Technical Report

Mathematical Model UnTRIM$^2$

User Interface Description

– Version May 2010 (1.1) –
Technical Report

Mathematical Model UnTRIM²

User Interface Description

– Version May 2010 (1.1) –

Contributors: Günther Lang (BAW, Germany)
              Vincenzo Casulli (Trento University, Italy)

Version-Date: May 2010
Version-No.: 1.1
Project-No.: A 395 5 03 70150

Trento · Hamburg — April 2011
Summary

This document is the user interface description for the mathematical model UnTRIM². This document can be regarded as an appendix to Mathematical Model UnTRIM² – Validation Document, which is unfortunately not yet available as a separate report. Only the document for an older version is available as a separate report.

Organization of this document

Chapter B contains a short overview for all user interface routines (see Section B.6 on page 114) together with a detailed description of all Set-routines (see Section B.3 on page 4) as well as Get-functions (please refer to Section B.4 on page 41). Also an overview on deleted, modified as well as new interface routines is given in Section B.2 on page 2.

Chapter C summarizes the available Set-routines and Get-functions in tabular form. This may help the experienced user of UnTRIM² to quickly access the correct routine or function, and informations about how to use them.

Chapter D contains short examples of the three standard input data files ("untrim.inp", "untrim.grd" and "untrim.srs") which are read by the core of the computational model.

Electronic user interface description document

This document is also available in electronic form in Portable Document Format (PDF). The electronic version may be read using ACROBAT READER™ software which is available for many computer platforms. See PDF version of this document.

Errors

In case the reader and/or user of UnTRIM² finds any error within this document, please do not hesitate to report them to one of the authors. Any contribution to improve this document can help to build better UnTRIM² applications and is therefore most welcome.

Acknowledgements

During the Seventh International UnTRIM Users Workshop in Trento, May 3–5, 2010, it was decided to update this document, to conform to the actual version of the software (May 2010). Positive and stimulating feedback from the users to continue working on this type of document is herewith gratefully acknowledged.
B.3.34 Interface set_radiation_time ........................................ 25
B.3.35 Interface set_rain .................................................... 26
B.3.36 Interface set_sediment ............................................. 26
B.3.37 Interface set_settling_velocity .................................. 27
B.3.38 Interface set_slope .................................................. 28
B.3.39 Interface set_source_file ......................................... 28
B.3.40 Interface set_subedge_depth ..................................... 28
B.3.41 Interface set_subpolygon_depth ................................. 29
B.3.42 Interface set_surface_alpha ...................................... 29
B.3.43 Interface set_surface_beta ........................................ 30
B.3.44 Interface set_surface_concentration .......................... 31
B.3.45 Interface set_time ................................................... 32
B.3.46 Interface set_turbulent_h_diffusivity ......................... 32
B.3.47 Interface set_turbulent_h_viscosity ............................ 33
B.3.48 Interface set_turbulent_v_diffusivity ......................... 34
B.3.49 Interface set_turbulent_v_viscosity ............................ 34
B.3.50 Interface set_velocity ............................................. 35
B.3.51 Interface set_wet_and_dry ........................................ 36
B.3.52 Interface set_wind_friction ...................................... 36
B.3.53 Interface set_wind_velocity ..................................... 37
B.3.54 Interface share_bottom_flux ..................................... 38
B.3.55 Interface share_flux ................................................ 38
B.3.56 Interface share_k ................................................... 38
B.3.57 Interface share_kh .................................................. 39
B.3.58 Interface share_kv .................................................. 39
B.3.59 Interface share_settling_velocity ............................. 39
B.3.60 Interface share_surface_flux .................................... 40

B.4 Get-functions .......................................................... 41
B.4.1 Interface get_adjacent_polygon .................................. 47
B.4.2 Interface get_angle .................................................. 47
B.4.3 Interface get_area .................................................... 48
B.4.4 Interface get_artificial_porosity .................................. 48
B.4.5 Interface get_atmospheric_pressure ............................ 48
B.4.6 Interface get_avr_bottom_deviation ............................. 49
B.4.7 Interface get_avr_depth ............................................. 49
B.4.8 Interface get_avr_edge_bottom_deviation ....................... 49
B.4.9 Interface get_avr_edge_depth ..................................... 49
B.4.10 Interface get_avr_edge_total_depth ........................... 50
B.4.11 Interface get_avr_face_height .................................. 50
B.4.12 Interface get_avr_prism_height ................................ 51
B.4.13 Interface get_avr_total_depth .................................. 51
B.4.14 Interface get_bottom_alpha ..................................... 52
B.4.15 Interface get_bottom_beta ........................................ 52
B.4.16 Interface get_bottom_concentration .............................. 53
B.4.17 Interface get_bottom_deviation .................................... 53
B.4.18 Interface get_bottom_face ......................................... 53
B.4.19 Interface get_bottom_flux ......................................... 54
B.4.20 Interface get_bottom_friction .................................... 54
B.4.21 Interface get_bottom_prism ........................................ 55
B.4.22 Interface get_bottom_stress ....................................... 55
B.4.23 Interface get_bottom_surface ..................................... 55
B.4.24 Interface get_bottom_subprism .................................. 56
B.4.25 Interface get_celerity ............................................ 56
B.4.26 Interface get_center_distance ................................... 57
B.4.27 Interface get_chezy ............................................... 57
B.4.28 Interface get_concentration ..................................... 57
B.4.29 Interface get_courant ............................................. 58
B.4.30 Interface get_density ............................................. 58
B.4.31 Interface get_dry_area ............................................ 59
B.4.32 Interface get_dry_length ......................................... 59
B.4.33 Interface get_dt_min ............................................... 60
B.4.34 Interface get_dx_min ............................................... 60
B.4.35 Interface get_dz_min ............................................... 60
B.4.36 Interface get_edge_begin ......................................... 60
B.4.37 Interface get_edge_bottom_deviation ............................ 61
B.4.38 Interface get_edge_center ....................................... 61
B.4.39 Interface get_edge_end ............................................ 61
B.4.40 Interface get_edge_length ........................................ 62
B.4.41 Interface get_edge_wet_area ..................................... 62
B.4.42 Interface get_elevation ........................................... 63
B.4.43 Interface get_elevation_tolerance ............................... 63
B.4.44 Interface get_face_index ........................................ 63
B.4.45 Interface get_flux_limiter ....................................... 63
B.4.46 Interface get_froude .............................................. 64
B.4.47 Interface get_gravity ............................................. 64
B.4.48 Interface get_grid_file ........................................... 64
B.4.49 Interface get_horizontal_diffusivity ............................ 65
B.4.50 Interface get_horizontal_velocity ................................ 65
B.4.51 Interface get_horizontal_velocity2 ............................. 65
B.4.52 Interface get_horizontal_velocity_x ............................ 66
B.4.53 Interface get_horizontal_velocity_y ............................ 67
B.4.54 Interface get_horizontal viscosity ............................... 67
B.4.55 Interface get_input_file .......................................... 67
B.4.56 Interface get_layer ............................................... 68
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

B.4.57 Interface get_laver_interface
B.4.58 Interface get_left_polygon
B.4.59 Interface get_length
B.4.60 Interface get_location
B.4.61 Interface get_mass
B.4.62 Interface get_max_courant
B.4.63 Interface get_max_depth
B.4.64 Interface get_max_edge_depth
B.4.65 Interface get_max_edge_total_depth
B.4.66 Interface get_max_face_depth
B.4.67 Interface get_max_froude
B.4.68 Interface get_max_prism_height
B.4.69 Interface get_max_total_depth
B.4.70 Interface get_max_velocity_courant
B.4.71 Interface get_nof_bottom_fluxes
B.4.72 Interface get_nof_boundary_polygons
B.4.73 Interface get_nof_diffusivities
B.4.74 Interface get_nof_edges
B.4.75 Interface get_nof_faces
B.4.76 Interface get_nof_fluxes
B.4.77 Interface get_nof_h_diffusivities
B.4.78 Interface get_nof_inflow_edges
B.4.79 Interface get_nof_inner_iterations
B.4.80 Interface get_nof_internal_edges
B.4.81 Interface get_nof_iterations
B.4.82 Interface get_nof_layers
B.4.83 Interface get_nof_max_edges
B.4.84 Interface get_nof_max_iterations
B.4.85 Interface get_nof_outer_iterations
B.4.86 Interface get_nof_point_sources
B.4.87 Interface get_nof_polygons
B.4.88 Interface get_nof_prisms
B.4.89 Interface get_nof_settling_velocities
B.4.90 Interface get_nof_species
B.4.91 Interface get_nof_subedges
B.4.92 Interface get_nof_subpolygons
B.4.93 Interface get_nof_substeps
B.4.94 Interface get_nof_surface_fluxes
B.4.95 Interface get_nof_v_diffusivities
B.4.96 Interface get_nof_vertices
B.4.97 Interface get_nof_wet_faces
B.4.98 Interface get_nof_wet_polygons
B.4.99 Interface get_nof_wet_prisms
B.4.100 Interface get_nof_wet_subedges
B.4.101 Interface get_nof_wet_subpolygons
B.4.102 Interface get_polygon
B.4.103 Interface get_polygon_area
B.4.104 Interface get_polygon_center
B.4.105 Interface get_polygon_edge
B.4.106 Interface get_polygon_vertex
B.4.107 Interface get_pressure
B.4.108 Interface get_pressure_tolerance
B.4.109 Interface get_printout_unit
B.4.110 Interface get_prism_index
B.4.111 Interface get_radiation_time
B.4.112 Interface get_right_polygon
B.4.113 Interface get_sediment
B.4.114 Interface get_settling_velocity
B.4.115 Interface get_slope
B.4.116 Interface get_source_concentration
B.4.117 Interface get_source_discharge
B.4.118 Interface get_source_file
B.4.119 Interface get_source_layer
B.4.120 Interface get_source_polygon
B.4.121 Interface get_subedge_depth
B.4.122 Interface get_subedge_length
B.4.123 Interface get_subedge_total_depth
B.4.124 Interface get_subpolygon_area
B.4.125 Interface get_subpolygon_depth
B.4.126 Interface get_subpolygon_total_depth
B.4.127 Interface get_surface_alpha
B.4.128 Interface get_surface_beta
B.4.129 Interface get_surface_concentration
B.4.130 Interface get_surface_flux
B.4.131 Interface get_theta
B.4.132 Interface get_time
B.4.133 Interface get_time_step
B.4.134 Interface get_top_face
B.4.135 Interface get_top_prism
B.4.136 Interface get_total_mass
B.4.137 Interface get_total_volume
B.4.138 Interface get_turbulent_h_diffusivity
B.4.139 Interface get_turbulent_h_viscosity
B.4.140 Interface get_turbulent_v_diffusivity
B.4.141 Interface get_turbulent_viscosity ................................. 97
B.4.142 Interface get_unerodible_bottom_face .......................... 98
B.4.143 Interface get_unerodible_bottom_prism ........................... 98
B.4.144 Interface get_velocity ............................................... 98
B.4.145 Interface get_velocity_courant ..................................... 99
B.4.146 Interface get_vertex_coordinates ................................. 99
B.4.147 Interface get_vertical_diffusivity .............................. 100
B.4.148 Interface get_vertical_velocity ................................. 100
B.4.149 Interface get_vertical_viscosity ............................... 100
B.4.150 Interface get_vertically_avr_velocity .......................... 101
B.4.151 Interface get_volume ............................................... 101
B.4.152 Interface get_wet_area ............................................. 102
B.4.153 Interface get_wet_length ........................................... 102
B.4.154 Interface get_which_bottom_flux ............................. 103
B.4.155 Interface get_which_diffusivity ............................. 103
B.4.156 Interface get_which_flux ......................................... 103
B.4.157 Interface get_which_h_diffusivity ............................ 104
B.4.158 Interface get_which_settling_velocity ......................... 104
B.4.159 Interface get_which_surface_flux ............................. 104
B.4.160 Interface get_which_v_diffusivity ............................ 105
B.4.161 Interface get_wind_friction ..................................... 105
B.4.162 Interface get_wind_stress ....................................... 105
B.4.163 Interface get_wind_velocity ................................... 106
B.4.164 Interface get_wind_velocity_x ............................... 106
B.4.165 Interface get_wind_velocity_y ............................... 106
B.4.166 Interface get_x_slope ............................................ 107
B.4.167 Interface get_x_vertex_coordinate ............................ 107
B.4.168 Interface get_y_slope ............................................ 107
B.4.169 Interface get_y_vertex_coordinate ............................ 107
B.4.170 Interface is_dry_edge ............................................ 108
B.4.171 Interface is_dry_polygon ........................................ 108
B.4.172 Interface is_wet_edge ........................................... 108
B.4.173 Interface is_wet_polygon ....................................... 108
B.4.174 Interface sediment .............................................. 109
B.4.175 Interface with_hydrodynamic_pressure ....................... 109
B.4.176 Interface with_scalar_transport .............................. 109
B.5 Check routines .............................................................. 110
B.5.1 Subroutine check_all ................................................. 110
B.5.2 Function check_continuity ......................................... 111
B.5.3 Subroutine check_grid ............................................... 112
B.6 User interface routines .................................................. 114
B.6.1 Subroutine user_set_input_files .................................. 114
B.6.2 Subroutine user set initial conditions .................. 114
B.6.3 Subroutine user set forcing terms .................. 116
B.6.4 Subroutine user get results .................. 127

C Tables for set-routines and get-functions .................. 131

D Example input data files .................................. 153
D.1 Grid file .................................. 153
D.1.1 Grid sorting .................................. 153
D.1.2 Short description .................................. 154
D.1.3 Example file .................................. 156
D.2 Input data file .................................. 157
D.2.1 Short description .................................. 157
D.2.2 Example file .................................. 159
D.3 Source and sink data file .................................. 160
D.3.1 Short description .................................. 160
D.3.2 Example file .................................. 161
List of Figures
List of Tables

1  Nomenclature (part 1)  .................................................. 1
2  Nomenclature (part 2) .................................................. 2
3  Set-routines (set artificial porosity – set inflow cc) .................. 132
4  Set-routines (set input file – set surface alpha) ..................... 133
5  Set-routines (set surface beta – share surface flux) ................. 134
6  Get-functions (get adjacent polygon – get avr edge total depth) .... 135
7  Get-functions (get avr face height – get bottom friction) .......... 136
8  Get-functions (get bottom prism – get density) .................... 137
9  Get-functions (get dry area – get edge length) .................... 138
10 Get-functions (get edge wet area – get horizontal velocity) ....... 139
11 Get-functions (get horizontal velocity2 – get location) ............ 140
12 Get-functions (get mass – get max velocity courant) ............... 141
13 Get-functions (get nof bottom fluxes – get nof internal edges) .... 142
14 Get-functions (get nof iterations – get nof species) ................ 143
15 Get-functions (get nof subedges – get nof wet subedges) .......... 144
16 Get-functions (get nof wet subpolygons – get prism index) ...... 145
17 Get-functions (get radiation time – get source polygon) .......... 146
18 Get-functions (get subedge depth – get surface flux) ............. 147
19 Get-functions (get theta – get turbulent v diffusivity) .......... 148
20 Get-functions (get turbulent v viscosity – get vertically avr velocity) 149
21 Get-functions (get volume – get which v diffusivity) .......... 150
22 Get-functions (get wind friction – is dry edge) ................ 151
23 Get-functions (is dry polygon – with scalar transport) ......... 152
B User interface

B.1 Nomenclature

<table>
<thead>
<tr>
<th>argument</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>a, a_x, a_y</td>
<td>one-dimensional array</td>
</tr>
<tr>
<td>b</td>
<td>two-dimensional array</td>
</tr>
<tr>
<td>i, i_1, i_2</td>
<td>polygon index</td>
</tr>
<tr>
<td>i^SG</td>
<td>subpolygon index within polygon i</td>
</tr>
<tr>
<td>i^3D</td>
<td>storage index for prism oriented data</td>
</tr>
<tr>
<td>j, j_1, j_2</td>
<td>edge (side) index</td>
</tr>
<tr>
<td>j^SG</td>
<td>subedge (-side) index along side j</td>
</tr>
<tr>
<td>j^3D</td>
<td>storage index for edge oriented data</td>
</tr>
<tr>
<td>k</td>
<td>layer index</td>
</tr>
<tr>
<td>l</td>
<td>side/vertex index within polygon</td>
</tr>
<tr>
<td>l_max</td>
<td>maximum sides/vertices within polygons</td>
</tr>
<tr>
<td>m, m_1, m_2</td>
<td>specie index</td>
</tr>
<tr>
<td>m_N_max</td>
<td>max. number of (outer) Newton iterations</td>
</tr>
<tr>
<td>n</td>
<td>source/sink location index, number, index</td>
</tr>
<tr>
<td>n^SG</td>
<td>number of subpolygons within polygon i</td>
</tr>
<tr>
<td>n^S_j</td>
<td>number of subedges along edge j</td>
</tr>
<tr>
<td>n_s_f</td>
<td>number of sides with flow boundary condition</td>
</tr>
<tr>
<td>no</td>
<td>logical unit number</td>
</tr>
<tr>
<td>s, s_x, s_y</td>
<td>scalar</td>
</tr>
<tr>
<td>st</td>
<td>character string</td>
</tr>
<tr>
<td>t</td>
<td>time</td>
</tr>
<tr>
<td>x, y, z</td>
<td>coordinates</td>
</tr>
</tbody>
</table>

Table 1: Nomenclature (part 1).
argument | description
--- | ---
$I_i$ | number of prisms above polygon $i$
$I_s$ | number of prisms
$J_j$ | number of faces above edge $j$
$J_s$ | number of faces
$J^*_s$ | number of faces at boundaries with prescribed flow
$N_c$ | number of species
$N_d$ | number of sources and sinks
$N_p$ | number of polygons
$N^p_p$ | number of polygons at boundaries with prescribed water level
$N^{SG}_p$ | number of subpolygons
$N_s$ | number of edges (sides)
$N^{SG}_s$ | number of subedges (-sides)
$N_{sj}$ | number of edges (sides) with flow
$N_{si}$ | number of internal edges (sides)
$N_v$ | number of vertices (nodes)
$N_{sw}$ | number of different settling velocities
$N_z$ | number of layers
$N_b$ | number of different bottom fluxes
$N^K_h$ | number of different horizontal diffusivities
$N^K_v$ | number of different vertical diffusivities
$N_f$ | number of different surface fluxes
$S_i$ | number of sides for polygon $i$
$\Delta t$ | time step
$\theta$ | implicitness factor

Table 2: Nomenclature (part 2).

B.2 Major changes

Subsequently all changes between this user interface description and the previous one — version August 2009 (1.1) — are listed.

B.2.1 Removed methods

- set routines: no routines removed.
- get functions: no functions removed.
- check routines: no routines removed.
B.2.2 Modified methods

- modified set routines
  - \texttt{set\_bottom\_friction}:
    1. now applicable to subedges $1 \leq j^{SG} \leq n^{SG}_{i,j}$ for all edges $j$ from $1 \leq j \leq N_j$.

- modified get functions
  - \texttt{get\_bottom\_friction}:
    1. now applicable to subedges $1 \leq j^{SG} \leq n^{SG}_{i,j}$ for all edges $j$ from $1 \leq j \leq N_j$;
    2. array of values returned for a single edge.
  - \texttt{get\_bottom\_stress}:
    1. now applicable to subedges $1 \leq j^{SG} \leq n^{SG}_{i,j}$ for all edges $j$ from $1 \leq j \leq N_j$;
    2. array of values returned for a single edge.

- modified check routines: no routines modified.

B.2.3 New methods

- new set routines: no new routines.

- new get functions
  - computational grid
    * dynamic structure
      - \texttt{get\_bottom\_subprism}
      - \texttt{get\_bottom\_subface}

- new check routines: no new routines.

B.2.4 Further modifications

Subsection B.6 was updated with respect to the removed, modified or new methods as described above.
The format of the grid file has not changed.
A new advection scheme is available. See subsection D.2 for further informations.
B.3 Set routines

Set-routines are going to be used if (internal) default values for variables and parameters are not appropriate for the scenario under investigation. In the following, all set-routines are grouped according to their functionality:

- file names for standard input files
  - steering data file
    * `set_input_file`
  - grid
    * `set_grid_file`
  - sources and sinks file
    * `set_source_file`

- logical unit numbers
  - printer file
    * `set_printout_unit`

- steering data
  - number of outer (newton) iterations
    * `set_wet_and_dry`
  - transport algorithm for scalar species
    * `set_flux_limiter`
  - scalar species
    * `set_sediment`
    * `set_new_bottom_flux`
    * `set_new_surface_flux`
    * `set_new_flux`
    * `set_new_k`
    * `set_new_kh`
    * `set_new_kv`
    * `set_new_settling_velocity`
    * `share_flux`
    * `share_bottom_flux`
    * `share_surface_flux`
    * `share_k`
    * `share_kh`
    * `share_kv`
- 
  * \texttt{share.settling.velocity} 
  - coriolis approximation 
    * \texttt{set.constant.coriolis} 
  - time 
    * \texttt{set.time} 

- initial data and variable coefficients 
  - (subgrid scale) bathymetry 
    * \texttt{set.subedge.depth} 
    * \texttt{set.subpolygon.depth} 
  - hydrodynamics 
    * \texttt{set.body.force} 
    * \texttt{set.density} 
    * \texttt{set.elevation} 
    * \texttt{set.horizontal.velocity} 
    * \texttt{set.pressure} 
    * \texttt{set.turbulent.h.viscosity} 
    * \texttt{set.turbulent.v.viscosity} 
    * \texttt{set.velocity} 
  - scalar species 
    * \texttt{set.concentration} 
    * \texttt{set.settling.velocity} 
    * \texttt{set.turbulent.h.diffusivity} 
    * \texttt{set.turbulent.v.diffusivity} 

- data at lateral open boundaries with \textit{prescribed water level} 
  - hydrodynamics 
    * \texttt{set.elevation.bc} 
    * \texttt{set.pressure.bc} 
    * \texttt{set.radiation.time} 
  - scalar species 
    * \texttt{set.concentration.bc} 

- data at lateral open boundaries with \textit{prescribed flow} 
  - hydrodynamics 
    * \texttt{set.inflow.bc} 
  - scalar species
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

* `set_inflow_cc`

- boundary data at the free surface
  - atmospheric data
    * `set_atmospheric_pressure`
    * `set_wind_friction`
    * `set_wind_velocity`
    * `set_rain`
  - data related to scalar species
    * `set_surface_alpha`
    * `set_surface_beta`
    * `set_surface_concentration`

- boundary data at the bottom surface
  - data related to bottom friction
    * `set_bottom_friction`
  - data related to scalar species
    * `set_bottom_alpha`
    * `set_bottom_beta`
    * `set_bottom_concentration`
  - data related to morphodynamics
    * `set_bottom_deviation`
    * `set_edge_bottom_deviation`

- sources and sinks
  - hydrodynamics
    * `set_point_sources_discharge`
  - scalar species
    * `set_point_sources_concentration`

- system properties
  - artificial porosity
    * `set_artificial_porosity`
    * `set_wet_and_dry`
  - slope
    * `set_slope`

All set routines available to the user are subsequently described and listed in alphabetical order.
B.3.1 Interface set_artificial_porosity

physical unit: m

default value: 0.0 (if interface is not used)

Assign artificial porosity $p_y$ at sides/edges with flow:

1. CALL set_artificial_porosity(s): a constant value $s$ [re | dp] is assigned to all flow edges ($1 \leq j \leq N_s$);

2. CALL set_artificial_porosity(j,s): artificial porosity is set for the $j$-th edge ($1 \leq j \leq N_s$) to a constant value $s$ [re | dp];

3. CALL set_artificial_porosity(a(:)): a spatially varying artificial porosity $a(N_s)$ [re | dp] is assigned to all flow edges $N_s$.

$p_y \geq 0.0$ must be fulfilled. $N_s$ is given by \texttt{get_nof_internal_edges()} and $N_s$ by $N_s + \texttt{get_nof_inflow_edges()}$.

B.3.2 Interface set_atmospheric_pressure

physical unit: m$^2$/s$^2$

default value: 0.0 (if interface is not used)

Assign normalized atmospheric pressure $p_a$ at polygon centers:

1. CALL set_atmospheric_pressure(s): a constant normalized atmospheric pressure $s$ [re | dp] is assigned to all polygons $N_p$;

2. CALL set_atmospheric_pressure(i,s): the normalized atmospheric pressure is set for the $i$-th polygon ($1 \leq i \leq N_p$) to a constant value $s$ [re | dp];

3. CALL set_atmospheric_pressure(a(:)): a spatially varying normalized atmospheric pressure $a(N_p)$ [re | dp] is assigned to all polygons $N_p$.

To determine $N_p$ use \texttt{get_nof_polygons()}. To obtain the normalized atmospheric pressure $p_a$, the (real physical) atmospheric pressure has to be divided by $\rho_0$.

B.3.3 Interface set_body_force

physical unit: m/s$^2$

default value: 0.0 (if interface is not used)
Assign horizontal normal body force $f$ at faces from user specified horizontal ($f_u,f_v$) body force components:

1. CALL set_body_force($f_u,f_v$): a constant horizontal body force with given $x$-$y$-components $f_u$, $f_v$ [re|dp] is assigned to the normal body force component $f$ for all computational faces $J_i$;

2. CALL set_body_force($j$, $f_u,f_v$): a constant horizontal body force with given $x$-$y$-components $f_u$, $f_v$ [re|dp] is assigned to the normal body force component $f$ for all computational faces lying above the $j$-th side ($1 \leq j \leq N_{si}$);

3. CALL set_body_force($j$, $k$, $f_u,f_v$): a constant horizontal body force with given $x$-$y$-components $f_u$, $f_v$ [re|dp] is assigned to the normal body force component $f$ for the computational face located in the $k$-th layer ($k_b(j) \leq k \leq k_l(j)$) above the $j$-th side ($1 \leq j \leq N_{si}$);

4. CALL set_body_force($j$, $f_u(:)$, $f_v$): a depth-varying horizontal body force with given variable $x$- and constant $y$-component $f_u(N_{si})$, $f_v$ [re|dp] is assigned to the normal body force component $f$ for the computational faces lying above the $j$-th side ($1 \leq j \leq N_{si}$);

5. CALL set_body_force($j$, $f_u,f_v(:)$): a depth-varying horizontal body force with given variable $y$- and constant $x$-component $f_u$, $f_v(N_{si})$ [re|dp] is assigned to the normal body force component $f$ for the computational faces lying above the $j$-th side ($1 \leq j \leq N_{si}$);

6. CALL set_body_force($j$, $f_u(:)$, $f_v(:)$): a depth-varying horizontal body force with given $x$-$y$-components $f_u(N_{si})$, $f_v(N_{si})$ [re|dp] is assigned to the normal body force component $f$ for the computational faces lying above the $j$-th side ($1 \leq j \leq N_{si}$);

7. CALL set_body_force($f_u(:)$, $f_v$): a spatially varying horizontal body force in $x$-direction and constant in $y$-direction with $f_u(J_i)$, $f_v$ [re|dp] is assigned to the normal velocity component $f$ at all the $J_i$ computational faces;

8. CALL set_body_force($f_u,f_v(:)$): a spatially varying horizontal body force in $y$-direction and constant in $x$-direction with $f_u$, $f_v(J_i)$ [re|dp] is assigned to the normal body force component $f$ at all the $J_i$ computational faces;

9. CALL set_body_force($f_u(:)$, $f_v(:)$): a spatially varying horizontal body force with given $x$-$y$-components $f_u(J_i)$, $f_v(J_i)$ [re|dp] is assigned to the normal body force component $f$ at all the $J_i$ computational faces.

The total number of faces $J_i$ is determined by means of [get_nof_faces()] whereas $N_{si}$ is obtained from [get_nof_internal_edges()]. The number of vertical layers $N_z$ can be retrieved using [get_nof_layers()] while the bottom layer index above edges $k_b(j)$ is given by [get_bottom_face($j$)] and $k_l(j)$ can be obtained from [get_top_face($j$)].
B.3.4 Interface set_bottom_alpha

**physical unit:** (physical unit of specie used) × m/s

**default value:** 0.0 (if interface is *not* used)

Assign bottom flux parameter $\alpha_s$ at polygon centers:

1. CALL set_bottom_alpha(s): a constant $s$ [re|dp] is assigned to the bottom flux parameter $\alpha_s$ for all scalar species $N_c$ and all polygons $N_p$;
2. CALL set_bottom_alpha(m,s): a constant $s$ [re|dp] is assigned to the bottom flux parameter $\alpha_s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) and all polygons $N_p$;
3. CALL set_bottom_alpha(m,i,s): a constant $s$ [re|dp] is assigned to the bottom flux parameter $\alpha_s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) in the $i$-th polygon ($1 \leq i \leq N_p$);
4. CALL set_bottom_alpha(m,a(:)): spatially varying data $a(N_p)$ [re|dp] are assigned to the bottom flux parameter $\alpha_s$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $N_p$ polygons;
5. CALL set_bottom_alpha(b(:,:)): spatially varying data $b(N_p,N_B)$ [re|dp], eventually different for some of the $N_c$ scalar species, are assigned to the bottom flux parameter $\alpha_s$ for all $N_s$ different bottom fluxes and all $N_p$ polygons.

$\alpha_s \geq 0.0$ must be fulfilled. $N_p$ can be determined by means of `get_nof_polygons()`, $N_c$ from `get_nof_species()` and $N_B$ from `get_nof_bottom_fluxes()`. In dependence on the situation at the bottom boundary the settings for $\alpha_s$, $\beta_B$ (set_bottom_beta) as well as $C_B$ (set_bottom_concentration) must be chosen in a coordinated way. Bottom fluxes may be shared between species — use `get_which_bottom_flux(m)`.

As an example please refer to paragraph *Flux of scalar species through the bottom boundary* in the validation document.

B.3.5 Interface set_bottom_beta

**physical unit:** m/s

**default value:** 0.0 (if interface is *not* used)

Assign bottom flux parameter $\beta_s$ at polygon centers:

1. CALL set_bottom_beta(s): a constant $s$ [re|dp] is assigned to the bottom flux parameter $\beta_s$ for all scalar species $N_c$ and all polygons $N_p$;
2. CALL set_bottom_beta(m,s): a constant $s$ [re|dp] is assigned to the bottom flux parameter $\beta_s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) and all polygons $N_p$.
3. CALL set\_bottom\_beta(m, i, s): a constant \( s \) [re|dp] is assigned to the bottom flux parameter \( \beta_B \) for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\) in the \( i \)-th polygon \((1 \leq i \leq N_p)\);

4. CALL set\_bottom\_beta(m, a(:)): spatially varying data \( a(N_p) \) [re|dp] are assigned to the bottom flux parameter \( \beta_B \) for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\) at all \( N_p \) polygons;

5. CALL set\_bottom\_beta(b(:,:),): spatially varying data \( b(N_p, N_b) \) [re|dp], eventually different for some of the \( N_c \) scalar species, are assigned to the bottom flux parameter \( \beta_B \) for all \( N_b \) different bottom fluxes and all \( N_p \) polygons.

\( \beta_B \geq 0.0 \) must be fulfilled. \( N_p \) can be determined by means of \texttt{get\_nof\_polygons()} \( N_c \) from \texttt{get\_nof\_species()} and \( N_b \) from \texttt{get\_nof\_bottom\_fluxes()}. In dependence on the situation at the bottom boundary the settings for \( \beta_B, \alpha_B \) \texttt{set\_bottom\_alpha} as well as \( C_B \) \texttt{set\_bottom\_concentration} must be chosen in a coordinated way. Bottom fluxes may be shared between species — use \texttt{get\_which\_bottom\_flux(m)}.

As an example please refer to paragraph \textit{Flux of scalar species through the bottom boundary} in the \texttt{validation document}.

### B.3.6 Interface set\_bottom\_concentration

**physical unit:** identical to physical unit of specie used

**default value:** 0.0 (if interface is \textit{not} used)

Assign bottom concentration \( C_B \) at polygon centers:

1. CALL set\_bottom\_concentration(s): a constant \( s \) [re|dp] is assigned to the bottom concentration \( C_B \) for all scalar species \( N_c \) and all polygons \( N_p \);

2. CALL set\_bottom\_concentration(m, s): a constant \( s \) [re|dp] is assigned to the bottom concentration \( C_B \) for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\) and all polygons \( N_p \);

3. CALL set\_bottom\_concentration(m, i, s): a constant \( s \) [re|dp] is assigned to the bottom concentration \( C_B \) for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\) in the \( i \)-th polygon \((1 \leq i \leq N_p)\);

4. CALL set\_bottom\_concentration(m, a(:)): spatially varying data \( a(N_p) \) [re|dp] are assigned to the bottom concentration \( C_B \) for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\) at all \( N_p \) polygons;

5. CALL set\_bottom\_concentration(b(:,:),): spatially varying data \( b(N_p, N_b) \) [re|dp], eventually different for some of the \( N_c \) scalar species, are assigned to the bottom concentration \( C_B \) for all \( N_b \) different bottom fluxes and all \( N_p \) polygons.
$N_p$ can be determined by means of \texttt{get\_nof\_polygons()}, $N_c$ from \texttt{get\_nof\_species()} and $N_b$ from \texttt{get\_nof\_bottom\_fluxes()}. In dependence on the situation at the bottom boundary the settings for $C_s$, $\alpha_b$ (\texttt{set\_bottom\_alpha}) as well as $\beta_b$ (\texttt{set\_bottom\_beta}) must be chosen in a coordinated way. Bottom fluxes may be shared between species — use \texttt{get\_which\_bottom\_flux(m)}.

As an example please refer to paragraph \textit{Flux of scalar species through the bottom boundary} in the validation document.

**B.3.7 Interface \texttt{set\_bottom\_deviation}**

**physical unit:** m/s (positive upwards)

**default value:** 0.0 (if interface is not used)

Assign vertical bottom velocity $w_b$ at subpolygons:

1. CALL \texttt{set\_bottom\_deviation(s)}: a constant $s$ [re|dp] is assigned to the vertical bottom velocity $w_b$ at all subpolygons $N_p^{SG}$;

2. CALL \texttt{set\_bottom\_deviation(i,s)}: a constant $s$ [re|dp] is assigned to the vertical bottom velocity $w_b$ for all subpolygons $n_p^{SG}$ of the $i$-th polygon ($1 \leq i \leq N_p$);

3. CALL \texttt{set\_bottom\_deviation(i,a(:))}: spatially varying data $a(n_p^{SG})$ [re|dp] are assigned to the vertical bottom velocity $w_b$ at all $n_p^{SG}$ subpolygons of the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL \texttt{set\_bottom\_deviation(i,isg,s)}: a constant vertical bottom velocity $s$ [re|dp] is assigned to the $i^{SG}$-th subpolygon ($1 \leq i^{SG} \leq n_p^{SG}$) for the $i$-th polygon ($1 \leq i \leq N_p$);

5. CALL \texttt{set\_bottom\_deviation(a(:))}: spatially varying data $a(N_p^{SG})$ [re|dp] are assigned to the vertical bottom velocity $w_b$ at all $N_p^{SG}$ subpolygons.

