AeoLiS Documentation

Release 1.0

Bas Hoonhout

Dec 21, 2017

Contents

AeoLiS is a process-based model for simulating aeolian sediment transport in situations where supply-limiting factors are important, like in coastal environments. Supply-limitations currently supported are soil moisture contents, sediment sorting and armouring, bed slope effects, air humidity and roughness elements.

This documentation describes the Python implementation of the AeoLiS model. The source code of the Python implementation can be found at https://github.com/openearth/aeolis-python.

CHAPTER 1

Contents

1.1 Model description

The model approach of *[dVvTdVvR+14]* is extended to compute the spatiotemporal varying sediment availability through simulation of the process of beach armoring. For this purpose the bed is discretized in horizontal grid cells and in vertical bed layers (2DV). Moreover, the grain size distribution is discretized into fractions. This allows the grain size distribition to vary both horizontally and vertically. A bed composition module is used to compute the sediment availability for each sediment fraction individually. This model approach is a generalization of existing model concepts, like the shear velocity threshold and critical fetch, and therefore compatible with these existing concepts.

1.1.1 Advection Scheme

A 1D advection scheme is adopted in correspondence with *[dVvTdVvR+14]* in which c [$\mathrm{kg/m^2}$] is the instantaneous sediment mass per unit area in transport:

$$\frac{\partial c}{\partial t} + u_z \frac{\partial c}{\partial x} = E - D \tag{1.1}$$

t [s] denotes time and x [m] denotes the cross-shore distance from a zero-transport boundary. E and D [$\mathrm{kg/m^2/s}$] represent the erosion and deposition terms and hence combined represent the net entrainment of sediment. Note that Equation (1.1) differs from Equation 9 in *[dVvTdVvR+14]* as they use the saltation height h [m] and the sediment concentration C_c [$\mathrm{kg/m^3}$]. As h is not solved for, the presented model computes the sediment mass per unit area $c = hC_\mathrm{c}$ rather than the sediment concentration C_c. For conciseness we still refer to c as the *sediment concentration*.

The net entrainment is determined based on a balance between the equilibrium or saturated sediment concentration c_sat [$\mathrm{kg/m^2}$] and the instantaneous sediment transport concentration c and is maximized by the available sediment in the bed m_a [$\mathrm{kg/m^2}$] according to:

$$E - D = \min\left(\frac{\partial m_\mathrm{a}}{\partial t} \quad ; \quad \frac{c_\mathrm{sat} - c}{T}\right) \tag{1.2}$$

T [s] represents an adaptation time scale that is assumed to be equal for both erosion and deposition. A time scale of 1 second is commonly used (*[dVvTdVvR+14]*).

The saturated sediment concentration c_{sat} is computed using an empirical sediment transport formulation (e.g. *[Bag37b]*):

$$q_{\mathrm{sat}} = \alpha C \frac{\rho_{\mathrm{a}}}{g} \sqrt{\frac{d_{\mathrm{n}}}{D_{\mathrm{n}}}} \left(u_z - u_{\mathrm{th}}\right)^3 \tag{1.3}$$

in which q_{sat} [kg/m/s] is the equilibrium or saturated sediment transport rate and represents the sediment transport capacity. u_z [m/s] is the wind velocity at height z [m] and u_{th} the velocity threshold [m/s]. The properties of the sediment in transport are represented by a series of parameters: C [–] is a parameter to account for the grain size distribution width, ρ_{a} [$\mathrm{kg/m^3}$] is the density of the air, g [$\mathrm{m/s^2}$] is the gravitational constant, d_{n} [m] is the nominal grain size and D_{n} [m] is a reference grain size. α is a constant to account for the conversion of the measured wind velocity to the near-bed shear velocity following Prandtl-Von Kármán's Law of the Wall: $\left(\frac{\kappa}{\ln z/z'}\right)^3$ in which z' [m] is the height at which the idealized velocity profile reaches zero and κ [-] is the Von Kármán constant.

The equilibrium sediment transport rate q_{sat} is divided by the wind velocity u_z to obtain a mass per unit area (per unit width):

$$c_{\mathrm{sat}} = \max\left(0 \quad ; \quad \alpha C \frac{\rho_{\mathrm{a}}}{g} \sqrt{\frac{d_n}{D_n}} \frac{\left(u_z - u_{\mathrm{th}}\right)^3}{u_z}\right)$$

in which C [–] is an empirical constant to account for the grain size distribution width, ρ_{a} [$\mathrm{kg/m^3}$] is the air density, g [$\mathrm{m/s^2}$] is the gravitational constant, d_{n} [m] is the nominal grain size, D_{n} [m] is a reference grain size, u_z [m/s] is the wind velocity at height z [m] and α [–] is a constant to convert from measured wind velocity to shear velocity.

Note that at this stage the spatial variations in wind velocity are not solved for and hence no morphological feedback is included in the simulation. The model is initially intended to provide accurate sediment fluxes from the beach to the dunes rather than to simulate subsequent dune formation.

1.1.2 Multi-fraction Erosion and Deposition

The formulation for the equilibrium or saturated sediment concentration c_{sat} (Equation `equilibrium-transport`) is capable of dealing with variations in grain size through the variables u_{th}, d_{n} and C (*[Bag37b]*). However, the transport formulation only describes the saturated sediment concentration assuming a fixed grain size distribution, but does not define how multiple fractions coexist in transport. If the saturated sediment concentration formulation would be applied to each fraction separately and summed up to a total transport, the total sediment transport would increase with the number of sediment fractions. Since this is unrealistic behavior the saturated sediment concentration c_{sat} for the different fractions should be weighted in order to obtain a realistic total sediment transport. Equation (1.2) therefore is modified to include a weighting factor $\hat{w}_k$ in which k represents the sediment fraction index:

$$E_k - D_k = \min\left(\frac{\partial m_{\mathrm{a},k}}{\partial t} \quad ; \quad \frac{\hat{w}_k \cdot c_{\mathrm{sat},k} - c_k}{T}\right) \tag{1.4}$$

It is common to use the grain size distribution in the bed as weighting factor for the saturated sediment concentration (e.g. *[Delft3DFManual14]*, section 11.6.4). Using the grain size distribution at the bed surface as a weighting factor assumes, in case of erosion, that all sediment at the bed surface is equally exposed to the wind.

Using the grain size distribution at the bed surface as weighting factor in case of deposition would lead to the behavior where deposition becomes dependent on the bed composition. Alternatively, in case of deposition, the saturated sediment concentration can be weighted based on the grain size distribution in the air. Due to the nature of saltation, in which continuous interaction with the bed forms the saltation cascade, both the grain size distribution in the bed and in the air are likely to contribute to the interaction between sediment fractions. The ratio between both contributions in the model is determined by a bed interaction parameter ζ.

The weighting of erosion and deposition of individual fractions is computed according to:

$$\hat{w}_k = \frac{w_k}{\sum_{k=1}^{n_k} w_k} \tag{1.5}$$

$$\text{where} \quad w_k = (1-\zeta) \cdot w_k^{\text{air}} + (1-\hat{S}_k) \cdot w_k^{\text{bed}} \tag{1.6}$$

in which k represents the sediment fraction index, n_k the total number of sediment fractions, w_k is the unnormalized weighting factor for fraction k, $\hat{w}_k$ is its normalized counterpart, w_k^{air} and w_k^{bed} are the weighting factors based on the grain size distribution in the air and bed respectively and $\hat{S}_k$ is the effective sediment saturation of the air. The weighting factors based on the grain size distribution in the air and the bed are computed using mass ratios:

$$w_k^{\text{air}} = \frac{c_k}{c_{\text{sat},k}} \quad ; \quad w_k^{\text{bed}} = \frac{m_{\text{a},k}}{\sum_{k=1}^{n_k} m_{\text{a},k}} \tag{1.7}$$

The sum of the ratio w_k^{air} over the fractions denotes the degree of saturation of the air column for fraction k. The degree of saturation determines if erosion of a fraction may occur. Also in saturated situations erosion of a sediment fraction can occur due to an exchange of momentum between sediment fractions, which is represented by the bed interaction parameter ζ. The effective degree of saturation is therefore also influenced by the bed interaction parameter and defined as:

$$\hat{S}_k = \min\left(1 \quad ; \quad (1-\zeta) \cdot \sum_{k=1}^{n_k} w_k^{\text{air}}\right)$$

When the effective saturation is greater than or equal to unity the air is (over)saturated and no erosion will occur. The grain size distribution in the bed is consequently less relevant and the second term in Equation (1.7) is thus minimized and zero in case $\zeta = 0$. In case the effective saturation is less than unity erosion may occur and the grain size distribution of the bed also contributes to the weighting over the sediment fractions. The weighting factors for erosion are then composed from both the grain size distribution in the air and the grain size distribution at the bed surface. Finally, the resulting weighting factors are normalized to sum to unity over all fractions ($\hat{w}_k$).

The composition of weighting factors for erosion is based on the saturation of the air column. The non-saturated fraction determines the potential erosion of the bed. Therefore the non-saturated fraction can be used to scale the grain size distribution in the bed in order to combine it with the grain size distribution in the air according to Equation (1.7). The non-saturated fraction of the air column that can be used for scaling is therefore $1 - \hat{S}_k$.

For example, if bed interaction is disabled ($\zeta = 0$) and the air is 70% saturated, then the grain size distribution in the air contributes 70% to the weighting factors for erosion, while the grain size distribution in the bed contributes the other 30% (Figure Fig. 1.1, upper left panel). In case of (over)saturation the grain size distribution in transport contributes 100% to the weighting factors and the grain size distribution in the bed is of no influence. Transport progresses in downwind direction without interaction with the bed.

To allow for bed interaction in saturated situations in which no net erosion can occur, the bed interaction parameter ζ is used (Figure Fig. 1.1). The bed interaction parameter can take values between 0.0 and 1.0 in which the weighting factors for the equilibrium or saturated sediment concentration in an (over)saturated situation are fully determined by the grain size distribution in the bed or in the air respectively. A bed interaction value of 0.2 represents the situation in which the grain size distribution at the bed surface contributes 20% to the weighting of the saturated sediment concentration over the fractions. In the example situation where the air is 70% saturated such value for the bed interaction parameter would lead to weighting factors that are constituted for $70\% \cdot (100\% - 20\%) = 56\%$ based on the grain size distribution in the air and for the other 44% based on the grain size distribution at the bed surface (Figure Fig. 1.1, upper right panel).

The parameterization of the exchange of momentum between sediment fractions is an aspect of saltation that is still poorly understood. Therefore calibration of the bed interaction parameter ζ is necessary. The model parameters in Equation `equilibrium-transport` can be chosen in accordance with the assumptions underlying multi-fraction sediment transport. C should be set to 1.5 as each individual sediment fraction is well-sorted, d_n should be chosen equal to D_n as the grain size dependency is implemented through u_{th}. u_{th} typically varies between 1 and 6 m/s for sand.

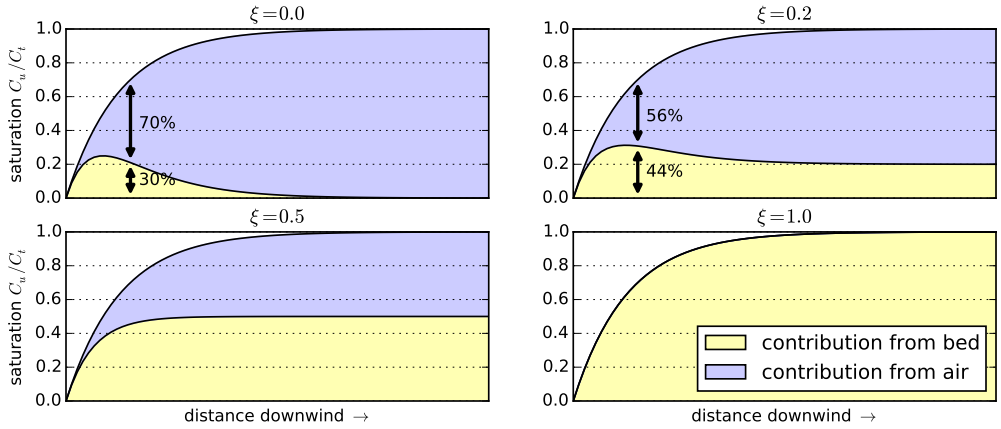

Fig. 1.1: Contributions of the grain size distribution in the bed and in the air to the weighting factors $\hat{w}_k$ for the equilibrium sediment concentration in Equation (1.4) for different values of the bed interaction parameter.

1.1.3 Simulation of Sediment Sorting and Beach Armoring

Since the equilibrium or saturated sediment concentration $c_{\mathrm{sat},k}$ is weighted over multiple sediment fractions in the extended advection model, also the instantaneous sediment concentration c_k is computed for each sediment fraction individually. Consequently, grain size distributions may vary over the model domain and in time. These variations are thereby not limited to the horizontal, but may also vary over the vertical since fine sediment may be deposited on top of coarse sediment or, reversely, fines may be eroded from the bed surface leaving coarse sediment to reside on top of the original mixed sediment. In order to allow the model to simulate the processes of sediment sorting and beach armoring the bed is discretized in horizontal grid cells and vertical bed layers (2DV; Figure Fig. 1.2).

The discretization of the bed consists of a minimum of three vertical bed layers with a constant thickness and an unlimited number of horizontal grid cells. The top layer is the *bed surface layer* and is the only layer that interacts with the wind and hence determines the spatiotemporal varying sediment availability and the contribution of the grain size distribution in the bed to the weighting of the saturated sediment concentration. One or more *bed composition layers* are located underneath the bed surface layer and form the upper part of the erodible bed. The bottom layer is the *base layer* and contains an infinite amount of erodible sediment according to the initial grain size distribution. The base layer cannot be eroded, but can supply sediment to the other layers.

