minmod limiter.
- Applies near-dry checks to skip processing cells that are effectively dry.
- Computes velocity components and partial Froude-limiter logic.

For 1D (ny=1), a simpler logic is used. For 2D, reconstruction is in both x- and y-directions.

`Pass1_SedTrans()`

Reconstruction step (Pass1) for sediment transport scalar. Uses the same generalized minmod approach to reconstruct sediment concentration at edges.

`Pass2()`

Flux computation step (Pass2):

- Computes fluxes at each cell edge in x and y directions using the function numerical flux.
- If sediment transport is enabled, calculates sediment fluxes similarly.

`Pass3`(*pred_or_corrector: ti.f32*)

Main time-update step (Pass3) for the SWE model:

- Updates eta, hu, hv, c for each cell based on fluxes and source terms (bed slope, friction, infiltration, etc.).
- Uses an explicit time scheme (Euler, predictor, predictor/corrector).
- Optionally includes a foam/breaking parameter if showBreaking>0.

Parameters
`pred_or_corrector` (*`int`*) – Stage in predictor-corrector scheme (1 => predictor, 2 => corrector).

`Pass3Bous`(*pred_or_corrector: ti.f32*)

Time-update step (Pass3) for the Boussinesq model:

- Updates wave height, hu, hv, concentration.
- Incorporates dispersion terms, bed slope, friction, infiltration, and wave breaking if enabled.
- Uses a predictor-corrector approach for higher-order accuracy.

Parameters
`pred_or_corrector` (*`int`*) – Stage in predictor-corrector scheme (1 => predictor, 2 => corrector).

`Pass3_SedTrans`(*pred_or_corrector: ti.f32*)

Time-update step (Pass3) for sediment transport scalar:

- Uses fluxes (XFlux_Sed, YFlux_Sed) plus simple diffusion.
- Adds erosion/deposition terms based on local shear velocity or critical Shields.

Parameters
`pred_or_corrector` (*`int`*) – Stage in predictor-corrector scheme (1 => predictor, 2 => corrector).

`Pass_Breaking`(*time: ti.f32*)

Wave-breaking model step (used if useBreakingModel == True):

- Applies Kennedy et al. or similar wave breaking logic to compute local dissipation flux and update the Breaking field.
- Incorporates eddy viscosity effects from breaking.

Parameters
`time` (*float*) – Current simulation time.

`Run_Tridiag_solver`()

Executes the parallel cyclic reduction (PCR) solver to handle the dispersion terms in the Boussinesq model. If model is 'SWE', no action is taken.

For 1D (ny=1), only the x-direction solver is applied. For 2D, runs PCR in x, then y directions.

`SolitaryWave`(*x0*, *y0*, *theta*, *x*, *y*, *t*, *d_here*)

Computes boundary conditions for a solitary wave type (WaveType=3).

Parameters

- **x0** (*float*) – Initial x-position of the wave crest.
- **y0** (*float*) – Initial y-position of the wave crest.
- **theta** (*float*) – Wave propagation angle in radians.
- **x** (*float*) – x-coordinate at boundary cell.
- **y** (*float*) – y-coordinate at boundary cell.
- **t** (*float*) – Current simulation time.
- **d_here** (*float*) – Local water depth, if any.

Returns
(eta, hu, hv) for a solitary wave boundary.

Return type
tuple of (float, float, float)

`TriDiag_PCRx`(*p: int*, *s: int*, *current_buffer: DummyTemplate*, *next_buffer: DummyTemplate*)

Parallel Cyclic Reduction step in x-direction for tridiagonal system (Boussinesq dispersion). This kernel performs the p-th level of reduction.

Parameters

- **p** (*int*) – Current level in PCR (log2 scale).
- **s** (*int*) – Step size (2^p).
- **current_buffer** (*ti.field*) – Current coefficients at p-1 level.
- **next_buffer** (*ti.field*) – Next coefficients to be filled for p-th level.