$N_p$ is obtained by means of \texttt{get\_nof\_polygons()} $N_p^{SG}$ is given by \texttt{get\_nof\_subpolygons(i)}, $N_p^{SG}$ is returned from \texttt{get\_nof\_subpolygons()}.

**B.3.8 Interface \texttt{set\_bottom\_friction}**

**physical unit:** —

**default value:** $g/C_z^2$ (if interface is not used), with $C_z$ taken from the user’s input file \texttt{"untrim.inp"}

Assign bottom friction coefficient $\gamma_b$ at internal sides/edges on subedge level:
1. CALL set_bottom_friction(s): a constant bottom friction $s \, [\text{re}\,|\,\text{dp}]$ is assigned to all subedges for all internal sides $N_i$;

2. CALL set_bottom_friction(j,s): bottom friction is set for all subedges of the $j$-th side ($1 \leq j \leq N_i$) to a constant value $s \, [\text{re}\,|\,\text{dp}]$;

3. CALL set_bottom_friction(j,jSG,s): bottom friction $s \, [\text{re}\,|\,\text{dp}]$ is assigned to the $jSG$-th subedge ($1 \leq jSG \leq n_{SG}$) of the $j$-th side ($1 \leq j \leq N_i$);

4. CALL set_bottom_friction(j,a(:)): a spatially varying bottom friction $a(n_{SG},j) \, [\text{re}\,|\,\text{dp}]$ is assigned to the subedges of the $j$-th side ($1 \leq j \leq N_i$);

5. CALL set_bottom_friction(a(:)): a spatially varying bottom friction $a(N_{SG}) \, [\text{re}\,|\,\text{dp}]$ is assigned to all subedges $N_{SG}$.

$\gamma_f \geq 0.0$ must be fulfilled. $N_i$ is given by $\text{get_no_of_internal_edges()}$. $N_{SG}$ is obtained from $\text{get_no_of_subedges()}$ and $n_{SG}$ from $\text{get_no_of_subedges(j)}$. In dependence on the computation of $\gamma_f$, different types of bottom friction laws can be applied.

### B.3.9 Interface set_concentration

**physical unit:** identical to physical unit of specie used

**default value:** 0.0 (if interface is not used)

Assign concentration $C$ for scalar species at prism centers:

1. CALL set_concentration(s): a constant $s \, [\text{re}\,|\,\text{dp}]$ is assigned to the specie concentration $C$ for all scalar species $N_c$ and all prisms $I_i$;

2. CALL set_concentration(m,s): a constant $s \, [\text{re}\,|\,\text{dp}]$ is assigned to the specie concentration $C$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms $I_i$;

3. CALL set_concentration(m,i,s): a constant $s \, [\text{re}\,|\,\text{dp}]$ is assigned to the specie concentration $C$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL set_concentration(m,i,k,s): a constant $s \, [\text{re}\,|\,\text{dp}]$ is assigned to the specie concentration $C$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) in the $k$-th layer ($k_b(i) \leq k \leq k_b(i)$) above the $i$-th polygon ($1 \leq i \leq N_p$);

5. CALL set_concentration(m,a(:)): spatially varying data $a(I_i) \, [\text{re}\,|\,\text{dp}]$ are assigned to the specie concentration $C$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $I_i$ prisms;

6. CALL set_concentration(m,i,a(:)): depth varying data $a(N_z) \, [\text{re}\,|\,\text{dp}]$ are assigned to the specie concentration $C$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);
7. CALL set\_concentration(b(:,,:)): spatially varying data \( b(I,N_c) \) [re|dp], different for each of the \( N_c \) scalar species, are assigned to the concentration \( C \) for all \( N_c \) species and all \( I \) prisms.

The total number of prisms \( I \) is given by \( \text{get\_nof\_prisms()} \) \( N_p \) can be determined by means of \( \text{get\_nof\_polygons()} \) whereas \( N_c \) can be obtained from \( \text{get\_nof\_species()} \). The number of vertical layers \( N_z \) can be retrieved using \( \text{get\_nof\_layers()} \) while the bottom layer index \( k_b(i) \) is given by \( \text{get\_bottom\_prism(i)} \) and \( k_t(i) \) can be obtained from \( \text{get\_top\_prism(i)} \).

**B.3.10 Interface set\_concentration\_bc**

**Physical unit:** identical to physical unit of specie used

**Default value:** 0.0 (if interface is *not* used)

Assign concentration \( C \) for scalar species at prism centers along open boundaries with prescribed water level:

1. CALL set\_concentration\_bc\( (s) \): a constant \( s \) [re|dp] is assigned to the specie concentration at lateral open boundaries for all scalar species \( N_c \) and all prisms \( I^* \) above open boundary polygons;

2. CALL set\_concentration\_bc\( (m,s) \): a constant \( s \) [re|dp] is assigned to the specie concentration at lateral open boundaries for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) and all prisms \( I^* \) above open boundary polygons;

3. CALL set\_concentration\_bc\( (m,i,s) \): a constant \( s \) [re|dp] is assigned to the specie concentration at lateral open boundaries for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) and all prisms belonging to the computational column above the \( i \)-th open boundary polygon \( 1 \leq i \leq N_p^* \);

4. CALL set\_concentration\_bc\( (m,i,k,s) \): a constant \( s \) [re|dp] is assigned to the specie concentration at lateral open boundaries for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) in the \( k \)-th layer \( k_b(i) \leq k \leq k_t(i) \) above open boundary polygon \( 1 \leq i \leq N_p^* \);

5. CALL set\_concentration\_bc\( (m,a(:)) \): spatially varying data \( a(I^*) \) [re|dp] are assigned to the specie concentration at lateral open boundaries for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) at all \( I^* \) prisms above open boundary polygons;

6. CALL set\_concentration\_bc\( (m,i,a(:)) \): depth varying data \( a(N_z) \) [re|dp] are assigned to the specie concentration at lateral open boundaries for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) and all prisms belonging to the computational column above the \( i \)-th open boundary polygon \( 1 \leq i \leq N_p^* \);
7. CALL set_concentration_bc(b(:, :) ): spatially varying data b(Ii, Ni) [re|dp],
different for each of the Ni scalar species, are assigned to the concentration at lateral
open boundaries for all Ni species and all Ii prisms above open boundary polygons.

The total number of open boundary prisms is given by \( I_i^* = \sum_{i=1}^{N_p} (\text{get_no}f_{\text{prisms}}(i) + 1) \). 
\( N_p \) can be determined from \text{get_no}f_{\text{boundary_polygons}}() whereas \( N_c \) can be ob-
tained from \text{get_no}f_{\text{species}}(). The number of vertical layers \( N_z \) can be retrieved using
\text{get_no}f_{\text{layers}}() while the bottom layer index \( k_b(i) \) is given by \text{get_bottom_prism}(i)
and \( k_t(i) \) can be obtained from \text{get_top_prism}(i).
Eventually, for \( I_i^* \), a new get method could be introduced later.

\section*{B.3.11 Interface set constant coriolis}
\textbf{physical unit:} (mean) geographic latitude \( \Phi \)
\textbf{default value:} beta plane approximation with geographic latitude \( \Phi \) prescribed in
"untrim.grd" (if interface is \textit{not} used)

Assign constant value to Coriolis acceleration:

1. CALL set constant coriolis(): f-plane approximation for Coriolis force will be
used for mean geographic latitude \( \Phi \) prescribed in grid file as variable angle;

2. CALL set constant coriolis(s): f-plane approximation for Coriolis force will
be used for mean geographic latitude \( s \ [re|dp] \); in case \( s=0.0 \), no Coriolis force will
be applied at all.

\section*{B.3.12 Interface set density}
\textbf{physical unit:} —
\textbf{default value:} 1.0 (if interface is \textit{not} used)

Assign normalized water density \( \rho/\rho_0 \) at prism centers:

1. CALL set density(s): a constant \( s \ [re|dp] \) is assigned to the normalized water
density \( \rho/\rho_0 \) for all prisms \( I_i \);

2. CALL set density(i, s): a constant \( s \ [re|dp] \) is assigned to the normalized wa-
ter density \( \rho/\rho_0 \) for all prisms belonging to the computational column above the \( i \)-th
polygon \( (1 \leq i \leq N_p) \);

3. CALL set density(i, k, s): a constant \( s \ [re|dp] \) is assigned to the normalized
water density \( \rho/\rho_0 \) in the \( k \)-th layer \( (k_b(i) \leq k \leq k_t(i)) \) above the \( i \)-th polygon \( (1 \leq i \leq N_p) \);
4. CALL set_density(i,a(:)): depth varying data \( a(N_z) \) \([\text{re}\,\text{dp}]\) are assigned to the normalized water density \( \rho/\rho_0 \) for all prisms belonging to the computational column above the \( i \)-th polygon \((1 \leq i \leq N_p)\);

5. CALL set_density(a(:)): spatially varying data \( a(I_s) \) \([\text{re}\,\text{dp}]\) are assigned to the normalized water density \( \rho/\rho_0 \) for all prisms \( I_s \).

\( \rho > 0 \) must be fulfilled. The total number of prisms \( I_s \) is given by \text{get\_nof\_prisms()}\), whereas \( N_p \) can be determined by means of \text{get\_nof\_polylgons()}\). The number of vertical layers \( N_z \) can be retrieved using \text{get\_nof\_layers()}\), while the bottom layer index \( k_b(i) \) is given by \text{get\_bottom\_prism(i)}\) and \( k_t(i) \) can be obtained from \text{get\_top\_prism(i)}\).

**B.3.13 Interface set\_edge\_bottom\_deviation**

**physical unit:** m/s (positive upwards)

**default value:** 0.0 (if interface is not used)

Assign vertical bottom velocity \( w_B \) for subedges at all edges with flow:

1. CALL set_edge_bottom_deviation(s): a constant \( s \) \([\text{re}\,\text{dp}]\) is assigned to the vertical bottom velocity \( w_B \) at all subedges \( N_{SG}^s \);  
2. CALL set_edge_bottom_deviation(j,s): a constant \( s \) \([\text{re}\,\text{dp}]\) is assigned to the vertical bottom velocity for all subedges \( n_{SG}^j \) along the \( j \)-th edge \((1 \leq j \leq N_f)\);  
3. CALL set_edge_bottom_deviation(j,jsg,s): a constant \( s \) \([\text{re}\,\text{dp}]\) is assigned to the vertical bottom velocity for the \( j_{SG} \)-th subedge \((1 \leq j_{SG} \leq n_{SG}^j)\) along the \( j \)-th edge \((1 \leq j \leq N_f)\);  
4. CALL set_edge_bottom_deviation(j,a(:)): spatially varying values \( a(n_{sj}^j) \) \([\text{re}\,\text{dp}]\) are assigned to the bottom velocity at all subedges \( n_{SJ}^j \) of the \( j \)-th edge \((1 \leq j \leq N_f)\);  
5. CALL set_edge_bottom_deviation(a(:)): spatially varying bottom velocities \( a(N_{SJ}^j) \) \([\text{re}\,\text{dp}]\) are assigned to all subedges \( N_{SJ}^j \).

\( N_s \) is given by \text{get\_nof\_internal\_edges()}\) and \( N_f \) by \text{get\_nof\_inflow\_edges()}\). \( n_{SJ}^j \) is given by \text{get\_nof\_subedges\_s()}}\) \( N_{SJ}^j \) is returned from \text{get\_nof\_subedges()}\).

**B.3.14 Interface set\_elevation**

**physical unit:** m [reference level]

**default value:** 0.0 (if interface is not used)
Assign water surface elevation $\eta$ at polygon centers; positive values indicate that the water surface is located above the reference level (e.g. mean sea level), whereas for negative values the water surface lies below the reference level:

1. CALL `set_elevation(s)`: a constant $s \ [\text{re|dp}]$ is assigned to the water surface elevation $\eta$ for all polygons $N_p$;
2. CALL `set_elevation(i,s)`: a constant $s \ [\text{re|dp}]$ is assigned to the water surface elevation $\eta$ at the $i$-th polygon ($1 \leq i \leq N_p$);
3. CALL `set_elevation(a(:))`: spatially varying data $a \ (N_p) \ [\text{re|dp}]$ are assigned to the water surface elevation $\eta$ for each of the $N_p$ polygons.

$N_p$ is given by `get_no_of_polygons()`

### B.3.15 Interface `set_elevation_bc`

**physical unit:** m [reference level]

**default value:** 0.0 (if interface is not used)

Assign water surface elevation $\eta$ at polygon centers along open boundaries with prescribed water level:

1. CALL `set_elevation_bc(s)`: a constant $s \ [\text{re|dp}]$ is assigned to the water surface elevation $\eta$ for all open boundary polygons $N_p^*$;
2. CALL `set_elevation_bc(i,s)`: a constant $s \ [\text{re|dp}]$ is assigned to the water surface elevation $\eta$ at the $i$-th open boundary polygon ($1 \leq i \leq N_p^*$);
3. CALL `set_elevation_bc(a(:))`: spatially varying data $a \ (N_p^*) \ [\text{re|dp}]$ are assigned to the water surface elevation $\eta$ for each of the $N_p^*$ open boundary polygons.

$N_p^*$ is given by `get_no_of_boundary_polygons()`

### B.3.16 Interface `set_flux_limiter`

**physical unit:** text string, no unit

**default value:** "0 (upwind)"

Select method to be used for advection of scalar species:

1. CALL `set_flux_limiter(ch)`: a constant ch [character, len=80] is assigned to the internal steering variable of the program for all species $m \ (1 \leq m \leq N_c)$; the following options are available at the moment:
(a) "Minmod";
(b) "van Leer";
(c) "Superbee".

2. CALL set_flux_limiter(m, ch): a constant ch [character, len=80] is assigned to the flux limiter used for the m-th specie (1 ≤ m ≤ Nc); options see above.

Nc can be retrieved using \texttt{get nof_species()}. Default is "0 (upwind)", which corresponds to the method used in classic UnTRIM-versions.

\textbf{B.3.17 Interface \texttt{set\_grid\_file}}

\textbf{physical unit:} text string, no unit

\textbf{default value:} no default, must be always explicitly specified

Assign path and name of the grid file:

1. CALL set_grid_file(ch): a constant name ch [character] is assigned to the grid file name.

\textbf{B.3.18 Interface \texttt{set\_horizontal\_velocity}}

\textbf{physical unit:} m/s

\textbf{default value:} 0.0 (if interface is not used)

Assign the normal velocity component \( u \) at faces from user specified horizontal x-y-velocity components:

1. CALL set_horizontal_velocity(uu, vv): a constant horizontal velocity with given x-y-components uu, vv [re|dp] is assigned to the normal velocity component \( u \) for all computational faces \( J_j \);

2. CALL set_horizontal_velocity(j, uu, vv): a constant horizontal velocity with given x-y-components uu, vv [re|dp] is assigned to the normal velocity component \( u \) for all computational faces lying above the \( j \)-th side (1 ≤ j ≤ \( N_s \));

3. CALL set_horizontal_velocity(j, k, uu, vv): a constant horizontal velocity with given x-y-components uu, vv [re|dp] is assigned to the normal velocity component \( u \) for the computational face located in the \( k \)-th layer (\( k_b(j) ≤ k ≤ k_s(j) \)) above the \( j \)-th side (1 ≤ j ≤ \( N_s \));
4. CALL set\_horizontal\_velocity\(j, uu(:), vv\): a depth-varying horizontal velocity with given variable \(x\)- and constant \(y\)-component \(uu(N_i), vv [re | dp]\) is assigned to the normal velocity component \(u\) for the computational faces lying above the \(j\)-th side \((1 \leq j \leq N_i)\);

5. CALL set\_horizontal\_velocity\(j, uu, vv(:)\): a depth-varying horizontal velocity with given variable \(y\)- and constant \(x\)-component \(uu, vv (N_i) [re | dp]\) is assigned to the normal velocity component \(u\) for the computational faces lying above the \(j\)-th side \((1 \leq j \leq N_i)\);

6. CALL set\_horizontal\_velocity\(j, uu(:), vv(:)\): a depth-varying horizontal velocity with given \(x\)-\(y\)-components \(uu(N_i), vv (N_i) [re | dp]\) is assigned to the normal velocity component \(u\) for the computational faces lying above the \(j\)-th side \((1 \leq j \leq N_i)\);

7. CALL set\_horizontal\_velocity\(uu(:), vv\): a spatially varying horizontal velocity in \(x\)-direction and constant in \(y\)-direction with \(uu(J_i), vv [re | dp]\) is assigned to the normal velocity component \(u\) at all the \(J_i\) computational faces;

8. CALL set\_horizontal\_velocity\(uu, vv(:)\): a spatially varying horizontal velocity in \(y\)-direction and constant in \(x\)-direction with \(uu, vv (J_i) [re | dp]\) is assigned to the normal velocity component \(u\) at all the \(J_i\) computational faces;

9. CALL set\_horizontal\_velocity\(uu(:), vv(:)\): a spatially varying horizontal velocity with given \(x\)-\(y\)-components \(uu(J_i), vv (J_i) [re | dp]\) is assigned to the normal velocity component \(u\) at all the \(J_i\) computational faces.

The total number of faces \(J_i\) is determined by means of \texttt{get nof faces()} whereas \(N_i\) is obtained from \texttt{get nof internal egdes()). The number of vertical layers \(N_z\) can be retrieved using \texttt{get nof layers()} while the bottom layer index above edges \(k_b(j)\) is given by \texttt{get bottom face(j)} and \(k_t(j)\) can be obtained from \texttt{get top face(j)}.

### B.3.19 Interface set\_inflow\_bc

**physical unit:** \(m^3/s\)

**default value:** 0.0 (if interface is not used)

Assign discharge of water along open boundaries with prescribed flow (a positive value corresponds to inflow, a negative value is correlated with outflow):

1. CALL set\_inflow\_bc\(s\): a constant integral discharge \(s [re | dp]\) is equally distributed (constant flow per unit area) to all flow boundary edges \((N_i + 1 \leq j \leq N_{sf})\) with prescribed flow boundary condition;
2. CALL set_inflow_bc(j, s): a constant integral discharge \( s \) \([\text{re}|\text{dp}]\) is equally distributed (constant flow per unit area) to all faces above edge \( j \) \((N_b + 1 \leq j \leq N_f)\) where the prescribed flow boundary condition shall be applied.

3. CALL set_inflow_bc(j1, j2, s): a constant integral discharge \( s \) \([\text{re}|\text{dp}]\) is equally distributed (constant flow per unit area) to all faces above all edges \( j_1 \leq j \leq j_2 \) with \( N_b + 1 \leq j_1 \leq N_f \) and \( N_b + 1 \leq j_2 \leq N_f \) where the prescribed flow boundary condition shall be applied.

4. CALL set_inflow_bc(j1, j2, k, s): a constant \( s \) \([\text{re}|\text{dp}]\) is equally distributed (constant flow per unit area) to all faces within the \( k \)-th layer \((k_b(j) \leq k \leq k_t(j))\) for all edges \( j_1 \leq j \leq j_2 \) with \( N_b + 1 \leq j_1 \leq N_f \) and \( N_b + 1 \leq j_2 \leq N_f \) where the prescribed flow boundary condition shall be applied.

\( N_b \) is given by \( \text{get \_total \_internal \_edges()} \) and \( N_f \) by \( \text{Nc} + \text{get \_total \_inflow \_edges()} \), while the bottom layer index above edges \( k_b(j) \) is given by \( \text{get \_top \_face(j)} \) and \( k_t(j) \) can be obtained from \( \text{get \_bottom \_face(j)} \).

### B.3.20 Interface set_inflow_cc

**physical unit:** identical to physical unit of specie used

**default value:** 0.0 (if interface is not used)

Assign prescribed concentration for species along open boundaries with prescribed flow:

1. CALL set_inflow_cc(s): a constant \( s \) \([\text{re}|\text{dp}]\) is assigned to the specie concentration for all edges \((N_b + 1 \leq j \leq N_f)\) with prescribed flow boundary condition for all scalar species \( N_c \);

2. CALL set_inflow_cc(m, s): a constant \( s \) \([\text{re}|\text{dp}]\) is assigned to the specie concentration for all edges \((N_b + 1 \leq j \leq N_f)\) with prescribed flow boundary condition for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\);

3. CALL set_inflow_cc(m, j, s): a constant \( s \) \([\text{re}|\text{dp}]\) is assigned to the specie concentration above edge \( j \)-th \((N_b + 1 \leq j \leq N_f)\) with prescribed flow boundary condition for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\);

4. CALL set_inflow_cc(m, j1, j2, s): a constant \( s \) \([\text{re}|\text{dp}]\) is assigned to the specie concentration above all edges \( j_1 \leq j \leq j_2 \) with \( N_b + 1 \leq j_1 \leq N_f \) and \( N_b + 1 \leq j_2 \leq N_f \) with prescribed flow boundary condition for the \( m \)-th scalar specie \((1 \leq m \leq N_c)\);

5. CALL set_inflow_cc(m, j1, j2, k, s): a constant \( s \) \([\text{re}|\text{dp}]\) is assigned to the \( m \)-th specie concentration \((1 \leq m \leq N_c)\) above all edges \( j_1 \leq j \leq j_2 \) with \( N_b + 1 \leq j_1 \leq N_f \) and \( N_b + 1 \leq j_2 \leq N_f \) with prescribed flow boundary condition in the \( k \)-th layer \((k_b(j) \leq k \leq k_t(j))\).
\[ N_s \text{ is given by } \text{get\_nof\_internal\_edges()} \text{ and } N_{sf} \text{ by } N_s + \text{get\_nof\_inflow\_edges()} \]\n\[ \text{while the bottom layer index above edges } k_{b}(j) \text{ is given by } \text{get\_bottom\_face(j)} \text{ and } k_{t}(j) \text{ can be obtained from } \text{get\_top\_face(j)} \text{ whereas } N_c \text{ can be retrieved using } \text{get\_nof\_species()}\]

**B.3.21 Interface set\_input\_file**

**physical unit:** text string, no unit

**default value:** no default, must be always explicitly specified

Assign path and name of the input file:

1. CALL set\_input\_file(ch): a constant name ch [character] is assigned to the input file name.

**B.3.22 Interface set\_new\_bottom\_flux**

**physical unit:** number, no unit

**default value:** all species use identical parameters for bottom flux (if interface is not used)

Assign new bottom flux parameters:

1. CALL set\_new\_bottom\_flux(m): individual bottom flux parameters can be assigned for the \(m\)-th scalar specie \((1 \leq m \leq N_c)\).

\(N_c\) can be retrieved using \text{get\_nof\_species()} \text{ Bottom flux can be prescribed using set\_bottom\_alpha, set\_bottom\_beta and set\_bottom\_concentration}\

**B.3.23 Interface set\_new\_flux**

**physical unit:** specie number, no unit

**default value:** all species use identical parameters for bottom and surface flux (if interface is not used)

Assign new bottom and surface flux parameters:

1. CALL set\_new\_flux(m): individual surface and bottom flux parameters can be assigned for the \(m\)-th scalar specie \((1 \leq m \leq N_c)\).

\(N_c\) can be retrieved using \text{get\_nof\_species()} \text{ See set\_new\_surface\_flux and set\_new\_bottom\_flux for further explanations.}
B.3.24 Interface set_new_k

**Physical unit:** specie number, no unit

**Default value:** all species use identical horizontal and vertical turbulent diffusivity (if interface is not used)

Assign new horizontal and vertical turbulent diffusivity:

1. **CALL set_new_k(m):** individual horizontal and vertical turbulent diffusivity can be assigned for the $m$-th scalar specie ($1 \leq m \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` See `set_new_kh` and `set_new_kv` for further explanations.

B.3.25 Interface set_new_kh

**Physical unit:** specie number, no unit

**Default value:** all species use identical horizontal turbulent diffusivity (if interface is not used)

Assign new horizontal turbulent diffusivity:

1. **CALL set_new_kh(m):** individual horizontal turbulent diffusivity can be assigned for the $m$-th scalar specie ($1 \leq m \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` Horizontal turbulent diffusivity can be prescribed using `set_turbulent_h_diffusivity`.

B.3.26 Interface set_new_kv

**Physical unit:** specie number, no unit

**Default value:** all species use identical vertical turbulent diffusivity (if interface is not used)

Assign new vertical turbulent diffusivity:

1. **CALL set_new_kv(m):** individual vertical turbulent diffusivity can be assigned for the $m$-th scalar specie ($1 \leq m \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` Vertical turbulent diffusivity can be prescribed using `set_turbulent_v_diffusivity`.
### B.3.27 Interface set_new_settling_velocity

**physical unit:** specie number, no unit  
**default value:** all species use identical settling velocity (if interface is not used)

Assign new settling velocity:

1. CALL `set_new_settling_velocity(m)`: individual settling velocity can be assigned for the m-th scalar specie ($1 \leq m \leq N_c$).

   $N_c$ can be retrieved using `get_nof_species()`. Settling velocity can be prescribed using `set_settling_velocity`.

### B.3.28 Interface set_new_surface_flux

**physical unit:** specie number, no unit  
**default value:** all species use identical parameters for surface flux (if interface is not used)

Assign new surface flux parameters:

1. CALL `set_new_surface_flux(m)`: individual surface flux parameters can be assigned for the m-th scalar specie ($1 \leq m \leq N_c$).

   $N_c$ can be retrieved using `get_nof_species()`. Surface flux can be prescribed using `set_surface_alpha`, `set_surface_beta` and `set_surface_concentration`.

### B.3.29 Interface set_point_sources_concentration

**physical unit:** identical to physical unit of specie used  
**default value:** 0.0 (if interface is not used)

Assign point source concentration $C$ at prism centers:

1. CALL `set_point_sources_concentration(s)`: a constant $s$ [re | dp] is assigned to the source/sink concentration for all scalar species $N_c$ and every source/sink location $N_d$;

2. CALL `set_point_sources_concentration(m, s)`: a constant $s$ [re | dp] is assigned to the source/sink concentration for the m-th scalar specie ($1 \leq m \leq N_c$) and all source/sink locations $N_d$;

3. CALL `set_point_sources_concentration(n, m, s)`: a constant $s$ [re | dp] is assigned to the source/sink concentration for the m-th scalar specie ($1 \leq m \leq N_c$) at the n-th source/sink location ($1 \leq n \leq N_d$);
4. CALL set_point_sources_concentration(m, a(:,)): spatially varying data \( a(N_d) \) are assigned for the \( m \)-th specie \( (1 \leq m \leq N_s) \) to the \( N_d \) source/sink locations; 

5. CALL set_point_sources_concentration(b(:,)): spatially varying data \( b(N_d, N_c) \) are assigned for all species \( N_c \) and all source/sink locations \( N_d \). 

\( N_d \) is given by \texttt{get_no_of_point_sources()} whereas \( N_c \) can be retrieved using \texttt{get_no_of_species()}. During outflow, application of this routine has no effect on computed results.

### B.3.30 Interface set_point_sources_discharge

**physical unit:** m\(^3\)/s

**default value:** 0.0 (if interface is not used)

Assign point source discharge at prism centers (a positive value corresponds to inflow, a negative value is correlated with outflow):

1. CALL set_point_sources_discharge(s): a constant source/sink discharge \( s \) \( [\text{re|dp}] \) is assigned to all source/sink locations \( N_d \);

2. CALL set_point_sources_discharge(n, s): the source/sink discharge \( s \) \( [\text{re|dp}] \) is assigned to the \( n \)-th source/sink location \( (1 \leq n \leq N_d) \);

3. CALL set_point_sources_discharge(a(:)): a spatially varying discharge \( a(N_d) \) \( [\text{re|dp}] \) is assigned to all the \( N_d \) source/sink locations. 

\( N_d \) is given by \texttt{get_no_of_point_sources()}.

### B.3.31 Interface set_pressure

**physical unit:** m\(^2\)/s\(^2\)

**default value:** 0.0 (if interface is not used)

Assign non-hydrostatic normalized pressure component \( q \) at prism centers:

1. CALL set_pressure(s): a constant \( s \) \( [\text{re|dp}] \) is assigned to the normalized pressure component \( q \) for all computational prisms \( I_3 \);

2. CALL set_pressure(i, s): a constant \( s \) \( [\text{re|dp}] \) is assigned to the normalized pressure component \( q \) for all prisms belonging to the computational column above polygon \( i \) \( (1 \leq i \leq N_p) \);
3. CALL set\_pressure(i, k, s): a constant s [re/\(\text{dp}\)] is assigned to the normalized pressure component \(q\) for the \(k\)-th layer prism \((k_b(i) \leq k \leq k_t(i))\) above polygon \(i\) \((1 \leq i \leq N_p)\);

4. CALL set\_pressure(i, a(:)): depth varying data \(a(N_z) [\text{re/\(\text{dp}\)]\ are assigned to the normalized pressure \(q\) for all prisms above polygon \(i\) \((1 \leq i \leq N_p)\);

5. CALL set\_pressure(a(:)): spatially varying data \(a(I_3) [\text{re/\(\text{dp}\)]\ are assigned to the normalized pressure \(q\) for all \(I_3\) computational prisms.

The total number of prisms \(I_3\) is given by \text{get\_nof\_prisms()} whereas \(N_p\) can be determined from \text{get\_nof\_polygons()} The number of vertical layers \(N_z\) can be retrieved using \text{get\_nof\_layers()} while the bottom layer index \(k_b(i)\) is given by \text{get\_bottom\_prism(i)} and \(k_t(i)\) can be obtained from \text{get\_top\_prism(i)} To obtain the normalized pressure \(p\), the (real physical) pressure has to be divided by \(\rho_0\).

B.3.32 Interface set\_pressure\_bc

physical unit: m\(^2\)/s\(^2\)

default value: 0.0 (if interface is not used)

Assign non-hydrostatic normalized pressure component \(q\) at prism centers along open boundaries with prescribed water level:

1. CALL set\_pressure\_bc(s): a constant s [re/\(\text{dp}\)] is assigned to the normalized pressure component \(q\) for all computational prisms \(I_3^*\) along the open boundary;

2. CALL set\_pressure\_bc(i, s): a constant s [re/\(\text{dp}\)] is assigned to the normalized pressure component \(q\) for all prisms belonging to the computational column above open boundary polygon \(i\) \((1 \leq i \leq N_p^*)\);

3. CALL set\_pressure\_bc(i, k, s): a constant s [re/\(\text{dp}\)] is assigned to the normalized pressure component \(q\) for the \(k\)-th layer prism \((k_b(i) \leq k \leq k_t(i))\) above open boundary polygon \(i\) \((1 \leq i \leq N_p^*)\);

4. CALL set\_pressure\_bc(i, a(:)): depth varying data \(a(N_z) [\text{re/\(\text{dp}\)]\ are assigned to the normalized pressure \(q\) for all prisms above open boundary polygon \(i\) \((1 \leq i \leq N_p^*)\);

5. CALL set\_pressure\_bc(a(:)): spatially varying data \(a(I_3^*) [\text{re/\(\text{dp}\)]\ are assigned to the normalized pressure \(q\) for all \(I_3^*\) computational prisms along the open boundary.
The total number of open boundary prisms is given by $I^*_p = \sum_{i=1}^{N^*_p} (\text{get\_nof\_prisms}(i) + 1)$. $N^*_p$ can be determined from \text{get\_nof\_boundary\_polygons()}. The number of vertical layers $N_z$ can be retrieved using \text{get\_nof\_layers()}, while the bottom layer index $k_b(i)$ is given by \text{get\_bottom\_prism(i)} and $k_t(i)$ can be obtained from \text{get\_top\_prism(i)}. To obtain the normalized pressure $p$, the (real physical) pressure has to be divided by $\rho_0$.

Eventually, for $I^*_p$, a new get method could be introduced later.

**B.3.33  Interface set\_printout\_unit**

**physical unit:** -

**default value:** 6 (standard output)

Assign logical unit number for output of informative messages to file UnTRIM.txt.

1. CALL set\_printout\_unit(s): a constant $s$ [int] is assigned to the unit number.

If method is not used informative messages are sent to standard output instead.

**B.3.34  Interface set\_radiation\_time**

**physical unit:** 1/s

**default value:** $t_R$, taken from the user’s input file "untrim.inp" (if interface is not used)

Assign (inverse) relaxation time at open boundary polygons:

1. CALL set\_radiation\_time(s): a constant relaxation time $s$ [re|dp] is assigned to all polygons at open boundaries $N^*_p$;

2. CALL set\_radiation\_time(i,s): relaxation time is set for the $i$-th polygon ($1 \leq i \leq N^*_p$) to a constant value $s$ [re|dp];

3. CALL set\_radiation\_time(a(:)): a spatially varying relaxation time $a(N^*_p)$ [re|dp] is assigned to all polygons at open boundaries $N^*_p$.

$N^*_p$ is given by \text{get\_nof\_boundary\_polygons()}. For high values of $t_R$, the water surface elevation at the open boundary will be closer to the prescribed boundary values assigned using \text{set\_elevation\_bc}. Small values for $t_R$ allow some degree of flexibility to which the resulting water level may depart from the prescribed one.
B.3.35 Interface set_rain

**physical unit:** mm/h

**default value:** 0.0 (if interface is not used)

Assign rate of rainfall (positive values) or evaporation (negative values) $R_r$ at polygon centers:

1. CALL `set_rain(s)`: a constant rate of rainfall or evaporation $s$ [re|dp] is assigned to all polygons $N_p$;
2. CALL `set_rain(i,s)`: rate of rainfall or evaporation is set for the $i$-th polygon ($1 \leq i \leq N_p$) to a constant value $s$ [re|dp];
3. CALL `set_rain(a(:))`: a spatially varying rate of rainfall or evaporation $a(N_p)$ [re|dp] is assigned to all polygons $N_p$.

$N_p$ is obtained from `get_nof_polygons()`.

B.3.36 Interface set_sediment

**physical unit:** —

**default value:** 0.0 (if interface is not used)

Assign a factor to each of the scalar species which indicates active settling ($> 0.0$), passive non-settling ($= 0.0$), or active floating ($< 0.0$) behaviour:

1. CALL `set_sediment(s)`: a constant factor $s$ [re|dp] is assigned to all species $N_c$;
2. CALL `set_sediment(a(:))`: a species-dependent constant factor $a(N_c)$ [re|dp] is assigned to all species $N_c$;
3. CALL `set_sediment(m,s)`: a constant factor $s$ [re|dp] is assigned to the $m$-th scalar species ($1 \leq m \leq N_c$).

$N_c$ is given by `get_nof_species()`. This interface allows the user to deal with several sediment fractions differing in settling velocity by a constant factor.