Each layer in each grid cell describes a grain size distribution over a predefined number of sediment fractions (Figure Fig. 1.2, detail). Sediment may enter or leave a grid cell only through the bed surface layer. Since the velocity threshold depends among others on the grain size, erosion from the bed surface layer will not be uniform over all sediment fractions, but will tend to erode fines more easily than coarse sediment (Figure Fig. 1.2, detail, upper left panel). If sediment is eroded from the bed surface layer, the layer is repleted by sediment from the lower bed composition layers. The repleted sediment has a different grain size distribution than the sediment eroded from the bed surface layer. If more fines are removed from the bed surface layer in a grid cell than repleted, the median grain size increases. If erosion of fines continues the bed surface layer becomes increasingly coarse. Deposition of fines or erosion of coarse material may resume the erosion of fines from the bed.

In case of deposition the process is similar. Sediment is deposited in the bed surface layer that then passes its excess sediment to the lower bed layers (Figure Fig. 1.2, detail, upper right panel). If more fines are deposited than passed to the lower bed layers the bed surface layer becomes increasingly fine.

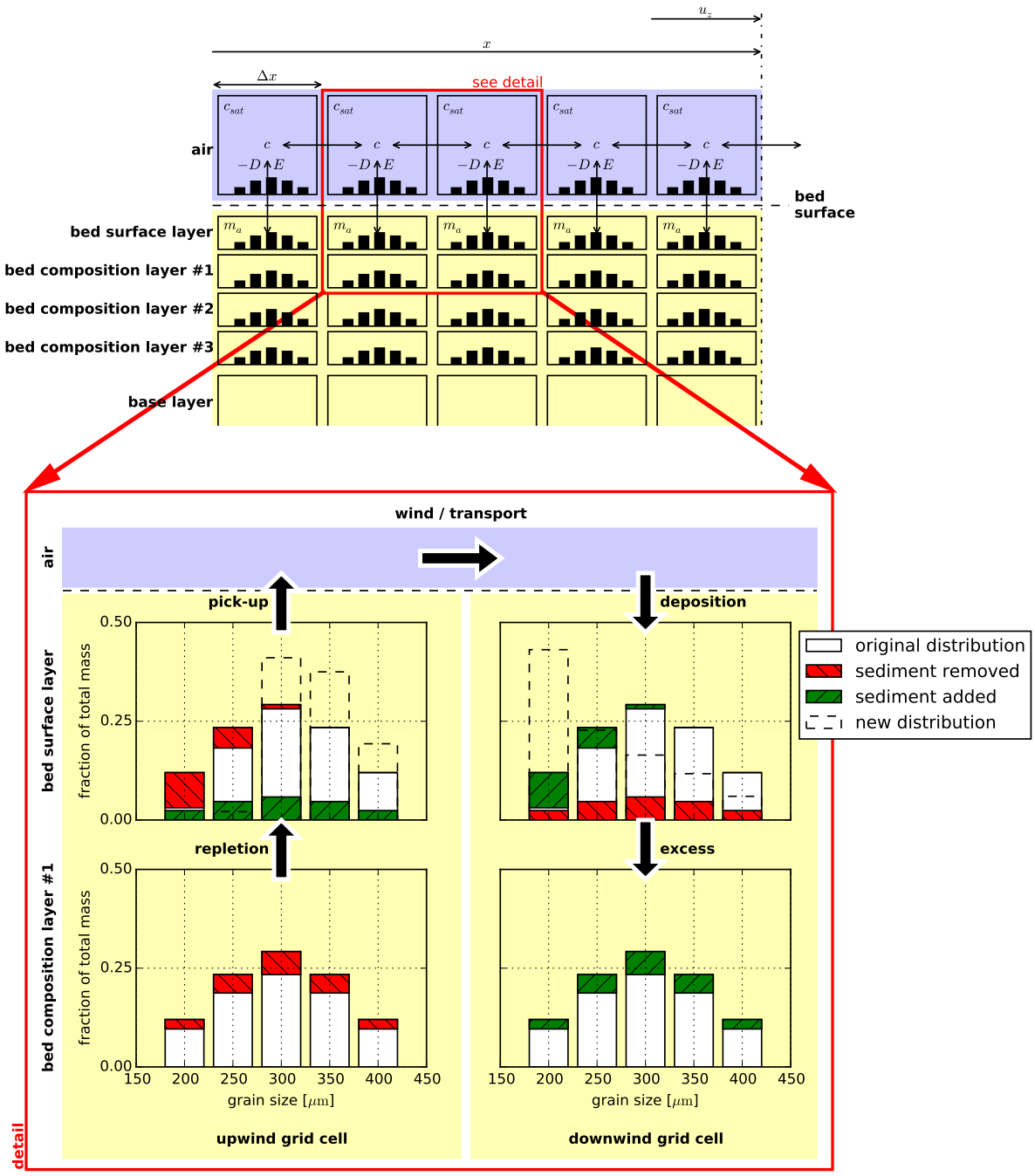

Fig. 1.2: Schematic of bed composition discretisation and advection scheme. Horizontal exchange of sediment may occur solely through the air that interacts with the *bed surface layer*. The detail presents the simulation of sorting and bed armouring where the bed surface layer in the upwind grid cell becomes coarser due to non-uniform erosion over the sediment fractions, while the bed surface layer in the downwind grid cell becomes finer due to non-uniform deposition over the sediment fractions. Symbols refer to Equations (1.1) and (1.2).

1.1.4 Simulation of the Emergence of Non-erodible Roughness Elements

Sediment sorting may lead to the emergence of non-erodible elements from the bed. Non-erodible roughness elements may shelter the erodible bed from wind erosion due to shear partitioning, resulting in a reduced sediment availability (*[RGL93]*). Therefore the equation of *[RGL93]* is implemented according to:

$$u_{*\mathrm{th,R}} = u_{*\mathrm{th}} \cdot \sqrt{\left(1 - m \cdot \sum_{k=k_0}^{n_\mathrm{k}} w_k^{\mathrm{bed}}\right)\left(1 + \frac{m\beta}{\sigma} \cdot \sum_{k=k_0}^{n_\mathrm{k}} w_k^{\mathrm{bed}}\right)} \tag{1.8}$$

in which σ is the ratio between the frontal area and the basal area of the roughness elements and β is the ratio between the drag coefficients of the roughness elements and the bed without roughness elements. m is a factor to account for the difference between the mean and maximum shear stress and is usually chosen 1.0 in wind tunnel experiments and may be lowered to 0.5 for field applications. The roughness density λ in the original equation of *[RGL93]* is obtained from the mass fraction in the bed surface layer w_k^{bed} according to:

$$\lambda = \frac{\sum_{k=k_0}^{n_\mathrm{k}} w_k^{\mathrm{bed}}}{\sigma}$$

in which k_0 is the index of the smallest non-erodible sediment fraction in current conditions and n_k is the total number of sediment fractions. It is assumed that the sediment fractions are ordered by increasing size. Whether a fraction is erodible depends on the sediment transport capacity.

1.1.5 Simulation of the Hydraulic Mixing, Infiltration and Evaporation

As sediment sorting due to aeolian processes can lead to armoring of a beach surface, mixing of the beach surface or erosion of course material may undo the effects of armoring. To ensure a proper balance between processes that limit and enhance sediment availability in the model both types of processes need to be sufficiently represented when simulating spatiotemporal varying bed surface properties and sediment availability.

A typical upwind boundary in coastal environments during onshore winds is the water line. For aeolian sediment transport the water line is a zero-transport boundary. In the presence of tides, the intertidal beach is flooded periodically. Hydraulic processes like wave breaking mix the bed surface layer of the intertidal beach, break the beach armoring and thereby influence the availability of sediment. Moreover, the hydraulic processes periodically wet the intertidal beach temporally increasing the shear velocity threshold. Infiltration and evaporation subsequently dry the beach.

In the model the mixing of sediment is simulated by averaging the sediment distribution over the depth of disturbance (Δz_d). The depth of disturbance is linearly related to the breaker height (e.g. *[Kin51]*, *[Wil71]*, *[MAROHare07]*). *[MAROHare07]* proposes an empirical factor $f_{\Delta z_\mathrm{d}}$ [-] that relates the depth of disturbance directly to the local breaker height according to:

$$\Delta z_\mathrm{d} = f_{\Delta z_\mathrm{d}} \cdot \min\left(H \quad ; \quad \gamma \cdot d\right)$$

in which the offshore wave height H [m] is taken as the local wave height maximized by a maximum wave height over depth ratio γ [-]. d [m] is the water depth that is provided to the model through an input time series of water levels. Typical values for $f_{\Delta z_\mathrm{d}}$ are 0.05 to 0.4 and 0.5 for γ.

The drying of the beach is simulated by simplified functions for infiltration and evaporation. Infiltration is represented by an exponential decay function that is governed by a drying time scale T_dry. Evaporation is simulated using an adapted version of the Penman-Monteith equation (*[Shu93]*) that is governed by meteorological time series of solar radiation, temperature and humidity.

Bibliography

1.2 Numerical implementation

The numerical implementation of the equations presented in *Model description* is explained here. The implementation is available as Python package through the OpenEarth GitHub repository at: http://www.github.com/openearth/aeolis-python/

1.2.1 Advection equation

The advection equation is implemented in two-dimensional form following:

$$\frac{\partial c}{\partial t} + u_{z,\mathrm{x}} \frac{\partial c}{\partial x} + u_{z,\mathrm{y}} \frac{\partial c}{\partial y} = \frac{c_{\mathrm{sat}} - c}{T} \tag{1.9}$$

in which c [$\mathrm{kg/m^2}$] is the sediment mass per unit area in the air, c_{sat} [$\mathrm{kg/m^2}$] is the maximum sediment mass in the air that is reached in case of saturation, $u_{z,\mathrm{x}}$ and $u_{z,\mathrm{y}}$ are the x- and y-component of the wind velocity at height z [m], T [s] is an adaptation time scale, t [s] denotes time and x [m] and y [m] denote cross-shore and alongshore distances respectively.

The formulation is discretized following a first order upwind scheme assuming that the wind velocity u_z is positive in both x-direction and y-direction:

$$\begin{aligned} \frac{c^{n+1}_{i,j,k} - c^n_{i,j,k}}{\Delta t^n} + u^n_{z,\mathrm{x}} \frac{c^n_{i+1,j,k} - c^n_{i,j,k}}{\Delta x_{i,j}} + u^n_{z,\mathrm{y}} \frac{c^n_{i,j+1,k} - c^n_{i,j,k}}{\Delta y_{i,j}} \\ = \frac{\hat{w}^n_{i,j,k} \cdot c^n_{\mathrm{sat},i,j,k} - c^n_{i,j,k}}{T} \end{aligned} \tag{1.10}$$

in which n is the time step index, i and j are the cross-shore and alongshore spatial grid cell indices and k is the grain size fraction index. w [-] is the weighting factor used for the weighted addition of the saturated sediment concentrations over all grain size fractions.

The discretization can be generalized for any wind direction as:

$$\begin{aligned} \frac{c^{n+1}_{i,j,k} - c^n_{i,j,k}}{\Delta t^n} + u^n_{z,\mathrm{x}+} c^n_{i,j,k,\mathrm{x}+} + u^n_{z,\mathrm{y}+} c^n_{i,j,k,\mathrm{y}+} \\ + u^n_{z,\mathrm{x}-} c^n_{i,j,k,\mathrm{x}-} + u^n_{z,\mathrm{y}-} c^n_{i,j,k,\mathrm{y}-} = \frac{\hat{w}^n_{i,j,k} \cdot c^n_{\mathrm{sat},i,j,k} - c^n_{i,j,k}}{T} \end{aligned} \tag{1.11}$$

in which:

$$\begin{aligned} u^n_{z,\mathrm{x}+} &= \max(0, u^n_{z,\mathrm{x}}) \quad ; \quad u^n_{z,\mathrm{y}+} = \max(0, u^n_{z,\mathrm{y}}) \\ u^n_{z,\mathrm{x}-} &= \min(0, u^n_{z,\mathrm{x}}) \quad ; \quad u^n_{z,\mathrm{y}-} = \min(0, u^n_{z,\mathrm{y}}) \end{aligned} \tag{1.12}$$

and

$$\begin{aligned} c^n_{i,j,k,\mathrm{x}+} &= \frac{c^n_{i+1,j,k} - c^n_{i,j,k}}{\Delta x} \quad ; \quad c^n_{i,j,k,\mathrm{y}+} = \frac{c^n_{i,j+1,k} - c^n_{i,j,k}}{\Delta y} \\ c^n_{i,j,k,\mathrm{x}-} &= \frac{c^n_{i,j,k} - c^n_{i-1,j,k}}{\Delta x} \quad ; \quad c^n_{i,j,k,\mathrm{y}-} = \frac{c^n_{i,j,k} - c^n_{i,j-1,k}}{\Delta y} \end{aligned} \tag{1.13}$$

Equation (1.11) is explicit in time and adheres to the Courant-Friedrich-Lewis (CFL) condition for numerical stability. Alternatively, the advection equation can be discretized implicitly in time for unconditional stability:

$$\begin{aligned} \frac{c^{n+1}_{i,j,k} - c^n_{i,j,k}}{\Delta t^n} + u^{n+1}_{z,\mathrm{x}+} c^{n+1}_{i,j,k,\mathrm{x}+} + u^{n+1}_{z,\mathrm{y}+} c^{n+1}_{i,j,k,\mathrm{y}+} \\ + u^{n+1}_{z,\mathrm{x}-} c^{n+1}_{i,j,k,\mathrm{x}-} + u^{n+1}_{z,\mathrm{y}-} c^{n+1}_{i,j,k,\mathrm{y}-} = \frac{\hat{w}^{n+1}_{i,j,k} \cdot c^{n+1}_{\mathrm{sat},i,j,k} - c^{n+1}_{i,j,k}}{T} \end{aligned} \tag{1.14}$$