`TriDiag_PCRy`(*p: int*, *s: int*, *current_buffer: DummyTemplate*, *next_buffer: DummyTemplate*)

Parallel Cyclic Reduction step in y-direction for tridiagonal system (Boussinesq dispersion).

Parameters

- **p** (*int*) – Current level in PCR (log2 scale).
- **s** (*int*) – Step size (2^p).

- **current_buffer** (*ti.field*) – Current coefficients at p-1 level.
- **next_buffer** (*ti.field*) – Next coefficients for p-th level.

copy_states(*src: DummyTemplate*, *dst: DummyTemplate*)

Copies state data from one Taichi field to another, ensuring shapes match.

Parameters

- **src** (*ti.field*) – Source Taichi field.
- **dst** (*ti.field*) – Destination Taichi field.

Raises

AssertionError – If shapes of src and dst do not match.

fill_bottom_field()

Fills the bottom vector field array with computed spatial derivatives (indices 0,1) and near-dry/auxiliary flags at index 3.

tridiag_coeffs_X()

Fills tridiagonal coefficient arrays in x-direction (coefMatx) for the Boussinesq model. These coefficients are used later in the parallel cyclic reduction solver (PCR) to handle dispersion terms.

tridiag_coeffs_Y()

Fills tridiagonal coefficient arrays in y-direction (coefMaty) for the Boussinesq model. These coefficients are used later in the parallel cyclic reduction solver (PCR) to handle dispersion terms.

class celeris.solver.**SedClass**(*d50=0.004*, *p=0.4*, *psi=0.0005*, *CriticalShields=0.045*, *rhorat=2.65*)

Runner module

class celeris.runner.**Evolve**(*domain=None*, *boundary_conditions=None*, *solver=None*, *maxsteps=1000*, *outdir=None*, *saveimg=False*, *vmin=-1.5*, *vmax=1.5*)

Controls and runs the main simulation loop of CelerisAi in various modes (headless, 1D, 2D with visualization, etc.).

This class ties together the *Domain*, *BoundaryConditions*, and *Solver* classes, and manages the time-stepping workflow. It includes methods for:

1. **Initialization** (*Evolve_0*):
   - Fills the bottom field with bathymetry/topography data.
   - Initializes solver states (water height, velocity, etc.).
   - Computes tridiagonal coefficients if using a Boussinesq model.
2. **Main Time-Stepping** (*Evolve_Steps*):
   - Runs reconstruction (Pass1) and flux computations (Pass2).
   - Handles wave breaking if enabled.
   - Integrates the solution one or more steps forward in time (Pass3, Pass3Bous, Pass3_SedTrans).
   - Updates boundary conditions.
   - Optionally solves tridiagonal systems for Boussinesq dispersion.
   - Copies or shifts data between fields for multi-stage time integrators.
3. **Headless Execution** (*Evolve_Headless*):

- Executes the simulation loop without rendering or displaying results, minimizing overhead and focusing on performance.
- Periodically logs timing information and can save the simulation states (e.g. *State* arrays).

4. **1D Visualization** (*Evolve_1D_Display*):
   - Specialized loop for 1D simulations, displaying free surface (eta) and bathymetry in a window using either taichi-gui or legacy GUI fallback.
5. **2D Visualization** (*Evolve_Display*):
   - Interactive loop for 2D simulations.
   - Renders wave height (h), free surface elevation (eta), or vorticity (vor) in real-time.
   - Allows saving images and assembling them into a GIF.
6. **Rendering and Color Mapping** (Kernels like *paint*, *paint_new*, *painting_h*, *painting_eta*, *painting_vor*, etc.):
   - Populates 2D Taichi fields (*self.image*, *self.solver.pixel*, etc.) based on solver results, for real-time visualization.
   - Supports multiple coloring strategies (e.g. realistic wave colors, topography shading, sediment rendering).