If settling velocities for different fractions should differ not only by a constant factor you may use an appropriate combination of the methods `set_new_settling_velocity`, `share_settling_velocity` and `set_settling_velocity`. 
### B.3.37 Interface `set_settling_velocity`

**physical unit:** m/s

**default value:** 0.0 (if interface is *not* used)

Assign a spatially varying settling velocity $w^s$ at the top of prisms for all actively settling species:

1. CALL `set_settling_velocity(s)`: a constant $s \ [\text{re|dp}]$ is assigned to the settling velocity $w^s$ for all computational prisms $I_i$ and all species $N_c$;

2. CALL `set_settling_velocity(m,s)`: a constant $s \ [\text{re|dp}]$ is assigned to the settling velocity $w^s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) and all prisms $I_i$;

3. CALL `set_settling_velocity(m,i,s)`: a constant $s \ [\text{re|dp}]$ is assigned to the settling velocity $w^s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL `set_settling_velocity(m,i,k,s)`: a constant $s \ [\text{re|dp}]$ is assigned to the settling velocity $w^s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) in the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$) above the $i$-th polygon ($1 \leq i \leq N_p$);

5. CALL `set_settling_velocity(m,a(:))`: spatially varying data $a(I_i) \ [\text{re|dp}]$ are assigned to the settling velocity $w^s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) at all $I_i$ prisms;

6. CALL `set_settling_velocity(m,i,a(:))`: depth varying data $a(N_z) \ [\text{re|dp}]$ are assigned to the settling velocity $w^s$ for the $m$-th scalar species ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);

7. CALL `set_settling_velocity(a(:,:))`: spatially varying data $a(I_i,N_{ws}) \ [\text{re|dp}]$, eventually different for the $N_c$ scalar species, are assigned to the settling velocity $w^s$ for each of the $I_i$ computational prisms.

The total settling velocity used in the computational run is determined by the values set using this interface and multiplied with specie-dependent factor (CALL `set_sediment`). The total number of prisms $I_i$ is given by `get_nof_prisms()` $N_p$ can be determined from `get_nof_polygons()`. $N_c$ can be obtained from `get_nof_species()` whereas $N_{ws}$ can be retrieved using `get_nof_settling_velocities()`. The number of vertical layers $N_z$ can be retrieved using `get_nof_layers()` while the bottom layer index $k_b(i)$ is given by `get_bottom_prism(i)` and $k_t(i)$ can be obtained from `get_top_prism(i)`. Settling velocities may be shared between species — use `get_which_settling_velocity(m)`.
**B.3.38 Interface set_slope**

**physical unit:** no unit

**default value:** [0.0,0.0] (if interface is not used)

Assign a prescribed slope in x- and y-direction:

1. CALL set_slope(sx,sy): a constant slope sx, sy [re|dp] is assigned to the x- as well as y-direction.

A positive slope corresponds to an increase of height in the respective co-ordinate direction. A slope of 0.001 is equivalent to a 1 m difference in height for each km.

**B.3.39 Interface set_source_file**

**physical unit:** text string, no unit

**default value:** no default, must be always explicitly specified

Assign path and name of the sources and sinks file:

1. CALL set_source_file(ch): a constant name ch [character] is assigned to the sources and sinks file name.

**B.3.40 Interface set_subedge_depth**

**physical unit:** m

**default value:** from file "untrim.grd" (if interface is not used)

Assign depth to subedges along sides/edges with flow:

1. CALL set_subedge_depth(s): a constant depth s [re|dp] is assigned to all subedges \( N_{s,i}^{SG} \);

2. CALL set_subedge_depth(j,s): a constant depth s [re|dp] is assigned to all subedges \( n_{s,i}^{SG} \) along the \( j \)-th edge \( (1 \leq j \leq N_{s,j}) \);

3. CALL set_subedge_depth(j,jsg,s): a constant depth s [re|dp] is assigned to the \( j^{SG} \)-th subedge \( (1 \leq j^{SG} \leq n_{s,j}^{SG}) \) for the \( j \)-th edge \( (1 \leq j \leq N_{s,j}) \);

4. CALL set_subedge_depth(j,a(:)): spatially varying depth values \( a(n_{s,j}^{SG}) \) [re|dp] are assigned to all subedges \( n_{s,j}^{SG} \) of the \( j \)-th edge \( (1 \leq j \leq N_{s,j}) \);
5. CALL set_subedge_depth(a(:)): spatially varying depth values 
a \( N_s^{SG} \) [re|dp] are assigned to all subedges \( N_s^{SG} \).

\( N_s \) is given by \( \text{get\_nof\_internal\_edges()} \) and \( N_s \) by \( \text{get\_nof\_inflow\_edges()} \).

\( n_s^{SG} \) is given by \( \text{set\_nof\_subedges(j)} \). \( N_s^{SG} \) is returned from \( \text{set\_nof\_subedges()} \).

### B.3.41 Interface set_subpolygon_depth

**physical unit:** m

**default value:** from file "untrim.grd" (if interface is not used)

Assign depth to subpolygons within computational polygons:

1. CALL set_subpolygon_depth(s): a constant depth \( s \) [re|dp] is assigned to all subpolygons \( N_s^{SG} \);

2. CALL set_subpolygon_depth(i,s): a constant depth \( s \) [re|dp] is assigned to all subpolygons \( n_s^{SG} \) of the \( i \)-th polygon \( 1 \leq i \leq N_p \);

3. CALL set_subpolygon_depth(i,isp,m,s): a constant depth \( s \) [re|dp] is assigned to the \( m \)-th subpolygon \( 1 \leq m \leq n_s^{SG} \) for the \( i \)-th polygon \( 1 \leq i \leq N_p \);

4. CALL set_subpolygon_depth(i,a(:)): spatially varying depth values \( a(n_s^{SG}) \) [re|dp] are assigned to all subpolygons \( n_s^{SG} \) of the \( i \)-th polygon \( 1 \leq i \leq N_p \);

5. CALL set_subpolygon_depth(a(:)): spatially varying depth values \( a(N_s^{SG}) \) [re|dp] are assigned to all subpolygons \( N_s^{SG} \).

\( N_p \) is obtained by means of \( \text{get\_nof\_polygons()} \), \( n_s^{SG} \) is given by \( \text{get\_nof\_subpolygons(1)} \), \( N_s^{SG} \) is returned from \( \text{get\_nof\_subpolygons()} \).

### B.3.42 Interface set_surface_alpha

**physical unit:** (physical unit of specie used) \( \times m/s \)

**default value:** 0.0 (if interface is not used)

Assign surface flux parameter \( \alpha_r \) at polygon centers:

1. CALL set_surface_alpha(s): a constant \( s \) [re|dp] is assigned to the surface flux parameter \( \alpha_r \) for all scalar species \( N_c \) and all polygons \( N_p \);

2. CALL set_surface_alpha(m,s): a constant \( s \) [re|dp] is assigned to the surface flux parameter \( \alpha_r \) for the \( m \)-th scalar specie \( 1 \leq m \leq N_c \) and all polygons \( N_p \).

---

B. User interface
3. CALL set_surface_alpha(m,i,s): scalar s [re|dp] is assigned to the surface flux parameter $\alpha_r$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) in the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL set_surface_alpha(m,a(:)): spatially varying data $a(N_p)$ [re|dp] are assigned to the surface flux parameter $\alpha_r$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $N_p$ polygons;

5. CALL set_surface_alpha(b(:,:)): spatially varying data $b(N_p,N_t)$ [re|dp], eventually different for some of the $N_c$ scalar species, are assigned to the surface flux parameter $\alpha_r$ for all $N_t$ different surface fluxes and all $N_p$ polygons.

$\alpha_r \geq 0.0$ must be fulfilled. $N_p$ can be determined by means of \texttt{get_no_polygons()} $N_c$ from \texttt{get_no_species()} and $N_t$ from \texttt{get_no_surface_fluxes()}.

In dependence on the situation at the water surface boundary the settings for $\alpha_r$, $\beta_r$ (\texttt{set_surface_beta}) as well as $C_r$ (\texttt{set_surface_concentration}) must be chosen in a coordinated way. Surface fluxes may be shared between species — use \texttt{get_which_surface_flux(m)}.

As an example please refer to paragraph \textit{Flux of scalar species through the free-surface boundary} in the validation document.

### B.3.43 Interface set_surface_beta

**physical unit:** m/s

**default value:** 0.0 (if interface is \textit{not} used)

Assign surface flux parameter $\beta_r$ at polygon centers:

1. CALL set_surface_beta(s): a scalar s [re|dp] is assigned to the surface flux parameter $\beta_r$ for all scalar species $N_c$ and polygons $N_p$;

2. CALL set_surface_beta(m,s): a scalar s [re|dp] is assigned to the surface flux parameter $\beta_r$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all polygons $N_p$;

3. CALL set_surface_beta(m,i,s): scalar s [re|dp] is assigned to the surface flux parameter $\beta_r$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) in the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL set_surface_beta(m,a(:)): spatially varying data $a(N_p)$ [re|dp] are assigned to the surface flux parameter $\beta_r$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $N_p$ polygons;

5. CALL set_surface_beta(b(:,:)): spatially varying data $b(N_p,N_r)$ [re|dp], eventually different for some of the $N_c$ scalar species, are assigned to the surface flux parameter $\beta_r$ for all $N_r$ different surface fluxes and all $N_p$ polygons.
\[ \beta_T \geq 0.0 \] must be fulfilled. \( N_p \) can be determined by means of \texttt{get\_nof\_polygons()} \( N_c \) from \texttt{get\_nof\_species()} and \( N_f \) from \texttt{get\_nof\_surface\_fluxes()} In dependence on the situation at the water surface boundary the settings for \( \beta_T, \alpha_T \) \texttt{set\_surface\_alpha} as well as \( C_T \) \texttt{set\_surface\_concentration} must be chosen in a coordinated way. Surface fluxes may be shared between species — use \texttt{get\_which\_surface\_flux(m)}

As an example please refer to paragraph \textit{Flux of scalar species through the free-surface boundary} in the \texttt{validation document}.

### B.3.44 Interface \texttt{set\_surface\_concentration}

**physical unit:** identical to physical unit of specie used

**default value:** 0.0 (if interface is \textit{not} used)

Assign surface concentration \( C_T \) at polygon centers:

1. \texttt{CALL set\_surface\_concentration(s)}: a constant \( s \) \{re|dp\} is assigned to the surface concentration \( C_T \) for all scalar species \( N_c \) and all polygons \( N_p \);

2. \texttt{CALL set\_surface\_concentration(m,s)}: a constant \( s \) \{re|dp\} is assigned to the surface concentration \( C_T \) for the \( m \)-th scalar specie \( (1 \leq m \leq N_c) \) and all polygons \( N_p \);

3. \texttt{CALL set\_surface\_concentration(m,i,s)}: scalar \( s \) \{re|dp\} is assigned to the surface concentration \( C_T \) for the \( m \)-th scalar specie \( (1 \leq m \leq N_c) \) in the \( i \)-th polygon \( (1 \leq i \leq N_p) \);

4. \texttt{CALL set\_surface\_concentration(m,a(:))}: spatially varying data \( a \) \( (N_p) \) \{re|dp\} are assigned to the surface concentration \( C_T \) for the \( m \)-th scalar specie \( (1 \leq m \leq N_c) \) at all \( N_p \) polygons;

5. \texttt{CALL set\_surface\_concentration(b(:,,:))}: spatially varying data \( b \) \( (N_p,N_f) \) \{re|dp\}, eventually different for some of the \( N_c \) scalar species, are assigned to the surface concentration \( C_T \) for all \( N_f \) different surface fluxes and all \( N_p \) polygons.

\( N_p \) can be determined by means of \texttt{get\_nof\_polygons()} \( N_c \) from \texttt{get\_nof\_species()} and \( N_f \) from \texttt{get\_nof\_surface\_fluxes()} In dependence on the situation at the water surface boundary the settings for \( \beta_T, \alpha_T \) \texttt{set\_surface\_alpha} as well as \( \beta_T \texttt{set\_surface\_beta} \) must be chosen in a coordinated way. Surface fluxes may be shared between species — use \texttt{get\_which\_surface\_flux(m)}

As an example please refer to paragraph \textit{Flux of scalar species through the free-surface boundary} in the \texttt{validation document}.
B.3.45 Interface set_time

physical unit: s

default value: 0.0 (if interface is not used)

Assign simulation time:

1. CALL set_time(s): value s [re|dp] is assigned to the simulation time.

B.3.46 Interface set_turbulent_h_diffusivity

physical unit: m²/s

default value: 0.0 (if interface is not used)

Assign horizontal turbulent diffusivity $K^h$ at face centers:

1. CALL set_turbulent_h_diffusivity(s): a constant s [re|dp] is assigned to the horizontal turbulent diffusivity $K^h$ for all computational faces $J_i$ and all scalar species $N_c$;

2. CALL set_turbulent_h_diffusivity(m,s): a constant s [re|dp] is assigned to the horizontal turbulent diffusivity $K^h$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all computational faces $J_i$;

3. CALL set_turbulent_h_diffusivity(m,j,s): a constant s [re|dp] is assigned to the horizontal turbulent diffusivity $K^h$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all faces above the $j$-th edge ($1 \leq j \leq N_s$);

4. CALL set_turbulent_h_diffusivity(m,j,k,s): scalar s [re|dp] is assigned to the horizontal turbulent diffusivity $K^h$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) in the $k$-th layer ($k_b(j) \leq k \leq k_t(j)$) and all faces above the $j$-th edge ($1 \leq j \leq N_s$);

5. CALL set_turbulent_h_diffusivity(m,a(:)): spatially varying data $a(J_i)$ [re|dp] are assigned to the horizontal turbulent diffusivity $K^h$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $J_i$ computational faces;

6. CALL set_turbulent_h_diffusivity(m,j,a(:)): depth varying data $a(N_z)$ [re|dp] are assigned to the horizontal turbulent diffusivity $K^h$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all faces above edge $j$ ($1 \leq j \leq N_s$);

7. CALL set_turbulent_h_diffusivity(a(:,)): spatially varying data $a(J_i,N_z^h)$ [re|dp], eventually different for some of the $N_c$ scalar species, are assigned to the horizontal turbulent diffusivity $K^h$ for each of the $J_i$ computational faces.
\( K^h \geq 0.0 \) must be fulfilled. The total number of faces \( J_i \) is determined by means of \texttt{get\_nof\_faces()} \( N_i \), obtained from \texttt{get\_nof\_internal\_edges()} \( N_i \), is obtained from \texttt{get\_nof\_species()} whereas \( N^h_k \) can be retrieved using \texttt{get\_nof\_h\_diffusivities()}.

The number of vertical layers \( N_z \) can be retrieved using \texttt{get\_nof\_layers()} while the bottom layer index above edges \( k_b(j) \) is given by \texttt{get\_bottom\_face(j)} and \( k_t(j) \) can be obtained from \texttt{get\_top\_face(j)}. Horizontal diffusivities may be shared between species — use \texttt{get\_which\_h\_diffusivity(m)}.

\begin{itemize}
  \item \textbf{B.3.47 Interface set\textunderscore turbulent\textunderscore h\textunderscore viscosity}
  \item \textbf{physical unit: m}^2/\textbf{s}
  \item \textbf{default value: 0.0 (if interface is \textit{not} used)}
\end{itemize}

Assign horizontal turbulent viscosity \( \nu^h \) at face centers:

1. CALL \texttt{set\_turbulent\_h\_viscosity(s)}: a constant \( s \) \( \texttt{[re|dp]} \) is assigned to the horizontal turbulent viscosity \( \nu^h \) for all computational faces \( J_i \);

2. CALL \texttt{set\_turbulent\_h\_viscosity(j,s)}: a constant \( s \) \( \texttt{[re|dp]} \) is assigned to the horizontal turbulent viscosity \( \nu^h \) for all computational faces located above the \( j \)-th edge \( (1 \leq j \leq N_i) \);

3. CALL \texttt{set\_turbulent\_h\_viscosity(j,k,s)}: scalar \( s \) \( \texttt{[re|dp]} \) is assigned to the horizontal turbulent viscosity \( \nu^h \) for computational face within the \( k \)-th layer \( (k_b(j) \leq k \leq k_t(j)) \) above the \( j \)-th edge \( (1 \leq j \leq N_i) \);

4. CALL \texttt{set\_turbulent\_h\_viscosity(j,a(:))}: depth varying data \( a(N_i) \) \( \texttt{[re|dp]} \) are assigned to the horizontal turbulent viscosity \( \nu^h \) for all computational faces above edge \( j \) \( (1 \leq j \leq N_i) \);

5. CALL \texttt{set\_turbulent\_h\_viscosity(a(:))}: spatially varying data \( a(J_i) \) \( \texttt{[re|dp]} \) are assigned to the horizontal turbulent viscosity \( \nu^h \) for each of the \( J_i \) computational faces.

\( \nu^h \geq 0.0 \) must be fulfilled. The total number of faces \( J_i \) is determined by means of \texttt{get\_nof\_faces()} whereas \( N_i \) is obtained from \texttt{get\_nof\_internal\_edges()} The number of vertical layers \( N_z \) can be retrieved using \texttt{get\_nof\_layers()} while the bottom layer index above edges \( k_b(j) \) is given by \texttt{get\_bottom\_face(j)} and \( k_t(j) \) can be obtained from \texttt{get\_top\_face(j)}.
B.3.48 Interface set_turbulent_v_diffusivity

physical unit: $m^2/s$

default value: 0.0 (if interface is not used)

Assign vertical turbulent diffusivity $K_v$ at top of prisms:

1. CALL set_turbulent_v_diffusivity(s): a constant $s$ [re|dp] is assigned to the vertical turbulent diffusivity $K_v$ for all scalar species $N_c$ and all prisms $I_i$;

2. CALL set_turbulent_v_diffusivity(m,s): a constant $s$ [re|dp] is assigned to the vertical turbulent diffusivity $K_v$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms $I_i$;

3. CALL set_turbulent_v_diffusivity(m,i,s): a constant $s$ [re|dp] is assigned to the vertical turbulent diffusivity $K_v$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);

4. CALL set_turbulent_v_diffusivity(m,i,k,s): scalar $s$ [re|dp] is assigned to the vertical turbulent diffusivity $K_v$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) in the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$) above the $i$-th polygon ($1 \leq i \leq N_p$);

5. CALL set_turbulent_v_diffusivity(m,a(:)): spatially varying data $a(I_i)$ [re|dp] are assigned to the vertical turbulent diffusivity $K_v$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) at all $I_i$ prisms;

6. CALL set_turbulent_v_diffusivity(m,i,a(:)): depth varying data $a(N_z)$ [re|dp] are assigned to the vertical turbulent diffusivity $K_v$ for the $m$-th scalar specie ($1 \leq m \leq N_c$) and all prisms belonging to the computational column above the $i$-th polygon ($1 \leq i \leq N_p$);

7. CALL set_turbulent_v_diffusivity(a(:,:)): spatially varying data $a(I_i,N_z)$ [re|dp], eventually different for the $N_c$ scalar species, are assigned to the vertical turbulent diffusivity $K_v$ for each of the $I_i$ computational prisms.

$K_v \geq 0.0$ must be fulfilled. The total number of prisms $I_i$ is given by get_nof_prisms(). $N_p$ can be determined from get_nof_polygons(). $N_c$ can be obtained from get_nof_species(). whereas $N_z$ can be retrieved using get_nof_v_diffusivities(). The number of vertical layers $N_z$ can be retrieved using get_nof_layers() while the bottom layer index $k_b(i)$ is given by get_bottom_prism(i) and $k_t(i)$ can be obtained from get_top_prism(i). Vertical diffusivities may be shared between species — use get_which_v_diffusivity(m).

B.3.49 Interface set_turbulent_v_viscosity

physical unit: $m^2/s$

B. User interface
**default value:** 0.0 (if interface is not used)

Assign vertical turbulent viscosity \( \nu^v \) at the top of prisms:

1. `CALL set_turbulent_v viscosity(s)`: a constant \( s \) [re|dp] is assigned to the vertical turbulent viscosity \( \nu^v \) for all computational prisms \( I_i \);

2. `CALL set_turbulent_v viscosity(i,s)`: a constant \( s \) [re|dp] is assigned to the vertical turbulent viscosity \( \nu^v \) for all computational prisms located above the \( i \)-th polygon \( (1 \leq i \leq N_p) \);

3. `CALL set_turbulent_v viscosity(i,k,s)`: scalar \( s \) [re|dp] is assigned to the vertical turbulent viscosity \( \nu^v \) for computational prism within the \( k \)-th layer \( (k_b(i) \leq k \leq k_t(i)) \) above the \( i \)-th polygon \( (1 \leq i \leq N_p) \);

4. `CALL set_turbulent_v viscosity(i,a(:))`: depth varying data \( a(N_z) \) [re|dp] are assigned to the vertical turbulent viscosity \( \nu^v \) for all computational prisms above polygon \( i \) \( (1 \leq i \leq N_p) \);

5. `CALL set_turbulent_v viscosity(a(:))`: spatially varying data \( a(I_j) \) [re|dp] are assigned to the vertical turbulent viscosity \( \nu^v \) for each of the \( I_j \) computational prisms.

\( \nu^v \geq 0.0 \) must be fulfilled. The total number of prisms \( I_i \) is given by `get nof prisms()`, whereas \( N_p \) can be determined from `get nof polygons()`. The number of vertical layers \( N_z \) can be retrieved using `get nof layers()` while the bottom layer index \( k_b(i) \) is given by `get bottom prism(i)` and \( k_t(i) \) can be obtained from `get top prism(i)`.

**B.3.50 Interface set_velocity**

**physical unit:** m/s

**default value:** 0.0 (if interface is not used)

Assign horizontal normal velocity component \( u \) at faces:

1. `CALL set_velocity(s)`: a constant \( s \) [re|dp] is assigned to the normal velocity component \( u \) for all computational faces \( J_j \);

2. `CALL set_velocity(j,s)`: a constant \( s \) [re|dp] is assigned to the normal velocity component \( u \) for all computational faces above the \( j \)-th side \( (1 \leq j \leq N_y) \);

3. `CALL set_velocity(j,k,s)`: scalar \( s \) [re|dp] is assigned to the normal velocity component \( u \) for the computational face lying in the \( k \)-th layer \( (k_b(j) \leq k \leq k_t(j)) \) above the \( j \)-th side \( (1 \leq j \leq N_y) \);
4. CALL set_velocity(j, a(:)): depth varying data a(Nz) [re | dp] are assigned to the normal velocity component $u$ for the computational faces lying above the $j$-th side ($1 \leq j \leq N_j$);

5. CALL set_velocity(a()): spatially varying data a(Js) [re | dp] are assigned to the normal velocity component $u$ at all the $J_s$ computational faces.

The total number of faces $J_s$ is determined by means of \texttt{get\_nof\_faces()} whereas $N_s$ is obtained from \texttt{get\_nof\_edges()} The number of vertical layers $N_z$ can be retrieved using \texttt{get\_nof\_layers()} while the bottom layer index above edges $k_b(j)$ is given by \texttt{get\_bottom\_face()} and $k_t(j)$ can be obtained from \texttt{get\_top\_face()} $N_s$ is given by \texttt{get\_nof\_internal\_edges()} and $N_f$ by $N_f + \texttt{get\_nof\_inflow\_edges()}$.

### B.3.51 Interface set_wet_and_dry

**physical unit:** m / no unit  
**default value:** 0.0 / 100 (if interface is not used)

Assign wet and dry parameters:

1. CALL set_wet_and_dry(s): a constant value $s$ [re | dp] is assigned to the artificial porosity $p_y$ for all flow edges ($1 \leq j \leq N_f$);

2. CALL set_wet_and_dry(n): the maximum number of (outer) Newton iterations $m_{\text{max}}^N$ is set to $n$ [int];

3. CALL set_wet_and_dry(n, s): the maximum number of (outer) Newton iterations $m_{\text{max}}^N$ is set to $n$ [int], and a constant value $s$ [re | dp] is assigned to the artificial porosity $p_y$ for all flow edges ($1 \leq j \leq N_f$). $p_y \geq 0.0$ as well as $m_{\text{max}}^N \geq 1$ must be fulfilled. $N_s$ is given by \texttt{get\_nof\_internal\_edges()} and $N_f$ by $N_i + \texttt{get\_nof\_inflow\_edges()}$.

### B.3.52 Interface set_wind_friction

**physical unit:** —  
**default value:** $1.75 \times 10^{-6}$ (if interface is not used)

Assign wind friction coefficient $\gamma_T$ at internal sides/edges:

1. CALL set_wind_friction(s): a constant wind friction $s$ [re | dp] is assigned to all internal sides $N_i$;
2. CALL set\_wind\_friction\(j, s\): wind friction is set for the \(j\)-th side \((1 \leq j \leq N_s)\) to a constant value \(s\) [re \(\mid\) dp];

3. CALL set\_wind\_friction\(a(:)\): a spatially varying wind friction \(a(N_s)\) [re \(\mid\) dp] is assigned to all internal sides \(N_s\).

\(\gamma_r \geq 0.0\) must be fulfilled. \(N_s\) is obtained from get\_nof\_internal\_edges(). In dependence on the computation of \(\gamma_r\), different types of wind friction laws can be applied. In case \(\gamma_r = 0.0\) no wind friction will be applied.

As an example please refer to paragraph Wind friction in the validation document.

### B.3.53 Interface set\_wind\_velocity

**physical unit:** m/s

**default value:** 0.0 (if interface is not used)

Assign components of the wind velocity \((u_w, v_w)\) at internal sides/edges:

1. CALL set\_wind\_velocity\((wx, wy)\): constant \(x\)- and \(y\)-components \(wx, wy\) [re \(\mid\) dp] are assigned to the wind velocity \((u_w, v_w)\) for all internal sides \(N_s\);

2. CALL set\_wind\_velocity\((j, wx, wy)\): \(x\)- and \(y\)-components \(wx, wy\) [re \(\mid\) dp] are assigned to the wind velocity \((u_w, v_w)\) at the \(j\)-th side \((1 \leq j \leq N_s)\);

3. CALL set\_wind\_velocity\((wx(:), wy)\): spatially varying \(x\)- and constant \(y\)-component \(wx(N_s), wy\) [re \(\mid\) dp] are assigned to the wind velocity \((u_w, v_w)\) for all internal sides \(N_s\);

4. CALL set\_wind\_velocity\((wx, wy(:))\): spatially varying \(y\)- and constant \(x\)-component \(wx, wy(N_s)\) [re \(\mid\) dp] are assigned to the wind velocity \((u_w, v_w)\) for all internal sides \(N_s\);

5. CALL set\_wind\_velocity\((wx(:), wy(:))\): spatially varying \(x\)- and \(y\)-components \(wx(N_s), wy(N_s)\) [re \(\mid\) dp] are assigned to the wind velocity \((u_w, v_w)\) for all internal sides \(N_s\).

\(N_s\) is obtained from get\_nof\_internal\_edges(). It is normally assumed, that the wind velocity is known 10 m above the free water surface. Wind friction vanishes in case of a zero wind friction coefficient only – see set\_wind\_friction – a zero wind velocity is not sufficient.
B.3.54 Interface share_bottom_flux

**physical unit:** specie number, no unit

**default value:** species may use different parameters for bottom flux if method `set_new_bottom_flux(m)` was applied before and this interface is *not* used.

Share bottom flux parameters:

1. CALL `share_bottom_flux(m1,m2)`: the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share all bottom flux parameters with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` See `set_new_bottom_flux` for further informations on how to set individual bottom flux parameters for different species.

B.3.55 Interface share_flux

**physical unit:** specie number, no unit

**default value:** species may use different parameters for bottom and surface flux if method `set_new_flux(m)` was applied before and this interface is *not* used.

Share bottom and surface flux parameters:

1. CALL `share_flux(m1,m2)`: the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share all bottom and surface flux parameters with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` See `share_bottom_flux` and `share_surface_flux` for further explanations.

B.3.56 Interface share_k

**physical unit:** specie number, no unit

**default value:** species may use different horizontal and vertical turbulent diffusivities if method `set_new_k(m)` was applied before and this interface is *not* used.

Share horizontal and vertical turbulent diffusivities:

1. CALL `share_k(m1,m2)`: the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share horizontal and vertical turbulent diffusivity with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using `get_nof_species()` See `set_new_k` for further informations on how to set individual horizontal and vertical turbulent diffusivities for different species.
B.3.57 Interface share_kh

physical unit: specie number, no unit

default value: species may use different horizontal turbulent diffusivities if method \texttt{set\_new\_kh(m)} was applied before and this interface is not used

Share horizontal turbulent diffusivities:

1. CALL share\_kh(m1,m2): the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share horizontal turbulent diffusivity with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using \texttt{get\_nof\_species()}. See \texttt{set\_new\_kh} for further informations on how to set individual horizontal turbulent diffusivities for different species.

B.3.58 Interface share_kv

physical unit: specie number, no unit

default value: species may use different vertical turbulent diffusivities if method \texttt{set\_new\_kv(m)} was applied before and this interface is not used

Share vertical turbulent diffusivities:

1. CALL share\_kv(m1,m2): the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share vertical turbulent diffusivity with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using \texttt{get\_nof\_species()}. See \texttt{set\_new\_kv} for further informations on how to set individual vertical turbulent diffusivities for different species.

B.3.59 Interface share\_settling\_velocity

physical unit: specie number, no unit

default value: species may use different settling velocity if method \texttt{set\_new\_settling\_velocity(m)} was applied before and this interface is not used

Share settling velocity:

1. CALL share\_settling\_velocity(m1,m2): the $m_2$-th scalar specie ($1 \leq m_2 \leq N_c$) will share settling velocity with the $m_1$-th scalar specie ($1 \leq m_1 \leq N_c$).

$N_c$ can be retrieved using \texttt{get\_nof\_species()}. See \texttt{set\_new\_settling\_velocity} for further informations on how to set individual settling velocities for different species.
B.3.60 Interface share_surface_flux

**physical unit:** specie number, no unit

**default value:** species may use different parameters for surface flux if method \[ \text{set.new.surface.flux}(m) \] was applied before and this interface is *not* used

Share surface flux parameters:

1. CALL \[ \text{share.surface.flux}(m_1,m_2) \]: the \( m_2 \)-th scalar specie (\( 1 \leq m_2 \leq N_c \)) will share all surface flux parameters with the \( m_1 \)-th scalar specie (\( 1 \leq m_1 \leq N_c \)).

\( N_c \) can be retrieved using \[ \text{get.nof.species()} \]. See \[ \text{set.new.surface.flux} \] for further informations on how to set individual surface flux parameters for different species.
B.4 Get-functions

The get-functions are grouped according to their functionality:

- logical unit numbers
  - printer file
    * `get_printout_unit`
- grid
  - static size
    * `get_nof_boundary_polygons`
    * `get_nof_inflow_edges`
    * `get_nof_internal_edges`
    * `get_nof_edges`
    * `get_nof_polygons`
    * `get_nof_subedges`
    * `get_nof_subpolygons`
    * `get_nof_vertices`
    * `get_nof_max_edges`
  - dynamic size
    * `get_nof_wet_polygons`
    * `get_nof_wet_subedges`
    * `get_nof_wet_subpolygons`
- static variables
  * `get_angle`
  * `get_dx_min`
  * `get_location`
- static structure
  * `get_adjacent_polygon`
  * `get_edge_begin`
  * `get_edge_end`
  * `get_left_polygon`
  * `get_right_polygon`
  * `get_polygon`
  * `get_polygon_edge`
  * `get_polygon_vertex`
- static geometry
  * `get_area`
* get_center_distance
* get_edge_length
* get_edge_center
* get_polygon_area
* get_polygon_center
* get_vertex_coordinates
* get_x_vertex_coordinate
* get_y_vertex_coordinate

- dynamic geometry
  * get_length
  * get_dry_length
  * get_wet_length
  * get_max_depth
  * get_avr_depth
  * get_max_total_depth
  * get_avr_total_depth
  * get_dry_area
  * get_wet_area
  * get_subpolygon_area
  * get_subpolygon_depth
  * get_subpolygon_total_depth
  * get_max_edge_depth
  * get_avr_edge_depth
  * get_max_edge_total_depth
  * get_avr_edge_total_depth
  * get_edge_wet_area
  * get_subedge_depth
  * get_subedge_total_depth
  * get_subedge_length
  * is_dry_edge
  * is_wet_edge
  * is_dry_polygon
  * is_wet_polygon

- computational grid
  - static size
    * get_nof_faces
    * get_nof_layers
• computational steering data
  – file names
    * get_grid_file
    * get_input_file
    * get_source_file
  – transport algorithm for scalar species
* `get_flux_limiter`
  - scalar species
    * `get_nof_point_sources`
    * `get_nof_species`
    * `get_nof_bottom_fluxes`
    * `get_nof_surface_fluxes`
    * `get_nof_fluxes`
    * `get_which_bottom_flux`
    * `get_which_surface_flux`
    * `get_which_flux`
    * `get_nof_h_diffusivities`
    * `get_nof_v_diffusivities`
    * `get_nof_diffusivities`
    * `get_which_h_diffusivity`
    * `get_which_v_diffusivity`
    * `get_which_diffusivity`
    * `get_nof_settling_velocities`
    * `get_which_settling_velocity`
  - logical data
    * `sediment`
    * `with_hydrodynamic_pressure`
    * `with_scalar_transport`
  - time
    * `get_time`
    * `get_time_step`
  - iterative solvers
    * `get_elevation_tolerance`
    * `get_pressure_tolerance`
    * `get_nof_iterations`
    * `get_nof_inner_iterations`
    * `get_nof_outer_iterations`
    * `get_nof_max_iterations`
  - transport equation solver
    * `get_nof_substeps`

- computational results

- hydrodynamics
• computational parameters and coefficients
  – scalars
    * \texttt{get\_dt\_min}
    * \texttt{get\_gravity}
    * \texttt{get\_chezy}
    * \texttt{get\_radiation\_time}
    * \texttt{get\_theta}
    * \texttt{get\_horizontal\_viscosity}
    * \texttt{get\_vertical\_viscosity}
    * \texttt{get\_horizontal\_diffusivity}
    * \texttt{get\_vertical\_diffusivity}
  – equation of state data
    * \texttt{get\_density}
  – turbulence related data
    * \texttt{get\_turbulent\_h\_viscosity}
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

* **get_turbulent_v-viscosity**
  * **get_turbulent_h-diffusivity**
  * **get_turbulent_v-diffusivity**

- data related to sources and sinks
  * **get_source_polygon**
  * **get_source_layer**
  * **get_source_discharge**
  * **get_source_concentration**

- hydrodynamic boundary data
  * **get_bottom_stress**
  * **get_bottom_friction**
  * **get_wind_stress**
  * **get_wind_friction**
  * **get_wind_velocity**
  * **get_wind_velocity_x**
  * **get_wind_velocity_y**
  * **get_atmospheric_pressure**

- species boundary data
  * **get_bottom_alpha**
  * **get_bottom_beta**
  * **get_bottom_concentration**
  * **get_bottom_flux**
  * **get_surface_alpha**
  * **get_surface_beta**
  * **get_surface_concentration**
  * **get_surface_flux**

- sediment data
  * **get_sediment**
  * **get_settling_velocity**

- data related to morphodynamics
  * **get_bottom_deviation**
  * **get_avr_bottom_deviation**
  * **get_edge_bottom_deviation**
  * **get_avr_edge_bottom_deviation**

- system properties
  - artificial porosity
The index of an adjacent polygon is less than or equal to zero if there is no adjacent polygon available. The latter is the case if the side of a polygon is a boundary edge. The number of sides $S_i$ for polygon $i$ can be determined by means of $\text{get\_nof\_edges}(i)$ whereas the total number of polygons $N_p$ can be retrieved from $\text{get\_nof\_polygons}()$.