Equation (1.11) and :eq:apx-implicit-generalized‘ can be rewritten as:

$$c_{i,j,k}^{n+1} = c_{i,j,k}^{n} - \Delta t^n \left[u_{z,\mathrm{x}+}^{n} c_{i,j,k,\mathrm{x}+}^{n} + u_{z,\mathrm{y}+}^{n} c_{i,j,k,\mathrm{y}+}^{n} + u_{z,\mathrm{x}-}^{n} c_{i,j,k,\mathrm{x}-}^{n} + u_{z,\mathrm{y}-}^{n} c_{i,j,k,\mathrm{y}-}^{n} + \frac{\hat{w}_{i,j,k}^{n} \cdot c_{\mathrm{sat},i,j,k}^{n} - c_{i,j,k}^{n}}{T} \right] \tag{1.15}$$

and

$$c_{i,j,k}^{n+1} + \Delta t^n \left[u_{z,\mathrm{x}+}^{n+1} c_{i,j,k,\mathrm{x}+}^{n+1} + u_{z,\mathrm{y}+}^{n+1} c_{i,j,k,\mathrm{y}+}^{n+1} + u_{z,\mathrm{x}-}^{n+1} c_{i,j,k,\mathrm{x}-}^{n+1} + u_{z,\mathrm{y}-}^{n+1} c_{i,j,k,\mathrm{y}-}^{n+1} + \frac{\hat{w}_{i,j,k}^{n+1} \cdot c_{\mathrm{sat},i,j,k}^{n+1} - c_{i,j,k}^{n+1}}{T} \right] = c_{i,j,k}^{n} \tag{1.16}$$

and combined using a weighted average:

$$\begin{aligned} c_{i,j,k}^{n+1} + \Gamma \Delta t^n & \left[u_{z,\mathrm{x}+}^{n+1} c_{i,j,k,\mathrm{x}+}^{n+1} + u_{z,\mathrm{y}+}^{n+1} c_{i,j,k,\mathrm{y}+}^{n+1} + u_{z,\mathrm{x}-}^{n+1} c_{i,j,k,\mathrm{x}-}^{n+1} + u_{z,\mathrm{y}-}^{n+1} c_{i,j,k,\mathrm{y}-}^{n+1} + \frac{\hat{w}_{i,j,k}^{n+1} \cdot c_{\mathrm{sat},i,j,k}^{n+1} - c_{i,j,k}^{n+1}}{T} \right] \\ = c_{i,j,k}^{n} - (1-\Gamma) \Delta t^n & \left[u_{z,\mathrm{x}+}^{n} c_{i,j,k,\mathrm{x}+}^{n} + u_{z,\mathrm{y}+}^{n} c_{i,j,k,\mathrm{y}+}^{n} + u_{z,\mathrm{x}-}^{n} c_{i,j,k,\mathrm{x}-}^{n} + u_{z,\mathrm{y}-}^{n} c_{i,j,k,\mathrm{y}-}^{n} + \frac{\hat{w}_{i,j,k}^{n} \cdot c_{\mathrm{sat},i,j,k}^{n} - c_{i,j,k}^{n}}{T} \right] \end{aligned} \tag{1.17}$$

in which Γ is a weight that ranges from 0 – 1 and determines the implicitness of the scheme. The scheme is implicit with $\Gamma = 0$, explicit with $\Gamma = 1$ and semi-implicit otherwise. $\Gamma = 0.5$ results in the semi-implicit Crank-Nicolson scheme.

Equation (1.13) is back-substituted in Equation (1.17):

$$\begin{aligned} c_{i,j,k}^{n+1} + \Gamma \Delta t^n & \left[u_{z,\mathrm{x}+}^{n+1} \frac{c_{i+1,j,k}^{n+1} - c_{i,j,k}^{n+1}}{\Delta x} + u_{z,\mathrm{y}+}^{n+1} \frac{c_{i,j+1,k}^{n+1} - c_{i,j,k}^{n+1}}{\Delta y} + u_{z,\mathrm{x}-}^{n+1} \frac{c_{i,j,k}^{n+1} - c_{i-1,j,k}^{n+1}}{\Delta x} + u_{z,\mathrm{y}-}^{n+1} \frac{c_{i,j,k}^{n+1} - c_{i,j-1,k}^{n+1}}{\Delta y} + \frac{\hat{w}_{i,j,k}^{n+1} \cdot c_{\mathrm{sat},i,j,k}^{n+1} - c_{i,j,k}^{n+1}}{T} \right] \\ = c_{i,j,k}^{n} - (1-\Gamma) \Delta t^n & \left[u_{z,\mathrm{x}+}^{n} \frac{c_{i+1,j,k}^{n} - c_{i,j,k}^{n}}{\Delta x} + u_{z,\mathrm{y}+}^{n} \frac{c_{i,j+1,k}^{n} - c_{i,j,k}^{n}}{\Delta y} + u_{z,\mathrm{x}-}^{n} \frac{c_{i,j,k}^{n} - c_{i-1,j,k}^{n}}{\Delta x} + u_{z,\mathrm{y}-}^{n} \frac{c_{i,j,k}^{n} - c_{i,j-1,k}^{n}}{\Delta y} + \frac{\hat{w}_{i,j,k}^{n} \cdot c_{\mathrm{sat},i,j,k}^{n} - c_{i,j,k}^{n}}{T} \right] \end{aligned} \tag{1.18}$$

and rewritten:

$$\begin{aligned} & \left[1 - \Gamma \left(u_{z,\mathrm{x}+}^{n+1} \frac{\Delta t^n}{\Delta x} + u_{z,\mathrm{y}+}^{n+1} \frac{\Delta t^n}{\Delta y} - u_{z,\mathrm{x}-}^{n+1} \frac{\Delta t^n}{\Delta x} - u_{z,\mathrm{y}-}^{n+1} \frac{\Delta t^n}{\Delta y} + \frac{\Delta t^n}{T} \right) \right] c_{i,j,k}^{n+1} \\ & + \Gamma \left(u_{z,\mathrm{x}+}^{n+1} \frac{\Delta t^n}{\Delta x} c_{i+1,j,k}^{n+1} + u_{z,\mathrm{y}+}^{n+1} \frac{\Delta t^n}{\Delta y} c_{i,j+1,k}^{n+1} - u_{z,\mathrm{x}-}^{n+1} \frac{\Delta t^n}{\Delta x} c_{i-1,j,k}^{n+1} - u_{z,\mathrm{y}-}^{n+1} \frac{\Delta t^n}{\Delta y} c_{i,j-1,k}^{n+1} \right) \\ & = \left[1 + (1-\Gamma) \left(u_{z,\mathrm{x}+}^{n} \frac{\Delta t^n}{\Delta x} + u_{z,\mathrm{y}+}^{n} \frac{\Delta t^n}{\Delta y} - u_{z,\mathrm{x}-}^{n} \frac{\Delta t^n}{\Delta x} - u_{z,\mathrm{y}-}^{n} \frac{\Delta t^n}{\Delta y} + \frac{\Delta t^n}{T} \right) \right] c_{i,j,k}^{n} \\ & + (1-\Gamma) \left(u_{z,\mathrm{x}+}^{n} \frac{\Delta t^n}{\Delta x} c_{i+1,j,k}^{n} + u_{z,\mathrm{y}+}^{n} \frac{\Delta t^n}{\Delta y} c_{i,j+1,k}^{n} - u_{z,\mathrm{x}-}^{n} \frac{\Delta t^n}{\Delta x} c_{i-1,j,k}^{n} - u_{z,\mathrm{y}-}^{n} \frac{\Delta t^n}{\Delta y} c_{i,j-1,k}^{n} \right) \\ & - \Gamma \hat{w}_{i,j,k}^{n+1} \cdot c_{\mathrm{sat},i,j,k}^{n+1} \frac{\Delta t^n}{T} - (1-\Gamma) \hat{w}_{i,j,k}^{n} \cdot c_{\mathrm{sat},i,j,k}^{n} \frac{\Delta t^n}{T} \end{aligned} \tag{1.19}$$

and simplified:

$$a_{i,j}^{0,0} c_{i,j,k}^{n+1} + a_{i,j}^{1,0} c_{i+1,j,k}^{n+1} + a_{i,j}^{0,1} c_{i,j+1,k}^{n+1} - a_{i,j}^{-1,0} c_{i-1,j,k}^{n+1} - a_{i,j}^{0,-1} c_{i,j-1,k}^{n+1} = y_{i,j,k} \tag{1.20}$$

where the implicit coefficients are defined as:

$$\begin{array}{rcl} a_{i,j}^{0,0} & = & \left[1 - \Gamma \left(u_{z,\mathrm{x}+}^{n+1} \frac{\Delta t^n}{\Delta x} + u_{z,\mathrm{y}+}^{n+1} \frac{\Delta t^n}{\Delta y} - u_{z,\mathrm{x}-}^{n+1} \frac{\Delta t^n}{\Delta x} - u_{z,\mathrm{y}-}^{n+1} \frac{\Delta t^n}{\Delta y} + \frac{\Delta t^n}{T} \right) \right] \\ a_{i,j}^{1,0} & = & \Gamma u_{z,\mathrm{x}+}^{n+1} \frac{\Delta t^n}{\Delta x} \\ a_{i,j}^{0,1} & = & \Gamma u_{z,\mathrm{y}+}^{n+1} \frac{\Delta t^n}{\Delta y} \\ a_{i,j}^{-1,0} & = & \Gamma u_{z,\mathrm{x}-}^{n+1} \frac{\Delta t^n}{\Delta x} \\ a_{i,j}^{0,-1} & = & \Gamma u_{z,\mathrm{y}-}^{n+1} \frac{\Delta t^n}{\Delta y} \end{array} \tag{1.21}$$

and the explicit right-hand side as:

$$\begin{aligned} y_{i,j,k}^n = & \left[1 + (1-\Gamma) \left(u_{z,\mathrm{x}+}^n \frac{\Delta t^n}{\Delta x} + u_{z,\mathrm{y}+}^n \frac{\Delta t^n}{\Delta y} - u_{z,\mathrm{x}-}^n \frac{\Delta t^n}{\Delta x} - u_{z,\mathrm{y}-}^n \frac{\Delta t^n}{\Delta y} + \frac{\Delta t^n}{T} \right) \right] c_{i,j,k}^n \\ & + (1-\Gamma) \left(u_{z,\mathrm{x}+}^n \frac{\Delta t^n}{\Delta x} c_{i+1,j,k}^n + u_{z,\mathrm{y}+}^n \frac{\Delta t^n}{\Delta y} c_{i,j+1,k}^n - u_{z,\mathrm{x}-}^n \frac{\Delta t^n}{\Delta x} c_{i-1,j,k}^n - u_{z,\mathrm{y}-}^n \frac{\Delta t^n}{\Delta y} c_{i,j-1,k}^n \right) \\ & - \Gamma \hat{w}_{i,j,k}^{n+1} \cdot c_{\mathrm{sat},i,j,k}^{n+1} \frac{\Delta t^n}{T} - (1-\Gamma) \hat{w}_{i,j,k}^n \cdot c_{\mathrm{sat},i,j,k}^n \frac{\Delta t^n}{T} \end{aligned} \tag{1.22}$$

The offshore boundary is defined to be zero-flux, the onshore boundary has a constant transport gradient and the lateral boundaries are circular:

$$\begin{array}{rcl} c_{1,j,k}^{n+1} & = & 0 \\ c_{n_\mathrm{x}+1,j,k}^{n+1} & = & 2c_{n_\mathrm{x},j,k}^{n+1} - c_{n_\mathrm{x}-1,j,k}^{n+1} \\ c_{i,1,k}^{n+1} & = & c_{i,n_\mathrm{y}+1,k}^{n+1} \\ c_{i,n_\mathrm{y}+1,k}^{n+1} & = & c_{i,1,k}^{n+1} \end{array} \tag{1.23}$$