Parameters

- **`domain`** (`Domain`) – The domain class containing spatial parameters.
- **`boundary_conditions`** (`BoundaryConditions`) – Class managing boundary setup (walls, waves, etc.).
- **`solver`** (`Solver`) – The main numerical solver class controlling the fluid model, time scheme, etc.
- **`maxsteps`** (*`int, optional`*) – Maximum number of time steps to simulate. Defaults to 1000.
- **`outdir`** (*`str, optional`*) – Output directory path for saving states, frames, etc. Defaults to None.
- **`saveimg`** (*`bool, optional`*) – If True, saves image frames at intervals (for creating GIFs or offline processing). Defaults to False.
- **`vmin`** (*`float, optional`*) – Minimum value for visualization color scaling (e.g. wave elevation). Defaults to -1.5.
- **`vmax`** (*`float, optional`*) – Maximum value for visualization color scaling. Defaults to 1.5.

`solver`

The numerical solver controlling fluid/morphodynamics.

Type

Solver

`maxsteps`

Number of time steps for the simulation run.

Type

int

`dt`

Time step size imported from the solver.

Type
float

`timeScheme`

Time integration scheme (Euler, predictor, predictor-corrector).

Type
int

`saveimg`

Flag indicating whether to save frames.

Type
bool

`vmin`

Minimum scale for color mapping wave or vorticity values.

Type
float

`vmax`

Maximum scale for color mapping wave or vorticity values.

Type
float

`outdir`

Directory for saving outputs.

Type
str

`image`

2D field to hold RGB color information for visualization.

Type
ti.Vector.field

`ocean`

1D array of color samples (RGB) for water visualization or general colormap usage.

Type
ti.Vector.field

`colormap_ocean`

A string identifier for the colormap used for water.

Type
str

`bottom1D, indexbottom1D, eta1D`

Fields used in 1D visualization of bottom topography and water surface.

`x_scale, y_scale`

Scaling factors for 1D plots in the GUI.

Type
float

Typical Usage:

```
>>> evolve = Evolve(domain=dom, boundary_conditions=bc, solver=sol, maxsteps=2000,␣
↪outdir="results")
>>> evolve.Evolve_Display(vmin=-1.0, vmax=1.0, variable='eta', cmapWater='Blues_r',␣
↪showSediment=True)
```

Note

- The class attempts to use Taichi's GGUI if available for improved performance and better UI control. If GGUI is not available, it falls back to legacy Taichi GUI.
- Various coloring kernels (*painting_h*, *painting_eta*, etc.) can be customized to match user-defined styles or to highlight specific flow features.

`Evolve_0``()`

One-time initialization steps:

- Fills bottom field (bathymetry/topo).
- Initializes solver states (fluid variables).
- Computes tridiagonal coefficients if the model is Boussinesq.
- Prints simulation parameters (model type, time step, etc.).

`Evolve_1D_Display``()`

Interactive loop for a 1D simulation of CelerisAi.

- Initializes bottom, runs the main solver steps, and displays the results in a small taichi-gui or GGUI window.
- Plots the free surface (eta) and bottom profile in each iteration.
- Optionally saves frames and can compile them into a GIF if desired.

`Evolve_Display``(`*vmin=None*, *vmax=None*, *variable='h'*, *cmapWater='Blues_r'*, *showSediment=False*`)`

Interactive loop for a 2D simulation of CelerisAi with real-time visualization.

- Calls *Evolve_0()* once to initialize solver fields.
- Creates a window (either GGUI or legacy GUI).
- Uses a custom colormap from *celeris_matplotlib()* if *showSediment* is True and the solver has sediment transport enabled.
- **Allows switching between different visualization variables:**
  - *h*: Water depth
  - *eta*: Free surface elevation
  - *vor*: Vorticity
- Saves frames if *saveimg* is True, can compile them into a GIF, and optionally saves solver states to *.npy*.