For easy access to the maximum value use $\text{get\_nof\_max\_edges}$.

B.4.2 Interface $\text{get\_angle}$

physical unit: degree north

Extract geographic latitude $\Phi$:

1. $\text{get\_angle}()$: returns scalar $s$ [dp] with geographic latitude ($-90 \leq \Phi \leq 90$).

Prescribed in grid file file untrim.grd or modified using $\text{set\_constant\_coriolis}(\Phi)$.
B.4.3 Interface `get_area`

**physical unit:** m²

Extract total computational area:

1. `get_area(i)`: returns scalar s [dp] with total area $\sum_{i=1}^{N_p} a_{i,p}^G$ (sum of subpolygon areas) for the $i$-th polygon ($1 \leq i \leq N_p$);

2. `get_area()`: returns scalar s [dp] with total area of the computational domain (sum of all subpolygon areas in all $N_p$ polygons).

The total number of polygons $N_p$ can be retrieved calling `get_no_of_polygons()`. If you want to retrieve the geometric area of a polygon use `get_polygon_area` instead; the geometric area $A(i)$ is always larger or equal than the area used in computation.

B.4.4 Interface `get_artificial_porosity`

**physical unit:** m

Retrieve artificial porosity at sides/edges with flow:

1. `get_artificial_porosity(j)`: returns scalar s [dp] with artificial porosity for the $j$-th edge ($1 \leq j \leq N_{sf}$);

2. `get_artificial_porosity()`: returns array $a(N_{sf})$ [dp] with artificial porosity for all edges ($1 \leq j \leq N_{sf}$).

$N_k$ is given by `get_no_of_internal_edges()` and $N_{sf}$ by $N_k + \text{get_no_of_inflow_edges()}$.

B.4.5 Interface `get_atmospheric_pressure`

**physical unit:** m²/s²

Retrieve normalized atmospheric pressure $p_a$ at polygon centers:

1. `get_atmospheric_pressure(i)`: returns scalar s [dp] with $p_a$ at polygon $i$ ($1 \leq i \leq N_p$);

2. `get_atmospheric_pressure()`: array $a(N_p)$ [dp] is returned with $p_a$ at all $N_p$ polygons.

To determine $N_p$ use `get_no_of_polygons()`. Please notice that to obtain the (real physical) atmospheric pressure the normalized atmospheric pressure $p_a$ has to be multiplied by $\rho_0$. 
B.4.6 Interface get_avr_bottom_deviation

**physical unit:** m/s (positive upwards)

retrieve average (weighted with subpolygon areas) vertical bottom velocity \( w_B \) at polygons:

1. `get_avr_bottom_deviation(i)`: scalar \( s \) [dp] is returned with average vertical bottom velocity for polygon \( i (1 \leq i \leq N_p) \);

2. `get_avr_bottom_deviation()`: array \( a(N_p) \) [dp] is returned with average vertical bottom velocity for all \( N_p \) polygons.

\( N_p \) is obtained by means of `get_nof_polygons()`.

For access to vertical bottom velocity on subgrid level use `get_bottom_deviation` instead.

B.4.7 Interface get_avr_depth

**physical unit:** m

Extract average depth within polygons, which is the (area weighted) average value of all subpolygon depth values within a polygon:

1. `get_avr_depth(i)`: returns scalar \( s \) [dp] with average depth for polygon \( i (1 \leq i \leq N_p) \);

2. `get_avr_depth()`: array \( a(N_p) \) [dp] is returned with average depth data for all \( N_p \) polygons, where \( a(i) \) is equivalent to the average depth at the \( i \)-th polygon.

The total number of polygons \( N_p \) can be determined by means of `get_nof_polygons()`.

For maximum depth use `get_max_depth` instead.

B.4.8 Interface get_avr_edge_bottom_deviation

**physical unit:** m/s (positive upwards)

Retrieve averaged (weighted with subedge lengths) vertical bottom velocity \( w_B \) at edges with flow:

1. `get_avr_edge_bottom_deviation(j)`: scalar \( s \) [dp] is returned with average vertical bottom velocity for edge \( j (1 \leq j \leq N_e) \);

2. `get_avr_edge_bottom_deviation()`: array \( a(N_e) \) [dp] is returned with average vertical bottom velocity for all \( N_e \) edges with flow.

\( N_e \) is given by `get_nof_internal_edges()` and \( N_f \) by \( N_e + \text{get_nof_inflow_edges()} \).

For access to vertical bottom velocity on subgrid level use `get_edge_bottom_deviation` instead.
B.4.9 Interface get_avr_edge_depth

**physical unit:** m

Extract average depth for sides/edges with flow, which is the (length weighted) average value of all subedge depths along an edge:

1. `get_avr_edge_depth(j)`: returns scalar \( s_{dp} \) with average depth along edge \( j \) \((1 \leq j \leq N_{sf})\);

2. `get_avr_edge_depth()`: array \( a(N_{sf}) \) \([dp]\) is returned with average depth data for all \( N_{sf} \) edges, where \( a(j) \) is equivalent to the average depth along the \( j \)-th edge.

\( N_{si} \) is given by `get_nof_internal_edges()` and \( N_{sf} \) by \( N_{si}+\text{get_nof_inflow_edges()} \). For maximum depth use `get_max_edge_depth` instead.

B.4.10 Interface get_avr_edge_total_depth

**physical unit:** m

Extract mean total water depth (from the free surface to the bottom) for all sides/edges with flow, which is the (length weighted) value evaluated for all respective subedges:

1. `get_avr_edge_total_depth(j)`: returns scalar \( s_{dp} \) with mean total water depth along edge \( j \) \((1 \leq j \leq N_{sf})\);

2. `get_avr_edge_total_depth()`: array \( a(N_{sf}) \) \([dp]\) is returned with mean total water depth data for all \( N_{sf} \) edges, where \( a(j) \) is equivalent to the mean total water depth along the \( j \)-th edge.

\( N_{si} \) is given by `get_nof_internal_edges()` and \( N_{sf} \) by \( N_{si}+\text{get_nof_inflow_edges()} \). For maximum total water depth use `get_max_edge_total_depth` instead.

B.4.11 Interface get_avr_face_height

**physical unit:** m

Extract average (weighted with subedge lengths) height \( \Delta z \) for faces above edges:

1. `get_avr_face_height(j,k)`: returns scalar \( s_{dp} \) with the actual average face height for the \( j \)-th edge \((1 \leq j \leq N_{sf})\) and \( k \)-th layer \((k_b(j) \leq k \leq k_i(j))\);

2. `get_avr_face_height(j)`: array \( a(N_z) \) \([dp]\) is returned, wherein \( a(k) \) corresponds to the actual average face height of the \( k \)-th layer for edge \( j \).
3. get_avr_face_height(): array \( b(J_i) \) [dp] with the actual average heights for all faces \( J_i \) is retrieved.

The total number of sides (edges) \( N_s \) can be determined calling get_no_of_edges() whereas the number of level surfaces (layers) \( N_z \) can be obtained from get_no_of_layers() \( J_s \) can be obtained from get_no_of_faces() while the bottom layer index \( k_b(j) \) is given by get_bottom_face() and the respective surface layer index results from get_top_face(). \( N_s \) is given by get_no_of_internal_edges() and \( N_{s_j} \) by \( N_s + \text{get_no_inflow_edges}() \).

For maximum face heights use get_max_face_height instead.

B.4.12 Interface get_avr_prism_height

physical unit: \( \text{m} \)

Extract average (weighted with subpolygon area) height (with water) for prisms:

1. get_avr_prism_height \((i, k)\): returns scalar \( s \) [dp] with average height at the \( i \)-th polygon \((1 \leq i \leq N_p)\) within layer \( k \) \((k_b(i) \leq k \leq k_t(i))\);

2. get_avr_prism_height \((i)\): array \( a(N_i) \) [dp] is returned with average heights for all computational prisms above polygon \( i \) \((1 \leq i \leq N_p)\), where \( a(k) \) corresponds to the average height within layer \( k \) \((k_b(i) \leq k \leq k_t(i))\);

3. get_avr_prism_height(): array \( a(I_3) \) [dp] is returned with average heights for all computational prisms \( I_3 \).

The total number of prisms \( I_3 \) is given by get_no_of_prisms() whereas \( N_p \) can be determined by means of get_no_of_polygons(). The number of vertical layers \( N_z \) can be retrieved using get_no_of_layers() while the bottom layer index \( k_b(i) \) is given by get_bottom_prism() and \( k_t(i) \) can be obtained from get_top_prism(). For maximum prism heights use get_max_prism_height instead.

B.4.13 Interface get_avr_total_depth

physical unit: \( \text{m} \)

Extract average total water depth (from the free surface to the bottom) within polygons, which is the (area weighted) average value for all subpolygon total water depth values within a polygon:

1. get_avr_total_depth \((i)\): returns scalar \( s \) [dp] with average total water depth for polygon \( i \) \((1 \leq i \leq N_p)\);
2. `get_avr_total_depth()`: array `a(Np)` [dp] is returned with average total water depth data for all `Np` polygons, where `a(i)` is equivalent to the average total water depth at the `i`-th polygon.

The total number of polygons `Np` can be determined by means of `get_nof_polygons()`. For maximum total water depth use `get_max_total_depth` instead.

### B.4.14 Interface `get_bottom_alpha`

**physical unit**: (physical unit of specie used) × m/s

Retrieve bottom flux parameter $\alpha_a$ at polygon centers:

1. `get_bottom_alpha(m,i)`: returns scalar s [dp] with $\alpha_a$ for the `m`-th specie ($1 \leq m \leq N_s$) at the `i`-th polygon ($1 \leq i \leq N_p$);
2. `get_bottom_alpha(m)`: array `a(Np)` [dp] is returned with $\alpha_a$ for the `m`-th specie ($1 \leq m \leq N_s$) at all `Np` polygons, where `a(i)` contains $\alpha_a$ for the `i`-th polygon;
3. `get_bottom_alpha()`: array `b(N_p,N_s)` [dp] is returned, where `b(i,m)` contains $\alpha_a$ for the `m`-th bottom flux component ($1 \leq m \leq N_s$) at polygon `i`.

`N_s` can be retrieved using `get_nof_species()`; `N_p` is obtained from `get_nof_polygons()`; `N_s` from `get_nof_bottom_fluxes()` and `m` from `get_which_bottom_flux(m)`.

### B.4.15 Interface `get_bottom_beta`

**physical unit**: m/s

Retrieve bottom flux parameter $\beta_a$ at polygon centers:

1. `get_bottom_beta(m,i)`: returns scalar s [dp] with $\beta_a$ for the `m`-th specie ($1 \leq m \leq N_s$) at the `i`-th polygon ($1 \leq i \leq N_p$);
2. `get_bottom_beta(m)`: array `a(Np)` [dp] is returned with $\beta_a$ for the `m`-th specie ($1 \leq m \leq N_s$) at all `Np` polygons, where `a(i)` contains $\beta_a$ for the `i`-th polygon;
3. `get_bottom_beta()`: array `b(N_p,N_s)` [dp] is returned, where `b(i,m)` contains $\beta_a$ for the `m`-th bottom flux component ($1 \leq m \leq N_s$) at polygon `i`.

`N_s` can be retrieved using `get_nof_species()`; `N_p` is obtained from `get_nof_polygons()`; `N_s` from `get_nof_bottom_fluxes()` and `m` from `get_which_bottom_flux(m)`.
B.4.16 Interface get_bottom_concentration

**physical unit:** identical to physical unit of specie used

Retrieve prescribed bottom concentration $C_b$ at polygon centers:

1. `get_bottom_concentration(m,i)`: returns scalar $s$ [dp] with $C_b$ for the $m$-th specie ($1 \leq m \leq N_c$) at the $i$-th polygon ($1 \leq i \leq N_p$);
2. `get_bottom_concentration(m)`: array $a(N_p)$ [dp] is returned with $C_b$ for the $m$-th specie ($1 \leq m \leq N_c$) at all $N_p$ polygons, where $a(i)$ contains $C_b$ for the $i$-th polygon;
3. `get_bottom_concentration()`: array $b(N_p,N_b)$ [dp] is returned, where $b(i,m_B)$ contains $C_b$ for the $m_B$-th bottom flux component ($1 \leq m_B \leq N_b$) at polygon $i$.

$N_c$ can be retrieved using `get_nof_species()`, $N_p$ is obtained from `get_nof_polygons()`, $N_b$ from `get_nof_bottom_fluxes()` and $m_B$ from `get_which_bottom_flux(m)`.

B.4.17 Interface get_bottom_deviation

**physical unit:** m/s (positive upwards)

Retrieve vertical bottom velocity $w_B$ at subpolygons:

1. `get_bottom_deviation(i,isg)`: scalar $s$ [dp] is returned with vertical bottom velocity for the $i_{SG}$-th subpolygon ($1 \leq i_{SG} \leq n_{SG}p$) within polygon $i$ ($1 \leq i \leq N_p$);
2. `get_bottom_deviation(i)`: array $a(n_{SG}p,i)$ [dp] is returned with vertical bottom velocity for all $n_{SG}p$ subpolygons belonging to polygon $i$ ($1 \leq i \leq N_p$);
3. `get_bottom_deviation()`: array $a(N_{SG}p)$ [dp] is returned with vertical bottom velocity for all $N_{SG}p$ subpolygons.

$N_p$ is obtained by means of `get_nof_polygons()`, $n_{SG}p$ is given by `get_nof_subpolygons(i)` and $N_{SG}p$ is returned from `get_nof_subpolygons()`.

Use `get_avr_bottom_deviation` to access average vertical bottom velocity on polygon level.

B.4.18 Interface get_bottom_face

**physical unit:** –

Evaluate layer index $k$ ($1 \leq k \leq N_z$) for the lowermost (bottom) face above the side of a polygon:
1. `get_bottom_face(j)`: returns scalar \( s[\text{int}] \) with bottom face layer index \( k \) for side \( j (1 \leq j \leq N_{sf}) \);

2. `get_bottom_face()`: array \( a(N_{sf})[\text{int}] \) is returned with bottom face layer indices for all \( N_{sf} \) sides with flow, where \( a(j) \) represents the bottom layer index for the \( j \)-th side of the grid.

\( N_{s} \) is given by `get_nof_internal_edges()` and \( N_{sf} \) by \( N_{s} + \text{get_nof_inflow_edges()} \).

For unerodible layer index use `get_unerodible_bottom_face` instead.

### B.4.19 Interface `get_bottom_flux`

**physical unit:** "physical unit of specie used" \( \times m^3 \)

Retrieve computed bottom flux \( q_{b} \) at polygon centers:

1. `get_bottom_flux(m,i)`: returns scalar \( s[\text{dp}] \) with \( q_{b} \) for the \( m \)-th specie \( (1 \leq m \leq N_{c}) \) at the \( i \)-th polygon \( (1 \leq i \leq N_{p}) \);

2. `get_bottom_flux(m)`: scalar \( s[\text{dp}] \) is returned with integral bottom flux \( q_{b} \) for the \( m \)-th specie \( (1 \leq m \leq N_{c}) \) summed for all \( N_{p} \) polygons;

3. `get_bottom_flux()`: array \( a(N_{c})[\text{dp}] \) is returned, where \( a(m) \) is the integral bottom flux \( q_{b} \) for specie \( m \) summed for all \( N_{p} \) polygons.

\( N_{c} \) can be retrieved using `get_nof_species()`. \( N_{p} \) is given by `get_nof_polygons()`.

### B.4.20 Interface `get_bottom_friction`

**physical unit:** —

Retrieve bottom friction coefficient \( \gamma_{b} \) at internal sides on subedge level:

1. `get_bottom_friction(j,jSG)`: returns scalar \( s[\text{dp}] \) with \( \gamma_{b} \) at the \( j^{SG} \)-th subedge of the \( j \)-th side \( (1 \leq j \leq N_{s}) \);

2. `get_bottom_friction(j)`: returns array \( a(n_{j}^{SG})[\text{dp}] \) with \( \gamma_{b} \) for all \( n_{j}^{SG} \) subedges of the \( j \)-th side \( (1 \leq j \leq N_{s}) \);

3. `get_bottom_friction()`: array \( a(N_{j}^{SG})[\text{dp}] \) is returned with \( \gamma_{b} \) for all \( N_{j}^{SG} \) subedges where \( a(j) \) contains \( \gamma_{b} \) for the \( j \)-th overall subedge.

\( N_{s} \) is given by `get_nof_internal_edges()`. \( N_{j}^{SG} \) is obtained from `get_nof_subedges()` and \( n_{j}^{SG} \) from `get_nof_subedges(j)`.
B.4.21 Interface get_bottom_prism

physical unit: —

Evaluate layer index $k$ ($1 \leq k \leq N_z$) for the lowermost (bottom) prism above a polygon:

1. get_bottom_prism(i): returns scalar $s$ [int] with bottom prism layer index $k$ for polygon $i$ ($1 \leq i \leq N_p$);
2. get_bottom_prism(): array $a(N_p)$ [int] is returned with bottom prism layer indices for all $N_p$ polygons, where $a(i)$ represents the bottom layer index for the $i$-th polygon.

The total number of polygons $N_p$ can be determined by means of get_nof_polygons(). For unerodible layer index use get_unerodible_bottom_prism instead.

B.4.22 Interface get_bottom_stress

physical unit: m$^2$/s$^2$

Retrieve (normalized) normal component of bottom shear stress at internal edges for all layers at subedge level:

1. get_bottom_stress(j, j$^{SG}$): returns scalar $s$ [dp] with stress for the $j^{SG}$-th subedge of the $j$-th side ($1 \leq j \leq N_s$);
2. get_bottom_stress(j): returns array $a(n_{s,j}^{SG})$ [dp] with stress for all $n_{s,j}^{SG}$ subedges of the $j$-th side ($1 \leq j \leq N_s$);
3. get_bottom_stress(): array $a(N_{s}^{SG})$ [dp] is returned with stress for all $N_{s}^{SG}$ subedges where $a(j)$ contains the stress for the $j$-th overall subedge.

$N_s$ is given by get_nof_internal_edges(). $N_{s}^{SG}$ is obtained from get_nof_subedges() and $n_{s,j}^{SG}$ from get_nof_subedges(). To obtain the (real physical) normal component of the bottom shear stress $\tau_B$, the retrieved normalized one has to be multiplied by $\rho_0$.

B.4.23 Interface get_bottom_subface

physical unit: —

Evaluate layer index $k$ ($1 \leq k \leq N_z$) for the lowermost (bottom) subface above a subedge:

1. get_bottom_subface(j, j$^{SG}$): returns scalar $s$ [int] with bottom subface layer index $k$ for edge $j$ ($1 \leq j \leq N_n + N_s$) and subedge $j^{SG}$ ($1 \leq j^{SG} \leq n_{s,j}^{SG}$);
2. `get_bottom_subface(j)`: array `a(nSG_j) [int]` is returned with bottom subface layer indices for all `nSG_j` subedges, where `a(j)` represents the bottom layer index for the `j`-th subedge.

3. `get_bottom_subface()`: array `a(NSG) [int]` is returned with bottom subface layer indices for all `NSG` subedges, where `a(j)` represents the bottom layer index for the `j`-th (overall) subedge.

The total number of subedges `NSG` can be determined by means of `get_nof_subedges()` and `nSG_j` from `get_nof_subedges(j)`.

---

**B.4.24 Interface get_bottom_subprism**

**Physical unit:** —

Evaluate layer index `k (1 ≤ k ≤ N)` for the lowermost (bottom) subprism above a subpolygon:

1. `get_bottom_subprism(i,iSG)`: returns scalar `s [int]` with bottom subprism layer index `k` for polygon `i (1 ≤ i ≤ Np)` and subpolygon `iSG (1 ≤ iSG ≤ nSG_p));

2. `get_bottom_subprism(i)`: array `a(nSG_p, i) [int]` is returned with bottom subprism layer indices for all `nSG_p` subpolygons, where `a(iSG)` represents the bottom layer index for the `iSG`-th subpolygon.

3. `get_bottom_subprism()`: array `a(NSG_p) [int]` is returned with bottom subprism layer indices for all `NSG_p` subpolygons, where `a(i)` represents the bottom layer index for the `i`-th (overall) subpolygon.

The total number of subpolygons `NSG_p` can be determined by means of `get_nof_subpolygons()` and `nSG_p` from `get_nof_subpolygons(i)`.

---

**B.4.25 Interface get_celerity**

**Physical unit:** m/s

Retrieve wave celerity `c = \sqrt{gH}` at edges:

1. `get_celerity(j)`: returns scalar `s [dp]` with wave celerity based on average edge depth `\overline{H}` for edge `j (1 ≤ j ≤ Nsj)`;

2. `get_celerity()`: returns array `a(Nsj) [dp]` with wave celerities for all edges with flow `Nsj`.

`Nsj` is given by `get_nof_internal_edges(i)` and `Nsj` by `Nsj + get_nof_inflow_edges()`.
B.4.26 Interface \texttt{get\_center\_distance}

physical unit: \textit{m}

Retrieve distance between adjacent polygon centers \(\delta\):

1. \texttt{get\_center\_distance}(\(j\)): returns scalar \(s\) [dp] with distance between the polygon centers adjacent to the \(j\)-th side (\(1 \leq j \leq N_{si}\));

2. \texttt{get\_center\_distance}(): array \(a(N_{si})\) [dp] is returned with \(\delta\) for all \(N_{si}\) sides, where \(a(j)\) corresponds to the center distance between polygons adjacent to the \(j\)-th side.

The number of internal sides \(N_{si}\) can be determined calling \texttt{get\_nof\_internal\_edges()}. The condition \(\delta \geq \delta_{\text{min}}\) always holds true. The minimum allowed distance between polygon centers \(\delta_{\text{min}}\) can be obtained from \texttt{get\_dx\_min()}. 

B.4.27 Interface \texttt{get\_chezy}

physical unit: \textit{m}^{1/2}/s

Retrieve constant Chezy friction factor (prescribed in steering data file \texttt{untrim.inp}):

1. \texttt{get\_chezy}(): scalar \(s\) [dp] is returned with actual value used for the Chezy coefficient.

Normally the bottom friction coefficient is varying in space and time. Use \texttt{get\_bottom\_friction} under these circumstances.

B.4.28 Interface \texttt{get\_concentration}

physical unit: identical to physical unit of specie used

Retrieve specie concentration \(C\) at prism centers:

1. \texttt{get\_concentration}(\(m, i, k\)): returns scalar \(s\) [dp] with \(C\) for the \(m\)-th specie (\(1 \leq m \leq N_{c}\)) at polygon \(i\) (\(1 \leq i \leq N_{p}\)) within layer \(k\) (\(k_{b}(i) \leq k \leq k_{t}(i)\));

2. \texttt{get\_concentration}(\(m, i\)): array \(a(N_{c})\) [dp] is returned with \(C\) for specie \(m\) (\(1 \leq m \leq N_{c}\)) at all prisms above polygon \(i\) (\(1 \leq i \leq N_{p}\)), where \(a(k)\) corresponds to \(C\) within the \(k\)-th layer (\(k_{b}(i) \leq k \leq k_{t}(i)\));

3. \texttt{get\_concentration}(\(m\)): array \(a(I_{3})\) [dp] is returned with \(C\) for the \(m\)-th specie (\(1 \leq m \leq N_{c}\)) at all \(I_{3}\) computational prisms;
4. `get_concentration()`: array `b(I, N_c) [dp]` is returned with `C` for all species `N_c` at all computational prisms `I_i`.

`N_c` is retrieved from `get_nof_species()`, `N_p` from `get_nof_polygons()` and `N_z` from `get_nof_layers()`. `I_i` can be determined using `get_nof_prisms()`. The bottom layer index `k_b(i)` for polygon `i` can be determined using `get_bottom_prism(i)` and `k_t(i)` from `get_top_prism(i)`.

### B.4.29 Interface `get_courant`

**physical unit:** -

Retrieve Courant number 
\[
Cr = \frac{\Delta t (|u| + \sqrt{gh})}{\delta}
\]
at edges:

1. `get_courant(j)`: returns scalar `s [dp]` with Courant number as given above for edge `j` (`1 ≤ j ≤ N_{sf}`);
2. `get_courant()`: returns array `a(N_{sf}) [dp]` with Courant numbers for all edges with flow `N_{sf}`.

`N_k` is given by `get_nof_internal_edges()` and `N_{sf}` by `N_k` + `get_nof_inflow_edges()`. Use `get_max_courant()` for easy access to maximum value.

### B.4.30 Interface `get_density`

**physical unit:** —

Retrieve normalized water density \(\rho/\rho_0\) at prism centers:

1. `get_density(i, k)`: returns scalar `s [dp]` with \(\rho/\rho_0\) at the `i`-th polygon (`1 ≤ i ≤ N_p`) within layer `k` (`k_b(i) ≤ k ≤ k_t(i)`);
2. `get_density(i)`: array `a(N_z) [dp]` is returned with \(\rho/\rho_0\) for all computational prisms above polygon `i` (`1 ≤ i ≤ N_p`), where `a(k)` corresponds to the normalized density within layer `k` (`k_b(i) ≤ k ≤ k_t(i)`);
3. `get_density()`: array `a(I_3) [dp]` is returned with \(\rho/\rho_0\) for all computational prisms `I_i`.

The total number of prisms `I_i` is given by `get_nof_prisms()` whereas `N_p` can be determined by means of `get_nof_polygons()`. The number of vertical layers `N_z` can be retrieved using `get_nof_layers()`, while the bottom layer index `k_b(i)` is given by `get_bottom_prism(i)` and `k_t(i)` can be obtained from `get_top_prism(i)`.
B.4.31 Interface `get_dry_area`  

**physical unit:** m²

Evaluate (total) dry area for polygons:

1. `get_dry_area(i,k)`: returns scalar s [dp] corresponding to the (total) dry area above polygon i (1 ≤ i ≤ \(N_p\)) up to the depth of layer k (\(k_b(i) ≤ k ≤ k_t(i)\));

2. `get_dry_area(i)`: returns scalar s [dp] with the (total) dry area above polygon i (1 ≤ i ≤ \(N_p\));

3. `get_dry_area()`: returns scalar s [dp] with the overall dry area in the computational domain.

\(N_p\) is retrieved using `get_nof_polygons()`, \(k_b(i)\) can be obtained from `get_bottom_prism(i)` and \(k_t(i)\) from `get_top_prism(i)`

If all subpolygons in polygon i are fully dry, the total dry area may still be less than the polygon area obtained from `get_polygon_area(i)` in case the subpolygons do not fully cover the whole polygon.

Use `get_area` to retrieve total polygon area available in computations.

B.4.32 Interface `get_dry_length`  

**physical unit:** m

Evaluate dry length along edges:

1. `get_dry_length(j,k)`: returns scalar s [dp] with dry length along edge j (1 ≤ j ≤ \(N_s\)) up to the depth of layer k (\(k_b(j) ≤ k ≤ k_t(j)\));

2. `get_dry_length(j)`: returns scalar s [dp] with the dry length along edge j (1 ≤ j ≤ \(N_s\));

3. `get_dry_length()`: returns array a(\(N_s\)) [dp] with the dry length for all \(N_s\) edges with flow.

\(N_s\) is retrieved using `get_nof_edges()`, \(k_b(j)\) can be obtained from `get_bottom_face(j)` and \(k_t(j)\) from `get_top_face(j)`.

\(N_s\) is given by `get_nof_internal_edges()` and \(N_s\) by \(N_b + get_nof_inflow_edges()\)

If all subedges are dry, the dry length of edge j may be still less than the (geometric) length of an edge obtained with `get_edge_length(j)` in case the subedges do not fully cover the edge.

Use `get_length` to retrieve total edge length available in computations.
B.4.33 **Interface get\_dt\_min**

**physical unit:** \( s \)

Extract minimum allowed time substep size \( \Delta t \):

1. `get_dt_min()`: return scalar \( s \) [dp] with \( \Delta t \), the minimum allowed substep size for advective scalar transport.

B.4.34 **Interface get\_dx\_min**

**physical unit:** \( m \)

Return minimum allowed distance \( \delta_{\text{min}} \) between adjacent polygon centers:

1. `get_dx_min()`: returns scalar \( s \) [dp] with minimum allowed distance \( \delta_{\text{min}} \) between adjacent polygon centers.

The distance \( \delta \) between polygon centers adjacent to sides is limited by \( \delta_{\text{min}} \) for reasons of numerical stability. \( \delta_{\text{min}} \) must be prescribed by the user in the input data file.

B.4.35 **Interface get\_dz\_min**

**physical unit:** \( m \)

Return minimum allowed layer depth \( H_{\text{min}} \):

1. `get_dz_min()`: returns scalar \( s \) [dp] with minimum allowed layer depth \( H_{\text{min}} \) at computational points.

\( H_{\text{min}} \) is prescribed by the user in the input data file.

B.4.36 **Interface get\_edge\_begin**

**physical unit:** —

Return index for the first vertex of a side (edge):

1. `get_edge_begin(j)`: returns scalar \( s \) [int] with the first vertex index for the \( j \)-th side (\( 1 \leq j \leq N_s \));

2. `get_edge_begin()`: array \( a(N_s) \) [int] is returned with all indices of the first vertex for all \( N_s \) sides, where \( a(j) \) corresponds to the index of the first vertex for the \( j \)-th side.

The total number of sides (edges) \( N_s \) can be determined calling `get_nof_edges()` \[get_nof_vertices()\] returns the total number of vertices \( N_s \) instead.
B.4.37 Interface get_edge_bottom_deviation

physical unit: m/s (positive upwards)

Retrieve vertical bottom velocity \( w_x \) at subedges for all edges with flow:

1. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x (j, jsg) \): scalar \( s \) [dp] is returned with vertical bottom velocity for the \( j^SG \)-th subedge \( (1 \leq j^SG \leq n^SG_x) \) along edge \( j \) \( (1 \leq j \leq N_x) \);

2. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x (j) \): array \( a(n^SG_x) \) [dp] is returned with vertical bottom velocity for all \( n^SG_x \) subedges belonging to edge \( j \) \((1 \leq j \leq N_x)\);

3. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x () \): array \( a(N^SG_x) \) [dp] is returned with vertical bottom velocity for all \( N^SG_x \) subedges.

\( N_x \) is given by \text{get_no_of_internal_edges()} \ and \( N_x \) by \( N_x + \text{get_no_of_inflow_edges()} \)

\( n^SG_x \) is given by \text{get_no_of_subedges(j)} \ and \( N^SG_x \) is returned from \text{get_no_of_subedges()}

For access to vertical bottom velocity on edge level use \text{get_avr_edge_bottom_deviation} instead.

B.4.38 Interface get_edge_center

physical unit: m [origin]

Retrieve edge (side) center coordinates \( x_j, y_j \):

1. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x (j) \): array \( a(2) \) [dp] is returned with the center coordinates for the \( j \)-th edge \( (1 \leq j \leq N_x) \), where \( a(1) \) corresponds to \( x_j \) and \( a(2) \) to \( y_j \);

2. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x () \): array \( a(2, N_x) \) [dp] is returned with the center coordinates for all \( N_x \) sides, wherein \( a(1, j) \) and \( a(2, j) \) are the center coordinates \( x_j, y_j \) for the \( j \)-th edge.

The total number of sides (edges) \( N_x \) can be determined calling \text{get_no_of_edges()} The edge center is located in the middle of an edge.

B.4.39 Interface get_edge_end

physical unit: —

Return index for the second vertex of a side (edge):

1. \( \text{get}_x \text{e}_d \text{g}_x \text{e}_x \text{d}_x (j) \): returns scalar \( s \) [int] with the second vertex index for the \( j \)-th side \( (1 \leq j \leq N_x) \);
2. `get_edge_end()`: array `a(Ns) [int]` is returned with all indices of the second vertex for all `Ns` sides, where `a(j)` corresponds to the index of the second vertex for the `j`-th side.

The total number of sides (edges) `Ns` can be determined calling `get_nof_edges()`.

### B.4.40 Interface `get_edge_length`

**physical unit:** m

Extract edge length \( \lambda \) (geometric length):

1. `get_edge_length(j)`: returns scalar `s [dp]` with the length of the `j`-th side \((1 \leq j \leq Ns)\);

2. `get_edge_length()`: array `a(Ns) [dp]` is returned with \( \lambda \) for all `Ns` sides with flow, where `a(j)` corresponds to the length of the `j`-th side.

The total number of sides (edges) `Ns` can be determined calling `get_nof_edges()`.

The active length of an edge used during computation may be less than or equal to this length. Use `get_length`, `get_dry_length` or `get_wet_length` for further inquiries.

### B.4.41 Interface `get_edge_wet_area`

**physical unit:** m²

Evaluate wet area above edges (area available for normal flow through edge):

1. `get_edge_wet_area(j, k)`: returns scalar `s [dp]` corresponding to the wet face area above edge `j` \((1 \leq j \leq Ns)\) of layer `k` \((k_b(j) \leq k \leq k_t(j))\);

2. `get_edge_wet_area(j)`: returns array `a(Ns) [dp]` with the wet face areas for all `Ns` layers above edge `j` \((1 \leq j \leq Ns)\);

3. `get_edge_wet_area()`: returns array `a(J3) [dp]` with the wet face areas for all `J3` faces.

`Ns` is given by `get_nof_layers()`, `Ns` is retrieved using `get_nof_edges()`, `k_b(j)` can be obtained from `get_bottom_face(j)` and `k_t(j)` from `get_top_face(j)`. `J3` is retrieved from `get_nof_faces()` and `N_k` by `N_k + get_nof_inflow_edges()`.
B.4.42 Interface \texttt{get\_elevation}

\textbf{physical unit: m [reference level]}

Retrieve water surface elevation $\eta$ at polygon centers:

1. \texttt{get\_elevation(i)}: returns scalar $s$ [dp] with water surface elevation $\eta_i$ at the $i$-th polygon ($1 \leq i \leq N_p$);

2. \texttt{get\_elevation()}: array $a(N_p)$ [dp] is returned with $\eta$ for all polygons $N_p$, where $a(i)$ corresponds the water surface elevation at polygon $i$.

$N_p$ can be obtained from \texttt{get\_nof\_polygons()}.

B.4.43 Interface \texttt{get\_elevation\_tolerance}

\textbf{physical unit: —}

Retrieve tolerance $\varepsilon_\eta$ for free-surface inner PCG iterative solver:

1. \texttt{get\_elevation\_tolerance()}: returns scalar $s$ [dp] with actual $\varepsilon_\eta$ used in the program.

$\varepsilon_\eta$ is specified by the user in the input data file "untrim.inp".

B.4.44 Interface \texttt{get\_face\_index}

\textbf{physical unit: —}

Evaluate storage index for a face within a one-dimensional array:

1. \texttt{get\_face\_index(j,k)}: returns scalar $s$ [int] with index of the storage location in a one-dimensional array for face-oriented data belonging to a face above edge $j$ ($1 \leq j \leq N_i$) layer $k (k_p^u(j) \leq k \leq N_z)$.