These boundary conditions can be combined with Equation (1.20), (1.21) and (1.22) into a linear system of equations:

$$\begin{bmatrix} A_1^0 & A_1^1 & \mathbf{0} & \cdots & \mathbf{0} & A_1^{n_\mathrm{y}+1} \\ A_2^{-1} & A_2^0 & \ddots & \ddots & & \mathbf{0} \\ \mathbf{0} & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \mathbf{0} \\ \mathbf{0} & & \ddots & \ddots & A_{n_\mathrm{y}}^0 & A_{n_\mathrm{y}}^1 \\ A_{n_\mathrm{y}+1}^{-n_\mathrm{y}-1} & \mathbf{0} & \cdots & \mathbf{0} & A_{n_\mathrm{y}+1}^{-1} & A_{n_\mathrm{y}+1}^0 \end{bmatrix} \begin{bmatrix} \vec{c}_1 \\ \vec{c}_2 \\ \vdots \\ \vdots \\ \vec{c}_{n_\mathrm{y}} \\ \vec{c}_{n_\mathrm{y}+1} \end{bmatrix} = \begin{bmatrix} \vec{y}_1 \\ \vec{y}_2 \\ \vdots \\ \vdots \\ \vec{y}_{n_\mathrm{y}} \\ \vec{y}_{n_\mathrm{y}+1} \end{bmatrix} \tag{1.24}$$

where each item in the matrix is again a matrix A_j^l and each item in the vectors is again a vector $\vec{c}_j$ and $\vec{y}_j$ respectively. The form of the matrix A_j^l depends on the diagonal index l and reads:

$$A_j^0 = \begin{bmatrix} 0 & 0 & 0 & 0 & \cdots & \cdots & 0 \\ a_{2,j}^{0,-1} & a_{2,j}^{0,0} & a_{2,j}^{0,1} & \ddots & & & \vdots \\ 0 & a_{3,j}^{0,-1} & a_{3,j}^{0,0} & a_{3,j}^{0,1} & \ddots & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & \ddots & a_{n_\mathrm{x}-1,j}^{0,-1} & a_{n_\mathrm{x}-1,j}^{0,0} & a_{n_\mathrm{x}-1,j}^{0,1} & 0 \\ \vdots & & & 0 & a_{n_\mathrm{x},j}^{0,-1} & a_{n_\mathrm{x},j}^{0,0} & a_{n_\mathrm{x},j}^{0,1} \\ 0 & \cdots & \cdots & 0 & 1 & -2 & 1 \end{bmatrix} \tag{1.25}$$

for $l = 0$ and

$$A_j^l = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 0 & a_{2,j}^{l,0} & \ddots & & & & \vdots \\ \vdots & \ddots & a_{3,j}^{l,0} & \ddots & & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & & \ddots & a_{n_\mathrm{x}-1,j}^{l,0} & \ddots & \vdots \\ \vdots & & & & \ddots & a_{n_\mathrm{x},j}^{l,0} & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & 1 \end{bmatrix} \tag{1.26}$$

for $l \neq 0$. The vectors $\vec{c}_{j,k}$ and $\vec{y}_{j,k}$ read:

$$\vec{c}_{j,k} = \begin{bmatrix} c_{1,j,k}^{n+1} \\ c_{2,j,k}^{n+1} \\ c_{3,j,k}^{n+1} \\ \vdots \\ c_{n_\mathrm{x}-1,j,k}^{n+1} \\ c_{n_\mathrm{x},j,k}^{n+1} \\ c_{n_\mathrm{x}+1,j,k}^{n+1} \end{bmatrix} \quad \text{and } \vec{y}_{j,k} = \begin{bmatrix} 0 \\ y_{2,j,k}^{n} \\ y_{3,j,k}^{n} \\ \vdots \\ y_{n_\mathrm{x}-1,j,k}^{n} \\ y_{n_\mathrm{x},j,k}^{n} \\ 0 \end{bmatrix}$$

n_x and n_y denote the number of spatial grid cells in x- and y-direction.

1.2.2 Implicit solver

The linear system defined in Equation (1.24) is solved by a sparse matrix solver for each sediment fraction separately in ascending order of grain size. Initially, the weights $\hat{w}_{i,j,k}^{n+1}$ are chosen according to the grain size distribution in the bed and the air. The sediment availability constraint is checked after each solve:

$$m_\mathrm{a} \geq \frac{\hat{w}_{i,j,k}^{n+1} c_{\mathrm{sat},i,j,k}^{n+1} - c_{i,j,k}^{n+1}}{T} \Delta t^n$$

If the constraint if violated, a new estimate for the weights is back-calculated following:

$$\hat{w}_{i,j,k}^{n+1} = \frac{c_{i,j,k}^{n+1} + m_\mathrm{a} \frac{\Delta t^n}{T}}{c_{\mathrm{sat},i,j,k}^{n+1}}$$

The system is solved again using the new weights. This procedure is repeated until a weight is found that does not violate the sediment availability constraint. If the time step is not too large, the procedure typically converges in only a few iterations. Finally, the weights of the larger grains are increased proportionally as to ensure that the sum of all weights remains unity. If no larger grains are defined, not enough sediment is available for transport and the grid cell is truly availability-limited. This situation should only occur occasionally as the weights in the next time step are computed based on the new bed composition and thus will be skewed towards the large fractions. If the situation occurs regularly, the time step is chosen too large compared to the rate of armoring.

1.2.3 Shear velocity threshold

The shear velocity threshold represents the influence of bed surface properties in the saturated sediment transport equation. The shear velocity threshold is computed for each grid cell and sediment fraction separately based on local bed surface properties, like moisture, roughness elements and salt content. For each bed surface property supported by the model a factor is computed to increase the initial shear velocity threshold:

$$u_{*\mathrm{th}} = f_{u_{*\mathrm{th}},\mathrm{M}} \cdot f_{u_{*\mathrm{th}},\mathrm{R}} \cdot f_{u_{*\mathrm{th}},\mathrm{S}} \cdot u_{*\mathrm{th},0} \tag{1.27}$$

The initial shear velocity threshold $u_{*\mathrm{th},0}$ [m/s] is computed based on the grain size following *[Bag37]*:

$$u_{*\mathrm{th},0} = A\sqrt{\frac{\rho_\mathrm{p} - \rho_\mathrm{a}}{\rho_\mathrm{a}} \cdot g \cdot d_\mathrm{n}}$$

where A [-] is an empirical constant, ρ_p [$\mathrm{kg/m^3}$] is the grain density, ρ_a [$\mathrm{kg/m^3}$] is the air density, g [$\mathrm{m/s^2}$] is the gravitational constant and d_n [m] is the nominal grain size of the sediment fraction.

Moisture content

The shear velocity threshold is updated based on moisture content following *[Bel64]*:

$$f_{u_{*\mathrm{th},\mathrm{M}}} = \max(1 \quad ; \quad 1.8 + 0.6 \cdot \log(p_\mathrm{g})) \tag{1.28}$$

where $f_{u_{*\mathrm{th},M}}$ [-] is a factor in Equation (1.27), p_g [-] is the geotechnical mass content of water, which is the percentage of water compared to the dry mass. The geotechnical mass content relates to the volumetric water content p_V [-] according to:

$$p_\mathrm{g} = \frac{p_\mathrm{V} \cdot \rho_\mathrm{w}}{\rho_\mathrm{p} \cdot (1-p)}$$

where ρ_w [$\mathrm{kg/m^3}$] and ρ_p [$\mathrm{kg/m^3}$] are the water and particle density respectively and p [-] is the porosity. Values for p_g smaller than 0.005 do not affect the shear velocity threshold (*[PT90]*). Values larger than 0.064 (or 10% volumetric content) cease transport (*[DF10]*), which is implemented as an infinite shear velocity threshold.

Exploratory model runs of the unsaturated soil with the HYDRUS1D (*[SimunekSejnavG98]*) hydrology model show that the increase of the volumetric water content to saturation is almost instantaneous with rising tide. The drying of the beach surface through infiltration shows an exponential decay. In order to capture this behavior the volumetric water content is implemented according to:

$$p_\mathrm{V}^{n+1} = \begin{cases} p & \text{if } \eta > z_\mathrm{b} \\ p_\mathrm{V}^n \cdot e^{\frac{\log(0.5)}{T_\mathrm{dry}} \cdot \Delta t^n} - E_\mathrm{v} \cdot \frac{\Delta t^n}{\Delta z} & \text{if } \eta \leq z_\mathrm{b} \end{cases} \tag{1.29}$$

where η [m+MSL] is the instantaneous water level, z_b [m+MSL] is the local bed elevation, p_V^n [-] is the volumetric water content in time step n, Δt^n [s] is the model time step and Δz is the bed composition layer thickness. T_dry [s] is the beach drying time scale, defined as the time in which the beach moisture content halves. E_v [m/s] is the evaporation rate that is implemented through an adapted version of the Penman equation (*[Shu93]*):

$$E_\mathrm{v} = \frac{m_\mathrm{v} \cdot R_\mathrm{n} + 6.43 \cdot \gamma_\mathrm{v} \cdot (1 + 0.536 \cdot u_2) \cdot \delta e}{\lambda_\mathrm{v} \cdot (m_\mathrm{v} + \gamma_\mathrm{v})} \cdot 9 \cdot 10^7 \tag{1.30}$$

where m_v [kPa/K] is the slope of the saturation vapor pressure curve, R_n [$\mathrm{MJ/m^2/day}$] is the net radiance, γ_v [kPa/K] is the psychrometric constant, u_2 [m/s] is the wind speed at 2 m above the bed, δe [kPa] is the vapor pressure deficit (related to the relative humidity) and λ_v [MJ/kg] is the latent heat vaporization. To obtain an evaporation rate in [m/s], the original formulation is multiplied by $9 \cdot 10^7$.

Roughness elements

The shear velocity threshold is updated based on the presence of roughness elements following *[RGL93]*:

$$f_{u_{*\mathrm{th},R}} = \sqrt{(1 - m \cdot \sum_{k=k_0}^{n_k} \hat{w}_k^\mathrm{bed})(1 + \frac{m\beta}{\sigma} \cdot \sum_{k=k_0}^{n_k} \hat{w}_k^\mathrm{bed})}$$

by assuming:

$$\lambda = \frac{\sum_{k=k_0}^{n_k} \hat{w}_k^\mathrm{bed}}{\sigma}$$

where $f_{u_{*\mathrm{th},R}}$ [-] is a factor in Equation (1.27), k_0 is the sediment fraction index of the smallest non-erodible fraction in current conditions and n_k is the number of sediment fractions defined. The implementation is discussed in detail in section ref{sec:roughness}.

Salt content

The shear velocity threshold is updated based on salt content following *[NE81]*:

$$f_{u_{*\mathrm{th},S}} = 1.03 \cdot \exp(0.1027 \cdot p_\mathrm{s})$$

where $f_{u_{*\mathrm{th},S}}$ [-] is a factor in Equation (1.27) and p_s [-] is the salt content [mg/g]. Currently, no model is implemented that predicts the instantaneous salt content. The spatial varying salt content needs to be specified by the user, for example through the BMI interface.

1.2.4 Basic Model Interface (BMI)

A Basic Model Interface (BMI, *[PHN13]*) is implemented that allows interaction with the model during run time. The model can be implemented as a library within a larger framework as the interface exposes the initialization, finalization and time stepping routines. As a convenience functionality the current implementation supports the specification of a callback function. The callback function is called at the start of each time step and can be used to exchange data with the model, e.g. update the topography from measurements.

An example of a callback function, that is referenced in the model input file or through the model command-line options as `callback.py:update`, is:

```
import numpy as np

def update(model):
  val = model.get_var('zb')
  val_new = val.copy()
  val_new[:,:] = np.loadtxt('measured_topography.txt')
  model.set_var('zb', val_new)
```

Bibliography

1.3 Source code documentation

1.3.1 Model classes

The AeoLiS model is based on two main model classes: *AeoLiS* and *AeoLiSRunner*. The former is the actual, low-level, BMI-compatible class that implements the basic model functions and numerical schemes. The latter is a convenience class that implements a time loop, netCDF4 output, a progress indicator and a callback function that allows the used to interact with the model during runtime.

The additional *WindGenerator* class to generate realistic wind time series is available from the same module.

AeoLiS

class model.**AeoLiS**(*configfile*)
AeoLiS model class

AeoLiS is a process-based model for simulating supply-limited aeolian sediment transport. This model class is compatible with the Basic Model Interface (BMI) and provides basic model operations, like initialization, time stepping, finalization and data exchange. For higher level operations, like a progress indicator and netCDF4 output is refered to the AeoLiS model runner class, see *AeoLiSRunner*.

Examples

```
>>> with AeoLiS(configfile='aeolis.txt') as model:
>>>     while model.get_current_time() <= model.get_end_time():
>>>         model.update()
```

```
>>> model = AeoLiS(configfile='aeolis.txt')
>>> model.initialize()
>>> zb = model.get_var('zb')
>>> model.set_var('zb', zb + 1)
>>> for i in range(10):
>>>     model.update(60.) # step 60 seconds forward
>>> model.finalize()
```

__init__(*configfile*)
Initialize class

Parameters **configfile** (*str*) – Model configuration file. See *read_configfile()*.

crank_nicolson()
Convenience function for implicit solver based on Crank-Nicolson scheme

See also:

model.AeoLiS.solve()

static dimensions(*var=None*)
Static method that returns named dimensions of all spatial grids

Parameters **var** (*str, optional*) – Name of spatial grid

Returns Tuple with named dimensions of requested spatial grid or dictionary with all named dimensions of all spatial grids. Returns nothing if requested spatial grid is not defined.

Return type tuple or dict

euler_backward()
Convenience function for implicit solver based on Euler backward scheme

See also:

model.AeoLiS.solve()

euler_forward()
Convenience function for explicit solver based on Euler forward scheme

See also:

model.AeoLiS.solve()

finalize()
Finalize model

get_count(*name*)
Get counter value

Parameters **name** (*str*) – Name of counter

get_current_time()

Returns Current simulation time

Return type float

get_end_time()

Returns Final simulation time

Return type float

get_start_time()

Returns Initial simulation time

Return type float

get_var(*var*)

Returns spatial grid or model configuration parameter

If the given variable name matches with a spatial grid, the spatial grid is returned. If not, the given variable name is matched with a model configuration parameter. If a match is found, the parameter value is returned. Otherwise, nothing is returned.

Parameters **var** (*str*) – Name of spatial grid or model configuration parameter

Returns Spatial grid or model configuration parameter

Return type np.ndarray or int, float, str or list

Examples

```
>>> # returns bathymetry grid
... model.get_var('zb')
```

```
>>> # returns simulation duration
... model.get_var('tstop')
```

See also:

model.AeoLiS.set_var()

get_var_count()

Returns Number of spatial grids

Return type int

get_var_name(*i*)

Returns name of spatial grid by index (in alphabetical order)

Parameters **i** (*int*) – Index of spatial grid

Returns Name of spatial grid or -1 in case index exceeds the number of grids

Return type str or -1

get_var_rank(*var*)

Returns rank of spatial grid

Parameters **var** (*str*) – Name of spatial grid

Returns Rank of spatial grid or -1 if not found

Return type int

get_var_shape(*var*)
Returns shape of spatial grid

Parameters **var** (*str*) – Name of spatial grid

Returns Dimensions of spatial grid or -1 if not found

Return type tuple or int

get_var_type(*var*)
Returns variable type of spatial grid

Parameters **var** (*str*) – Name of spatial grid

Returns Variable type of spatial grid or -1 if not found

Return type str or int

initialize()
Initialize model

Read model configuration file and initialize parameters and spatial grids dictionary and load bathymetry and bed composition.

set_timestep(*dt=-1.0*)
Determine optimal time step

If no time step is given the optimal time step is determined. For explicit numerical schemes the time step is based in the Courant-Frierichs-Lewy (CFL) condition. For implicit numerical schemes the time step specified in the model configuration file is used. Alternatively, a preferred time step is given that is maximized by the CFL condition in case of an explicit numerical scheme.