Parameters

- **vmin** (*float, optional*) – Minimum colormap value for rendering. Defaults to None (class-level vmin).

- **vmax** (*float, optional*) – Maximum colormap value for rendering. Defaults to None (class-level vmax).
- **variable** (*str, optional*) – Which variable to render (*h*, *eta*, or *vor*). Defaults to 'h'.
- **cmapWater** (*str, optional*) – Matplotlib colormap name for water. Defaults to 'Blues_r'.
- **showSediment** (*bool, optional*) – If True, merges in a sediment colormap when sediment transport is active. Defaults to False.

Evolve_Headless()

Runs CelerisAi without any visualization, printing timing info periodically and optionally saving states to disk.

Steps:

1. Calls Evolve_0() to initialize fields and solver state.
2. Loops over *maxsteps*, calling Evolve_Steps() each iteration.
3. Logs simulation time and performance metrics every 100 steps.
4. If an output directory is specified, saves solver state arrays to .npy files.

Evolve_Steps(*step=0*)

Advances the solution by one time step (or one sub-step) according to the selected time scheme.

Internally calls:

- Pass1 (and Pass1_SedTrans if sediment is enabled)
- Pass2
- Optional wave breaking model
- Pass3 or Pass3Bous for predictor step
- BoundaryPass to enforce boundary conditions
- Run_Tridiag_solver for Boussinesq models
- (If 4th-order predictor-corrector) a second cycle of Pass1, Pass2, Pass3 or Pass3Bous
- Copies or shifts old/predicted states for multi-stage time integrators

InitColors(*arr: ndarray(dtype=ti.f32, ndim=2)*)

Copies an external NumPy array of shape (N, 3) (RGB colors) into the internal taichi field *self.ocean*.

Parameters
arr (*np.ndarray*) – (N, 3) array of float16 color data (e.g., from a Matplotlib colormap).

bottom_paint()

Fills *bottom1D* with scaled bottom data for 1D visualization. Also populates *indexbottom1D* so that line plotting can connect them in order.

brk_color(*x*, *y0*, *y1*, *x0*, *x1*)

Interpolates between two values (y0, y1) based on x in [x0, x1].

Used to smoothly vary color or other scalar values between two extremes:
(x0 -> y0) to (x1 -> y1).

eta_paint()

Fills *eta1D* with scaled free-surface data for 1D visualization.

paint()

General paint kernel that merges bottom topography and free-surface height.

- Assigns color based on bottom topography if there's no water.
- Interpolates water color if flow depth > 0.0001.
- If sediment transport is enabled, merges sediment concentration into final color.

paint_new()

A simple rendering kernel mixing bottom topography and wave height. Uses a linear interpolation (brk_color) to assign colors to each cell based on water depth or topography.

painting_eta()

Kernel for visualizing free surface elevation (eta).

- Normalizes eta to [vmin, vmax] for color lookup.
- Distinguishes water areas from wet sand and land similar to *painting_h*.

painting_h()

Kernel for visualizing water depth (h) in a "realistic wave" style.

- Normalizes water depth to [0, base_depth].
- Chooses colors from the *ocean* array based on the normalized depth (linear interpolation).
- Make a difference between the water areas (flow > 0.25) from shallow/wet sand and land.

painting_vor()

Kernel for visualizing vorticity (vor).

- Approximates vorticity by differences in velocity between adjacent cells.
- Normalizes the result into [vmin, vmax] for color lookups in *ocean*.
- Distinguishes water vs. land similar to *painting_h*/*painting_eta*.

Utils module

celeris.utils.**CalcUV**(*h*, *hu*, *hv*, *hc*, *epsilon*, *dB_max*)

Computes velocity and scalar concentration at cell edges given water height and momentum.

This function takes the water depth (h), x-momentum (hu), y-momentum (hv), and a scalar quantity (hc) at the cell edges (usually indexed [N, E, S, W]) and returns the velocity components (u, v) and scalar concentration (c) at those edges. It applies a limiting factor to avoid division by a near-zero depth.