$N_i$ is obtained from \texttt{get\_nof\_edges()} $k_p^u(j)$ from \texttt{get\_unerodible\_bottom\_face(j)} and $N_z$ is given by \texttt{get\_nof\_layers()}.

B.4.45 Interface \texttt{get\_flux\_limiter}

\textbf{physical unit: text string or array, no unit}

Retrieve name of flux limiter used to compute advection for scalar species:
1. **get_flux_limiter(m)**: returns scalar \(c \times 80\) [char] with the name of the flux limiter used for the \(m\)-th specie \((1 \leq m \leq N_c)\);

2. **get_flux_limiter()**: returns array \(c \times 80\) [char] \((N_c)\) with the names of the flux limiters used for all \(N_c\) species.

\(N_c\) can be retrieved using **get_nof_species()**.

### B.4.46 Interface **get_froude**

**physical unit:** -

Retrieve Froude number \(Fr = \left| \frac{u}{\sqrt{gh}} \right|\) at edges:

1. **get_froude(j)**: returns scalar \(s\) [dp] with Froude number based on depth averaged normal velocity \(\left| \overline{u} \right|\) (modulus) and average depth \(\overline{H}\) for edge \(j\) \((1 \leq j \leq N_{sf})\);

2. **get_froude()**: returns array \(a(N_{sf})\) [dp] with Froude numbers for all edges with flow \(N_{sf}\).

\(N_{sf}\) is given by **get_nof_internal_edges()** and \(N_{sf}\) by \(N_{si} + \text{get_nof_inflow_edges}()\). Use **get_max_froude** for easy access to maximum Froude number.

### B.4.47 Interface **get_gravity**

**physical unit:** \(m/s^2\)

Extract gravitational acceleration \(g\):

1. **get_gravity()**: returns scalar \(s\) [dp] with actual value for \(g\) used in the simulation.

\(g\) is internally computed from the modelling domains (mean) geographic latitude \(\Phi\).

### B.4.48 Interface **get_grid_file**

**physical unit:** text string, no unit

Extract grid file name used:

1. **get_input_file()**: returns \(c \times 80\) [char] with the name of the grid file.

Standard name is "untrim.grd"
B.4.49 Interface get_horizontal_diffusivity

physical unit: \( \text{m}^2/\text{s} \)

Retrieve scalar constant for horizontal molecular diffusivity:

1. `get_horizontal_diffusivity()`: returns scalar \( s \) [dp] with scalar constant horizontal molecular diffusivity.

This value is set by the user in the input file "untrim.inp" and is treated as a constant for all computational points.

*Notice:* The total horizontal diffusivity applied during computation is the sum of this value plus the turbulent horizontal diffusivity set through `CALL set_turbulent_h_diffusivity`.

B.4.50 Interface get_horizontal_velocity

physical unit: \( \text{m}/\text{s} \)

Retrieve interpolated (linear interpolation scheme) velocity components \( u \) (x-direction) and \( v \) (y-direction):

1. `get_horizontal_velocity(x,y)`: returns array \( a(2,N_z) \) [dp] with \( u \) and \( v \) for all layers above position \( x,y \) [re|dp], where \( a(:,k) \) corresponds to the velocity components within the \( k \)-th layer (\( k_b(i) \leq k \leq k_t(i) \));

2. `get_horizontal_velocity(x,y,i,k)`: returns array \( a(2) \) [dp] with \( u \) and \( v \) at position \( x,y \) [re|dp], which is assumed to lie inside polygon \( i \) (\( 1 \leq i \leq N_p \)) within layer \( k \) (\( k_b(i) \leq k \leq k_t(i) \));

3. `get_horizontal_velocity(x,y,i)`: returns array \( a(2,N_z) \) [dp] with \( u \) and \( v \) for all layers above position \( x,y \) [re|dp], which is assumed to lie inside polygon \( i \) (\( 1 \leq i \leq N_p \)), where \( a(:,k) \) corresponds to the velocity components within the \( k \)-th layer (\( k_b(i) \leq k \leq k_t(i) \)).

\( N_p \) can be determined from `get_no_polygons()`, \( k_b(i) \) from `get_bottom_prism(i)` and \( k_t(i) \) from `get_top_prism(i)`. \( N_z \) is given by `get_no_layers()`. Please notice that \( u \) and \( v \) are interpolated in horizontal direction by means of a linear interpolation scheme (routine velo). In \( z \)-direction no interpolation scheme is applied.

B.4.51 Interface get_horizontal_velocity2

physical unit: \( \text{m}/\text{s} \)
Retrieve interpolated velocity (quadratic interpolation scheme) components $u$ ($x$-direction) and $v$ ($y$-direction):

1. `get_horizontal_velocity2(x,y)`: returns array $a(2,N_z)$ [dp] with $u$ and $v$ for all layers above position $x,y$ [re|dp], where $a(:,k)$ corresponds to the velocity components within the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$);

2. `get_horizontal_velocity2(x,y,i,k)`: returns array $a(2)$ [dp] with $u$ and $v$ at position $x,y$ [re|dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$) within layer $k$ ($k_b(i) \leq k \leq k_t(i)$);

3. `get_horizontal_velocity2(x,y,i)`: returns array $a(2,N_z)$ [re|dp] with $u$ and $v$ for all layers above position $x,y$ [dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$), where $a(:,k)$ corresponds to the velocity components within the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$).

$N_p$ can be determined from `get_nof_polygons()` $k_b(i)$ from `get_bottom_prism(i)` and $k_t(i)$ from `get_top_prism(i)`. $N_z$ is given by `get_nof_layers()` Please notice that $u$ and $v$ are interpolated in horizontal direction by means of a quadratic interpolation scheme (routine `vola`). In $z$-direction no interpolation scheme is applied.

### B.4.52 Interface `get_horizontal_velocity_x`

Physical unit: m/s

Retrieve horizontally interpolated (linear interpolation scheme) velocity $u$ ($x$-direction):

1. `get_horizontal_velocity_x(x,y)`: returns array $a(N_z)$ [dp] with $u$ for all layers above position $x,y$ [re|dp], where $a(k)$ corresponds to the $u$-component within the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$);

2. `get_horizontal_velocity_x(x,y,i,k)`: scalar $s$ [dp] is returned with $u$ at position $x,y$ [re|dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$) within layer $k$ ($k_b(i) \leq k \leq k_t(i)$);

3. `get_horizontal_velocity_x(x,y,i)`: returns array $a(N_z)$ [dp] with $u$ for all layers above position $x,y$ [re|dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$), where $a(k)$ corresponds to the $u$-component within the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$).

$N_p$ can be determined from `get_nof_polygons()` $k_b(i)$ from `get_bottom_prism(i)` and $k_t(i)$ from `get_top_prism(i)`. $N_z$ is given by `get_nof_layers()` Please notice that $u$ is interpolated in horizontal direction by means of a linear interpolation scheme. In $z$-direction no interpolation scheme is applied.
**B.4.53 Interface **\texttt{get\_horizontal\_velocity\_y} \\
\textbf{physical unit: m/s} \\
Retrieve horizontally interpolated (linear interpolation scheme) velocity $v$ ($y$-direction): \\
\begin{enumerate}
\item \texttt{get\_horizontal\_velocity\_y(x,y)}: returns array $a(Nz)$ [dp] with $v$ for all layers above position $x,y$ [re dp], where a $(k)$ corresponds to the $v$-component within the $k$-th layer $(k_b(i) \leq k \leq k_t(i))$;
\item \texttt{get\_horizontal\_velocity\_y(x,y,i,k)}: scalar $s$ [dp] is returned with $v$ at position $x,y$ [re dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$) within layer $k$ $(k_b(i) \leq k \leq k_t(i))$;
\item \texttt{get\_horizontal\_velocity\_y(x,y,i)}: returns array $a(Nz)$ [dp] with $v$ for all layers above position $x,y$ [re dp], which is assumed to lie inside polygon $i$ ($1 \leq i \leq N_p$), where a $(k)$ corresponds to the $v$-component within the $k$-th layer $(k_b(i) \leq k \leq k_t(i))$.
\end{enumerate}

$N_p$ can be determined from \texttt{get\_nof\_polygons()}, $k_b(i)$ from \texttt{get\_bottom\_prism(i)} and $k_t(i)$ from \texttt{get\_top\_prism(i)} $N_z$ is given by \texttt{get\_nof\_layers()} Please notice that $v$ is interpolated in horizontal direction by means of a linear interpolation scheme. In $z$-direction no interpolation scheme is applied.

**B.4.54 Interface **\texttt{get\_horizontal\_viscosity} \\
\textbf{physical unit: m$^2$/s} \\
Retrieve scalar constant for horizontal kinematic viscosity: \\
\begin{enumerate}
\item \texttt{get\_horizontal\_viscosity()}: returns scalar $s$ [dp] with scalar constant horizontal kinematic viscosity.
\end{enumerate}

This value is set by the user in the input file "untrim.inp" and is treated as a constant for all computational points. \\
\textit{Notice:} The total horizontal viscosity applied during computation is the sum of this value plus the horizontal turbulent viscosity set through \texttt{CALL set\_turbulent\_h\_viscosity}

**B.4.55 Interface **\texttt{get\_input\_file} \\
\textbf{physical unit: text string, no unit} \\
Extract input file name used: \\
\begin{enumerate}
\item \texttt{get\_input\_file()}: returns c*80 [char] with the name of the input file.
\end{enumerate}

Standard name is "untrim.inp"
B.4.56 **Interface get_layer**

**physical unit:** —

Return layer index \( k \) for a given depth \( z \):

1. **get_layer(\( z \))**: for a given depth \( z \) [\( \text{re, dp} \) a scalar \( s \) [\( \text{int} \) is returned, which represents the actual layer index \( k \) \( (1 \leq k \leq N_z) \) for a given depth \( z \).

The number of level surfaces (layers) \( N_z \) can be obtained from \( \text{get_no\_layers()} \).

B.4.57 **Interface get_layer_interface**

**physical unit:** m [reference level]

Retrieve \( z \)-coordinate of level surface:

1. **get_layer_interface(\( k \))**: returns scalar \( s \) [\( \text{dp} \) with:
   
   (a) \( k = 0 \), maximum (actual) polygon depth;
   
   (b) \( k = N_z \), height, one meter above the (actual) maximum water level at polygons;
   
   (c) \( 1 < k < N_z \), \( z \)-coordinate \( z_k+\frac{1}{2} \) for the \( k \)-th level surface.

2. **get_layer_interface()**: array \( a(N_z-1) \) [\( \text{dp} \) is returned, wherein \( a(k) \) is the \( z \)-coordinate \( z_k+\frac{1}{2} \) for the \( k \)-th level surface.

The number of level surfaces (layers) \( N_z \) can be obtained from \( \text{get_no\_layers()} \).

B.4.58 **Interface get_left_polygon**

**physical unit:** —

Return polygon index \( i \) for the left polygon adjacent to a given side (edge):

1. **get_left_polygon(\( j \))**: returns scalar \( s \) [\( \text{int} \) with index \( i \) for the left neighbouring polygon adjacent to the \( j \)-th side \( (1 \leq j \leq N_s) \);

2. **get_left_polygon()**: array \( a(N_s) \) [\( \text{int} \) with indices for left neighbouring polygons for all \( N_s \) sides is returned, wherein \( a(j) \) is the polygon index for the left neighbour of the \( j \)-th side.

If \( i \leq 0 \) is returned no neighbouring polygon exists to the left, which should never be the case. The total number of sides (edges) \( N_s \) is obtained from \( \text{get_no\_edges()} \).
**B.4.59  Interface get_length**

**physical unit:** m

Extract total length of subedges along edges:

1. `get_length(j)` returns scalar s [dp] with the total length \( \sum_{jSG=1}^{nSG} l_{j,jSG} \) of all subedges belonging to the \( j \)-th side (\( 1 \leq j \leq N_{sj} \));

2. `get_length()` returns array \( a(N_{sf}) \) [dp] is returned with the summed subedge lengths for all edges with flow \( N_{sf} \).

\( N_{sj} \) is given by `get_no_of_internal_edges()` and \( N_{sf} \) by \( N_{sj} + \) `get_no_of_inflow_edges()`.

This length is always less than or equal to the (geometric) length of an edge obtained with `get_edge_length(j)` in case the subedges do not fully cover the edge.

**B.4.60  Interface get_location**

**physical unit:** text string, no unit

Extract location name for the modelling domain:

1. `get_location()` returns \( c*80 \) [char] with the name of the modelling domain.

The name of the modelling domain is specified by the user in the grid input file "untrim.grd".

**B.4.61  Interface get_mass**

**physical unit:** "physical unit of specie used"*m\(^3\)

Determine (local) specie (salt, heat, sediment, etc.) mass present:

1. `get_mass(m,i,k)` returns scalar s [dp] with specie mass present for the \( m \)-th specie (\( 1 \leq m \leq N_{c} \)) at the \( i \)-th polygon (\( 1 \leq i \leq N_{p} \)) within layer \( k \) (\( k_{b}(i) \leq k \leq k_{i}(i) \));

2. `get_mass(m,i)` returns array \( a(N_{z}) \) [dp] with specie mass for the \( m \)-th specie (\( 1 \leq m \leq N_{c} \)) in all \( N_{z} \) layers of the computational column above polygon \( i \) (\( 1 \leq i \leq N_{p} \));

3. `get_mass(m)` returns array \( a(I_{3}) \) [dp] with specie mass for the \( m \)-th specie (\( 1 \leq m \leq N_{c} \)) in all \( I_{3} \) computational prisms;

4. `get_mass()` returns array \( b(I_{3},N_{c}) \) [dp] is returned with specie masses for all \( N_{c} \) species and \( I_{3} \) computational prisms.
$N_c$ can be retrieved using \texttt{get\_nof\_species()}$ and $N_p$ from \texttt{get\_nof\_polygons()} $k_b(i)$ results from \texttt{get\_bottom\_prism(i)} and $k_t(i)$ from \texttt{get\_top\_prism(i)} $N_z$ is obtained from \texttt{get\_nof\_layers()} while $I_3$ is given by \texttt{get\_nof\_prisms()}. For total mass use \texttt{get\_total\_mass} instead.

### B.4.62 Interface \texttt{get\_max\_courant}

**Physical unit:** -

Retrieve maximum Courant number $Cr = \frac{\Delta t (|\bar{u}| + \sqrt{gh})}{\delta}$ at edges:

1. \texttt{get\_max\_courant()} returns scalar \[ \text{dp} \] with maximum Courant number based on depth averaged normal velocity $|\bar{u}|$ (modulus) and average depth $H$ for all active edges with flow.

Use \texttt{get\_courant} for access to individual values at edges.

### B.4.63 Interface \texttt{get\_max\_depth}

**Physical unit:** m

Extract maximum depth $h_i$ within polygons, which is the maximum depth value of all sub-polygon depths $h_{i,SG}$ within a polygon:

1. \texttt{get\_max\_depth(i)}: returns scalar \[ \text{dp} \] with maximum depth at polygon $i (1 \leq i \leq N_p)$;

2. \texttt{get\_max\_depth()} array \[ a(N_p) \text{ [dp]} \] is returned with maximum depth data for all $N_p$ polygons, $a(i)$ is equivalent to the maximum depth at the $i$-th polygon.

The total number of polygons $N_p$ can be determined by means of \texttt{get\_nof\_polygons()}.

For average depth use \texttt{get\_avr\_depth} instead.

### B.4.64 Interface \texttt{get\_max\_edge\_depth}

**Physical unit:** m

Extract maximum depth $h_j$ for all sides/edges with flow, which is the maximum depth value of all subedge depths $h_{j,SG}$ along an edge:

1. \texttt{get\_max\_edge\_depth(j)}: returns scalar \[ \text{dp} \] with maximum depth along edge $j (1 \leq j \leq N_{ij})$;
2. \texttt{get\_max\_edge\_depth()}: array \(a(N_f)\) \([dp]\) is returned with maximum depth data for all \(N_f\) edges, \(a(j)\) is equivalent to the maximum depth along the \(j\)-th edge.

\(N_f\) is given by \texttt{get\_nof\_internal\_edges()} and \(N_s\) by \(N_s+N_f+\texttt{get\_nof\_inflow\_edges()}\). For average depth use \texttt{get\_avr\_edge\_depth} instead.

**B.4.65 Interface \texttt{get\_max\_edge\_total\_depth}**

**physical unit:** m

Extract maximum total water depth (from the free surface to the bottom) for all sides/edges with flow, which is the maximum value evaluated for all respective subedges:

1. \texttt{get\_max\_edge\_total\_depth(j)}: returns scalar \(s\) \([dp]\) with maximum total water depth along edge \(j\) \((1 \leq j \leq N_f)\);

2. \texttt{get\_max\_edge\_total\_depth()}: array \(a(N_f)\) \([dp]\) is returned with maximum total water depth data for all \(N_f\) edges, \(a(j)\) is equivalent to the maximum total water depth along the \(j\)-th edge.

\(N_f\) is given by \texttt{get\_nof\_internal\_edges()} and \(N_s\) by \(N_s+N_f+\texttt{get\_nof\_inflow\_edges()}\). For average total water depth use \texttt{get\_avr\_edge\_total\_depth} instead.

**B.4.66 Interface \texttt{get\_max\_face\_height}**

**physical unit:** m

Extract maximum height \(\Delta z\) for faces above edges:

1. \texttt{get\_max\_face\_height(j,k)}: returns scalar \(s\) \([dp]\) with the actual maximum face height for the \(j\)-th edge \((1 \leq j \leq N_f)\) and \(k\)-th layer \((k_b(j) \leq k \leq k_t(j))\);

2. \texttt{get\_max\_face\_height(j)}: array \(a(N_z)\) \([dp]\) is returned, wherein \(a(k)\) corresponds to the actual maximum face height of the \(k\)-th layer for edge \(j\) \((1 \leq j \leq N_f)\);

3. \texttt{get\_max\_face\_height()}: array \(b(J_3)\) \([dp]\) with the actual maximum heights for all faces \(J_3\) is retrieved.

The total number of sides (edges) \(N_s\) can be determined calling \texttt{get\_nof\_edges()}, whereas the number of level surfaces (layers) \(N_z\) can be obtained from \texttt{get\_nof\_layers()} \(J_3\) can be obtained from \texttt{get\_nof\_faces()} while the bottom layer index \(k_b(j)\) is given by \texttt{get\_bottom\_face(j)} and the respective surface layer index results from \texttt{get\_top\_face(j)} \(N_s\) is given by \texttt{get\_nof\_internal\_edges()} and \(N_f\) by \(N_s+N_f+\texttt{get\_nof\_inflow\_edges()}\). For average face heights use \texttt{get\_avr\_face\_height} instead.
B.4.67 Interface \texttt{get\_max\_froude}

physical unit: 

Retrieve maximum Froude number \( Fr = \frac{\bar{u}}{\sqrt{gH}} \) at edges:

1. \texttt{get\_max\_froude()} returns scalar \([\text{dp}]\) with maximum Froude number based on depth averaged normal velocity \(|\bar{u}|\) (modulus) and average depth \(H\) for all active edges with flow.

Use \texttt{get\_froude} to access individual values at edges.

B.4.68 Interface \texttt{get\_max\_prism\_height}

physical unit: \textit{m}

Extract maximum height (with water) for prisms:

1. \texttt{get\_max\_prism\_height(i,k)} returns scalar \([\text{dp}]\) with maximum height at the \(i\)-th polygon \((1 \leq i \leq N_p)\) within layer \(k\) \((k_b(i) \leq k \leq k_t(i))\);

2. \texttt{get\_max\_prism\_height(i)}: array \(a(\text{Nz})\) \([\text{dp}]\) is returned with maximum heights for all computational prisms above polygon \(i\) \((1 \leq i \leq N_p)\), where \(a(k)\) corresponds to the actual height within layer \(k\) \((k_b(i) \leq k \leq k_t(i))\);

3. \texttt{get\_max\_prism\_height()} returns maximum heights for all computational prisms \(I_3\).

The total number of prisms \(I_3\) is given by \texttt{get\_nof\_prisms()} whereas \(N_p\) can be determined by means of \texttt{get\_nof\_polygons()}. The number of vertical layers \(\text{Nz}\) can be retrieved using \texttt{get\_nof\_layers()} while the bottom layer index \(k_b(i)\) is given by \texttt{get\_bottom\_prism(i)} and \(k_t(i)\) can be obtained from \texttt{get\_top\_prism(i)}.

For average prism heights use \texttt{get\_avr\_prism\_height} instead.

B.4.69 Interface \texttt{get\_max\_total\_depth}

physical unit: \textit{m}

Extract maximum (total) water depth (from the free surface to the bottom) within polygons:

1. \texttt{get\_max\_total\_depth(i)} returns scalar \([\text{dp}]\) with maximum total water depth at polygon \(i\) \((1 \leq i \leq N_p)\);

2. \texttt{get\_max\_total\_depth()} returns maximum total water depth data for all \(N_p\) polygons, \(a(i)\) is equivalent to the maximum total water depth at the \(i\)-th polygon.
The total number of polygons \( N_p \) can be determined by means of \texttt{get_nof_polygons()}. For average total water depth use \texttt{get_avr_total_depth} instead.

**B.4.70 Interface \texttt{get\_max\_velocity\_courant}**

**physical unit: -**

Retrieves maximum value for velocity Courant number \( C_{ru} = \frac{\Delta t |u|}{\delta} \) at faces:

1. \texttt{get\_max\_velocity\_courant(j)}: returns scalar s [dp] with maximum velocity Courant number as given above for all active faces with flow at edge \( j (1 \leq j \leq N_f) \);

2. \texttt{get\_max\_velocity\_courant()}: returns scalar s [dp] with maximum velocity Courant number as given above detected for all active faces with flow. 

\( N_f \) is given by \texttt{get\_nof\_internal\_edges()} and \( N_f \) by \( N_f + \texttt{get\_nof\_inflow\_edges()} \). Use \texttt{get\_velocity\_courant} to access individual data at faces.

**B.4.71 Interface \texttt{get\_nof\_bottom\_fluxes}**

**physical unit: —**

Extracts number of different bottom fluxes \( N_B \):

1. \texttt{get\_nof\_bottom\_fluxes()}: returns scalar s [int] with the number of different bottom fluxes \( N_B \) used with all species \( N_c \).

\( N_c \) can be obtained from \texttt{get\_nof\_species()}.

**B.4.72 Interface \texttt{get\_nof\_boundary\_polygons}**

**physical unit: —**

Retrieves the number of polygons \( N_p^* \) located along the open boundary of the modelling domain with \emph{prescribed water level}:

1. \texttt{get\_nof\_boundary\_polygons()}: returns scalar s [int] with actual number of boundary polygons \( N_p^* \).

This value is prescribed by the user in the grid input file "untrim.grd".
B.4.73 Interface `get_nof_diffusivities`

**physical unit:** —

Retrieve number of different horizontal and vertical diffusivities in use:

1. `get_nof_diffusivities()`: an array `a(2) [int]` with the actual number `a(1)` of different horizontal diffusivities `Nh` as well as the actual number `a(2)` of different vertical diffusivities `Nv` used is returned.

The program’s default value is 1 for each number. During runtime this number may become larger than one due to the use of `set_new_k`, `set_new_kh`, `set_new_kv`, `share_k`, `share_kh`, and `share_kv`.

B.4.74 Interface `get_nof_edges`

**physical unit:** —

Retrieve number of sides (edges):

1. `get_nof_edges(i)`: returns scalar `s [int]` with the actual number of sides `Si` for polygon `i (1 ≤ i ≤ Np)`;
2. `get_nof_edges()`: scalar `s [int]` with actual number of sides `Ns` for all polygons `Np` is returned.

The total number of polygons `Np` can be retrieved calling `get_nof_polygons()`.

B.4.75 Interface `get_nof_faces`

**physical unit:** —

Extract number of computational faces:

1. `get_nof_faces(j)`: returns scalar `s [int]` with number of faces `J_j` above the `j`-th side `(1 ≤ j ≤ Nsf)`;
2. `get_nof_faces()`: a scalar `s [int]` with the total number of faces `J` above all `Nsf` sides is returned.

`Ns` is given by `get_nof_internal_edges()` and `Nsf` by `Ns + get_nof_inflow_edges()`
### B.4.76 Interface `get_nof_fluxes`

**Physical unit:** —

Extract number of different bottom and surface fluxes:

1. `get_nof_fluxes()`: an array `a(2) [int]` with the number of different surface `a(1)` as well as bottom fluxes `a(2)` used with all species `N_e` is returned.

`N_e` can be obtained from `get_nof_species()`.

### B.4.77 Interface `get_nof_h_diffusivities`

**Physical unit:** —

Retrieve number of different horizontal diffusivities `N_h^K` in use:

1. `get_nof_h_diffusivities()`: a scalar `s [int]` with the actual number `N_h^K` of different horizontal diffusivities used is returned.

This program default value is 1. During runtime this number may become larger than one due to the use of `set_new_k` and `set_new_kh`.

### B.4.78 Interface `get_nof_inflow_edges`

**Physical unit:** —

Extract number of inflow edges along the open boundary of the model domain with prescribed flow.

1. `get_nof_inflow_edges()`: returns scalar `s [int]` with number of edges `n_{sf}` with prescribed flow.

This value is (indirectly) set by the user in the grid file "untrim.grd". `N_{si}` is given by `get_nof_internal_edges()` and `N_{sj}` by `N_{si} + get_nof_inflow_edges()`.

### B.4.79 Interface `get_nof_inner_iterations`

**Physical unit:** —

Extract (total) number of inner (PCG) iterations for the free surface as well as the hydrodynamic pressure solver used during the current time step:

1. `get_nof_inner_iterations()`: returns array `a(1|2) [int]` with the total number of (inner) iterations used during the current time step:
• computation using hydrostatic pressure:
  a(1) : total number of inner (PCG) iterations used to solve the free surface equation.
• computation using non-hydrostatic pressure:
  a(1) : total number of inner (PCG) iterations used to solve the free surface equation;
  a(2) : total number of inner (PCG) iterations used to solve the pressure equation.

2. get_nof_inner_iterations(n): returns array a(1|2) [int] with the number of (inner) iterations used during n-th outer (Newton) iteration (1 ≤ n ≤ n_N^N) within the current time step:
   • computation using hydrostatic pressure:
     a(1) : number of inner (PCG) iterations used to solve the free surface equation within the n-th outer (Newton) iteration.
   • computation using non-hydrostatic pressure:
     a(1) : number of inner (PCG) iterations used to solve the free surface equation within the n-th outer (Newton) iteration;
     a(2) : number of inner (PCG) iterations used to solve the pressure equation within the n-th outer (Newton) iteration.

Use with hydrodynamic pressure() to check for the pressure approximation used. n_N^N can be obtained from get_nof_outer_iterations.

B.4.80 Interface get_nof_internal_edges

physical unit: —

Extract number of internal edges (internal edges are sides shared by two polygons):

1. get_nof_internal_edges(): returns scalar s [int] with number of internal edges N_S_i.

This value is set by the user in the grid file "untrim.grd".

B.4.81 Interface get_nof_iterations

physical unit: —

Extract (total) number of inner (PCG) and outer (Newton) iterations for the free surface as well as the hydrodynamic pressure solver used during the current time step:

1. get_nof_iterations(): returns array a(2|4) [int] with the number of iterations used during the current time step:
• computation using hydrostatic pressure:
  a(1) : total number of inner (PCG) iterations used to solve the free surface equation;
  a(2) : number of outer (Newton) iterations used to solve the free surface equation.

• computation using non-hydrostatic pressure:
  a(1) : total number of inner (PCG) iterations used to solve the free surface equation;
  a(2) : total number of inner (PCG) iterations used to solve the pressure equation;
  a(3) : number of outer (Newton) iterations used to solve the free surface equation;
  a(4) : number of outer (Newton) iterations used to solve the pressure equation.

Use `with_hydrodynamic_pressure()` to check for the pressure approximation used.

### B.4.82 Interface `get_nof_layers`

**physical unit: —**

Retrieve the number of level surfaces \( N_z \):

1. `get_nof_layers()`: returns scalar \( s \) [int] with the number of level surfaces \( N_z \).

This value is prescribed by the user in the input file "untrim.inp".

### B.4.83 Interface `get_nof_max_edges`

**physical unit: —**

Extract maximum number of edges (vertices) within all polygons (either 3 or 4):

1. `get_nof_max_edges()`: returns scalar \( s \) [int] with the maximum number of edges (vertices).

For individual access on polygon level use `get_nof_edges(i)` instead.

### B.4.84 Interface `get_nof_max_iterations`

**physical unit: —**

Extract maximum number of inner (PCG) and outer (Newton) iterations allowed during one time step:
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM^2
User Interface Description – Version May 2010 (1.1)

1. `get_nof_maximum_iterations()`: returns array `a(2)[int]` with the maximum number of iterations allowed during one time step:
   - `a(1)`: maximum number of inner (PCG) iterations `m_{PCG}^{max}` allowed to solve the free surface as well as the pressure equation;
   - `a(2)`: maximum number of outer (Newton) iterations `m_{N}^{max}` allowed to solve the free surface as well as the pressure equation.

   `m_{PCG}^{max}` is usually prescribed in the input file "untrim.inp". `m_{N}^{max}` can be set during run time using `set_wet_and_dry(m_{N}^{max})` to a value different from its default 100 used in the program.

B.4.85 Interface `get_nof_outer_iterations`

physical unit: —

Extract number of outer (Newton) iterations for the free surface as well as the hydrodynamic pressure solver used during the current time step:

1. `get_nof_outer_iterations()`: returns array `a(1|2)[int]` with the number of (outer) iterations used during the current time step:
   - computation using hydrostatic pressure:
     - `a(1)`: number of outer (Newton) iterations used to solve the free surface equation.
   - computation using non-hydrostatic pressure:
     - `a(1)`: number of outer (Newton) iterations used to solve the free surface equation;
     - `a(2)`: number of outer (Newton) iterations used to solve the pressure equation.

Use `with_hydrodynamic_pressure()` to check for the pressure approximation used.

B.4.86 Interface `get_nof_point_sources`

physical unit: —

Retrieve the number of point sources `N_d`:

1. `get_nof_point_sources()`: returns scalar `s[int]` with the number of point sources `N_d`.

This value is prescribed by the user in the input file "untrim.srs".
B.4.87 Interface get_nof_polygons

physical unit: —

Retrieve the number of polygons $N_p$:

1. `get_nof_polygons()`: returns scalar $s$ [int] with the number of polygons $N_p$.

This value is prescribed by the user in the grid file "untrim.grd".

B.4.88 Interface get_nof_prisms

physical unit: —

Extract the number of computational prisms:

1. `get_nof_prisms(i)`: returns scalar $s$ [int] with number of prisms $I_i$ above the $i$-th polygon ($1 \leq i \leq N_p$);

2. `get_nof_prisms()`: a scalar $s$ [int] with the total number of computational prisms $I_3$ above all $N_p$ polygons is returned.

The total number of polygons $N_p$ can be retrieved calling `get_nof_polygons()`.

B.4.89 Interface get_nof_settling_velocities

physical unit: —

Retrieve number of different settling velocities $N_{ws}$ in use:

1. `get_nof_settling_velocities()`: a scalar $s$ [int] with the actual number $N_{ws}$ of different settling velocities used is returned.

The programs default value is 1. During runtime this number may become larger than one due to the use of `set_new_settling_velocity` and `share_settling_velocity`.

B.4.90 Interface get_nof_species

physical unit: —

Retrieve number of species $N_c$:

1. `get_nof_species()`: a scalar $s$ [int] with the actual number of species $N_c$ is returned.

This value is prescribed by the user in the input file "untrim.inp".
B.4.91 Interface get_nof_subedges

physical unit: —

Retrieve the number of subedges:

1. get_nof_subedges(j): returns scalar s [int] with the number $n_{SG}^{s,j}$ of subedges for the $j$-th edge ($1 \leq j \leq N_{sf}$);

2. get_nof_subedges(): returns scalar s [int] with the overall number $N_{SG}^s$ of subedges for all edges $N_{sf}$.

These values are prescribed by the user in the grid file "untrim.grd". $N_{si}$ is given by get_nof_internal_edges() and $N_{sf}$ by $N_{si} +$ get_nof_inflow_edges().

B.4.92 Interface get_nof_subpolygons

physical unit: —

Retrieve the number of subpolygons:

1. get_nof_subpolygons(i): returns scalar s [int] with the number $n_{SG}^{p,i}$ of subpolygons for the $i$-th polygon ($1 \leq i \leq N_{p}$);

2. get_nof_subpolygons(): returns scalar s [int] with the overall number $N_{SG}^p$ of subpolygons for all polygons $N_{p}$.

These values are prescribed by the user in the grid file "untrim.grd". The total number of polygons $N_{p}$ can be retrieved calling get_nof_polygons().

B.4.93 Interface get_nof_substeps

physical unit: —

Retrieve number of substeps used in solver for transport equation:

1. get_nof_substeps(m): a scalar s [int] with the actual number of substeps for the $m$-th specie ($1 \leq m \leq N_{c}$) is returned;

2. get_nof_substeps(): an array a($N_{c}$) [int] with the actual numbers of substeps for all species $N_{c}$ is returned.

$N_{c}$ is obtained from get_nof_species()
### B.4.94 Interface `get_noof_surface_fluxes`  

**physical unit:** —  

Extract number of different surface fluxes $N_f$:

1. `get_noof_surface_fluxes()`: returns scalar s [int] with the number of different surface fluxes $N_f$ used with all species $N_c$.

   $N_c$ can be obtained from `get_noof_species()`.

### B.4.95 Interface `get_noof_v_diffusivities`  

**physical unit:** —  

Retrieve number of different vertical diffusivities $N^v_k$ in use:

1. `get_noof_v_diffusivities()`: a scalar s [int] with the actual number $N^v_k$ of different vertical diffusivities used is returned.

   This program default value is 1. During runtime this number may become larger than one due to the use of `set_new_k`, `set_new_kv`, `share_k`, and `share_kv`.

### B.4.96 Interface `get_noof_vertices`  

**physical unit:** —  

Retrieve number of vertices:

1. `get_noof_vertices(i)`: returns scalar s [int] with number of vertices $S_i$ for the $i$-th polygon ($1 \leq i \leq N_p$);

2. `get_noof_vertices()`: returns scalar s [int] with the overall number of vertices $N_v$.

   This value is set by the user in the grid file "untrim.grd". The total number of polygons $N_p$ can be retrieved calling `get_noof_polygons()`.

### B.4.97 Interface `get_noof_wet_faces`  

**physical unit:** —  

Determine the actual number of wet faces:

1. `get_noof_wet_faces(j)`: returns scalar s [int] with number of wet faces for the $j$-th side ($1 \leq j \leq N_s$);
2. `get_no_of_wet_faces()`: returns scalar `s [int]` with the overall number of wet faces.

The total number of sides (edges) `N_s` is obtained from `get_no_of_edges()`.