Returns True except when:

1. No time step could be determined, for example when there is no wind and the numerical scheme is explicit. In this case the time step is set arbitrarily to one second.

2. Or when the time step is smaller than -1. In this case the time is updated with the absolute value of the time step, but no model execution is performed. This funcionality can be used to skip fast-forward in time.

Parameters **df** (*float, optional*) – Preferred time step

Returns False if determination of time step was unsuccessful, True otherwise

Return type bool

set_var(*var*, *val*)
Sets spatial grid or model configuration parameter

If the given variable name matches with a spatial grid, the spatial grid is set. If not, the given variable name is matched with a model configuration parameter. If a match is found, the parameter value is set. Otherwise, nothing is set.

Parameters

- **var** (*str*) – Name of spatial grid or model configuration parameter
- **val** (*np.ndarray or int, float, str or list*) – Spatial grid or model configuration parameter

Examples

```
>>> # set bathymetry grid
... model.set_var('zb', np.array([[0.,0., ... ,0.]]))
```

```
>>> # set simulation duration
... model.set_var('tstop', 3600.)
```

See also:

model.AeoLiS.get_var()

set_var_index(*i*, *val*)
Set spatial grid by index (in alphabetical order)

Parameters

- **i** (*int*) – Index of spatial grid
- **val** (*np.ndarray*) – Spatial grid

solve(*alpha=0.5*, *beta=1.0*)
Implements the explicit Euler forward, implicit Euler backward and semi-implicit Crank-Nicolson numerical schemes

Determines weights of sediment fractions, sediment pickup and instantaneous sediment concentration. Returns a partial spatial grid dictionary that can be used to update the global spatial grid dictionary.

Parameters

- **alpha** (*float, optional*) – Implicitness coefficient (0.0 for Euler forward, 1.0 for Euler backward or 0.5 for Crank-Nicolson, default=0.5)
- **beta** (*float, optional*) – Centralization coefficient (1.0 for upwind or 0.5 for centralized, default=1.0)

Returns Partial spatial grid dictionary

Return type dict

Examples

```
>>> model.s.update(model.solve(alpha=1., beta=1.) # euler backward
```

```
>>> model.s.update(model.solve(alpha=.5, beta=1.) # crank-nicolson
```

See also:

model.AeoLiS.euler_forward(), *model.AeoLiS.euler_backward()*, *model.AeoLiS.crank_nicolson()*, *transport.compute_weights()*, *transport.renormalize_weights()*

update(*dt=-1*)
Time stepping function

Takes a single step in time. Interpolates wind and hydrodynamic time series to the current time, updates the soil moisture, mixes the bed due to wave action, computes wind velocity threshold and the equilibrium sediment transport concentration. Subsequently runs one of the available numerical schemes to compute the instantaneous sediment concentration and pickup for the next time step and updates the bed accordingly.

For explicit schemes the time step is maximized by the Courant-Friedrichs-Lewy (CFL) condition. See *set_timestep()*.

Parameters **dt** (*float, optional*) – Time step in seconds. The time step specified in the model configuration file is used in case dt is smaller than zero. For explicit numerical schemes the time step is maximized by the CFL confition.

AeoLiSRunner

class model.**AeoLiSRunner**(*configfile='aeolis.txt'*)

AeoLiS model runner class

This runner class is a convenience class for the BMI-compatible AeoLiS model class (*AeoLiS()*). It implements a time loop, a progress indicator and netCDF4 output. It also provides the definition of a callback function that can be used to interact with the AeoLiS model during runtime.

The command-line function `aeolis` is available that uses this class to start an AeoLiS model run.

Examples

```
>>> # run with default settings
... AeoLiSRunner().run()
```

```
>>> AeoLiSRunner(configfile='aeolis.txt').run()
```

```
>>> model = AeoLiSRunner(configfile='aeolis.txt')
>>> model.run(callback=lambda model: model.set_var('zb', zb))
```

```
>>> model.run(callback='bar.py:add_bar')
```

See also:

console.aeolis

__init__(*configfile='aeolis.txt'*)

Initialize class

Reads model configuration file without parsing all referenced files for the progress indicator and netCDF output. If no configuration file is given, the default settings are used.

Parameters **configfile** (*str, optional*) – Model configuration file. See *read_configfile()*.

dump_restartfile()

Dump model state to restart file

get_statistic(*var*, *stat='avg'*)

Return statistic of spatial grid

Parameters

- **var** (*str*) – Name of spatial grid
- **stat** (*str*) – Name of statistic (avg, sum, var, min or max)

Returns Statistic of spatial grid

Return type numpy.ndarray

get_var(*var*, *clear=True*)
Returns spatial grid, statistic or model configuration parameter

Overloads the *get_var()* function and extends it with the functionality to return statistics on spatial grids by adding a postfix to the variable name (e.g. Ct_avg). Supported statistics are avg, sum, var, min and max.

Parameters

- **var** (*str*) – Name of spatial grid or model configuration parameter. Spatial grid name can be extended with a postfix to request a statistic (_avg, _sum, _var, _min or _max).
- **clear** (*bool*) – Clear output statistics afterwards.

Returns Spatial grid, statistic or model configuration parameter

Return type np.ndarray or int, float, str or list

Examples

```
>>> # returns average sediment concentration
... model.get_var('Ct_avg')
```

```
>>> # returns variance in wave height
... model.get_var('Hs_var')
```

See also:

model.AeoLiS.get_var()

initialize()
Initialize model

Overloads the *initialize()* function, but also initializes output statistics.

load_hotstartfiles()
Load model state from hotstart files

Hotstart files are plain text representations of model state variables that can be used to hotstart the (partial) model state. Hotstart files should have the name of the model state variable it contains and have the extension *.hotstart*. Hotstart files differ from restart files in that restart files contain entire model states and are pickled Python objects.

See also:

model.AeoLiSRunner.load_restartfile()

load_restartfile(*restartfile*)
Load model state from restart file

Parameters **restartfile** (*str*) – Path to previously written restartfile.

output_clear()
Clears output statistics dictionary

Creates a matrix for minimum, maximum, variance and summed values for each output variable and sets the time step counter to zero.

output_init()
Initialize netCDF4 output file and output statistics dictionary

output_update()
Updates output statistics dictionary

Updates matrices with minimum, maximum, variance and summed values for each output variable with current spatial grid values and increases time step counter with one.

output_write()
Appends output to netCDF4 output file

If the time since the last output is equal or larger than the set output interval, append current output to the netCDF4 output file. Computes the average and variance values based on available output statistics and clear output statistics dictionary.

parse_callback(*callback*)
Parses callback definition and returns function

The callback function can be specified in two formats:

- As a native Python function
- As a string refering to a Python script and function, separated by a colon (e.g. `example/callback.py:function`)

Parameters **callback** (*str or function*) – Callback definition

Returns Python callback function

Return type function

print_params()
Print model configuration parameters to screen

print_progress(*fraction=0.1*, *min_interval=1.0*, *max_interval=60.0*)
Print progress to screen

Parameters

- **fraction** (*float, optional*) – Fraction of simulation at which to print progress (default: 10%)
- **min_interval** (*float, optional*) – Minimum time in seconds between subsequent progress prints (default: 1s)
- **max_interval** (*float, optional*) – Maximum time in seconds between subsequent progress prints (default: 60s)

print_stats()
Print model run statistics to screen

run(*callback=None*, *restartfile=None*)
Start model time loop

Changes current working directory to the model directory, prints model configuration parameters and progress indicator to the screen, writes netCDF4 output and calls a callback function upon request.

Parameters

- **callback** (*str or function*) – The callback function is called at the start of every single time step and takes the AeoLiS model object as input. The callback function can be used to interact with the model during simulation (e.g. update the bed with new measurements). See for syntax *parse_callback()*.
- **restartfile** (*str*) – Path to previously written restartfile. The model state is loaded from this file after initialization of the model.

See also:

model.AeoLiSRunner.parse_callback()

set_configfile(*configfile*)
Set model configuration file name

set_params(***kwargs*)
Set model configuration parameters

update(*dt=-1*)
Time stepping function

Overloads the *update()* function, but also updates output statistics and clears output statistics upon request.

Parameters **dt** (*float, optional*) – Time step in seconds.

write_params()
Write updated model configuration to configuration file

Creates a backup in case the model configration file already exists.

See also:

inout.backup()

WindGenerator

class model.**WindGenerator**(*mean_speed=9.0*, *max_speed=30.0*, *dt=60.0*, *n_states=30*, *shape=2.0*, *scale=2.0*)
Wind velocity time series generator

Generates a random wind velocity time series with given mean and maximum wind speed, duration and time resolution. The wind velocity time series is generated using a Markov Chain Monte Carlo (MCMC) approach based on a Weibull distribution. The wind time series can be written to an AeoLiS-compatible wind input file assuming a constant wind direction of zero degrees.

The command-line function `aeolis-wind` is available that uses this class to generate AeoLiS wind input files.

Examples

```
>>> wind = WindGenerator(mean_speed=10.).generate(duration=24*3600.)
>>> wind.write_time_series('wind.txt')
>>> wind.plot()
>>> wind.hist()
```

See also:

console.wind

1.3.2 Physics modules

Bathymetry and bed composition

bed.**initialize**(*s*, *p*)
Initialize bathymetry and bed composition

Initialized bathymetry, computes cell sizes and orientation, bed layer thickness and bed composition.

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

bed.**mixtoplayer**(*s*, *p*)
Mix grain size distribution in top layer of the bed

Simulates mixing of the top layers of the bed by wave action. The wave action is represented by a local wave height maximized by a maximum wave hieght over depth ratio `gamma`. The mixing depth is a fraction of the local wave height indicated by `facDOD`. The mixing depth is used to compute the number of bed layers that should be included in the mixing. The grain size distribution in these layers is then replaced by the average grain size distribution over these layers.

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

bed.**prevent_negative_mass**(*m*, *dm*, *pickup*)
Handle situations in which negative mass may occur due to numerics

Negative mass may occur by moving sediment to lower layers down to accomodate deposition of sediments. In particular two cases are important:

1. A net deposition cell has some erosional fractions.

   In this case the top layer mass is reduced according to the existing sediment distribution in the layer to accomodate deposition of fresh sediment. If the erosional fraction is subtracted afterwards, negative values may occur. Therefore the erosional fractions are subtracted from the top layer beforehand in this function. An equal mass of deposition fractions is added to the top layer in order to keep the total layer mass constant. Subsequently, the distribution of the sediment to be moved to lower layers is determined and the remaining deposits are accomodated.

2. Deposition is larger than the total mass in a layer.

   In this case a non-uniform distribution in the bed may also lead to negative values as the abundant fractions are reduced disproportionally as sediment is moved to lower layers to accomodate the deposits. This function fills the top layers entirely with fresh deposits and moves the existing sediment down such that the remaining deposits have a total mass less than the total bed layer mass. Only the remaining deposits are fed to the routine that moves sediment through the layers.

Parameters

- **m** (*np.ndarray*) – Sediment mass in bed (nx*ny, nl, nf)
- **dm** (*np.ndarray*) – Total sediment mass exchanged between layers (nx*ny, nf)
- **pickup** (*np.ndarray*) – Sediment pickup (nx*ny, nf)

Returns

- *np.ndarray* – Sediment mass in bed (nx*ny, nl, nf)
- *np.ndarray* – Total sediment mass exchanged between layers (nx*ny, nf)

- *np.ndarray* – Sediment pickup (nx*ny, nf)

Note: The situations handled in this function can also be prevented by reducing the time step, increasing the layer mass or increasing the adaptation time scale.

`bed.`**`update`**(*s*, *p*)
Update bathymetry and bed composition

Update bed composition by moving sediment fractions between bed layers. The total mass in a single bed layer does not change as sediment removed from a layer is repleted with sediment from underlying layers. Similarly, excess sediment added in a layer is moved to underlying layers in order to keep the layer mass constant. The lowest bed layer exchanges sediment with an infinite sediment source that follows the original grain size distribution as defined in the model configuration file by `grain_size` and `grain_dist`. The bathymetry is updated following the cummulative erosion/deposition over the fractions if `bedupdate` is `True`.

Parameters

- **`s`** (*`dict`*) – Spatial grids
- **`p`** (*`dict`*) – Model configuration parameters

Returns Spatial grids

Return type dict

Wind velocity and direction

`wind.`**`get_velocity_at_height`**(*u*, *z*, *z0*, *z1=None*)
Compute shear velocity from wind velocity following Prandl-Karman's Law of the Wall

Parameters

- **`u`** (*`numpy.ndarray`*) – Spatial wind field
- **`z`** (*`float`*) – Height above bed where `u` is measured
- **`z0`** (*`float`*) – Roughness length
- **`z1`** (*`float, optional`*) – Height above bed for which to return wind speeds. Returns wind shear if not given.

Returns Array of size `u` with wind speeds at height `z1`

Return type numpy.ndarray

`wind.`**`initialize`**(*s*, *p*)
Initialize wind model

`wind.`**`interpolate`**(*s*, *p*, *t*)
Interpolate wind velocity and direction to current time step

Interpolates the wind time series for velocity and direction to the current time step. The cosine and sine of the direction angle are interpolated separately to prevent zero-crossing errors. The wind velocity is decomposed in two grid components based on the orientation of each individual grid cell. In case of a one-dimensional model only a single positive component is used.