The key step involves computing:
divide_by_h = 2.0 * h / (h * h + ti.max(h * h, epsilon_c))

where:

- *epsilon_c = max(epsilon, dB_max)*
- *epsilon* is a small threshold to prevent division by zero,
- *dB_max* represents the maximum bed-elevation difference across edges, ensuring the local depth used for velocity calculation is not less than the difference in water depth across an edge.

Parameters

- **h** (*ti.types.vector*) – Water depth at edges, shaped [N, E, S, W].
- **hu** (*ti.types.vector*) – Momentum in the x-direction at edges.

- **hv** (*ti.types.vector*) – Momentum in the y-direction at edges.
- **hc** (*ti.types.vector*) – Scalar quantity (e.g. concentration) at edges.
- **epsilon** (*float*) – Small threshold to avoid division by zero in near-dry cells.
- **dB_max** (*ti.types.vector or float*) – Maximum bed-elevation difference used to limit depth.

Returns

(u, v, c) where each is a vector shaped [N, E, S, W].

- **u**: x-velocity at edges.
- **v**: y-velocity at edges.
- **c**: Scalar concentration at edges.

Return type

tuple of ti.types.vector

`celeris.utils.`**`CalcUV_Sed`**`(`*h*, *hc1*, *hc2*, *hc3*, *hc4*, *epsilon*, *dB_max*`)`

Computes sediment scalar concentrations at cell edges given the water height.

This function calculates four sediment-related quantities (e.g., scalar concentrations or sediment fractions) at the edges of a cell. It applies a limiting factor based on water depth (*h*) to avoid division by a near-zero depth and to account for significant bed-elevation differences.

Parameters

- **h** (*float*) – Water depth at the cell edge.
- **hc1** (*float*) – Sediment/scalar quantity #1 at the edge.
- **hc2** (*float*) – Sediment/scalar quantity #2 at the edge.
- **hc3** (*float*) – Sediment/scalar quantity #3 at the edge.
- **hc4** (*float*) – Sediment/scalar quantity #4 at the edge.
- **epsilon** (*float*) – Small threshold to avoid division by zero.
- **dB_max** (*float*) – Maximum bed-elevation difference, used in limiting depth.

Returns

A 4-component vector containing the scaled sediment/scalar values *[c1, c2, c3, c4]* after applying the depth-limiting factor.

Return type

ti.Vector([float, float, float, float])

`celeris.utils.`**`ColorsfromMPL`**`(`*cmap='Blues'*`)`

Extracts a small set of color values from a Matplotlib colormap.

This function accesses any Matplotlib colormap by name, samples 16 color entries from it, and returns them as a NumPy array of type float16. Users can choose any of the built-in Matplotlib colormaps (e.g., “viridis”, “jet”, “inferno”, “Blues”, etc.).

Parameters

cmap (*str, optional*) – Name of the Matplotlib colormap to sample. Defaults to “Blues”.

Returns

A (16, 4) array of RGBA color values (float16).

The array index corresponds to a discrete point along the specified colormap.

Return type
numpy.ndarray

`celeris.utils.`**`FrictionCalc`**(*hu*, *hv*, *h*, *base_depth*, *delta*, *isManning*, *g*, *friction*)

Computes a bottom friction term for shallow-water or Boussinesq-type flows.

This function calculates a friction coefficient based on either a constant friction parameter (*friction*) or Manning's formula (if *isManning == 1*), and then applies it to the momentum components. The water depth *h* is scaled by the *base_depth* for numerical stability and to avoid singularities near dry cells.