**B.4.98 Interface `get_no_of_wet_polygons`**

physical unit: —

Determine the actual number of wet polygons:

1. `get_no_of_wet_polygons()`: returns scalar `s [int]` with the overall number of wet polygons.

A polygon `i` is considered wet if `A_{i}^{wet} > 0.0` holds true.

**B.4.99 Interface `get_no_of_wet_prisms`**

physical unit: —

Determine the actual number of wet prisms:

1. `get_no_of_wet_prisms(i)`: returns scalar `s [int]` with number of wet prisms above polygon `i` (1 ≤ `i` ≤ `N_p`);

2. `get_no_of_wet_prisms()`: returns scalar `s [int]` with the overall number of wet prisms.

The total number of polygons `N_p` can be retrieved calling `get_no_of_polygons()`.

**B.4.100 Interface `get_no_of_wet_subedges`**

physical unit: —

Determine the actual number of wet subedges:

1. `get_no_of_wet_subedges(j)`: returns scalar `s [int]` with actual number of wet subedges for the `j`-th edge (1 ≤ `j` ≤ `N_s`);

2. `get_no_of_wet_subedges()`: returns scalar `s [int]` with actual number of wet subedges for all edges with flow `N_s`.

`N_s` is given by `get_no_of_internal_edges()` and `N_f` by `N_s + get_no_of_inflow_edges()`.
B.4.101 Interface `get_nof_wet_subpolygons`

physical unit: —

Determine the actual number of *wet* subpolygons:

1. `get_nof_wet_subpolygons(i)`: returns scalar \( s \) [int] with actual number of wet subpolygons for the \( i \)-th computational polygon \( (1 \leq i \leq N_p) \);

2. `get_nof_wet_subpolygons()`: returns scalar \( s \) [int] with actual number of wet subpolygons in all computational polygons \( N_p \).

To determine \( N_p \) use `get_nof_polygons()`.

B.4.102 Interface `get_polygon`

physical unit: —

Determine polygon index \( i \) from given position \( x,y \):

1. `get_polygon(x,y)`: returns scalar \( s \) [int] with index \( i \) of polygon wherein \( x, y \) [re|dp] is located (if the position is out of the model area, \( i = 0 \) will be returned instead).

B.4.103 Interface `get_polygon_area`

physical unit: \( \text{m}^2 \)

Determine polygon area \( P \) (geometric area):

1. `get_polygon_area(i)`: returns scalar \( s \) [souble] with area \( P_i \) for the \( i \)-th polygon \( (1 \leq i \leq N_p) \);

2. `get_polygon_area()`: array \( a(N_p) \) [dp] with actual areas for all polygons \( N_p \) is returned, where \( a(i) \) is equivalent to \( P_i \), the area of the \( i \)-th polygon.

The total number of polygons \( N_p \) can be retrieved calling `get_nof_polygons()`.

The active area of a polygon used in computation may be less than or equal to this area. Use `get_area` `get_dry_area` or `get_wet_area` for further inquiries.
B.4.104 Interface get_polygon_center

physical unit: m [origin]

Retrieve center coordinates \( x_i, y_i \) for polygons:

1. \( \text{get\_polygon\_center}(i) \): array \( a(2) \) [dp] is returned with the center coordinates for the \( i \)-th polygon (\( 1 \leq i \leq N_p \)), where \( a(1) \) corresponds to \( x_i \) and \( a(2) \) to \( y_i \);

2. \( \text{get\_polygon\_center}() \): array \( b(2,N_p) \) [dp] is returned with the center coordinates for all \( N_p \) polygons, wherein \( b(1,i) \) and \( b(2,i) \) are the center coordinates \( x_i, y_i \) for the \( i \)-th polygon;

The total number of polygons \( N_p \) can be retrieved calling \( \text{get\_nof\_polygons}() \).

B.4.105 Interface get_polygon_edge

physical unit: —

Evaluate indices for sides (edges) of polygons:

1. \( \text{get\_polygon\_edge}(i,l) \): returns scalar \( s \) [int] with index \( j \) for the \( l \)-th (\( 1 \leq l \leq S_i \)) side of polygon \( i \) (\( 1 \leq i \leq N_p \));

2. \( \text{get\_polygon\_edge}(i) \): array \( a(4) \) [int] of side indices \( j(i,:) \) is returned for the \( i \)-th polygon, where \( a(1) \) is the side index \( j \) for the \( l \)-th side of the \( i \)-th polygon (with \( l = 1,S_i \));

3. \( \text{get\_polygon\_edge}() \): array \( b(4,N_p) \) [int] of side indices for all \( N_p \) polygons is returned, where \( b(1,i) \) represents the side index for the \( l \)-th (\( 1 \leq l \leq S_i \)) side of polygon \( i \).

The number of sides \( S_i \) for polygon \( i \) can be determined by means of \( \text{get\_nof\_edges}(i) \) whereas the total number of polygons \( N_p \) can be retrieved from \( \text{get\_nof\_polygons}() \).

B.4.106 Interface get_polygon_vertex

physical unit: —

Evaluate indices for vertices (nodes) of polygons:

1. \( \text{get\_polygon\_vertex}(i,l) \): returns scalar \( s \) [int] with index \( n \) for the \( l \)-th (\( 1 \leq l \leq S_i \)) node of polygon \( i \) (\( 1 \leq i \leq N_p \));

2. \( \text{get\_polygon\_vertex}(i) \): array \( a(4) \) [int] of vertex indices is returned for the \( i \)-th polygon, where \( a(1) \) is the vertex index \( n \) for the \( l \)-th node of the \( i \)-th polygon;
3. `get_polygon_vertex()`: array `a(4,Np) [int]` of vertex indices is returned for all polygon, where `a(1,i)` is the vertex index `n` for the `l`-th node of the `i`-th polygon. The total number of polygons `Np` can be retrieved from `get_noof_polygons()` whereas the actual number of edges `Si` for polygon `i` is obtained from `get_noof_edges(i)`.

### B.4.107 Interface `get_pressure`

**physical unit:** m²/s²

Retrieve normalized non-hydrostatic pressure component `q` at prism centers:

1. `get_pressure(i,k)`: returns scalar `s [dp]` with `q` at polygon `i` (`1 ≤ i ≤ Np`) within layer `k` (`kb(i) ≤ k ≤ kt(i)`);
2. `get_pressure(i)`: array `a(Nz) [dp]` is returned with `q` at all prisms above polygon `i` (`1 ≤ i ≤ Np`), where `a(k)` corresponds to `q` within the `k`-th layer (`kb(i) ≤ k ≤ kt(i)`);
3. `get_pressure()`: array `a(I3) [dp]` is returned with `q` at all `I3` computational prisms. `Np` can be retrieved from `get_noof_polygons()` and `Nz` from `get_noof_layers()` `I3` can be determined using `get_noof_prisms()`. For the `i`-th polygon the bottom and top layer indices `kb(i)` and `kt(i)` can be obtained from `get_bottom_prism(i)` and `get_top_prism(i)` respectively. Multiplication of `q` with `ρ₀` gives the (real physical) non-hydrostatic pressure component.

### B.4.108 Interface `get_pressure_tolerance`

**physical unit:** —

Retrieve tolerance `ε_q` for non-hydrostatic pressure (inner) iterative PCG solver:

1. `get_pressure_tolerance()`: returns scalar `s [dp]` with actual `ε_q` used in the program.

This value is prescribed by the user in the input file "untrim.inp".

### B.4.109 Interface `get_printout_unit`

**physical unit:** —

Retrieve logical unit number for output of informative messages to file UnTRIM.txt.

1. `get_printout_unit()`: returns scalar `s [int]` with actual logical unit number `no` used in the program.
B.4.110 Interface `get_prism_index`

**physical unit:** —

Evaluate storage index for a prism within a one-dimensional array:

1. `get_prism_index(i,k)`: returns scalar \( s \) [int] with index of the storage location in a one-dimensional array for prism-oriented data belonging to a prism above polygon \( i \) \((1 \leq i \leq N_p)\) layer \( k \) \((k_u(i) \leq k \leq N_z)\).

\( N_p \) is obtained from `get_nof_polygons()` \( k_u(i) \) from `get_unerodible_bottom_prism(i)` and \( N_z \) is given by `get_nof_layers()`.

B.4.111 Interface `get_radiation_time`

**physical unit:** 1/s

Retrieve radiation time \( t_r \):

1. `get_radiation_time(i)`: returns scalar \( s \) [dp] with actual \( t_r \) used at the \( i \)-th polygon \((1 \leq i \leq N^*_p)\) in the program;

2. `get_radiation_time()`: return array \( a(N^*_p) \) [dp] with actual \( t_r \) values used in all open boundary polygons \( N^*_p \).

\( N^*_p \) is given by `get_nof_boundary_polygons()` \( t_r \) is either taken from the user’s input file "untrim.inp" or set by means of `set_radiation_time()` during run time.

B.4.112 Interface `get_right_polygon`

**physical unit:** —

Return polygon index \( i \) for the right polygon adjacent to a given side (edge):

1. `get_right_polygon(j)`: returns scalar \( s \) [int] with index \( i \) for the right neighbouring polygon adjacent to the \( j \)-th side \((1 \leq j \leq N_s)\);

2. `get_right_polygon()`: array \( a(N_s) \) [int] with indices for right neighbouring polygons for all \( N_s \) sides is returned, wherein \( a(j) \) is the polygon index of the right neighbour of the \( j \)-th side.

If \( i \leq 0 \) is returned no neighbouring polygon exists to the right, which may be the case for e.g. sides along boundaries. The total number of sides (edges) \( N_s \) is obtained from `get_nof_edges()`.
B.4.113 Interface get_sediment

physical unit: —

Retrieve settling velocity factor $f_s$ for species:

1. `get_sediment(m)`: returns scalar $s$ [dp] with $f_s$ for the $m$-th specie ($1 \leq m \leq N_c$);
2. `get_sediment()`: returns array $a(N_c)$ [dp] with settling velocity factors for all species $N_c$.

$N_c$ is obtained from `get_nof_species()`.

B.4.114 Interface get_settling_velocity

physical unit: m/s

Retrieve settling velocity $w_s$ either at top of prisms or interpolated in $z$-direction, for species which are labelled as sediment:

1. `get_settling_velocity(m,i,k)`: returns scalar $s$ [dp] with $w_s$ for the $m$-th specie ($1 \leq m \leq N_c$) at polygon $i$ ($1 \leq i \leq N_p$) at top of layer $k$ ($k_b(i) \leq k \leq k_t(i)$);
2. `get_settling_velocity(m,i,z)`: returns scalar $s$ [dp] with vertically interpolated $w_s$ at position $z$ [dp] above polygon $i$ ($1 \leq i \leq N_p$) for the $m$-th specie ($1 \leq m \leq N_c$);
3. `get_settling_velocity(m,i)`: array $a(0:N_z)$ [dp] is returned for the $m$-th specie ($1 \leq m \leq N_c$) with $w_s$ for the computational column above polygon $i$ ($1 \leq i \leq N_p$), where $a(k)$ is the settling velocity for the top of the $k$-th layer ($k_b(i) \leq k \leq k_t(i)$);
4. `get_settling_velocity(m)`: array $a(I_3)$ [dp] is returned for the $m$-th specie ($1 \leq m \leq N_c$) with $w_s$ for all computational prisms;
5. `get_settling_velocity()`: array $b(I_3,N_{ws})$ [dp] is returned with $w_s$ for all species $N_{ws}$ at all computational prisms $I_3$ and all different settling velocities $N_{ws}$.

$I_3$ can be retrieved using `get_nof_prisms()` whereas $N_c$ is given by `get_nof_layers()`.

The bottom and top layer indices $k_b(i)$ and $k_t(i)$ can be determined for the $i$-th polygon by means of `get_bottom_prism(i)` and `get_top_prism(i)`. $N_{ws}$ is given by `get_nof_settling_velocities`.

Please notice that $w_s > 0.0$ corresponds to downward settling of particles. If a $z$-coordinate is prescribed in the function call, $w_s$ is computed by means of a linear interpolation in vertical direction. Horizontally no interpolation scheme is applied. The total settling velocity, as applied during computation, may differ for actively settling species by a constant factor from the values returned, if factors $\neq 1$ were used in `CALL set_sediment`. 
B.4.115 Interface **get_slope**

**physical unit:** —

Retrieve actual slope of the system in x- and/or y-direction:

1. `get_slope(n)`: returns scalar \( s \) with actual slope in
   
   \( n = 1 \), x-direction, or
   \( n = 2 \), y-direction.

2. `get_slope()`: returns array \( a(2) \) with actual slope in x-direction \( a(1) \) as well as in y-direction \( a(2) \).

B.4.116 Interface **get_source_concentration**

**physical unit:** identical to physical unit of specie used

Retrieve source concentration \( C \):

1. `get_source_concentration(n,m)`: returns scalar \( s \) with actual \( C \) for the \( n \)-th source \((1 \leq n \leq N_d)\) and the \( m \)-th specie \((1 \leq m \leq N_c)\);

2. `get_source_concentration(m)`: array \( a(N_d) \) is returned with actual \( C \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at all \( N_d \) sources, where \( a(n) \) is the respective specie concentration at the \( n \)-th source;

3. `get_source_concentration()`: array \( b(N_d,N_c) \) is returned with actual \( C \) for all \( N_c \) species and all \( N_d \) sources, where \( b(n,m) \) is the concentration for the \( m \)-th specie at the \( n \)-th source.

\( N_d \) can be obtained from `get_no_point_sources()` and \( N_c \) from `get_no_species()`.

B.4.117 Interface **get_source_discharge**

**physical unit:** \( m^3/s \)

Extract discharge for sources and sinks:

1. `get_source_discharge(n)`: returns scalar \( s \) with actual discharge of the \( n \)-th source \((1 \leq n \leq N_d)\);

2. `get_source_discharge()`: array \( a(N_d) \) with actual discharge for all \( N_d \) sources is returned, where \( a(n) \) is the discharge of the \( n \)-th source.

\( N_d \) can be obtained from `get_no_point_sources()`.
B.4.118 Interface get_source_file

physical unit: text string, no unit

Extract source file name used:

1. `get_source_file()`: returns `c*80 [char]` with the name of the source file.

Standard name is "untrim.srs"

B.4.119 Interface get_source_layer

physical unit: —

Retrieve layer index \( k (1 \leq k \leq N_z) \) of sources/sinks:

1. `get_source_layer(n)`: returns scalar \( s \) [int] with layer index \( k \) within which the \( n \)-th source \( (1 \leq n \leq N_d) \) is located in;

2. `get_source_layer()`: array \( a(N_d) \) [int] with layer indices for all \( N_d \) sources is returned, where \( a(n) \) is the layer index of the \( n \)-th source.

\( N_d \) can be obtained from `get_nof_point_sources()`.

B.4.120 Interface get_source_polygon

physical unit: —

Retrieve polygon index \( i (1 \leq i \leq N_p) \) of sources/sinks:

1. `get_source_polygon(n)`: returns scalar \( s \) [int] with polygon index \( i \) within which the \( n \)-th source \( (1 \leq n \leq N_d) \) is located in;

2. `get_source_polygon()`: array \( a(N_d) \) [int] with polygon indices for all \( N_d \) sources is returned, where \( a(n) \) is the polygon index of the \( n \)-th source.

\( N_d \) can be obtained from `get_nof_point_sources()`.

B.4.121 Interface get_subedge_depth

physical unit: m

Extract depth for subedges along flow edges:

1. `get_subedge_depth(j, jsg)`: scalar \( s \) [dp] is returned with depth for the \( j^{th} \) subedge \( (1 \leq j \leq N_s) \) along edge \( j (1 \leq j \leq N_f) \);
2. `get_subedge_depth(j)`: array \((n^{SG}_{s,j})\) [dp] is returned with depth for all \(n^{SG}_{s,j}\) subedges belonging to edge \(j (1 \leq j \leq N_s)\);

3. `get_subedge_depth()`: array \((N^S_{s})\) [dp] is returned with depth for all \(N^S_{s}\) subedges.

\(N_s\) is given by `get_nof_internal_edges()` and \(N_{sf}\) by \(N_s + \text{get_nof_inflow_edges()}\)
\(n^{SG}_{s,j}\) is given by `get_nof_subedges(j)` and \(N^S_{s}\) is returned from `get_nof_subedges()`.

For total water depth at subedges use `get_subedge_total_depth`.

### B.4.122 Interface `get_subedge_length`

**physical unit:** m

Extract length for subedges along flow edges:

1. `get_subedge_length(j,jsg)`: scalar \(s\) [dp] is returned with length for the \(j^S_{g}\)-th subedge \((1 \leq j^S_{g} \leq n^{SG}_{s,j})\) along edge \(j (1 \leq j \leq N_s)\);

2. `get_subedge_length(j)`: array \((n^S_{s,j})\) [dp] is returned with length for all \(n^S_{s,j}\) subedges belonging to edge \(j (1 \leq j \leq N_s)\);

3. `get_subedge_length()`: array \((N^S_{s})\) [dp] is returned with length for all \(N^S_{s}\) subedges.

\(N_s\) is given by `get_nof_internal_edges()` and \(N_{sf}\) by \(N_s + \text{get_nof_inflow_edges()}\)
\(n^{SG}_{s,j}\) is given by `get_nof_subedges(j)` and \(N^S_{s}\) is returned from `get_nof_subedges()`.

Use `get_length` to retrieve the actual total length, or `get_edge_length` for the (geometric) length of an edge.

### B.4.123 Interface `get_subedge_total_depth`

**physical unit:** m

Extract total water depth for subedges along flow edges:

1. `get_subedge_total_depth(j,jsg)`: scalar \(s\) [dp] is returned with total water depth for the \(j^S_{g}\)-th subedge \((1 \leq j^S_{g} \leq n^{SG}_{s,j})\) along edge \(j (1 \leq j \leq N_s)\);

2. `get_subedge_total_depth(j)`: array \((n^S_{s,j})\) [dp] is returned with total water depth for all \(n^S_{s,j}\) subedges belonging to edge \(j (1 \leq j \leq N_s)\);

3. `get_subedge_total_depth()`: array \((N^S_{s})\) [dp] is returned with total water depth for all \(N^S_{s}\) subedges.
$N_s$ is given by $\text{get\_nof\_internal\_edges()}$ and $N_f$ by $\text{get\_nof\_inflow\_edges()}$.

$n_{SG}^f$ is given by $\text{get\_nof\_subedges()}$ and $N_{SG}^f$ is returned from $\text{get\_nof\_subedges()}$.

For depth at subedges use $\text{get\_subedge\_depth}$. 

### B.4.124 Interface get\_subpolygon\_area

**physical unit:** $m^2$

Extract area for subpolygons:

1. get\_subpolygon\_area($i$, $isg$): scalar $s$ [dp] is returned with area for the $i_{SG}$-th subpolygon ($1 \leq i_{SG} \leq n_{SG}^p$) within polygon $i$ ($1 \leq i \leq N_p$);

2. get\_subpolygon\_area($i$): array $a(n_{SG}^p)$ [dp] is returned with area for all $n_{SG}^p$ subpolygons belonging to polygon $i$ ($1 \leq i \leq N_p$);

3. get\_subpolygon\_area(): array $a(N_{SG}^p)$ [dp] is returned with area for all $N_{SG}^p$ subpolygons.

$N_p$ is obtained by means of $\text{get\_nof\_polygons()}$, $n_{SG}^p$ is given by $\text{get\_nof\_subpolygons()}$, and $N_{SG}^p$ is returned from $\text{get\_nof\_subpolygons()}$.

For (geometric) polygon area use $\text{get\_polygon\_area}$ instead.

### B.4.125 Interface get\_subpolygon\_depth

**physical unit:** $m$

Extract depth for subpolygons:

1. get\_subpolygon\_depth($i$, $isg$): scalar $s$ [dp] is returned with depth for the $i_{SG}$-th subpolygon ($1 \leq i_{SG} \leq n_{SG}^p$) within polygon $i$ ($1 \leq i \leq N_p$);

2. get\_subpolygon\_depth($i$): array $a(n_{SG}^p)$ [dp] is returned with depth for all $n_{SG}^p$ subpolygons belonging to polygon $i$ ($1 \leq i \leq N_p$);

3. get\_subpolygon\_depth(): array $a(N_{SG}^p)$ [dp] is returned with depth for all $N_{SG}^p$ subpolygons.

$N_p$ is obtained by means of $\text{get\_nof\_polygons()}$, $n_{SG}^p$ is given by $\text{get\_nof\_subpolygons()}$, and $N_{SG}^p$ is returned from $\text{get\_nof\_subpolygons()}$.

For retrieval of polygon related depth data use $\text{get\_subpolygon\_total\_depth}$ instead.

Use $\text{get\_max\_depth}$ or $\text{get\_avr\_depth}$.
B.4.126 Interface get_subpolygon_total_depth

physical unit: m

Extract total water depth (free surface to bottom) for subpolygons:

1. get_subpolygon_total_depth(i, isg): scalar \( s \) [dp] is returned with total water depth for the \( i^{SG} \)-th subpolygon (\( 1 \leq i^{SG} \leq n_{p}^{SG} \)) within polygon \( i \) (\( 1 \leq i \leq N_p \));

2. get_subpolygon_total_depth(i): array \( a(n_{p}^{SG}) \) [dp] is returned with total water depth for all \( n_{p}^{SG} \) subpolygons belonging to polygon \( i \) (\( 1 \leq i \leq N_p \));

3. get_subpolygon_total_depth(): array \( a(N_{p}^{SG}) \) [dp] is returned with total water depth for all \( N_{p}^{SG} \) polygons.

\( N_p \) is obtained by means of get_nof_polygons(), \( n_{p}^{SG} \) is given by get_nof_subpolygons(i), and \( N_{p}^{SG} \) is returned from get_nof_subpolygons().

For depths use get_subpolygon_depth instead.

Use get_max_total_depth or get_avr_total_depth for retrieval of polygon related total water depth data.

B.4.127 Interface get_surface_alpha

physical unit: (physical unit of specie used) \( \times \) m/s

Retrieve surface flux parameter \( \alpha_r \) at polygon centers:

1. get_surface_alpha(m, i): returns scalar \( s \) [dp] with \( \alpha_r \) for the \( m \)-th specie (\( 1 \leq m \leq N_c \)) at the \( i \)-th polygon (\( 1 \leq i \leq N_p \));

2. get_surface_alpha(m): array \( a(N_p) \) [dp] is returned with \( \alpha_r \) for the \( m \)-th specie (\( 1 \leq m \leq N_c \)) at all \( N_p \) polygons, where \( a(i) \) contains \( \alpha_r \) for the \( i \)-th polygon;

3. get_surface_alpha(): array \( b(N_p, N_r) \) [dp] is returned, where \( b(i, m_r) \) contains \( \alpha_r \) for the \( m_r \)-th surface flux component (\( 1 \leq m_r \leq N_r \)) at polygon \( i \).

\( N_c \) can be retrieved using get_nof_species(), \( N_p \) is obtained from get_nof_polygons(), \( N_r \) from get_nof_surface_fluxes(), and \( m_r \) from get_which_surface_flux(m).
1. \texttt{get\_surface\_beta}(m, i): returns scalar \( s \) [dp] with \( \beta_r \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at the \( i \)-th polygon \((1 \leq i \leq N_p)\);

2. \texttt{get\_surface\_beta}(m): array \( a(N_p) \) [dp] is returned with \( \beta_r \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at all \( N_p \) polygons, where \( a(i) \) contains \( \beta_r \) for the \( i \)-th polygon;

3. \texttt{get\_surface\_beta}(): array \( b(N_p, N_t) \) [dp] is returned, where \( b(i, m_r) \) contains \( \beta_r \) for the \( m_r \)-th surface flux component \((1 \leq m_r \leq N_t)\) at polygon \( i \).

\( N_c \) can be retrieved using \texttt{get\_nof\_species()} \( N_p \) is obtained from \texttt{get\_nof\_polygons()} \( N_t \) from \texttt{get\_nof\_surface\_fluxes(m)} and \( m_r \) from \texttt{get\_which\_surface\_flux(m)}

**B.4.129 Interface get\_surface\_concentration**

**physical unit:** identical to physical unit of specie used

Retrieve prescribed surface concentration \( C_T \) at polygon centers:

1. \texttt{get\_surface\_concentration}(m, i): returns scalar \( s \) [dp] with \( C_T \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at the \( i \)-th polygon \((1 \leq i \leq N_p)\);

2. \texttt{get\_surface\_concentration}(m): array \( a(N_p) \) [dp] is returned with \( C_T \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at all \( N_p \) polygons, where \( a(i) \) contains \( C_T \) for the \( i \)-th polygon;

3. \texttt{get\_surface\_concentration}(): array \( b(N_p, N_t) \) [dp] is returned, where \( b(i, m_r) \) contains \( C_T \) for the \( m_r \)-th surface flux component \((1 \leq m_r \leq N_t)\) at polygon \( i \).

\( N_c \) can be retrieved using \texttt{get\_nof\_species()} \( N_p \) is obtained from \texttt{get\_nof\_polygons()} \( N_t \) from \texttt{get\_nof\_surface\_fluxes()} and \( m_r \) from \texttt{get\_which\_surface\_flux(m)}

**B.4.130 Interface get\_surface\_flux**

**physical unit:** ”physical unit of specie used”*m^3

Retrieve computed free surface flux \( q_T \) at polygon centers:

1. \texttt{get\_surface\_flux}(m, i): returns scalar \( s \) [dp] with \( q_T \) for the \( m \)-th specie \((1 \leq m \leq N_c)\) at the \( i \)-th polygon \((1 \leq i \leq N_p)\);

2. \texttt{get\_surface\_flux}(m): scalar \( s \) [dp] is returned with integral free surface flux \( q_T \) for the \( m \)-th specie \((1 \leq m \leq N_c)\), summed for all \( N_p \) polygons;

3. \texttt{get\_surface\_flux}(): array \( a(N_c) \) [dp] is returned, where \( a(m) \) is the integral free surface flux \( q_T \) for specie \( m \), summed for all \( N_p \) polygons.

\( N_c \) can be retrieved using \texttt{get\_nof\_species()} \( N_p \) is given by \texttt{get\_nof\_polygons()}

---

B. User interface
### B.4.131 Interface `get_theta`

**physical unit:** —

Retrieve implicitness factor $\theta$:

1. `get_theta()`: returns scalar $s$ [dp] with actual $\theta$ used in the program.

$\theta$ is prescribed by the user in the input data file "untrim.inp".

### B.4.132 Interface `get_time`

**physical unit:** s

Extract time $t$ (in seconds):

1. `get_time()`: returns scalar $s$ [dp] with actual time $t$ used in the program.

### B.4.133 Interface `get_time_step`

**physical unit:** s

Extract time step $\Delta t$ (in seconds):

1. `get_time_step()`: returns scalar $s$ [dp] with actual $\Delta t$ used in the program.

$\Delta t$ is prescribed by the user in the input data file "untrim.inp".

### B.4.134 Interface `get_top_face`

**physical unit:** —

Evaluate layer index $k$ ($1 \leq k \leq N_z$) for the topmost (surface) face above the side of a polygon:

1. `get_top_face(j)`: returns scalar $s$ [int] with surface face layer index $k$ for side $j$ ($1 \leq j \leq N_{sf}$);

2. `get_top_face()`: array $a(N_{sf})$ [int] is returned with surface face layer indices for all $N_{sf}$ sides with flow, where $a(j)$ represents the surface layer index for the $j$-th side of the grid.

$N_{si}$ is given by `get_nof_internal_edges()` and $N_{sf}$ by $N_{si}+\text{get_nof_inflow_edges()}$.
B.4.135 Interface get_top_prism

physical unit: —

Evaluate layer index $k$ ($1 \leq k \leq N_z$) for the topmost (surface) prism above a polygon:

1. `get_top_prism(i)`: returns scalar $s$ [int] with surface prism layer index $k$ for polygon $i$ ($1 \leq i \leq N_p$);

2. `get_top_prism()`: array $a(N_p)$ [int] is returned with surface prism layer indices for all $N_p$ polygons, where $a(i)$ represents the surface layer index for the $i$-th polygon.

The total number of polygons $N_p$ can be determined by means of `get_nof_polygons()`.

B.4.136 Interface get_total_mass

physical unit: "physical unit of specie used" · m$^3$

Determine (total) specie (salt, heat, sediment, etc.) mass present:

1. `get_total_mass(m,i,k)`: returns scalar $s$ [dp] with (total) specie mass present for the $m$-th specie ($1 \leq m \leq N_c$) at the $i$-polygon ($1 \leq i \leq N_p$) within layer $k$ ($k_b(i) \leq k \leq k_t(i)$);

2. `get_total_mass(m,i)`: returns scalar $s$ [dp] with accumulated (total) specie mass for the $m$-th specie ($1 \leq m \leq N_c$) available in the computational column above polygon $i$ ($1 \leq i \leq N_p$);

3. `get_total_mass(m)`: returns scalar $s$ [dp] with accumulated (total) specie mass for the $m$-th specie ($1 \leq m \leq N_c$) within the overall computational domain;

4. `get_total_mass()`: array $a(N_c)$ [dp] is returned with accumulated (total) specie masses for all $N_c$ species within the overall computational domain, where $a(m)$ represents the accumulated mass for the $m$-th specie.

$N_c$ can be retrieved using `get_nof_species()` and $N_p$ from `get_nof_polygons()`. $k_b(i)$ results from `get_bottom_prism(i)` and $k_t(i)$ from `get_top_prism(i)`.

Local, prism oriented mass can be retrieved with `get_mass`.

B.4.137 Interface get_total_volume

physical unit: m$^3$

Computes the (total) volume for computational prisms:

1. `get_total_volume(i,k)`: returns scalar $s$ [dp] corresponding to the fluid volume inside the computational prism above polygon $i$ ($1 \leq i \leq N_p$) within layer $k$ ($k_b(i) \leq k \leq k_t(i)$);
2. get_total_volume(i): returns scalar s [dp] with the total fluid volume summed for all prisms above polygon i (1 ≤ i ≤ Np);

3. get_total_volume(): returns scalar s [dp] with the total fluid volume summed for all prisms within the overall computational domain.

Np is obtained from get_no_polygons.
For inquiries about prism related volumes use get_volume instead.

B.4.138  Interface get_turbulent_h_diffusivity

physical unit: m²/s

Retrieve horizontal turbulent diffusivity $K^h$ at faces:

1. get_turbulent_h_diffusivity(m, j, k): returns scalar s [dp] with $K^h$ for the m-th specie (1 ≤ m ≤ Nz) at the j-th side (1 ≤ j ≤ Ns) within layer k ($k_b(j) ≤ k ≤ k_t(j)$);

2. get_turbulent_h_diffusivity(m, j): array a(Ns) [dp] is returned for the m-th specie (1 ≤ m ≤ Nz) with $K^h$ for all computational faces above side j (1 ≤ j ≤ Ns), where a(k) is the corresponding value for the k-th layer ($k_b(j) ≤ k ≤ k_t(j)$);

3. get_turbulent_h_diffusivity(m): array a(Js) [dp] is returned for the m-th specie (1 ≤ m ≤ Nz) with $K^h$ for all computational faces;

4. get_turbulent_h_diffusivity(): array a(Js, Ns) [dp] is returned with $K^h$ for all different horizontal diffusivities $N_s^h$ at all computational faces $J_s$.

Js is retrieved from get_no_faces(); Nz is given by get_no_internal_edges(); Ns from get_no_layers(); Nz from get_no_species() and $N_s^h$ from get_no_h_diffusivities(). Bottom face layer index $k_b(j)$ is given by get_bottom_face(j) and $k_t(j)$ can be retrieved from get_top_face(j).

B.4.139  Interface get_turbulent_h_viscosity

physical unit: m²/s

Retrieve horizontal turbulent viscosity $v^h$ at faces:

1. get_turbulent_h_viscosity(j, k): returns scalar s [dp] with $v^h$ at the j-th side (1 ≤ j ≤ Nz) within layer k ($k_b(j) ≤ k ≤ k_t(j)$);

2. get_turbulent_h_viscosity(j): array a(Ns) [dp] is returned with $v^h$ for all computational faces above side j (1 ≤ j ≤ Nz), where a(k) is the corresponding value for the k-th layer ($k_b(j) ≤ k ≤ k_t(j)$);
3. \texttt{get\_turbulent\_h\_viscosity()}: array \(a \{J_i\} [dp]\) is returned with \(v^h\) for all computational faces.  

\(J_i\) is retrieved from \texttt{get\_nof\_faces()} \(N_c\) is given by \texttt{get\_nof\_internal\_edges()} and \(N_z\) from \texttt{get\_nof\_layers()} Bottom face layer index \(k_b(j)\) is given by \texttt{get\_bottom\_face\_j()} and \(k_t(j)\) can be retrieved from \texttt{get\_top\_face\_j()}.

### B.4.140 Interface \texttt{get\_turbulent\_v\_diffusivity}

Physical unit: \(m^2/s\)

Retrieve vertical turbulent diffusivity \(K_v\) at top of prisms:

1. \texttt{get\_turbulent\_v\_diffusivity(m, i, k)}: returns scalar \(s [dp]\) with \(K_v\) for the \(m\)-th specie \((1 \leq m \leq N_c)\) at the \(i\)-th polygon \((1 \leq i \leq N_p)\) for top of layer \(k (k_b(i) \leq k \leq k_t(i))\);

2. \texttt{get\_turbulent\_v\_diffusivity(m, i)}: array \(a \{N_z\} [dp]\) is returned for the \(m\)-th specie \((1 \leq m \leq N_c)\) with \(K_v\) for all computational prisms above polygon \(i (1 \leq i \leq N_p)\), where \(a \{k\}\) is the corresponding value for the top of the \(k\)-th layer \((k_b(i) \leq k \leq k_t(i))\);

3. \texttt{get\_turbulent\_v\_diffusivity(m)}: array \(a \{I_z\} [dp]\) is returned for the \(m\)-th specie \((1 \leq m \leq N_c)\) with \(K_v\) for all computational prisms;

4. \texttt{get\_turbulent\_v\_diffusivity()} : array \(a \{I_z, N_{v_k}\} [dp]\) is returned with \(K_v\) for all computational prisms and all different vertical turbulent diffusivities \(N_{v_k}\).

\(I_z\) is retrieved from \texttt{get\_nof\_prisms()} \(N_p\) can be obtained from \texttt{get\_nof\_polynomials()} \(N_z\) from \texttt{get\_nof\_layers()} \(N_c\) from \texttt{get\_nof\_species()} and \(N_{v_k}\) using \texttt{get\_nof\_v\_diffusivities()} The bottom prism layer index \(k_b(i)\) is given by \texttt{get\_bottom\_prism\(I_z\)} and \(k_t(i)\) results from \texttt{get\_top\_prism\(I_z\)}.