Parameters

- **`s`** (*`dict`*) – Spatial grids
- **`p`** (*`dict`*) – Model configuration parameters

- **t** (*float*) – Current time

Returns Spatial grids

Return type dict

class shear.**WindShear**(*x*, *y*, *z*, *dx=1.0*, *dy=1.0*, *buffer_width=100.0*, *buffer_relaxation=None*, *L=100.0*, *z0=0.001*, *l=10.0*)

Class for computation of 2DH wind shear perturbations over a topography.

The class implements a 2D FFT solution to the wind shear perturbation on curvilinear grids. As the FFT solution is only defined on an equidistant rectilinear grid with circular boundary conditions that is aligned with the wind direction, a rotating computational grid is automatically defined for the computation. The computational grid is extended in all directions using a logistic sigmoid function as to ensure full coverage of the input grid for all wind directions, circular boundaries and preservation of the alongshore uniformity. An extra buffer distance can be used as to minimize the disturbence from the borders in the input grid. The results are interpolated back to the input grid when necessary.

Frequencies related to wave lengths smaller than a computational grid cell are filtered from the 2D spectrum of the topography using a logistic sigmoid tapering. The filtering aims to minimize the disturbance as a result of discontinuities in the topography that may physically exists, but cannot be solved for in the computational grid used.

Example

```
>>> w = WindShear(x, y, z)
>>> w(u0=10., udir=30.).add_shear(taux, tauy)
```

Notes

To do:

- **Actual resulting values are still to be compared with the results** from Kroy et al. (2002)
- Grid interpolation can still be optimized
- Separation bubble is still to be implemented
- Avalanching is still to be implemented

__call__(*u0*, *udir*)

Compute wind shear for given wind speed and direction

Parameters

- **u0** (*float*) – Free-flow wind speed
- **udir** (*float*) – Wind direction in degrees

add_shear(*taux*, *tauy*)

Add wind shear perturbations to a given wind shear field

Parameters

- **taux** (*numpy.ndarray*) – Wind shear in x-direction
- **tauy** (*numpy.ndarray*) – Wind shear in y-direction

Returns

- **taux** (*numpy.ndarray*) – Wind shear including perturbations in x-direction

- **tauy** (*numpy.ndarray*) – Wind shear including perturbations in y-direction

compute_shear(*u0*, *nfilter=(1.0*, *2.0)*)
Compute wind shear perturbation for given free-flow wind speed on computational grid

Parameters

- **u0** (*float*) – Free-flow wind speed
- **nfilter** (*2-tuple*) – Wavenumber range used for logistic sigmoid filter. See `filter_highfrequencies()`

filter_highfrequenies(*kx*, *ky*, *hs*, *nfilter=(1*, *2)*, *p=0.01*)
Filter high frequencies from a 2D spectrum

A logistic sigmoid filter is used to taper higher frequencies from the 2D spectrum. The range over which the sigmoid runs from 0 to 1 with a precision `p` is given by the 2-tuple `nfilter`. The range is defined as wavenumbers in terms of gridcells, i.e. a value 1 corresponds to a wave with length `dx`.

Parameters

- **kx** (*numpy.ndarray*) – Wavenumbers in x-direction
- **ky** (*numpy.ndarray*) – Wavenumbers in y-direction
- **hs** (*numpy.ndarray*) – 2D spectrum
- **nfilter** (*2-tuple*) – Wavenumber range used for logistic sigmoid filter
- **p** (*float*) – Precision of sigmoid range definition

Returns **hs** – Filtered 2D spectrum

Return type numpy.ndarray

static get_borders(*x*)
Returns borders of a grid as one-dimensional array

static get_exact_grid(*xmin*, *xmax*, *ymin*, *ymax*, *dx*, *dy*)
Returns a grid with given gridsizes approximately within given bounding box

get_shear()
Returns wind shear perturbation field

Returns

- **dtaux** (*numpy.ndarray*) – Wind shear perturbation in x-direction
- **dtauy** (*numpy.ndarray*) – Wind shear perturbation in y-direction

get_sigmoid(*x*)
Get sigmoid function value

Get bed level multiplication factor in buffer area based on buffer specificationa and distance to input grid boundary.

Parameters **x** (*float or numpy.ndarray*) – Distance(s) to input grid boundary

Returns Bed level multiplication factor (z = factor * z_boundary)

Return type float or numpy.ndarray

interpolate(*x*, *y*, *z*, *xi*, *yi*)
Interpolate a grid onto another grid

static interpolate_projected_point(*a*, *b*, *p*)
Project point to line segment and return distance and interpolated value

Parameters

- **a** (*iterable*) – Start vector for line segment
- **b** (*iterable*) – End vector for line segment
- **p** (*iterable*) – Point vector to be projected

Returns

- **d** (*numpy.ndarray*) – Distance from point p to projected point q
- **z** (*float*) – Interpolated value at projected point q

plot(*ax=None*, *cmap='Reds'*, *stride=10*, *computational_grid=False*, ***kwargs*)
Plot wind shear perturbation

Parameters

- **ax** (*matplotlib.pyplot.Axes, optional*) – Axes to plot onto
- **cmap** (*matplotlib.cm.Colormap or string, optional*) – Colormap for topography (default: Reds)
- **stride** (*int, optional*) – Stride to apply to wind shear vectors (default: 10)
- **computational_grid** (*bool, optional*) – Plot on computational grid rather than input grid (default: False)
- **kwargs** (*dict*) – Additional arguments to `matplotlib.pyplot.quiver()`

Returns **ax** – Axes used for plotting

Return type matplotlib.pyplot.Axes

populate_computational_grid(*alpha*)
Interpolate input topography to computational grid

Rotates computational grid to current wind direction and interpolates the input topography to the rotated grid. Any grid cells that are not covered by the input grid are filled using a sigmoid function.

Parameters **alpha** (*float*) – Rotation angle in degrees

static rotate(*x*, *y*, *alpha*, *origin=(0, 0)*)
Rotate a matrix over given angle around given origin

set_computational_grid()
Define computational grid

The computational grid is square with dimensions equal to the diagonal of the bounding box of the input grid, plus twice the buffer width.

set_topo(*z*)
Update topography

Parameters **z** (*numpy.ndarray*) – 2D array with topography of input grid

Wind velocity threshold

threshold.**compute**(*s*, *p*)
Compute wind velocity threshold based on bed surface properties

Computes wind velocity threshold based on grain size fractions, bed slope, soil moisture content, air humidity and the presence of roughness elements. All bed surface properties increase the current wind velocity threshold,

except for the grain size fractions. Therefore, the computation is initialized by the grain size fractions and subsequently altered by the other bed surface properties.

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

See also:

compute_grainsize(), *compute_bedslope()*, *compute_moisture()*, *compute_humidity()*, *compute_roughness()*

threshold.**compute_bedslope**(*s*, *p*)

Modify wind velocity threshold based on bed slopes following Dyer (1986)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

threshold.**compute_grainsize**(*s*, *p*)

Compute wind velocity threshold based on grain size fractions following Bagnold (1937)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

threshold.**compute_humidity**(*s*, *p*)

Modify wind velocity threshold based on air humidity following Arens (1996)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

threshold.**compute_moisture**(*s*, *p*)

Modify wind velocity threshold based on soil moisture content following Belly (1964) or Hotta (1984)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

threshold.**compute_roughness**(*s*, *p*)

Modify wind velocity threshold based on the presence of roughness elements following Raupach (1993)

Raupach (1993) presents the following amplification factor for the shear velocity threshold due to the presence of roughness elements.

$$R_t = \frac{u_{*,th,s}}{u_{*,th,r}} = \sqrt{\frac{\tau_s''}{\tau}} = \frac{1}{\sqrt{(1 - m\sigma\lambda)\,(1 + m\beta\lambda)}}$$

m is a constant smaller or equal to unity that accounts for the difference between the average stress on the bed surface τ_s and the maximum stress on the bed surface τ_s''.

β is the stress partition coefficient defined as the ratio between the drag coefficient of the roughness element itself C_r and the drag coefficient of the bare surface without roughness elements C_s.

σ is the shape coefficient defined as the basal area divided by the frontal area: $\frac{A_b}{A_f}$. For hemispheres $\sigma = 2$, for spheres $\sigma = 1$.

λ is the roughness density defined as the number of elements per surface area $\frac{n}{S}$ multiplied by the frontal area of a roughness element A_f, also known as the frontal area index:

$$\lambda = \frac{nbh}{S} = \frac{nA_f}{S}$$

If multiplied by σ the equation simplifies to the mass fraction of non-erodible elements:

$$\sigma\lambda = \frac{nA_b}{S} = \sum_{k=n_0}^{n_k} \hat{w}_k^{\mathrm{bed}}$$

where k is the fraction index, n_0 is the smallest non-erodible fraction, n_k is the largest non-erodible fraction and $\hat{w}_k^{\mathrm{bed}}$ is the mass fraction of sediment fraction k. It is assumed that the fractions are ordered by increasing size.

Substituting the derivation in the Raupach (1993) equation gives the formulation implemented in this function:

$$u_{*,th,r} = u_{*,th,s} * \sqrt{\left(1 - m \sum_{k=n_0}^{n_k} \hat{w}_k^{\mathrm{bed}}\right)\left(1 + m\frac{\beta}{\sigma} \sum_{k=n_0}^{n_k} \hat{w}_k^{\mathrm{bed}}\right)}$$

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

threshold.**compute_salt**(*s*, *p*)

Modify wind velocity threshold based on salt content following Nickling and Ecclestone (1981)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

Tides, meteorology and soil moisture content

hydro.**interpolate**(*s*, *p*, *t*)
Interpolate hydrodynamic and meteorological conditions to current time step

Interpolates the hydrodynamic and meteorological time series to the current time step, if available. Meteorological parameters are stored as dictionary rather than a single value.

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters
- **t** (*float*) – Current time

Returns Spatial grids

Return type dict

hydro.**saturation_pressure**(*T*)
Compute saturation pressure based on air temperature

Parameters **T** (*float*) – Air temperature in degrees Celcius

Returns Saturation pressure

Return type float

hydro.**update**(*s*, *p*, *dt*)
Update soil moisture content

Updates soil moisture content in all cells. The soil moisure content in flooded cells is set to the porosity. The soil moisture content in non-flooded cells is decreased by simulating infiltration using an exponential decay function with a rate `F` following:

$$M = M \cdot e^{-F \cdot \Delta t}$$

Cells are considered flooded if the water depth is larger than `eps`. Aditionally, is meteorological conditions are provided the soil moisture content is decreased by simulating evaporation following the Penman equation:

$$M = M - \frac{m \cdot R + \gamma \cdot 6.43 \cdot (1 + 0.536 \cdot u]) \cdot \delta}{\lambda \cdot (m + \gamma)}$$

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters
- **dt** (*float*) – Current time step

Returns Spatial grids

Return type dict

hydro.**vaporation_pressure_slope**(*T*)
Compute vaporation pressure slope based on air temperature

Parameters **T** (*float*) – Air temperature in degrees Celcius

Returns Vaporation pressure slope

Return type float

Sediment transport

transport.**compute_weights**(*s*, *p*)
Compute weights for sediment fractions

Multi-fraction sediment transport needs to weigh the transport of each sediment fraction to prevent the sediment transport to increase with an increasing number of sediment fractions. The weighing is not uniform over all sediment fractions, but depends on the sediment availibility in the air and the bed and the bed interaction parameter `bi`.

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Array with weights for each sediment fraction

Return type numpy.ndarray

transport.**equilibrium**(*s*, *p*)
Compute equilibrium sediment concentration following Bagnold (1937)

Parameters

- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters

Returns Spatial grids

Return type dict

transport.**renormalize_weights**(*w*, *ix*)
Renormalizes weights for sediment fractions

Renormalizes weights for sediment fractions such that the sum of all weights is unity. To ensure that the erosion of specific fractions does not exceed the sediment availibility in the bed, the normalization only modifies the weights with index equal or larger than `ix`.

Parameters

- **w** (*numpy.ndarray*) – Array with weights for each sediment fraction
- **ix** (*int*) – Minimum index to be modified

Returns Array with weights for each sediment fraction

Return type numpy.ndarray

1.3.3 Helper modules

Input/Output

inout.**backup**(*fname*)
Creates a backup file of the provided file, if it exists

inout.**check_configuration**(*p*)
Check model configuration validity

Checks if required parameters are set and if the references files for bathymetry, wind, tide and meteorological input are valid. Throws an error if one or more requirements are not met.

Parameters **p** (*dict*) – Model configuration dictionary with parsed files

See also:

read_configfile()

inout.**get_backupfilename**(*fname*)
Returns a non-existing backup filename

inout.**parse_value**(*val*, *parse_files=True*, *force_list=False*)
Casts a string to the most appropriate variable type

Parameters

- **val** (*str*) – String representation of value
- **parse_files** (*bool*) – If True, files referred to by string parameters are parsed by `numpy.loadtxt`
- **force_list** – If True, interpret the value as a list, even if it consists of a single value

Returns Casted value

Return type misc

Examples

```
>>> type(parse_value('T'))
    bool
>>> type(parse_value('F'))
    bool
>>> type(parse_value('123'))
    int
>>> type(parse_value('123.2'))
    float
>>> type(parse_value('euler_forward'))
    str
>>> type(parse_value(''))
    NoneType
>>> type(parse_value('zb zs Ct'))
    numpy.ndarray
>>> type(parse_value('zb', force_list=True))
    numpy.ndarray
>>> type(parse_value('0.1 0.2 0.3')[0])
    float
>>> type(parse_value('wind.txt'), parse_files=True)
    numpy.ndarray
>>> type(parse_value('wind.txt'), parse_files=False)
    str
```

inout.**read_configfile**(*configfile*, *parse_files=True*, *load_defaults=True*)
Read model configuration file

Updates default model configuration based on a model configuration file. The model configuration file should be a text file with one parameter on each line. The parameter name and value are seperated by an equal sign (=). Any lines that start with a percent sign (%) or do not contain an equal sign are omitted.