The steps are:

1. Scale the water depth (*h_scaled = h / base_depth*) and compute powers (*h2 = h_scaled*^2, *h4 = h2*^2).
2. Compute a term *divide_by_h2* which further scales friction based on squared depth.
3. Ensure the local depth *h* is not below a small threshold *delta* to prevent division by zero.
4. If *isManning == 1*, convert the *friction* input into Mannings n and compute:

   f = g * (friction^2) * (1 / h^(1/3))

   otherwise, keep a constant friction value.
5. Clamp the friction factor to a maximum of 0.5 as a safety measure.
6. Multiply by the flow speed (computed from *hu*, *hv*) and the scaling factor *divide_by_h2*.

Parameters

- **hu** (*float*) – Momentum in the x-direction.
- **hv** (*float*) – Momentum in the y-direction.
- **h** (*float*) – Local water depth.
- **base_depth** (*float*) – Reference (base) depth for scaling.
- **delta** (*float*) – Minimum threshold for water depth to avoid division by zero.
- **isManning** (*int*) – If 1, uses Mannings n formulation; otherwise uses a constant friction.
- **g** (*float*) – Gravitational acceleration.
- **friction** (*float*) – Either a constant friction coefficient or Mannings n value depending on *isManning*.

Returns
Computed friction term, capped at 0.5, that will be applied to momentum.

Return type
float

`celeris.utils.`**`MinMod`**(*a*, *b*, *c*)

Computes a simple minmod function of three values.

The minmod function returns:

- The minimum among (a, b, c) if all three are positive.
- The maximum among (a, b, c) if all three are negative.
- Zero otherwise.

Parameters

- **a** (*float*) – First value.
- **b** (*float*) – Second value.
- **c** (*float*) – Third value.

Returns
The minmod result based on the sign of the inputs.

Return type
float

`celeris.utils.`**`NumericalFlux`**`(`*aplus*, *aminus*, *Fplus*, *Fminus*, *Udifference*`)`

Computes a wave-speed-based numerical flux between two adjacent cells.

This function calculates the flux across a cell interface using the wave speeds *aplus* (maximum positive speed) and *aminus* (maximum negative speed) along with flux values from the "plus" and "minus" sides (*Fplus*, *Fminus*) and the state difference (*Udifference*). If the wave speeds cancel each other out (*aplus - aminus == 0.0*), the flux is set to zero.

The formula implemented is:

flux = (aplus * Fminus - aminus * Fplus + (aplus * aminus) * Udifference) / (aplus - aminus)

Parameters

- **aplus** (*float*) – Maximum positive wave speed at the cell interface.
- **aminus** (*float*) – Maximum negative wave speed at the cell interface.
- **Fplus** (*float*) – Flux contribution from the "plus" (right) side.
- **Fminus** (*float*) – Flux contribution from the "minus" (left) side.
- **Udifference** (*float*) – Difference in the conserved variable across the interface (e.g., U_right - U_left).

Returns
The computed numerical flux. Returns 0.0 if *(aplus - aminus) == 0.0*.

Return type
float

`celeris.utils.`**`Reconstruct`**`(`*west*, *here*, *east*, *TWO_THETAc*`)`

Performs a piecewise linear reconstruction of a variable using a generalized minmod limiter.

This function takes three consecutive cell-centered values (*west*, *here*, and *east*) along with a limiter parameter (*TWO_THETAc*) and returns two reconstructed interface values at the current cell interfaces (left/right or west/east edges).

The reconstruction logic:

- Computes slopes (z1, z2, z3) that scale differences between neighboring cells.
- Finds the minimum among those slopes (when all have the same sign) or zero otherwise.
- Applies a factor of 0.25 to that minimum slope to limit the reconstruction (i.e., controlling oscillations).
- Returns the reconstructed values at the left (west) and right (east) edges of the current cell.

Parameters

- **west** (*float*) – Value of the variable at the cell immediately to the left (j-1).

- **here** (*float*) – Value of the variable at the current cell (j).
- **east** (*float*) – Value of the variable at the cell immediately to the right (j+1).
- **TWO_THETAc** (*float*) – Limiter parameter, typically 2 * theta, where theta is in range [1, 2] for generalized minmod-type limiters.