### B.4.141 Interface \texttt{get\_turbulent\_v\_viscosity}

Physical unit: \(m^2/s\)

Retrieve vertical turbulent viscosity \(v^v\) at top of prisms:

1. \texttt{get\_turbulent\_v\_viscosity(i, k)}: returns scalar \(s [dp]\) with \(v^v\) at the \(i\)-th polygon \((1 \leq i \leq N_p)\) for top of layer \(k (k_b(i) \leq k \leq k_t(i))\);

2. \texttt{get\_turbulent\_v\_viscosity(i)}: array \(a \{N_z\} [dp]\) is returned with \(v^v\) for all computational prisms above polygon \(i (1 \leq i \leq N_p)\), where \(a \{k\}\) is the corresponding value for the top of the \(k\)-th layer \((k_b(i) \leq k \leq k_t(i))\);

3. \texttt{get\_turbulent\_v\_viscosity()} : array \(a \{I_z\} [dp]\) is returned with \(v^v\) for all computational prisms.
$I_i$ is retrieved from `get_nof_prisms()` $N_p$ can be obtained from `get_nof_polygons()` and $N_z$ from `get_nof_layers()` The bottom prism layer index $k_b(i)$ is given by `get_bottom_prism(i)` and $k_t(i)$ results from `get_top_prism(i)`.

### B.4.142 Interface `get_unerodible_bottom_face`

**physical unit:** —

Extract layer index $k (1 \leq k \leq N_z)$ of the unerodible bottom face:

1. `get_unerodible_bottom_face(j)`: returns scalar $s$ [int] with unerodible bottom face layer index $k$ for edge $j (1 \leq j \leq N_s)$;

2. `get_unerodible_bottom_face()`: array $a(N_s)$ [int] is returned with unerodible bottom face layer indices for all $N_s$ sides with flow, where $a(j)$ represents the bottom layer index for the $j$-th side.

$N_s$ is given by `get_nof_internal_edges()` and $N_u$ by $N_s + get_nof_inflow_edges()$

For the actual bottom layer index use `get_bottom_face` instead.

### B.4.143 Interface `get_unerodible_bottom_prism`

**physical unit:** —

Extract layer index $k (1 \leq k \leq N_z)$ of the unerodible bottom prism:

1. `get_unerodible_bottom_prism(i)`: returns scalar $s$ [int] with unerodible bottom prism layer index $k$ for polygon $i (1 \leq i \leq N_p)$;

2. `get_unerodible_bottom_prism()`: array $a(N_p)$ [int] is returned with unerodible bottom prism layer indices for all $N_p$ polygons, where $a(i)$ represents the bottom layer index for the $i$-th polygon.

The total number of polygons $N_p$ can be determined by means of `get_nof_polygons()`

For the actual bottom layer index use `get_bottom_prism` instead.

### B.4.144 Interface `get_velocity`

**physical unit:** $\text{m/s}$

Retrieve horizontal normal velocity $u$ at computational faces:

1. `get_velocity(j,k)`: returns scalar $s$ [dp] with $u$ at the $j$-th side (edge) $(1 \leq j \leq N_s)$ within layer $k (k_b(j) \leq k \leq k_t(j))$;

---

**B. User interface**    
Page 98
2. get_velocity(j): array a(Nz) [dp] is returned with u for all computational faces above side j (1 ≤ j ≤ Ns), where a(k) contains u for the k-th layer (kb(j) ≤ k ≤ kt(j));

3. get_velocity(): array a(J3) [dp] is returned with u at all J3 faces.

Ns can be retrieved using get_nof_edges() and Nz is obtained from get_nof_layers(). The bottom face layer index kb(j) can be determined using get_bottom_face(j) whereas kt(j) is given by get_top_face(j). J3 is retrieved from get_nof_faces().

Use get_vertically_avr_velocity to retrieve depth averaged normal velocities.

B.4.145 Interface get_velocity_courant

physical unit: -

Retrieve velocity Courant number Cr_u = Δt |u| / δ at faces:

1. get_velocity_courant(j,k): returns scalar s [dp] with velocity Courant number as given above for layer k (kb(j) ≤ k ≤ kt(j)) above edge j (1 ≤ j ≤ Ns);

2. get_velocity_courant(j): returns array a(Nz) [dp] with velocity Courant numbers for all layers Nz above edge j (1 ≤ j ≤ Ns).

Ns is given by get_nof_internal_edges() and Nz by Nb + get_nof_inflow_edges(). Nz is obtained from get_nof_layers(). The bottom face layer index kb(j) can be determined using get_bottom_face(j) whereas kt(j) is given by get_top_face(j).

For easy access to maximum values use get_max_velocity_courant instead.

B.4.146 Interface get_vertex_coordinates

physical unit: m [origin]

Retrieve coordinates (x, y) for vertices:

1. get_vertex_coordinates(n): array a(2) [dp] is returned with coordinates of the n-th vertex (1 ≤ n ≤ Nv), where a(1) and a(2) are the respective x- and y-coordinates of the n-th vertex;

2. get_vertex_coordinates(): array a(2,Nv) [dp] is returned with the coordinates for all Nv vertices, wherein a(1,n) and a(2,n) are the respective x- and y-coordinates of the n-th vertex.

The overall number of vertices Nv can be obtained from get_nof_vertices().
B.4.147  Interface get_vertical_diffusivity

physical unit: m²/s

Retrieve scalar constant for vertical molecular diffusivity:

1. get_vertical_diffusivity(): returns scalar s [dp] with scalar constant vertical molecular diffusivity.

This value is set by the user in the input file "untrim.inp" and is treated as a constant for all computational points.

Notice: The total vertical diffusivity applied during computation is the sum of this value plus the vertical turbulent diffusivity set through CALL set_turbulent_v_diffusivity.

B.4.148  Interface get_vertical_velocity

physical unit: m/s

Retrieve vertical velocity component \( w \) either at top of prisms or interpolated in \( z \)-direction:

1. get_vertical_velocity(i,k): returns scalar s [dp] with \( w \) at the \( i \)-th polygon (\( 1 \leq i \leq N_p \)) for top of layer \( k (k_b(i) \leq k \leq k_t(i)) \);

2. get_vertical_velocity(i,z): returns scalar s [dp] with \( w \) vertically interpolated to position \( z \) [dp] above polygon \( i (1 \leq i \leq N_p) \);

3. get_vertical_velocity(i): array \( a(0:N_z) \) [dp] is returned with \( w \) for all computational prisms above polygon \( i (1 \leq i \leq N_p) \), where \( a(k) \) contains \( w \) for the top of the \( k \)-th layer \( (k_b(i) \leq k \leq k_t(i)) \);

4. get_vertical_velocity(): array \( a(I_3) \) [dp] is returned with \( w \) for all \( I_3 \) computational prisms.

\( I_3 \) is given by get_nof_prisms(). \( N_p \) can be determined from get_nof_polygons(). \( k_b(i) \) results from get_bottom_prism(i) and \( k_t(i) \) from get_top_prism(i). \( N_z \) is given by get_nof_layers(). If a \( z \)-coordinate is prescribed in the function call, \( w \) is computed by means of linear interpolation in vertical direction. Horizontally no interpolation scheme is applied.

B.4.149  Interface get_vertical_viscosity

physical unit: m²/s

Retrieve scalar constant for vertical kinematic viscosity:
1. get_vertical_viscosity(): returns scalar \( s \) [dp] with scalar constant vertical kinematic viscosity.

This value is set by the user in the input file "untrim.inp" and is treated as a constant for all computational points.

Notice: The total vertical viscosity applied during computation is the sum of this value plus the turbulent vertical viscosity set through `CALL set_turbulent_viscosity`.

### B.4.150 Interface get_vertically_avr_velocity

**Physical unit**: m/s

Retrieve vertically averaged normal velocity:

1. get_vertically_avr_velocity(j): returns scalar \( s \) [dp] with the depth averaged normal velocity at edge \( j \) (\( 1 \leq j \leq N_s \));

2. get_vertically_avr_velocity(): returns array \( a(N_s) \) [dp] with the depth averaged normal velocities for all edges with flow \( N_s \).

\( N_s \) is given by `get_no_of_internal_edges()` and \( N_s \) by \( N_s + \) `get_no_of_inflow_edges()`.

**Notice**: Use `get_velocity` to retrieve depth structured normal velocities.

### B.4.151 Interface get_volume

**Physical unit**: m³

Computes the (local) volume for computational prisms:

1. get_volume(i,k): returns scalar \( s \) [dp] corresponding to the fluid volume inside the computational prism above polygon \( i \) (\( 1 \leq i \leq N_p \)) within layer \( k \) (\( k_b(i) \leq k \leq k_t(i) \));

2. get_volume(i): returns array \( a(N_z) \) [dp] with the fluid volume belonging to prisms at polygon \( i \) (\( 1 \leq i \leq N_p \));

3. get_volume(): returns array \( a(I_3) \) [dp] with the fluid volume belonging to all prisms \( I_3 \).

\( N_p \) is obtained from `get_no_of_polygons` \( k_b(i) \) from `get_bottom_prism(i)` and \( k_t(i) \) from `get_top_prism(i)`.

For \( N_z \), use `get_no_of_layers()` and for \( I_3 \), use `get_no_of_prisms()`.

**Notice**: For inquiries concerning total volume use `get_total_volume` instead.
B.4.152 Interface `get_wet_area`

**physical unit**: \( m^2 \)

Retrieve (total) wet area for polygons:

1. `get_wet_area(i, k)`: returns scalar \( s \) [dp] corresponding to the (total) wet area above polygon \( i \) (\( 1 \leq i \leq N_p \)) up to the depth of layer \( k \) (\( k_b(i) \leq k \leq k_t(i) \)); this is not necessarily the bottom area touched by subprisms within layer \( k \);

2. `get_wet_area(i)`: returns scalar \( s \) [dp] with the (total) wet area above polygon \( i \) (\( 1 \leq i \leq N_p \));

3. `get_wet_area()`: returns scalar \( s \) [dp] with the overall wet area in the computational domain.

\( N_p \) is retrieved using `get_nof_polygons()`. \( k_b(i) \) can be obtained from `get_bottom_prism(i)` and \( k_t(i) \) from `get_top_prism(i)`.

If all subpolygons in polygon \( i \) are fully covered with water, the wet area may still be less than the polygon area obtained from `get_polygon_area(i)` in case the subpolygons do not fully cover the whole polygon.

Use `get_area` to retrieve total polygon area available in computations.

B.4.153 Interface `get_wet_length`

**physical unit**: \( m \)

Evaluate wet length along edges:

1. `get_wet_length(j, k)`: returns scalar \( s \) [dp] with wet length along edge \( j \) (\( 1 \leq j \leq N_s \)) up to the depth of layer \( k \) (\( k_b(j) \leq k \leq k_t(j) \));

2. `get_wet_length(j)`: returns scalar \( s \) [dp] with the wet (total) length along edge \( j \) (\( 1 \leq j \leq N_s \));

3. `get_wet_length()`: returns array \( a(N_s) \) [dp] with the wet length for all \( N_s \) edges with flow.

\( N_s \) is retrieved using `get_nof_edges()`. \( k_b(j) \) can be obtained from `get_bottom_face(j)` and \( k_t(j) \) from `get_top_face(j)`. \( N_s \) is given by `get_nof_internal_edges()` and \( N_i \) by \( N_k + get_nof_inflow_edges() \).

If all subedges are wet, the wet length of edge \( j \) may be still less than the (geometric) length of an edge obtained with `get_edge_length(j)` in case the subedges do not fully cover the edge.

Use `get_length` to retrieve total edge length available in computations.
### B.4.154 Interface `get which bottom flux`

**Physical unit:** —

Retrieve which bottom flux (index) is applied to species:

1. `get which bottom flux(m)`: returns scalar `s [int]` for the `m`-th specie `1 ≤ m ≤ Nc` with index for the bottom flux used during transport of this specie;

2. `get which bottom flux()`: returns array `a(Nc [int])` for all species `Nc` with indices for the bottom fluxes used during transport of all species.

`Nc` is obtained from `get nof species()`.

### B.4.155 Interface `get which diffusivity`

**Physical unit:** —

Retrieve (indices) horizontal as well as vertical diffusivity applied to species:

1. `get which diffusivity(m)`: returns array `a(2 [int])` for the `m`-th specie `1 ≤ m ≤ Nc` with index `a(1)` for the horizontal as well as index `a(2)` for the vertical diffusivity used during transport of this specie;

2. `get which diffusivity()`: returns array `a(Nc, 2 [int])` for all species `Nc` with indices `a(:, 1)` for the horizontal as well as indices `a(:, 2)` for the vertical diffusivities used during transport of all species.

`Nc` is obtained from `get nof species()`.

### B.4.156 Interface `get which flux`

**Physical unit:** —

Retrieve which fluxes (indices) are applied to species:

1. `get which flux(m)`: returns array `a(2 [int])` for the `m`-th specie `1 ≤ m ≤ Nc` with index `a(1)` for the surface as well as index `a(2)` for the bottom flux used during transport of this specie;

2. `get which flux()`: returns array `a(Nc, 2 [int])` for all species `Nc` with indices `a(:, 1)` for the surface as well as indices `a(:, 2)` for the bottom fluxes used during transport of all species.

`Nc` is obtained from `get nof species()`.
**B.4.157 Interface get\_which\_h\_diffusivity**

**physical unit:** —

Retrieve (index) horizontal diffusivity applied to species:

1. `get\_which\_h\_diffusivity(m)`: returns scalar \(s\) \([\text{int}]\) for the \(m\)-th specie \((1 \leq m \leq N_c)\) with index for the horizontal diffusivity used during transport of this specie;

2. `get\_which\_h\_diffusivity()`: returns array \(a(N_c)\) \([\text{int}]\) for all species \(N_c\) with indices for the horizontal diffusivities used during transport of all species.

\(N_c\) is obtained from `get\_nof\_species()`.

**B.4.158 Interface get\_which\_settling\_velocity**

**physical unit:** —

Retrieve (index) of settling velocity applied to species:

1. `get\_which\_settling\_velocity(m)`: returns scalar \(s\) \([\text{int}]\) for the \(m\)-th specie \((1 \leq m \leq N_c)\) with index \(s\) for the settling velocity used during transport of this specie;

2. `get\_which\_settling\_velocity()`: returns array \(a(N_c)\) \([\text{int}]\) for all species \(N_c\) with indices \(a(:)\) for the settling velocity used during transport of all species.

\(N_c\) is obtained from `get\_nof\_species()`.

**B.4.159 Interface get\_which\_surface\_flux**

**physical unit:** —

Retrieve which surface flux (index) is applied to species:

1. `get\_which\_surface\_flux(m)`: returns scalar \(s\) \([\text{int}]\) for the \(m\)-th specie \((1 \leq m \leq N_c)\) with index for the surface flux used during transport of this specie;

2. `get\_which\_surface\_flux()`: returns array \(a(N_c)\) \([\text{int}]\) for all species \(N_c\) with indices for the surface fluxes used during transport of all species.

\(N_c\) is obtained from `get\_nof\_species()`.
B.4.160  Interface get\_which\_v\_diffusivity

**physical unit:** —

Retrieve (index) vertical diffusivity applied to species:

1. get\_which\_v\_diffusivity(m): returns scalar s [int] for the m-th specie (1 ≤ m ≤ \( N_c \)) with index for the vertical diffusivity used during transport of this specie;

2. get\_which\_v\_diffusivity(): returns array a(Nc) [int] for all species \( N_c \) with indices for the vertical diffusivities used during transport of all species.

\( N_c \) is obtained from get\_nof\_species().

B.4.161  Interface get\_wind\_friction

**physical unit:** —

Retrieve wind friction coefficient \( \gamma_f \) at internal sides:

1. get\_wind\_friction(j): returns scalar s [dp] with \( \gamma_f \) at the j-th side (1 ≤ j ≤ \( N_{si} \));

2. get\_wind\_friction(): array a(Nsi) [dp] is returned with \( \gamma_f \) at all \( N_{si} \) sides, where a(j) contains \( \gamma_f \) for the j-th side.

\( N_{si} \) is obtained from get\_nof\_internal\_edges().

B.4.162  Interface get\_wind\_stress

**physical unit:** \( m^2/s^2 \)

Retrieve the normalized normal component of wind shear stress at internal sides:

1. get\_wind\_stress(j): return scalar s [dp] for the j-th side (1 ≤ j ≤ \( N_{si} \));

2. get\_wind\_stress(): array a(Nsi) [dp] is returned for all \( N_{si} \) sides, where a(j) is the respective value for the j-th side.

\( N_{si} \) can be obtained from get\_nof\_internal\_edges(). The (real physical) normal component of the actual wind stress \( \tau_f \) is obtained by multiplication with \( \rho_0 \).
B.4.163 Interface get_wind_velocity
physical unit: m/s

Retrieve components of the wind velocity \((u_a, v_a)\) at internal sides:

1. \texttt{get\_wind\_velocity}(j): array \(a(2)\) [dp] is returned for the \(j\)-th side \((1 \leq j \leq N_{si})\), where \(a(1)\) and \(a(2)\) correspond to the \(x\)- and \(y\)-components of the wind velocity \((u_a, v_a)\);

2. \texttt{get\_wind\_velocity}(): array \(a(2, N_{si})\) [dp] is returned with wind velocity at all \(N_{si}\) sides, where \(a(:, j)\) gives the respective value for the \(j\)-th side.

\(N_{si}\) is obtained from \texttt{get\_nof\_internal\_edges()}

B.4.164 Interface get_wind_velocity_x
physical unit: m/s

Retrieve \(x\)-component \(u_a\) of the wind velocity at internal sides:

1. \texttt{get\_wind\_velocity\_x}(j): returns scalar \(s\) [dp] with the \(x\)-component \(u_a\) for the \(j\)-th side \((1 \leq j \leq N_{si})\);

2. \texttt{get\_wind\_velocity\_x}(): array \(a(N_{si})\) [dp] is returned with \(x\)-component \(u_a\) at all \(N_{si}\) sides.

\(N_{si}\) is obtained from \texttt{get\_nof\_internal\_edges()}

B.4.165 Interface get_wind_velocity_y
physical unit: m/s

Retrieve \(y\)-component \(v_a\) of the wind velocity at internal sides:

1. \texttt{get\_wind\_velocity\_y}(j): returns scalar \(s\) [dp] with the \(y\)-component \(v_a\) for the \(j\)-th side \((1 \leq j \leq N_{si})\);

2. \texttt{get\_wind\_velocity\_y}(): array \(a(N_{si})\) [dp] is returned with \(y\)-component \(v_a\) at all \(N_{si}\) sides.

\(N_{si}\) is obtained from \texttt{get\_nof\_internal\_edges()}
B.4.166 Interface get\_x\_slope
physical unit: —

Retrieve actual slope of the system in x-direction:

1. get\_x\_slope(): returns scalar \(s\) [dp] with actual slope in x-direction.

B.4.167 Interface get\_x\_vertex\_coordinate
physical unit: m [origin]

Retrieve x-coordinate for vertices:

1. get\_x\_vertex\_coordinate(n): scalar \(s\) [dp] is returned with the x-coordinate for the \(n\)-th vertex \((1 \leq n \leq N_v)\);
2. get\_x\_vertex\_coordinate(): array \(a(N_v)\) [dp] ist returned with the x-coordinates for all \(N_v\) vertices.

The overall number of vertices \(N_v\) can be obtained from get\_nof\_vertices().

B.4.168 Interface get\_y\_slope
physical unit: —

Retrieve actual slope of the system in y-direction:

1. get\_y\_slope(): returns scalar \(s\) [dp] with actual slope in y-direction.

B.4.169 Interface get\_y\_vertex\_coordinate
physical unit: m [origin]

Retrieve y-coordinate for vertices:

1. get\_y\_vertex\_coordinate(n): scalar \(s\) [dp] is returned with the y-coordinate for the \(n\)-th vertex \((1 \leq n \leq N_v)\);
2. get\_y\_vertex\_coordinate(): array \(a(N_v)\) [dp] ist returned with the y-coordinates for all \(N_v\) vertices.

The overall number of vertices \(N_v\) can be obtained from get\_nof\_vertices().
B.4.170 Interface is_dry_edge

**physical unit:** logical, no unit

Returns status whether an edge is *dry* or not:

1. \( \text{is\_dry\_edge}(j) \): returns scalar \( s \) [log] which is *true* if the \( j \)-th edge \((1 \leq j \leq N_s)\) is dry or *false* otherwise.

\( N_s \) is given by \text{get\_nof\_internal\_edges()} and \( N_s \) by \( N_s + \text{get\_nof\_inflow\_edges()} \)

B.4.171 Interface is_dry_polygon

**physical unit:** logical, no unit

Returns status whether a polygon is *dry* or not:

1. \( \text{is\_dry\_polygon}(i) \): returns scalar \( s \) [log] which is *true* if the \( i \)-th polygon \((1 \leq i \leq N_p)\) is dry or *false* otherwise.

\( N_p \) is given by \text{get\_nof\_polygons()}

B.4.172 Interface is_wet_edge

**physical unit:** logical, no unit

Returns status whether an edge is *wet* or not:

1. \( \text{is\_wet\_edge}(j) \): returns scalar \( s \) [log] which is *true* if the \( j \)-th edge \((1 \leq j \leq N_s)\) is wet or *false* otherwise.

\( N_s \) is given by \text{get\_nof\_internal\_edges()} and \( N_s \) by \( N_s + \text{get\_nof\_inflow\_edges()} \)

B.4.173 Interface is_wet_polygon

**physical unit:** logical, no unit

Returns status whether a polygon is *wet* or not:

1. \( \text{is\_wet\_polygon}(i) \): returns scalar \( s \) [log] which is *true* if the \( i \)-th polygon \((1 \leq i \leq N_p)\) is wet or *false* otherwise.

\( N_p \) is given by \text{get\_nof\_polygons()}

---

B. User interface
Interface sediment

physical unit: logical, no unit

Check if a specie is a sediment with a nonzero settling velocity:

1. `sediment(m)` returns scalar $s \ [\text{log}]$ which is `true` if the $m$-th specie ($1 \leq m \leq N_c$) is a sediment with a nonzero settling velocity or `false` otherwise.

$N_c$ can be determined from `get_no_of_species()`.

Interface with_hydrodynamic_pressure

physical unit: logical, no unit

Check if the hydrodynamic pressure solver has been turned on in the ongoing simulation:

1. `with_hydrodynamic_pressure()` returns scalar $s \ [\text{log}]$ which is `true` if the ongoing simulation is performed with the assumption that the pressure is non-hydrostatic or `false` otherwise (hydrostatic assumption).

The result of this function is dependent on user’s input data in file "untrim.inp".

Interface with_scalar_transport

physical unit: logical, no unit

Check if scalar transport has been turned on in the ongoing simulation:

1. `with_scalar_transport()` returns scalar $s \ [\text{log}]$ which is `true` if the ongoing simulation is performed with transport of scalar species ($N_c > 0$) or `false` otherwise ($N_c = 0$).

The result of this function is dependent on user’s input data, variable `nsp`, in file "untrim.inp".
B.5 Check routines

- check routines
  - grid structure and grid quality
    * \texttt{check\_grid} (subroutine)
  - continuity
    * \texttt{check\_continuity} (function)
  - parameters
    * \texttt{check\_all} (subroutine)

All check routines available to the user are subsequently described and listed in alphabetical order.

B.5.1 Subroutine check\_all

This subroutine has no parameters. The routine can be called during each time step. The following checks are performed and in case of any violation of one of the following tests the execution of the program will be terminated.

The following tests are performed if the routine is called for the first time:

1. valid number of layers $N_z \geq 1$ ($nk$ in "untrim.inp");
2. valid indicator for hydrostatic/non-hydrostatic pressure approximation ($nq$ in "untrim.inp");
3. valid number of species used $N_c \geq 0$ ($nsp$ in "untrim.inp");
4. valid maximum number of iterations $m_{PCG}^{\max} \geq 0$ ($maxiter$ in "untrim.inp");
5. valid time step $\Delta t > 0$ ($delt$ in "untrim.inp");
6. valid minimum layer depth $H_{\text{min}} > 0$ ($dzmin$ in "untrim.inp");
7. valid minimum time substep size $\Delta \tau > 0$ ($dtmin$ in "untrim.inp");
8. valid horizontal kinematic viscosity $\nu_h \geq 0$ ($hvis$ in "untrim.inp");
9. valid vertical kinematic viscosity $\nu_v \geq 0$ ($vvis$ in "untrim.inp");
10. valid horizontal molecular diffusivity $K_h \geq 0$ ($hdif$ in "untrim.inp");
11. valid vertical kinematic viscosity $K_v \geq 0$ ($vdif$ in "untrim.inp");
12. valid Chezy friction coefficient $C_z > 0$ ($Cz$ in "untrim.inp");
13. valid tolerance $\epsilon_\eta$ for free-surface inner PCG iterative solver ($epsi$ in "untrim.inp");
14. valid tolerance $\varepsilon_q$ for non-hydrostatic pressure (inner) iterative PCG solver ($q_{psi}$ in "untrim.inp");

15. valid radiation time $t_r > 0$ ($tr_x$ in "untrim.inp");

16. valid indicator for advection terms off/on [0|1] ($noli$ in "untrim.inp");

17. valid indicator for slip at lateral boundaries [-1,1] ($slip$ in "untrim.inp");

18. valid implicitness factor $0.5 \leq \theta \leq 1.0$ ($theta$ in "untrim.inp");

19. valid layer sequence $h_k > k_{k-1}$ (input data in "untrim.inp").

The following tests are performed if the routine is called for the second and all subsequent times:

1. check radiation time $t_R > 0$ (⇒ set radiation time);

2. check horizontal turbulent viscosity $\nu_h \geq 0$ (⇒ set turbulent h viscosity);

3. check vertical turbulent viscosity $\nu_v \geq 0$ (⇒ set turbulent v viscosity);

4. check artificial porosity $p_y \geq 0$ (⇒ set artificial porosity);

5. check bottom friction coefficient $\gamma_b \geq 0$ (⇒ set bottom friction);

6. check wind friction coefficient $\gamma_t \geq 0$ (⇒ set wind friction).

If species are present, the following checks are also executed:

1. check horizontal turbulent diffusivity $K_h \geq 0$ (⇒ set turbulent h diffusivity);

2. check vertical turbulent diffusivity $K_v \geq 0$ (⇒ set turbulent v diffusivity);

3. check bottom flux parameter $\alpha_b \geq 0$ (⇒ set bottom alpha);

4. check bottom flux parameter $\beta_b \geq 0$ (⇒ set bottom beta);

5. check surface flux parameter $\alpha_r \geq 0$ (⇒ set surface alpha);

6. check surface flux parameter $\beta_r \geq 0$ (⇒ set surface beta).

### B.5.2 Function check_continuity

This function has no parameters. The routine can be called during each time step. It returns array $a(3) \ [dp]$ with the numbers computed below:

1. maximum error for computed water levels;

2. maximum volume error within a polygon;
3. total volume error for the computational domain.

This routine is very helpful to check accuracy of the numerical solution. Results may be used to optimize some of the solver parameters in file "untrim.inp".

B.5.3 Subroutine check_grid

This subroutine has no parameters. The routine may be called once after the grid has been read by the computational core of the program. The following actions are carried through:

1. print elementary informations about grid size;
2. determine smallest/largest area for all polygons;
3. determine smallest/largest length for all sides (edges);
4. determine smallest/largest distance between adjacent polygon centers for all edges;
5. determine smallest/largest area for all subpolygons;
6. determine smallest/largest length for all subedges;
7. compute statistics for all distances between adjacent polygon centers;
8. determine the number of centers which are not contained in their polygon;
9. compute maximum violation of orthogonality condition;
10. determine the number of edges with incorrect orientation (edges with negative distance between adjacent polygon centers);
11. compute minimum/average accuracy of momentum (second order if all edge centers are exactly located in the middle between adjacent polygon centers);
12. compute minimum/average accuracy of continuity (second order if all edge centers are exactly located in the middle of an edge);
13. stop execution if red-black-sorting is incorrect;
14. stop execution if inflow edges \((N_s + 1 \leq j \leq N_f)\) have a right neighbour polygon (only left neighbours allowed);
15. stop execution if there are negative polygon areas;
16. stop execution in case of invalid sources and/or sinks locations;
17. warning in case of zero subpolygon areas;
18. warning in case of zero subedge lengths;
19. stop execution in case of negative subpolygon areas;
20. stop execution in case of negative subedge lengths;
21. stop execution in case (geometric) polygon area is smaller than sum of respective subpolygon areas;
22. stop execution in case (geometric) edge length is smaller than sum of respective subedge lengths.
B.6 User interface routines

In the following sections for each of the typical operations within the user interface routines the most important get- and set-methods are listed. These listings are in general not comprehensive, and the user may find that additional methods out of the vast amount of available subroutines and functions (see section B.3 on page 4 and section B.4 on page 41) might be useful in case.

B.6.1 Subroutine user_set_input_files

This routine is called once during the initialization phase by the computational core of the software. The user must set user-specific paths and names for the three different input files (no default values used in the program). The following routines and/or functions can be useful in this respect:

- path and name for the steering file, e.g. "untrim.inp"
  
  - set_input_file

- path and name for the grid file, e.g. "untrim.grd"
  
  - set_grid_file

- path and name for the sources and sinks file, e.g. "untrim.srs"
  
  - set_source_file

- logical unit number for informative message data output to file "UnTRIM.txt"
  
  - set_printout_unit
  - get_printout_unit

Notice: if one or several of the before mentioned set-routines are not used the respective file names will be initialized to their default names.

B.6.2 Subroutine user_set_initial_conditions

This routine is called once during the initialization phase by the computational core of the software. The user has the possibility to set the initial conditions as required. The following routines and/or functions may be useful in this context:

- logical unit number for informative message data output to file "UnTRIM.txt"
  
  - set_printout_unit

- problem size
  
  - grid
* `get_nof_polygons`
* `get_nof_edges`
* `get_nof_vertices`
* `get_nof_layers`
* `get_nof_faces`
* `get_nof_prisms`
* `get_nof_max_edges`

  - subgrid
    * `get_nof_subedges`
    * `get_nof_subpolygons`

  - others
    * `get_nof_species`

• inquiry functions
  
  - `with_scalar_transport`
  - `with_hydrodynamic_pressure`
  - `get_face_index`
  - `get_prism_index`

• steering data
  
  - Coriolis approximation
    * `set_constant_coriolis`
  
  - transport algorithm for scalar species
    * `set_flux_limiter`

  - scalar species
    * active species (with settling velocity)
      · `set_sediment`
      · `set_new_settling_velocity`
      · `share_settling_velocity`
    * fluxes at the bottom and the surface
      · `set_new_bottom_flux`
      · `set_new_surface_flux`
      · `set_new_flux`
      · `share_bottom_flux`
      · `share_surface_flux`
      · `share_flux`

  * horizontal and vertical turbulent diffusivities
Notice: if one or several of the before mentioned set-routines are not used, the respective data will be typically initialized to zero.

**B.6.3 Subroutine user_set_forcing_terms**

This routine is called once during each time step performed by the computational core of the software. The user has the possibility to set the boundary data as required. On the following pages, the most useful routines and/or functions are listed in several sections.

- logical unit number for informative message data output to file "UnTRIM.txt"
  - get_printout_unit
**open boundary data** Boundary data at open model boundaries are normally set by the user to non-default values. Ideally the data should be available for time level $t^{n+1}$. The following routines and/or functions may be of use for this purpose:

- **time and problem size**
  - `get_time`
  - `get_time_step`
  - `get_no_boundary_polygons`
  - `get_no_internal_edges`
  - `get_no_inflow_edges`

- **vertical structure**
  - `get_no_layers`
  - `get_layer_interface`
  - `get_layer`
  - `get_bottom_face`
  - `get_top_face`
  - `get_bottom_prism`
  - `get_top_prism`
  - `get_unerodible_bottom_face`
  - `get_unerodible_bottom_prism`

- **inquiry functions**
  - `with_scalar_transport`
  - `with_hydrodynamic_pressure`
  - `get_edge_center`
  - `get_polygon_center`
  - `is_dry_edge`
  - `is_wet_edge`
  - `is_dry_polygon`
  - `is_wet_polygon`

- **hydrodynamic open boundary data (with prescribed water level)**
  - `CALL set_elevation_bc`
  - `CALL set_pressure_bc`
  - `CALL set_radiation_time`
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

- hydrodynamic inflow boundary data (with prescribed flow)
  - CALL set_inflow_bc
- species open boundary data (with prescribed water level)
  - CALL set_concentration_bc
- species inflow boundary data (with prescribed flow)
  - CALL set_inflow_cc

Notice: if one or several of the before mentioned set-routines are never used the respective boundary data will be initialized with their default values (normally zero). Some of the functions are acting on different parts (polygons with prescribed water level and edges with prescribed flow) of the open boundary.

**morphodynamics and slope**  In case of a time-varying bathymetry the user has to set the evolution of the bottom, typically available from a separate morphodynamic simulation package. Ideally the data should be available for time level $t^{n+1}$. The following routines and/or functions may be of use for this purpose:

- time and problem size
  - get_time
  - get_time_step
  - get_nof_polygons
  - get_nof_edges
  - get_nof_max_edges

- inquiry functions
  - get_edge_center
  - get_polygon_center
  - is_dry_edge
  - is_wet_edge
  - is_dry_polygon
  - is_wet_polygon

- bottom evolution
  - CALL set_bottom_deviation
  - CALL set_edge_bottom_deviation
  - CALL set_slope

Notice: if one or several of the before mentioned methods are never used the respective boundary data will be initialized with their default values (normally zero) which corresponds to static bathymetry, no morphodynamics.
body forces  If additional body forces, e. g. due to waves have to be taken into account, they have to be prescribed by the user during each time step. Ideally data should be available for time level $t^{n+1}$. The following routines and/or functions may be of use for this purpose:

- time and problem size
  - `get_time`
  - `get_time_step`
  - `get_nof_edges`
  - `get_nof_layers`
  - `get_nof_max_edges`

- vertical structure
  - `get_bottom_face`
  - `get_top_face`

- inquiry functions
  - `get_edge_center`
  - `is_dry_edge`
  - `is_wet_edge`
  - `is_dry_polygon`
  - `is_wet_polygon`

- body force
  - `set_body_force`

Notice: if one or several of the before mentioned methods are never used the respective boundary data will be initialized with their default values (normally zero) which corresponds to no additional (horizontal) body forces applied.

sources and sinks  If the transport of species is modelled the respective concentration for sources and sinks should be set. The following routines and/or functions may be useful in this respect:

- time and problem size
  - `get_time`
  - `get_time_step`
  - `get_nof_species`
  - `get_nof_point_sources`
• inquiry functions
  - \texttt{with\_scalar\_transport}
  - \texttt{get\_concentration}
  - \texttt{get\_source\_concentration}
  - \texttt{get\_source\_discharge}
  - \texttt{get\_source\_layer}
  - \texttt{get\_source\_polygon}
  - \texttt{is\_dry\_edge}
  - \texttt{is\_wet\_edge}
  - \texttt{is\_dry\_polygon}
  - \texttt{is\_wet\_polygon}

• strength of sources and sinks
  - \texttt{CALL set\_point\_sources\_discharge}

• concentration for sources and sinks
  - \texttt{CALL set\_point\_sources\_concentration}

Notice: if some of the before mentioned set-routines are never used, the respective sources and sinks data will remain with their default values (normally zero).