Parameters are casted into the best matching variable type. If the variable type is `str` it is optionally interpreted as a filename. If the corresponding file is found it is parsed using the `numpy.loadtxt` function.

Parameters

- **configfile** (*str*) – Model configuration file

- **parse_files** (*bool*) – If True, files referred to by string parameters are parsed
- **load_defaults** (*bool*) – If True, default settings are loaded and overwritten by the settings from the configuration file

Returns Dictionary with casted and optionally parsed model configuration parameters

Return type dict

See also:

write_configfile(), *check_configuration()*

inout.**write_configfile**(*configfile*, *p=None*)

Write model configuration file

Writes model configuration to file. If no model configuration is given, the default configuration is written to file. Any parameters with a name ending with *_file* and holding a matrix are treated as separate files. The matrix is then written to an ASCII file using the `numpy.savetxt` function and the parameter value is replaced by the name of the ASCII file.

Parameters

- **configfile** (*str*) – Model configuration file
- **p** (*dict, optional*) – Dictionary with model configuration parameters

Returns Dictionary with casted and optionally parsed model configuration parameters

Return type dict

See also:

read_configfile()

netCDF4 output

netcdf.**append**(*outputfile*, *variables*)

Append variables to existing netCDF4 output file

Increments the time axis length with one and appends the provided spatial grids along the time axis. The `variables` dictionary should at least have the `time` field indicating the current simulation time. The CF time bounds are updated accordingly.

Parameters

- **outputfile** (*str*) – Name of netCDF4 output file
- **variables** (*dict*) – Dictionary with spatial grids and time

Examples

```
>>> netcdf.append('aeolis.nc', {'time', 3600.,
...                             'Ct', np.array([[0.,0., ... ,0.]]),
...                             'Cu', np.array([[1.,1., ... ,1.]]))
```

See also:

set_bounds()

netcdf.**dump**(*outputfile*, *dumpfile*, *var='mass'*, *ix=-1*)
Dumps time slice from netCDF4 output file to ASCII file

This function can be used to use a specific time slice from a netCDF4 output file as input file for another AeoLiS model run. For example, the bed composition from a spinup run can be used as initial composition for other runs reducing the spinup time.

Parameters

- **outputfile** (*str*) – Name of netCDF4 output file
- **dumpfile** (*str*) – Name of ASCII dump file
- **var** (*str, optional*) – Name of spatial grid to be dumped (default: mass)
- **ix** (*int*) – Time slice index to be dumped (default: -1)

Examples

```
>>> # use bedcomp_file = bedcomp.txt in model configuration file
... netcdf.dump('aeolis.nc', 'bedcomp.txt', var='mass')
```

netcdf.**initialize**(*outputfile*, *outputvars*, *s*, *p*, *dimensions*)
Create empty CF-compatible netCDF4 output file

Parameters

- **outputfile** (*str*) – Name of netCDF4 output file
- **outputvars** (*dictionary*) – Spatial grids to be written to netCDF4 output file
- **s** (*dict*) – Spatial grids
- **p** (*dict*) – Model configuration parameters
- **dimensions** (*dict*) – Dictionary that specifies a tuple with the named dimensions for each spatial grid (e.g. ('ny', 'nx', 'nfractions'))

Examples

```
>>> netcdf.initialize('aeolis.nc',
...                   ['Ct', 'Cu', 'zb'],
...                   ['avg', 'max'],
...                   s, p, {'Ct':('ny','nx','nfractions'),
...                          'Cu':('ny','nx','nfractions'),
...                          'zb':('ny','nx')})
```

netcdf.**parse_metadata**(*outputvars*)
Parse metadata from constants.py

Parses the Python comments in constants.py to extract meta data, like units, for the model state variables that can be used as netCDF4 meta data.

Parameters **outputvars** (*dictionary*) – Spatial grids to be written to netCDF4 output file

Returns **meta** – Dictionary with meta data for the output variables

Return type dict

netcdf.**set_bounds**(*outputfile*)
Sets CF time bounds

Parameters **outputfile** (*str*) – Name of netCDF4 output file

Plotting

plot.**profile**(*outputfile*, *var*, *ix=-1*, *subplots_kw={'figsize': (10, 4)}*)
Plots profile

Parameters

- **outputfile** (*str*) – Name of netCDF4 output file
- **var** (*str*) – Name of spatial grid
- **ix** (*int, optional*) – Time slice (default: -1)
- **subplots_kw** (*dict*) – Keyword options to subplots function

Command-line tools

console.**aeolis**()
aeolis : a process-based model for simulating supply-limited aeolian sediment transport

Usage: aeolis <config> [–callback=FUNC] [–restart=FILE] [–verbose=LEVEL]

Positional arguments: config configuration file

Options:

-h, --help	show this help message and exit
--callback=FUNC	reference to callback function (e.g. example/callback.py:callback)
--restart=FILE	model restart file
--verbose=LEVEL	write logging messages [default: 20]

console.**wind**()
aeolis-wind : a wind time series generation tool for the aeolis model

Usage: aeolis-wind <file> [–mean=MEAN] [–max=MAX] [–duration=DURATION] [–timestep=TIMESTEP]

Positional arguments: file output file

Options:

-h, --help	show this help message and exit
--mean=MEAN	mean wind speed [default: 10]
--max=MAX	maximum wind speed [default: 30]
--duration=DURATION	duration of time series [default: 3600]
--timestep=TIMESTEP	timestep of time series [default: 60]

Miscellaneous

utils.**apply_mask**(*arr*, *mask*)
Apply complex mask

The real part of the complex mask is multiplied with the input array. Subsequently the imaginary part is added and the result returned.

The shape of the mask is assumed to match the first few dimensions of the input array. If the input array is larger than the mask, the mask is repeated for any additional dimensions.

Parameters

- **arr** (*numpy.ndarray*) – Array or matrix to which the mask needs to be applied
- **mask** (*numpy.ndarray*) – Array or matrix with complex mask values

Returns **arr** – Array or matrix to which the mask is applied

Return type numpy.ndarray

utils.**format_log**(*msg*, *ncolumns=2*, ***props*)

Format log message into columns

Prints log message and additional data into a column format that fits into a 70 character terminal.

Parameters

- **msg** (*str*) – Main log message
- **ncolumns** (*int*) – Number of columns
- **props** (*key/value pairs*) – Properties to print in column format

Returns Formatted log message

Return type str

Note: Properties names starting with `min`, `max` or `nr` are respectively replaced by `min.`, `max.` or `#`.

---

utils.**interp_array**(*x*, *xp*, *fp*, *circular=False*, ***kwargs*)

Interpolate multiple time series at once

Parameters

- **x** (*array_like*) – The x-coordinates of the interpolated values.
- **xp** (*1-D sequence of floats*) – The x-coordinates of the data points, must be increasing.
- **fp** (*2-D sequence of floats*) – The y-coordinates of the data points, same length as `xp`.
- **circular** (*bool*) – Use the *interp_circular()* function rather than the `numpy.interp()` function.
- **kwargs** (*dict*) – Keyword options to the `numpy.interp()` function

Returns The interpolated values, same length as second dimension of `fp`.

Return type ndarray

utils.**interp_circular**(*x*, *xp*, *fp*, ***kwargs*)

One-dimensional linear interpolation.

Returns the one-dimensional piecewise linear interpolant to a function with given values at discrete data-points. Values beyond the limits of `x` are interpolated in circular manner. For example, a value of `x > x.max()` evaluates as `f(x-x.max())` assuming that `x.max() - x < x.max()`.

Parameters

- **x** (*array_like*) – The x-coordinates of the interpolated values.

- **xp** (*1-D sequence of floats*) – The x-coordinates of the data points, must be increasing.
- **fp** (*1-D sequence of floats*) – The y-coordinates of the data points, same length as xp.
- **kwargs** (*dict*) – Keyword options to the numpy.interp() function

Returns **y** – The interpolated values, same shape as x.

Return type {float, ndarray}

Raises ValueError – If xp and fp have different length

utils.**isarray**(*x*)
Check if variable is an array

utils.**isiterable**(*x*)
Check if variable is iterable

utils.**makeiterable**(*x*)
Ensure that variable is iterable

utils.**normalize**(*x*, *ref=None*, *axis=0*, *fill=0.0*)
Normalize array

Normalizes an array to make it sum to unity over a specific axis. The procedure is safe for dimensions that sum to zero. These dimensions return the fill value instead.

Parameters

- **x** (*array_like*) – The array to be normalized
- **ref** (*array_like, optional*) – Alternative normalization reference, if not specified, the sum of x is used
- **axis** (*int, optional*) – The normalization axis (default: 0)
- **fill** (*float, optional*) – The return value for all-zero dimensions (default: 0.)

utils.**prevent_tiny_negatives**(*x*, *max_error=1e-10*, *replacement=0.0*)
Replace tiny negative values in array

Parameters

- **x** (*np.ndarray*) – Array with potential tiny negative values
- **max_error** (*float*) – Maximum absolute value to be replaced
- **replacement** (*float*) – Replacement value

Returns Array with tiny negative values removed

Return type np.ndarray

utils.**print_value**(*val*, *fill='<novalue>'*)
Construct a string representation from an arbitrary value

Parameters

- **val** (*misc*) – Value to be represented as string
- **fill** (*str, optional*) – String representation used in case no value is given

Returns String representation of value

Return type str

1.4 Default settings

The AeoLiS model can be configured using a model configuration file. For any configuration parameters not defined in the model configuration file, or in case the model configuration file is absent, the default model configuration is used. The default model configuration is listed below.

```
#: AeoLiS model default configuration
DEFAULT_CONFIG = {
    'process_wind'        : True,               # Enable the process of wind
    'process_shear'       : False,              # Enable the process of wind shear
    'process_tide'        : True,               # Enable the process of tides
    'process_wave'        : True,               # Enable the process of waves
    'process_runup'       : True,               # Enable the process of wave runup
    'process_moist'       : True,               # Enable the process of moist
    'process_mixtoplayer' : True,               # Enable the process of mixing
    'process_threshold'   : True,               # Enable the process of threshold
    'process_transport'   : True,               # Enable the process of transport
    'process_bedupdate'   : True,               # Enable the process of bed updating
    'process_meteo'       : False,              # Enable the process of meteo
    'process_salt'        : False,              # Enable the process of salt
    'process_humidity'    : False,              # Enable the process of humidity
    'th_grainsize'        : True,               # Enable wind velocity threshold␣
↪based on grainsize
    'th_bedslope'         : False,              # Enable wind velocity threshold␣
↪based on bedslope
    'th_moisture'         : True,               # Enable wind velocity threshold␣
↪based on moisture
    'th_humidity'         : False,              # Enable wind velocity threshold␣
↪based on humidity
    'th_salt'             : False,              # Enable wind velocity threshold␣
↪based on salt
    'th_roughness'        : True,               # Enable wind velocity threshold␣
↪based on roughness
    'xgrid_file'          : None,               # Filename of ASCII file with x-
↪coordinates of grid cells
    'ygrid_file'          : None,               # Filename of ASCII file with y-
↪coordinates of grid cells
    'bed_file'            : None,               # Filename of ASCII file with bed␣
↪level heights of grid cells
    'wind_file'           : None,               # Filename of ASCII file with time␣
↪series of wind velocity and direction
    'tide_file'           : None,               # Filename of ASCII file with time␣
↪series of water levels
    'wave_file'           : None,               # Filename of ASCII file with time␣
↪series of wave heights
    'meteo_file'          : None,               # Filename of ASCII file with time␣
↪series of meteorlogical conditions
    'bedcomp_file'        : None,               # Filename of ASCII file with initial␣
↪bed composition
    'threshold_file'      : None,               # Filename of ASCII file with shear␣
↪velocity threshold
    'wave_mask'           : None,               # Filename of ASCII file with mask␣
↪for wave height
    'tide_mask'           : None,               # Filename of ASCII file with mask␣
↪for tidal elevation
    'threshold_mask'      : None,               # Filename of ASCII file with mask␣
↪for the shear velocity threshold
    'nx'                  : 0,                  # [-] Number of grid cells in x-
↪dimension
```

```
    'ny'                    : 0,                    # [-] Number of grid cells in y-
↪dimension
    'dt'                    : 60.,                  # [s] Time step size
    'CFL'                   : 1.,                   # [-] CFL number to determine time␣
↪step in explicit scheme
    'accfac'                : 1.,                   # [-] Numerical acceleration factor
    'tstart'                : 0.,                   # [s] Start time of simulation
    'tstop'                 : 3600.,                # [s] End time of simulation
    'restart'               : None,                 # [s] Interval for which to write␣
↪restart files
    'output_times'          : 60.,                  # [s] Output interval in seconds of␣
↪simulation time
    'output_file'           : None,                 # Filename of netCDF4 output file
    'output_vars'           : ['zb', 'zs',
                               'Ct', 'Cu',
                               'uw', 'uth',
                               'mass', 'pickup'], # Names of spatial grids to be␣
↪included in output
    'output_types'          : [],                   # Names of statistical parameters to␣
↪be included in output (avg, sum, var, min or max)
    'grain_size'            : [225e-6],             # [m] Average grain size of each␣
↪sediment fraction
    'grain_dist'            : [1.],                 # [-] Initial distribution of␣
↪sediment fractions
    'nfractions'            : 1,                    # [-] Number of sediment fractions
    'nlayers'               : 3,                    # [-] Number of bed layers
    'layer_thickness'       : .01,                  # [m] Thickness of bed layers
    'g'                     : 9.81,                 # [m/s^2] Gravitational constant
    'rhoa'                  : 1.25,                 # [kg/m^3] Air density
    'rhop'                  : 2650.,                # [kg/m^3] Grain density
    'rhow'                  : 1025.,                # [kg/m^3] Water density
    'porosity'              : .4,                   # [-] Sediment porosity
    'A'                     : .085,                 # [-] Constant in formulation for␣
↪wind velocity threshold based on grain size
    'z'                     : 10.,                  # [m] Measurement height of wind␣
↪velocity
    'h'                     : None,                 # [m] Representative height of␣
↪saltation layer
    'k'                     : 0.01,                 # [m] Bed roughness
    'Cb'                    : 1.5,                  # [-] Constant in formulation for␣
↪equilibrium sediment concentration
    'm'                     : .5,                   # [-] Factor to account for␣
↪difference between average and maximum shear stress
    'sigma'                 : 4.2,                  # [-] Ratio between basal area and␣
↪frontal area of roughness elements
    'beta'                  : 130.,                 # [-] Ratio between drag coefficient␣
↪of roughness elements and bare surface
    'bi'                    : 1.,                   # [-] Bed interaction factor
    'T'                     : 1.,                   # [s] Adaptation time scale in␣
↪advection equation
    'Tdry'                  : 3600.*1.5,            # [s] Adaptation time scale for soil␣
↪drying
    'Tsalt'                 : 3600.*24.*30.,        # [s] Adaptation time scale for␣
↪salinitation
    'eps'                   : 1e-3,                 # [m] Minimum water depth to consider␣
↪a cell "flooded"
    'gamma'                 : .5,                   # [-] Maximum wave height over depth␣
↪ratio
```

```
    'xi'                    : .3,                   # [-] Surf similarity parameter
    'facDOD'                : .1,                   # [-] Ratio between depth of␣
↪disturbance and local wave height
    'csalt'                 : 35e-3,                # [-] Maximum salt concentration in␣
↪bed surface layer
```