Returns
A 2-component vector representing the reconstructed value at: - [0]: The left (west) interface of the current cell. - [1]: The right (east) interface of the current cell.

Return type
ti.types.vector(2, float)

celeris.utils.**ScalarAntiDissipation**(*uplus*, *uminus*, *aplus*, *aminus*, *epsilon*)

Computes an anti-dissipation factor based on local wave speeds and state magnitudes.

This function calculates a dimensionless ratio *R* that adjusts numerical dissipation in a flux-based scheme. The ratio depends on the maximum wave speed (*aplus* or *aminus*) and the magnitudes of the state variables *uplus* and *uminus*. If both wave speeds are non-zero, a local "Froude-like" number is formed by dividing the larger magnitude of *uplus* or *uminus* by the respective wave speed. This number is then augmented by a small threshold *epsilon* to yield a final ratio between 0 and 1. If either wave speed is zero, the ratio is set to *epsilon*.

Parameters

- **uplus** (*float*) – The "plus" state or velocity component.
- **uminus** (*float*) – The "minus" state or velocity component.
- **aplus** (*float*) – Positive wave speed at the cell interface.
- **aminus** (*float*) – Negative wave speed at the cell interface.
- **epsilon** (*float*) – Small threshold to prevent division by zero or extreme values.

Returns
Anti-dissipation ratio *R*. A value near 1 indicates lower numerical dissipation, while a value near 0 increases dissipation. Defaults to 0 if neither condition applies.

Return type
float

celeris.utils.**celeris_matplotlib**(*water='seismic'*, *land='terrain'*, *sediment='default'*, *SedTrans=False*)

Creates a customized matplotlib color map for visualizing water, land/topography, and optionally sediment transport.

This function merges three main color segments:

1. **Water**: Ranges from 0 to 0.75 or 0 to 0.5 (depending on sediment usage).
2. **Sediment** (optional): Placed between the water and land segments if *SedTrans* is True.
3. **Land/Topography: Assigned to the higher range of the color bar (e.g., 0.75 - 1 or 0.75 - 1**
   when *SedTrans* is False, and 0.75 - 1 when *SedTrans* is True).

Parameters

- **water** (*str, optional*) – Name of the colormap to use for water (default "seismic").
- **land** (*str, optional*) – Name of the colormap to use for land/topo (default "terrain").
- **sediment** (*str, optional*) – Name of the colormap to use for sediment. If "default", a hard-coded set of color stops (skyblue, tan, peru, saddlebrown) is used. Otherwise, a user-specified colormap is merged (default "default").

- **SedTrans** (*bool, optional*) – Indicates whether sediment transport is active. If True, the colormap includes an additional segment for sediment; if False, only water and land segments are used (default False).

Returns
A single merged colormap with the specified segments for water, (optionally) sediment, and land.

Return type
matplotlib.colors.LinearSegmentedColormap

Example

```
>>> cmap_no_sed = celeris_matplotlib(SedTrans=False)
>>> cmap_sed = celeris_matplotlib(water="Blues", sediment="Reds", SedTrans=True)
```

celeris.utils.**celeris_waves**()

Creates a custom color map (colormap) designed to represent realistic sea water gradients.

This function defines a series of color stops spanning blues, greens, and yellows, which can be used to visualize water-related data (e.g., wave heights or velocities). The color map transitions from light blues (representing shallower or clearer water) through darker blues/greens, and finally into yellowish tones that can highlight areas of foam or breaking waves.

Returns
A color map object suitable for use with matplotlib plotting functions or other visualization frameworks.

Return type
matplotlib.colors.LinearSegmentedColormap

celeris.utils.**cosh**(*x*)

Returns the hyperbolic cosine of *x*.

The hyperbolic cosine is defined as:
cosh(x) = (e^x + e^(-x)) / 2

Parameters
x (*float*) – The input value.

Returns
The value of the hyperbolic cosine of *x*.

Return type
float

celeris.utils.**sineWave**(*x*, *y*, *t*, *d*, *amplitude*, *period*, *theta*, *phase*, *g*, *wave_type*)

Computes a sine wave (and related momentum terms) at a given point (x, y) and time t.

This function uses a dispersion relation and hyperbolic tangent of wave depth to approximate wave number (k) and phase speed (c). It then calculates free-surface elevation (eta) and horizontal momentum components(hu, hv) based on wave parameters. An optional decay term is applied for certain wave types.

Parameters

- **x** (*float*) – x-coordinate where the wave is evaluated.
- **y** (*float*) – y-coordinate where the wave is evaluated.
- **t** (*float*) – Current time in the simulation.
- **d** (*float*) – Local water depth.

- **amplitude** (*float*) – Wave amplitude.
- **period** (*float*) – Wave period.
- **theta** (*float*) – Wave propagation angle in radians.
- **phase** (*float*) – Additional phase offset.
- **g** (*float*) – Gravitational acceleration.
- **wave_type** (*int*) – Wave type indicator. If *wave_type == 2*, a decay multiplier is applied to the wave for demonstration/limiting purposes.

Returns

A 3-component vector:

- **eta**: Free-surface elevation (wave height) above still water level.
- **hu**: Momentum in the x-direction (wave speed * wave height).
- **hv**: Momentum in the y-direction (wave speed * wave height).

Return type

ti.Vector([float, float, float])

Note

- The calculation for wave number *k* uses a simplified relationship assuming linear wave theory with a hyperbolic tangent term for finite depth.
- The term *ti.min(1.0, t / period)* is used to gradually ramp up the wave from zero at t=0 (avoid sudden wave onset).
- If *wave_type == 2*, an additional decay factor is applied as *t* approaches *num_waves * period* (here *num_waves* is hard-coded to 4 in the example). For a transient pulse
- The returned *hu* and *hv* are computed as a fraction of *g * eta / (c * k) * tanh(k * d)*, scaled by the direction cosines *(cos(theta), sin(theta))*.

Current Version

CelerisAi is a Python-Taichi-based software designed for nearshore wave modeling. This solver offers high-performance simulations on various hardware platforms and seamlessly integrates with machine learning and artificial intelligence environments. The solver leverages the flexibility of Python for customization and interoperability, while Taichi's high-performance parallel programming capabilities ensure efficient computations.

This package contains modules for running domain-specific simulations, solving problems, and providing general utilities:

- **domain.py**: Contains classes and functions defining the problem domain.
- **runner.py**: Manages the execution flow, including initialization and orchestration.
- **solver.py**: Implements the core solver logic for the defined domain and mathematical models.
- **utils.py**: A collection of helper utilities used across the package.

Example

```
>>> from celeris import domain, runner, solver, utils
>>> domain_obj = domain.Domain(params={...})
>>> solver_obj = solver.Solver(domain=domain_obj)
>>> runner_obj = runner.Evolve(solver = solver_obj)
>>> runner_obj.Evolve_Headless()
```

`celeris.__version__`

The current version of the CelerisAi.

Type

str

BIBLIOGRAPHY

[TavakkolLynett2020] Tavakkol, S., & Lynett, P. (2020). Celeris Base: An interactive and immersive Boussinesq-type nearshore wave simulation software. Computer Physics Communications, 248, 106966. https://doi.org/10.1016/j.cpc.2019.106966

[MadseSorensen1992] Madsen, P. A., & Sørensen, O. R. (1992). A new form of the Boussinesq equations with improved linear dispersion characteristics. Part 2. A slowly-varying bathymetry. Coastal Engineering, 18(3–4), 183–204. https://doi.org/10.1016/0378-3839(92)90019-Q

PYTHON MODULE INDEX

INDEX