1.5 Model state/output

The AeoLiS model state is described by a collection of spatial grid variables with at least one value per horizontal grid cell. Specific model state variables can also be subdivided over bed composition layers and/or grain size fractions. All model state variables can be part of the model netCDF4 output. The current model state variables are listed below.

```
    ('ny', 'nx') : (
        'x',                                # [m] Real-world x-coordinate of grid␣
↪cell center
        'y',                                # [m] Real-world y-coordinate of grid␣
↪cell center
        'ds',                               # [m] Real-world grid cell size in x-
↪direction
        'dn',                               # [m] Real-world grid cell size in y-
↪direction
        'dsdn',                             # [m^2] Real-world grid cell surface area
        'dsdni',                            # [m^-2] Inverse of real-world grid cell␣
↪surface area
        'alfa',                             # [rad] Real-world grid cell orientation
        'uw',                               # [m/s] Wind velocity
        'uws',                              # [m/s] Component of wind velocity in x-
↪direction
        'uwn',                              # [m/s] Component of wind velocity in y-
↪direction
        'tau',                              # [m/s] Wind shear velocity
        'taus',                             # [m/s] Component of wind shear velocity␣
↪in x-direction
        'taun',                             # [m/s] Component of wind shear velocity␣
↪in y-direction
        'dtaus',                            # [-] Component of the wind shear␣
↪perturbation in x-direction
        'dtaun',                            # [-] Component of the wind shear␣
↪perturbation in y-direction
        'udir',                             # [rad] Wind direction
        'zb',                               # [m] Bed level above reference
        'zs',                               # [m] Water level above reference
        'Hs',                               # [m] Wave height
    ),
    ('ny','nx','nfractions') : (
        'Cu',                               # [kg/m^2] Equilibrium sediment␣
↪concentration integrated over saltation height
        'Ct',                               # [kg/m^2] Instantaneous sediment␣
↪concentration integrated over saltation height
        'q',                                # [kg/m/s] Instantaneous sediment flux
        'qs',                               # [kg/m/s] Instantaneous sediment flux in␣
↪x-direction
        'qn',                               # [kg/m/s] Instantaneous sediment flux in␣
↪y-direction
        'pickup',                           # [kg/m^2] Sediment entrainment
```

```
        'w',                                    # [-] Weights of sediment fractions
        'w_init',                               # [-] Initial guess for ``w''
        'w_air',                                # [-] Weights of sediment fractions based␣
↪on grain size distribution in the air
        'w_bed',                                # [-] Weights of sediment fractions based␣
↪on grain size distribution in the bed
        'uth',                                  # [m/s] Shear velocity threshold
    ),
    ('ny','nx','nlayers') : (
        'thlyr',                                # [m] Bed composition layer thickness
        'moist',                                # [-] Moisure content
        'salt',                                 # [-] Salt content
    ),
    ('ny','nx','nlayers','nfractions') : (
```

1.6 Installation

1.6.1 Requirements

Python packages

- bmi-python: http://github.com/openearth/bmi-python
- numpy
- scipy
- netCDF4
- docopt

External libraries (Windows)

These libraries are needed on Windows if the Python package netCDF4 is installed manually.

- Microsoft Visual C++ Compiler for Python 2.7: http://aka.ms/vcpython27
- msinttypes for stdint.h: https://code.google.com/archive/p/msinttypes/
- HDF5 headers: https://www.hdfgroup.org/HDF5/release/obtain5.html
- netCDF4 headers: https://github.com/Unidata/netcdf-c/releases
- Set environment variables HDF5_DIR and NETCDF_DIR to the respective installation paths

1.7 What's New

1.7.1 v1.1.3 (unreleased)

Breaking changes

- Removed support for statistical variable names with dot-notation (e.g. *.avg* and *.sum*).

Improvements

None.

New functions/methods

None.

Bug fixes

None.

Tests

None.

1.7.2 v1.1.2 (21 December 2017)

Breaking changes

- Changed name of statistics variables that describe the average, minimum, maximum, cumulative values, or variance of a model state variable. The variables names that used to end with *.avg*, *.sum*, etc. now end with *_avg*, *_sum*, etc. The new naming convention was already adopted in the netCDF output in order to be compatible with the CF-1.6 convention, but is now also adopted in, for example, the Basic Model Interface (BMI). Old notation is deprecated but still supported.

Improvements

- Prepared for continuous integration through CircleCI.
- Prepared for code coverage checking through codecov.

New functions/methods

None.

Bug fixes

- Use percentages (0-100) rather than fractions (0-1) in the formulation of Belly and Johnson that describes the effect of soil moisture on the shear velocity threshold. Thanks to Dano Roelvink and Susana Costas (b3d992b).

Tests

- Reduced required accuracy for mass conservation tests from 0.00000000000001% to 1%.

1.7.3 v1.1.1 (15 November 2017)

Breaking changes

None.

Improvements

- Made code compatible with Python 3.x.
- Prepared and uploaded package to PyPI.
- Switch back to original working directory after finishing simulation.
- Removed double definition of model state. Not only defined in *constants.MODEL_STATE*.
- Also write initial model state to output.
- Made netCDF output compatible with CF-1.6 convention.

New functions/methods

- Added support to run a default model for testing purposes by setting the configuration file as "DEFAULT".
- Added generic framework for reading and applying spatial masks. Implemented support for wave, tide and threshold masks specifically.
- Added option to include a reference date in netCDF output.
- Added experimental option for constant boundary conditions.
- Added support for reading and writing hotstart files to load a (partial) model state upon initialisation.
- Added preliminary wind shear perturbation module. Untested.
- Added support to switch on or off specific processes.
- Added support for immutable model state variables. This functionality can be combined with BMI or hotstart files to prevent external process results to be overwritten by the model.
- Added option to specify wind direction convention (nautical or cartesian).

Bug fixes

- Fixed conversion from volume to mass using porosity and density (fe9aa52).
- Update water level with bed updates to prevent loss of water due to bed level change (fe9aa52).
- Fixed mass bug in base layer that drained sediment from bottom layers, resulting in empty layers (f612760).
- Made removal of negative concentrations mass conserving by scraping the concentrations from all other grid cells (03de813).

Tests

- Added tests to check mass conservation in bed mixing routines.
- Added integration tests.

1.7.4 v1.1.0 (27 July 2016)

Initial release

CHAPTER 2

Acknowledgements

AeoLiS is initially developed by Bas Hoonhout at Delft University of Technology with support from the ERC-Advanced Grant 291206 Nearshore Monitoring and Modeling (NEMO) and Deltares. AeoLiS is currently maintained by Bas Hoonhout at Deltares and Sierd de Vries at Delft University of Technology.

CHAPTER 3

Indices and tables

- genindex
- search

Bibliography

[Bag37] RA Bagnold. The size-grading of sand by wind. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 250–264, 1937.

[Bag37b] RA Bagnold. The transport of sand by wind. *Geographical journal*, pages 409–438, 1937.

[Bel64] P Y Belly. Sand movement by wind. Technical Report 1, U.S. Army Corps of Engineers CERC, 1964. 38 pp.

[dVvTdVvR+14] S de Vries, J S M van Thiel de Vries, L C van Rijn, S M Arens, and R Ranasinghe. Aeolian sediment transport in supply limited situations. *Aeolian Research*, 12:75–85, 2014. doi:10.1016/j.aeolia.2013.11.005.

[DF10] I Delgado-Fernandez. A review of the application of the fetch effect to modelling sand supply to coastal foredunes. *Aeolian Research*, 2:61–70, 2010. doi:10.1016/j.aeolia.2010.04.001.

[Kin51] C. A. M. King. Depth of disturbance of sand on sea beaches by waves. *Journal of Sedimentary Petrology*, 21(3):131–140, 1951. URL: http://archives.datapages.com/data/sepm/journals/v01-32/data/021/021003/pdfs/0131.pdf.

[MAROHare07] G Masselink, N Auger, P Russell, and T O'Hare. Short-term morphological change and sediment dynamics in the intertidal zone of a macrotidal beach. *Sedimentology*, 54:39–53, 2007. doi:10.1111/j.1365-3091.2006.00825.x.

[NE81] W G Nickling and M Ecclestone. The effects of soluble salts on the threshold shear velocity of fine sand. *Sedimentology*, 28:505–510, 1981.

[PHN13] S. D. Peckham, E. W. H. Hutton, and B. Norris. A component-based approach to integrated modeling in the geosciences: the design of CSDMS. *Computers and Geosciences*, 53:3–12, 2013. doi:10.1016/j.cageo.2012.04.002.

[PT90] K. Pye and H. Tsoar. *Aeolian Sand and Sand Dunes*. Unwin Hyman, London, 1990.

[RGL93] MR Raupach, DA Gillette, and JF Leys. The effect of roughness elements on wind erosion threshold. *Journal of Geophysical Research: Atmospheres*, 98(D2):3023–3029, 1993. doi:10.1029/92JD01922.

[Shu93] W J Shuttleworth. Evaporation. In D R Maidment, editor, *Handbook of Hydrology*, pages 4.1–4.53. McGraw-Hill, New York, 1993.

[Wil71] A. T. Williams. An analysis of some factors involved in the depth of disturbance of beach sand by waves. *Marine Geology*, 11(3):145–158, 1971. doi:10.1016/0025-3227(71)90003-X.

[Delft3DFManual14] Delft3D-FLOW Manual. *Delft3D - 3D/2D modelling suite for integral water solutions - Hydro-Morphodynamics*. Deltares, Delft, May 2014. Version 3.15.34158.

[SimunekSejnavG98] J. Šimůnek, M. Šejna, and M. Th. van Genuchten. *The HYDRUS-1D software package for simulating the one-dimensional movement of water, heat, and multiple solutes in variably- saturated media*. International Ground Water Modeling Center, Colorado School of Mines, Golden, Colorado, version 1.0. igwmc - tps - 70 edition, 1998. 186pp.

[Bag37] RA Bagnold. The size-grading of sand by wind. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 250–264, 1937.

[Bel64] P Y Belly. Sand movement by wind. Technical Report 1, U.S. Army Corps of Engineers CERC, 1964. 38 pp.

[DF10] I Delgado-Fernandez. A review of the application of the fetch effect to modelling sand supply to coastal foredunes. *Aeolian Research*, 2:61–70, 2010. doi:10.1016/j.aeolia.2010.04.001.

[NE81] W G Nickling and M Ecclestone. The effects of soluble salts on the threshold shear velocity of fine sand. *Sedimentology*, 28:505–510, 1981.

[PHN13] S. D. Peckham, E. W. H. Hutton, and B. Norris. A component-based approach to integrated modeling in the geosciences: the design of CSDMS. *Computers and Geosciences*, 53:3–12, 2013. doi:10.1016/j.cageo.2012.04.002.

[PT90] K. Pye and H. Tsoar. *Aeolian Sand and Sand Dunes*. Unwin Hyman, London, 1990.

[RGL93] MR Raupach, DA Gillette, and JF Leys. The effect of roughness elements on wind erosion threshold. *Journal of Geophysical Research: Atmospheres*, 98(D2):3023–3029, 1993. doi:10.1029/92JD01922.

[Shu93] W J Shuttleworth. Evaporation. In D R Maidment, editor, *Handbook of Hydrology*, pages 4.1–4.53. McGraw-Hill, New York, 1993.

[SimunekSejnavG98] J. Šimůnek, M. Šejna, and M. Th. van Genuchten. *The HYDRUS-1D software package for simulating the one-dimensional movement of water, heat, and multiple solutes in variably- saturated media*. International Ground Water Modeling Center, Colorado School of Mines, Golden, Colorado, version 1.0. igwmc - tps - 70 edition, 1998. 186pp.

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W