**bottom friction boundary data**  Normally the user has to prescribe a friction coefficient at the bottom. In dependence on the parametrization used, this may be either a constant or spatially varying, e.g. in dependence on time-dependent water depth. The following routines and/or functions may be useful in this respect:

• time and problem size
  - \texttt{get\_time}
  - \texttt{get\_time\_step}
  - \texttt{get\_nof\_internal\_edges}

• inquiry functions
  - \texttt{get\_edge\_center}
  - \texttt{get\_max\_edge\_total\_depth}
  - \texttt{get\_avr\_edge\_total\_depth}
  - \texttt{get\_edge\_wet\_area}
  - \texttt{get\_dry\_length}
  - \texttt{get\_wet\_length}
- is_dry_edge
- is_wet_edge
- is_dry_polygon
- is_wet_polygon

• bottom friction coefficient
  
  - CALL set_bottom_friction

Notice: if the before mentioned set-routine is never used the bottom friction coefficient is assumed to be equal to \( \frac{g}{c_z^2} \), with \( c_z \) prescribed in file "untrim.inp".

**atmospheric boundary data**  The user is enabled to set wind speed and atmospheric pressure at the free surface. Normally wind speed as measured or computed 10 m above the free surface is used. In reality the wind friction coefficient is varying in space and time. The following routines and/or functions may be useful:

• time and problem size

  - get_time
  - get_time_step
  - get_nof_internal_edges
  - get_nof_polygons

• inquiry functions

  - get_edge_center
  - get_polygon_center
  - is_dry_edge
  - is_wet_edge
  - is_dry_polygon
  - is_wet_polygon

• wind friction coefficient

  - CALL set_wind_friction

• wind speed

  - CALL set_wind_velocity

• atmospheric pressure

  - CALL set_atmospheric_pressure
• rain and evaporation
  
  \texttt{- CALL set\_rain}

Notice: if some of the aforementioned set-routines are never used respective atmospheric data will be initialized to default values. Zero wind friction is is obtained by explicitly requesting \texttt{CALL set\_wind\_friction(0.0)}.

\textbf{density (equation of state)} If there are species present, density is normally no longer a constant. The user is enabled to compute and set the density according to his or her requirements in dependence on the actual species concentrations. The following routines and/or functions may be useful in this respect:

• time and problem size
  
  \texttt{- get\_time}
  \texttt{- get\_time\_step}
  \texttt{- get\_nof\_layers}
  \texttt{- get\_nof\_prisms}
  \texttt{- get\_nof\_species}

• vertical structure
  
  \texttt{- get\_bottom\_prism}
  \texttt{- get\_top\_prism}

• inquiry functions
  
  \texttt{- with\_scalar\_transport}
  \texttt{- get\_concentration}
  \texttt{- is\_dry\_edge}
  \texttt{- is\_wet\_edge}
  \texttt{- is\_dry\_polygon}
  \texttt{- is\_wet\_polygon}

• density
  
  \texttt{- CALL set\_density}

Notice: if the before mentioned set-routine is never used, normalized density is assumed to be 1.0 ($\rho = \rho_0$).
turbulent viscosities and diffusivities  The user is enabled to assign meaningful values to the vertical and horizontal turbulent viscosities ($\nu_v^t$, $\nu_h^t$) and diffusivities ($K_v^t$, $K_h^t$). They normally strongly depend on space and time. The following routines and/or functions may be useful in this respect:

- time and problem size
  - `get_time`
  - `get_time_step`
  - `get_nof_layers`
  - `get_nof_faces`
  - `get_nof_prisms`
  - `get_nof_species`

- vertical structure
  - `get_bottom_face`
  - `get_top_face`
  - `get_bottom_prism`
  - `get_top_prism`
  - `get_avr_prism_height`
  - `get_max_prism_height`

- inquiry functions
  - `with_scalar_transport`
  - `get_gravity`
  - `get_edge_center`
  - `get_polygon_center`
  - `get_horizontal_velocity`
  - `get_horizontal_velocity_x`
  - `get_horizontal_velocity_y`
  - `get_max_total_depth`
  - `get_avr_total_depth`
  - `get_wet_area`
  - `get_volume`
  - `get_max_edge_total_depth`
  - `get_avr_edge_total_depth`
  - `get_edge_wet_area`
- `get_wet_length`
- `get_avr_face_height`
- `get_max_face_height`
- `get_which_h_diffusivity`
- `get_which_v_diffusivity`
- `get_which_diffusivity`
- `is_dry_edge`
- `is_wet_edge`
- `is_dry_polygon`
- `is_wet_polygon`

**• turbulent viscosity**
- `CALL set_turbulent_h_viscosity`
- `CALL set_turbulent_v_viscosity`

**• turbulent diffusivity (if scalar transport of tracers is taken into account)**
- `CALL set_turbulent_h_diffusivity`
- `CALL set_turbulent_v_diffusivity`

Notice: the before mentioned functions enable the user to set the spatially- and time-varying component of the total viscosities and diffusivities as used in the program; the total values do also take into account the constant contribution from the steering data file `untrim.inp`; if the above-mentioned set-routines are not used, total diffusivities and viscosities are equivalent to the values from "untrim.inp".

**fluxes of tracers through the surface and the bottom**  The user is enabled to set the boundary data at the free surface as well as at the bottom individually for each of the species, if required. The following routines and/or functions may be useful:

**• time and problem size**
- `get_time`
- `get_time_step`
- `get_nof_polygons`
- `get_nof_species`

**• vertical structure**
- `get_bottom_prism`
- `get_top_prism`
• inquiry functions
  - `with_scalar_transport`
  - `sediment`
  - `get_concentration`
  - `get_sediment`
  - `get_settling_velocity`
  - `get_wet_area`
  - `get_which_bottom_flux`
  - `get_which_surface_flux`
  - `get_which_flux`
  - `is_dry_edge`
  - `is_wet_edge`
  - `is_dry_polygon`
  - `is_wet_polygon`

• species bottom flux data
  - `CALL set_bottom_alpha`
  - `CALL set_bottom_beta`
  - `CALL set_bottom_concentration`

• species surface flux data
  - `CALL set_surface_alpha`
  - `CALL set_surface_beta`
  - `CALL set_surface_concentration`

Notice: if one or several of the before mentioned set-routines are never used the respective flux data will be initialized to zero. This means that a zero-flux condition will be applied at the free surface, and for the bottom.

settling velocity of active tracers The user is enabled to assign meaningful values to the settling velocity \( w_s \) for suspended sediments, which e.g. may depend on turbulence intensity. Settling velocity may vary in space and time. Settling velocities between different tracers may differ by a constant factor. The following routines and/or functions may be useful in this respect:

• time and problem size
  - `get_time`
- `get_time_step`
- `get_nof_polygons`
- `get_nof_species`

• vertical structure
  - `get_bottom_prism`
  - `get_top_prism`

• inquiry functions
  - `with_scalar_transport`
  - `sediment`
  - `get_concentration`
  - `get_gravity`
  - `get_turbulent_viscosity`
  - `get_turbulent_v_diffusivity`
  - `get_which_settling_velocity`
  - `is_dry_edge`
  - `is_wet_edge`
  - `is_dry_polygon`
  - `is_wet_polygon`

• settling velocity
  - `CALL set_sediment`
  - `CALL set_settling_velocity`

The settling velocity (set by means of `CALL set_settling_velocity`) will be multiplied for each tracer with a constant factor prescribed through `CALL set_sediment`.

Notice: if the aforementioned set-routine `CALL set_sediment` was never used for a specific specie, its settling velocity will remain zero.

**checks and balances** There are a few functions available which allow to retrieve valuable informations concerning numerical accuracy as well as performance of the computational core of the program:

• check iterations done by the iterative solvers
  - `get_nof_iterations`
  - `get_nof_inner_iterations`
  - `get_nof_outer_iterations`
- `get_nof_max_iterations`

- water volume
  - `get_total_volume`
  - `get_volume`

- tracer mass
  - `get_mass`
  - `get_total_mass`

- fluxes of tracer at the free surface and at the bottom
  - `get_surface_flux`
  - `get_bottom_flux`

- continuity of water volume
  - `check_continuity`

- wave celerity, Froude number and others
  - `get_celerity`
  - `get_courant`
  - `get_velocity_courant`
  - `get_froude`
  - `get_max_courant`
  - `get_max_velocity_courant`
  - `get_max_froude`

- correctness of parameters used
  - `CALL check_all`

---

### B.6.4 Subroutine `user_get_results`

This routine is called once for each time step. The user is offered the possibility to output any results he/she is interested in. The following routines and/or functions may be useful in this respect:

- logical unit number for informative message data output to file "UnTRIM.txt"
  - `get_printout_unit`

- time and problem size
- `get_time`
- `get_time_step`
- `get_no_polygons`
- `get_no_edges`
- `get_no_vertices`
- `get_no_faces`
- `get_no_prisms`
- `get_no_species`
- `get_no_layers`
- `get_no_point_sources`
- `get_no_max_edges`

• inquiry functions
  - `with_hydrodynamic_pressure`
  - `with_scalar_transport`
  - `is_dry_edge`
  - `is_wet_edge`
  - `is_dry_polygon`
  - `is_wet_polygon`
  - `get_face_index`
  - `get_prism_index`

• computational results
  - `get_elevation`
  - `get_velocity`
  - `get_concentration`
  - `get_pressure`
  - `get_horizontal_velocity`
  - `get_horizontal_velocity_x`
  - `get_horizontal_velocity_y`
  - `get_vertically_avr_velocity`
  - `get_vertical_velocity`
  - `get_wind_stress`
  - `get_bottom_stress`
  - `get_mass`
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

- `get_total_mass`

- **parameters and coefficients**
  - `get_horizontal_diffusivity`
  - `get_horizontal_viscosity`
  - `get_turbulent_h_diffusivity`
  - `get_turbulent_h_viscosity`
  - `get_vertical_diffusivity`
  - `get_vertical_viscosity`
  - `get_turbulent_v_diffusivity`
  - `get_turbulent_v_viscosity`
  - `get_wind_friction`
  - `get_bottom_friction`
  - `get_density`
  - `get_surface_alpha`
  - `get_surface_beta`
  - `get_surface_concentration`
  - `get_bottom_alpha`
  - `get_bottom_beta`
  - `get_bottom_concentration`
  - `get_settling_velocity`
  - `get_bottom_deviation`
  - `get_avr_bottom_deviation`
  - `get_edge_bottom_deviation`
  - `get_avr_edge_bottom_deviation`
  - `get_slope`

- **boundary data**
  - `get_wind_velocity`
  - `get_wind_velocity_x`
  - `get_wind_velocity_y`
  - `get_atmospheric_pressure`
  - `get_surface_flux`
  - `get_bottom_flux`

- **sources and sinks**
- `get_source_discharge`
- `get_source_concentration`

- drying and wetting
  - `get_dry_area`
  - `get_wet_area`
C  Tables for set-routines and get-functions

In the following subsections all interfaces for set-routines as well as get-functions are presented in tabular form. For each interface the available set of arguments is given. The tables should help the experienced user to quickly select any routine together with the required arguments. For the arguments used throughout Table 3 on the following page to Table 23 on page 152 the nomenclature shown in Table 1 on page 1 and Table 2 on page 2 applies.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_artificial_porosity</td>
<td>((s \mid j, s \mid a(N_p)))</td>
</tr>
<tr>
<td>set_atmospheric_pressure</td>
<td>((s \mid i, a(N_p)))</td>
</tr>
<tr>
<td>set_body_force</td>
<td>((s_x, s_y \mid j, k, s_x, s_y \mid j, a_i(N_x), s_y \mid j, s_x, a_j(N_x) \mid a_i(N_x), a_j(N_x) \mid \ldots)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>set_bottom_alpha</td>
<td>((s \mid m, s \mid m, a(N_p) \mid b(N_p, N_q)))</td>
</tr>
<tr>
<td>set_bottom_beta</td>
<td>((s \mid m, s \mid m, a(N_p) \mid b(N_p, N_q)))</td>
</tr>
<tr>
<td>set_bottom_concentration</td>
<td>((s \mid m, s \mid m, a(N_p) \mid b(N_p, N_q)))</td>
</tr>
<tr>
<td>set_bottom_deviation</td>
<td>((s \mid i, s \mid a(N_p) \mid i, a^S(N_p)))</td>
</tr>
<tr>
<td>set_bottom_friction</td>
<td>((s \mid j, s \mid j, a^S(N_p) \mid j, a^S(N_p)))</td>
</tr>
<tr>
<td>set_concentration</td>
<td>((s \mid m, s \mid m, i, k, s \mid m, a(N_i) \mid m, i, a(N_i) \mid b(N_i, N_q)))</td>
</tr>
<tr>
<td>set_concentration_bc</td>
<td>((s \mid m, s \mid m, i, k, s \mid m, a(N_i) \mid b(N_i, N_q)))</td>
</tr>
<tr>
<td>set_constant_coriolis</td>
<td>((s \mid s))</td>
</tr>
<tr>
<td>set_density</td>
<td>((s \mid i, s \mid i, k, s \mid i, a(N_i) \mid a(N_i)))</td>
</tr>
<tr>
<td>set_edge_bottom_deviation</td>
<td>((s \mid j, s \mid j, a^S(N_p) \mid j, a^S(N_p)))</td>
</tr>
<tr>
<td>set_elevation</td>
<td>((s \mid i, s \mid a(N_p)))</td>
</tr>
<tr>
<td>set_elevation.bc</td>
<td>((s \mid s \mid a(N_p)))</td>
</tr>
<tr>
<td>set_flux_limiter</td>
<td>((s \mid m, s \mid m, a(N_p) \mid b(N_p, N_q)))</td>
</tr>
<tr>
<td>set_grid_file</td>
<td>((s \mid m, s \mid m, a(N_p) \mid b(N_p, N_q)))</td>
</tr>
<tr>
<td>set_horizontal_velocity</td>
<td>((s_x, s_y \mid j, k, s_x, s_y \mid j, a_i(N_x), s_y \mid j, s_x, a_j(N_x) \mid a_i(N_x), a_j(N_x) \mid \ldots)</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>set_inflow.bc</td>
<td>((s \mid j, s \mid j, f_j, k, s))</td>
</tr>
<tr>
<td>set_inflow.cc</td>
<td>((s \mid m, s \mid m, j, f_j, k, s))</td>
</tr>
</tbody>
</table>

Table 3: Available interfaces for set-routines together with allowed arguments (set_artificial_porosity – set_inflow.cc). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_input_file</td>
<td>(st[len=80])</td>
</tr>
<tr>
<td>set_new_bottom_flux</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_flux</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_k</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_kh</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_kv</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_settling_velocity</td>
<td>(m)</td>
</tr>
<tr>
<td>set_new_surface_flux</td>
<td>(m)</td>
</tr>
<tr>
<td>set_point_sources_concentration</td>
<td>(s</td>
</tr>
<tr>
<td>set_point_sources_discharge</td>
<td>(s</td>
</tr>
<tr>
<td>set_pressure</td>
<td>(s</td>
</tr>
<tr>
<td>set_pressure_bc</td>
<td>(s</td>
</tr>
<tr>
<td>set_printout_unit</td>
<td>(no)</td>
</tr>
<tr>
<td>set_radiation_time</td>
<td>(s</td>
</tr>
<tr>
<td>set_rain</td>
<td>(s</td>
</tr>
<tr>
<td>set_sediment</td>
<td>(s</td>
</tr>
<tr>
<td>set_settling_velocity</td>
<td>(s</td>
</tr>
<tr>
<td>set_slope</td>
<td>(s_x, s_y)</td>
</tr>
<tr>
<td>set_source_file</td>
<td>(st[len=80])</td>
</tr>
<tr>
<td>set_subedge_depth</td>
<td>(s</td>
</tr>
<tr>
<td>set_subpolygon_depth</td>
<td>(s</td>
</tr>
<tr>
<td>set_surface_alpha</td>
<td>(s</td>
</tr>
</tbody>
</table>

Table 4: Available interfaces for set-routines together with allowed arguments (set_input_file – set_surface_alpha). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments</th>
</tr>
</thead>
<tbody>
<tr>
<td>set_surface_beta</td>
<td>((s</td>
</tr>
<tr>
<td>set_surface_concentration</td>
<td>((s</td>
</tr>
<tr>
<td>set_time</td>
<td>((s))</td>
</tr>
<tr>
<td>set_turbulent_h_diffusivity</td>
<td>((s</td>
</tr>
<tr>
<td>set_turbulent_h_viscosity</td>
<td>((s</td>
</tr>
<tr>
<td>set_turbulent_v_diffusivity</td>
<td>((s</td>
</tr>
<tr>
<td>set_turbulent_v_viscosity</td>
<td>((s</td>
</tr>
<tr>
<td>set_velocity</td>
<td>((s</td>
</tr>
<tr>
<td>set_wet_and_dry</td>
<td>((s</td>
</tr>
<tr>
<td>set_wind_friction</td>
<td>((s</td>
</tr>
<tr>
<td>share_bottom_flux</td>
<td>((s_x,s_y</td>
</tr>
<tr>
<td>share_flux</td>
<td>((m_1,m_2))</td>
</tr>
<tr>
<td>share_k</td>
<td>((m_1,m_2))</td>
</tr>
<tr>
<td>share_kh</td>
<td>((m_1,m_2))</td>
</tr>
<tr>
<td>share_kv</td>
<td>((m_1,m_2))</td>
</tr>
<tr>
<td>share_settling_velocity</td>
<td>((m_1,m_2))</td>
</tr>
<tr>
<td>share_surface_flux</td>
<td>((m_1,m_2))</td>
</tr>
</tbody>
</table>

Table 5: Available interfaces for set-routines together with allowed arguments (set_surface_beta – share_surface_flux). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_adjacent_polygon</td>
<td>arg $(i, l)$ (i) ()</td>
</tr>
<tr>
<td></td>
<td>resINT $i$ $a(4)$ $b(4, N_p)$</td>
</tr>
<tr>
<td>get_angle</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$</td>
</tr>
<tr>
<td>get_area</td>
<td>arg $(i)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $s$</td>
</tr>
<tr>
<td>get_artificial_porosity</td>
<td>arg $(j)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_s)$</td>
</tr>
<tr>
<td>get_atmospheric_pressure</td>
<td>arg $(i)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_p)$</td>
</tr>
<tr>
<td>get_avr_bottom_deviation</td>
<td>arg $(i)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_p)$</td>
</tr>
<tr>
<td>get_avr_depth</td>
<td>arg $(i)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_p)$</td>
</tr>
<tr>
<td>get_avr_edge_bottom_deviation</td>
<td>arg $(j)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_s)$</td>
</tr>
<tr>
<td>get_avr_edge_depth</td>
<td>arg $(j)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_s)$</td>
</tr>
<tr>
<td>get_avr_edge_total_depth</td>
<td>arg $(j)$ ()</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_s)$</td>
</tr>
</tbody>
</table>

Table 6: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_adjacent_polygon – get_avr_edge_total_depth). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
</table>
| get_avr_face_height             | arg  
|                                 | (j,k) 
|                                 | (j)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Nj)  
|                                 | a(Ij) |
| get_avr_prism_height            | arg  
|                                 | (i,k) 
|                                 | (i)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Ni)  
|                                 | a(Ii)|
| get_avr_total_depth             | arg  
|                                 | (i)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Np)  |
| get_bottom_alpha                | arg  
|                                 | (m,i) 
|                                 | (m)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Np)  
|                                 | b(Np,Nb)|
| get_bottom_beta                 | arg  
|                                 | (m,i) 
|                                 | (m)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Np)  
|                                 | b(Np,Nb)|
| get_bottom_concentration        | arg  
|                                 | (m,i) 
|                                 | (m)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(Np)  
|                                 | b(Np,Nb)|
| get_bottom_deviation            | arg  
|                                 | (i,NjG) 
|                                 | (j)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(njG)  
|                                 | a(NjG)|
| get_bottom_face                 | arg  
|                                 | (j)  
|                                 | ()   
|                                 | resINT 
|                                 | k    
|                                 | a(Nj)  |
| get_bottom_flux                 | arg  
|                                 | (m,i) 
|                                 | (m)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | s    
|                                 | a(Nc)  |
| get_bottom_friction             | arg  
|                                 | (j,NjG) 
|                                 | (j)  
|                                 | ()   
|                                 | resDP 
|                                 | s    
|                                 | a(njG)  
|                                 | a(NjG)|

Table 7: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_avr_face_height – get_bottom_friction). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
The Federal Waterways Engineering and Research Institute (BAW)
Mathematical Model UnTRIM²
User Interface Description – Version May 2010 (1.1)

<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_bottom_prism</td>
<td>arg (i) (k) a(N_p)</td>
</tr>
<tr>
<td>get_bottom_stress</td>
<td>arg (j, j^SG) (j) a(N_s^SG)</td>
</tr>
<tr>
<td>get_bottom_subface</td>
<td>arg (j, j^SG) (j) a(N_s^SG)</td>
</tr>
<tr>
<td>get_bottom_subprism</td>
<td>arg (j, j^SG) (j) a(N_s^SG)</td>
</tr>
<tr>
<td>get_celerity</td>
<td>arg (j) (j) a(N_s)</td>
</tr>
<tr>
<td>get_center_distance</td>
<td>arg (j) (j) a(N_s)</td>
</tr>
<tr>
<td>get_chezy</td>
<td>arg () (j) a(N_s)</td>
</tr>
<tr>
<td>get_concentration</td>
<td>arg (m, i, k) (m, i) (m) (m) (j) a(I_i) b(I_i, N_c)</td>
</tr>
<tr>
<td>get_courant</td>
<td>arg (j) (j) a(N_s)</td>
</tr>
<tr>
<td>get_density</td>
<td>arg (i, k) (i) a(N_s) a(I_i)</td>
</tr>
</tbody>
</table>

Table 8: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get\_bottom\_prism – get\_density).
For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_dry_area</td>
<td>arg ( (i, k) ) ( (i) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( s ) ( s )</td>
</tr>
<tr>
<td>get_dry_length</td>
<td>arg ( (j, k) ) ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( s ) ( a(N_{sf}) )</td>
</tr>
<tr>
<td>get_dt_min</td>
<td>arg ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s )</td>
</tr>
<tr>
<td>get_dx_min</td>
<td>arg ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s )</td>
</tr>
<tr>
<td>get_dz_min</td>
<td>arg ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s )</td>
</tr>
<tr>
<td>get_edge_begin</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resINT ( s ) ( a(N_s) )</td>
</tr>
<tr>
<td>get_edge_bottom_deviation</td>
<td>arg ( (j, j^{SG}) ) ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(n_{bj}^{SG}) ) ( a(N_{SG}) )</td>
</tr>
<tr>
<td>get_edge_center</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( a(2) ) ( b(2, N_s) )</td>
</tr>
<tr>
<td>get_edge_end</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resINT ( s ) ( a(N_s) )</td>
</tr>
<tr>
<td>get_edge_length</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(N_s) )</td>
</tr>
</tbody>
</table>

Table 9: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_dry_area – get_edge_length). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
### Table 10: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_edge_wet_area – get_horizontal_velocity). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.

<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_edge_wet_area</td>
<td>arg $(j,k)$ $(j)$ $(i)$ $(i)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N)$ $a(J)$</td>
</tr>
<tr>
<td>get_elevation</td>
<td>arg $(i)$ $(i)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_p)$</td>
</tr>
<tr>
<td>get_elevation_tolerance</td>
<td>arg $()$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$</td>
</tr>
<tr>
<td>get_face_index</td>
<td>arg $(j,k)$ $(i,k)$</td>
</tr>
<tr>
<td></td>
<td>resINT $j^3D$</td>
</tr>
<tr>
<td>get_flux_limiter</td>
<td>arg $(m)$ $(m)$</td>
</tr>
<tr>
<td></td>
<td>resCH80 $st$ $st(N)$ $st(N)$</td>
</tr>
<tr>
<td>get_froude</td>
<td>arg $(j)$ $(j)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N)$ $a(N_f)$</td>
</tr>
<tr>
<td>get_gravity</td>
<td>arg $(N)$ $(N)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N_f)$ $a(N_f)$</td>
</tr>
<tr>
<td>get_grid_file</td>
<td>arg $()$</td>
</tr>
<tr>
<td></td>
<td>resCH80 $st$</td>
</tr>
<tr>
<td>get_horizontal_diffusivity</td>
<td>arg $(N)$ $(N)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$ $a(N)$ $a(N_f)$</td>
</tr>
<tr>
<td>get_horizontal_velocity</td>
<td>arg $(x,y)$ $(x,y,i,k)$ $(x,y,i)$</td>
</tr>
<tr>
<td></td>
<td>resDP $b(2,N)$ $a(2)$ $b(2,N)$</td>
</tr>
</tbody>
</table>
### Available Interfaces for Get-Functions

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Allowed Sets of Arguments and Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_horizontal_velocity2</td>
<td>arg (x, y)</td>
</tr>
<tr>
<td></td>
<td>resDP b(2, Nz)</td>
</tr>
<tr>
<td>get_horizontal_velocity_x</td>
<td>arg (x, y)</td>
</tr>
<tr>
<td></td>
<td>resDP a(Nz)</td>
</tr>
<tr>
<td>get_horizontal_velocity_y</td>
<td>arg (x, y)</td>
</tr>
<tr>
<td></td>
<td>resDP a(Nz)</td>
</tr>
<tr>
<td>get_horizontal_viscosity</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resDP s</td>
</tr>
<tr>
<td>get_input_file</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resCH80 st</td>
</tr>
<tr>
<td>get_layer</td>
<td>arg (z)</td>
</tr>
<tr>
<td></td>
<td>resINT k</td>
</tr>
<tr>
<td>get_layer_interface</td>
<td>arg (k)</td>
</tr>
<tr>
<td></td>
<td>resDP s a(Nz - 1)</td>
</tr>
<tr>
<td>get_left_polygon</td>
<td>arg (j)</td>
</tr>
<tr>
<td></td>
<td>resINT i a(Ni)</td>
</tr>
<tr>
<td>get_length</td>
<td>arg (j)</td>
</tr>
<tr>
<td></td>
<td>resDP s a(Nji)</td>
</tr>
<tr>
<td>get_location</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resCH80 st</td>
</tr>
</tbody>
</table>

Table 11: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_horizontal_velocity2 – get_location). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>get_mass</strong></td>
<td>arg ((m,i,k)(m,i)(m)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_c)) (a(I_c)) (b(I_c,N_c))</td>
</tr>
<tr>
<td><strong>get_max_courant</strong></td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) ()</td>
</tr>
<tr>
<td><strong>get_max_depth</strong></td>
<td>arg ((i)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{p})) ()</td>
</tr>
<tr>
<td><strong>get_max_edge_depth</strong></td>
<td>arg ((j)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{s_{y}})) ()</td>
</tr>
<tr>
<td><strong>get_max_edge_total_depth</strong></td>
<td>arg ((j)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{s_y})) ()</td>
</tr>
<tr>
<td><strong>get_max_face_height</strong></td>
<td>arg ((j,k)(j)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{c})) (a(J_{c}))</td>
</tr>
<tr>
<td><strong>get_max_froude</strong></td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) ()</td>
</tr>
<tr>
<td><strong>get_max_prism_height</strong></td>
<td>arg ((i,k)(i)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{c})) (a(I_{c}))</td>
</tr>
<tr>
<td><strong>get_max_total_depth</strong></td>
<td>arg ((i)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (a(N_{p})) ()</td>
</tr>
<tr>
<td><strong>get_max_velocity_courant</strong></td>
<td>arg ((j)) ()</td>
</tr>
<tr>
<td></td>
<td>resDP (s) (s)</td>
</tr>
</tbody>
</table>

Table 12: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_mass–get_max_velocity_courant). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>get_nof_bottom_fluxes</code></td>
<td><code>arg</code> (()) <code>resINT</code> $N_b$</td>
</tr>
<tr>
<td><code>get_nof_boundary_polygons</code></td>
<td><code>arg</code> (()) <code>resINT</code> $N^s_p$</td>
</tr>
<tr>
<td><code>get_nof_diffusivities</code></td>
<td><code>arg</code> (()) <code>resINT</code> $a(2)$</td>
</tr>
<tr>
<td><code>get_nof_edges</code></td>
<td><code>arg</code> (i) (()) <code>resINT</code> $S_i$ $N_s$</td>
</tr>
<tr>
<td><code>get_nof_faces</code></td>
<td><code>arg</code> (j) (()) <code>resINT</code> $J_j$ $J_s$</td>
</tr>
<tr>
<td><code>get_nof_fluxes</code></td>
<td><code>arg</code> (()) <code>resINT</code> $a(2)$</td>
</tr>
<tr>
<td><code>get_noh_diffusivities</code></td>
<td><code>arg</code> (()) <code>resINT</code> $N^h$</td>
</tr>
<tr>
<td><code>get_nof_inflow_edges</code></td>
<td><code>arg</code> (()) <code>resINT</code> $n_{sf}$</td>
</tr>
<tr>
<td><code>get_nof_inner_iterations</code></td>
<td><code>arg</code> (n) (()) <code>resINT</code> $a(1 \mid 2)$ $a(1 \mid 2)$</td>
</tr>
<tr>
<td><code>get_nof_internal_edges</code></td>
<td><code>arg</code> (()) <code>resINT</code> $N_i$</td>
</tr>
</tbody>
</table>

Table 13: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (`get_nof_bottom_fluxes`–`get_nof_internal_edges`). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_nof_iterations</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( a(2 \mid 4) )</td>
</tr>
<tr>
<td>get_nof_layers</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( N_z )</td>
</tr>
<tr>
<td>get_nof_max_edges</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( p_{\text{max}} )</td>
</tr>
<tr>
<td>get_nof_max_iterations</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( a(2) )</td>
</tr>
<tr>
<td>get_nof_outer_iterations</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( a(1 \mid 2) )</td>
</tr>
<tr>
<td>get_nof_point_sources</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( N_d )</td>
</tr>
<tr>
<td>get_nof_polygons</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( N_p )</td>
</tr>
<tr>
<td>get_nof_prisms</td>
<td>arg (i) ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( I_i, I_i )</td>
</tr>
<tr>
<td>get_nof_settling_velocities</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( N_{\text{ws}} )</td>
</tr>
<tr>
<td>get_nof_species</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>res_INT ( N_c )</td>
</tr>
</tbody>
</table>

Table 14: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get\_nof\_iterations – get\_nof\_species). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
### Table 15: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get\_nof\_subedges – get\_nof\_wet\_subedges).

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Arguments</th>
<th>Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_nof_subedges</td>
<td>arg ((j))</td>
<td>res(N^S_j)</td>
</tr>
<tr>
<td>get_nof_subpolygons</td>
<td>arg ((i))</td>
<td>res(N^P_i)</td>
</tr>
<tr>
<td>get_nof_substeps</td>
<td>arg ((m))</td>
<td>res(s) (a(N_c))</td>
</tr>
<tr>
<td>get_nof_surface_fluxes</td>
<td>arg ())</td>
<td>res(N_f)</td>
</tr>
<tr>
<td>get_nof_diffusivities</td>
<td>arg ())</td>
<td>res(N^v_k)</td>
</tr>
<tr>
<td>get_nof_vertices</td>
<td>arg ((i))</td>
<td>res(S_i) (N_v)</td>
</tr>
<tr>
<td>get_nof_wet_faces</td>
<td>arg ((j))</td>
<td>res(s) (s)</td>
</tr>
<tr>
<td>get_nof_wet_polygons</td>
<td>arg ())</td>
<td>res(s)</td>
</tr>
<tr>
<td>get_nof_wet_prisms</td>
<td>arg ((i))</td>
<td>res(s) (s)</td>
</tr>
<tr>
<td>get_nof_wet_subedges</td>
<td>arg ((j))</td>
<td>res(s) (s)</td>
</tr>
</tbody>
</table>

For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_nof_wet_subpolygons</td>
<td>arg (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resINT s s</td>
</tr>
<tr>
<td>get_polygon</td>
<td>arg (x,y)</td>
</tr>
<tr>
<td></td>
<td>resINT i</td>
</tr>
<tr>
<td>get_polygon_area</td>
<td>arg (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resDP s a(N_p)</td>
</tr>
<tr>
<td>get_polygon_center</td>
<td>arg (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resDP a(2) b(2,N_p)</td>
</tr>
<tr>
<td>get_polygon_edge</td>
<td>arg (i,l) (i) (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resINT j a(4) b(4,N_p)</td>
</tr>
<tr>
<td>get_polygon_vertex</td>
<td>arg (i,l) (i) (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resINT n a(4) b(4,N_p)</td>
</tr>
<tr>
<td>get_pressure</td>
<td>arg (i,k) (i) (i) ( )</td>
</tr>
<tr>
<td></td>
<td>resDP s a(N_z) a(I_z)</td>
</tr>
<tr>
<td>get_pressure_tolerance</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>resDP s</td>
</tr>
<tr>
<td>get_printout_unit</td>
<td>arg ( )</td>
</tr>
<tr>
<td></td>
<td>resINT no</td>
</tr>
<tr>
<td>get_prism_index</td>
<td>arg (i,k)</td>
</tr>
<tr>
<td></td>
<td>resINT i^{3D}</td>
</tr>
</tbody>
</table>

Table 16: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get\_nof\_wet\_subpolygons – get\_prism\_index). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_radiation_time</td>
<td>arg $(i)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$, $a(N_p)$</td>
</tr>
<tr>
<td>get_right_polygon</td>
<td>arg $(j)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resINT $i$, $a(N_i)$</td>
</tr>
<tr>
<td>get_sediment</td>
<td>arg $(m)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$, $a(N_c)$</td>
</tr>
<tr>
<td>get_settling_velocity</td>
<td>arg $(m,i,k)$, $(m,i,z)$, $(m,i)$, $(m)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$, $s$, $a(0:N_z)$, $a(I_z)$, $b(I_z,N_{w'})$</td>
</tr>
<tr>
<td>get_slope</td>
<td>arg $(1</td>
</tr>
<tr>
<td></td>
<td>resDP $s_x$, $s_y$, $a(2)$</td>
</tr>
<tr>
<td>get_source_concentration</td>
<td>arg $(n,m)$, $(m)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$, $a(N_d)$, $b(N_d,N_c)$</td>
</tr>
<tr>
<td>get_source_discharge</td>
<td>arg $(n)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resDP $s$, $a(N_d)$</td>
</tr>
<tr>
<td>get_source_file</td>
<td>arg $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resCH80 $st$</td>
</tr>
<tr>
<td>get_source_layer</td>
<td>arg $(n)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resINT $s$, $a(N_d)$</td>
</tr>
<tr>
<td>get_source_polygon</td>
<td>arg $(n)$, $(\cdot)$</td>
</tr>
<tr>
<td></td>
<td>resINT $i$, $a(N_d)$</td>
</tr>
</tbody>
</table>

Table 17: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_radiation_time–get_source_polygon). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
### Interface Name

<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Allowed Sets of Arguments and Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_subedge_depth</td>
<td>arg ((j, j_{SG}) ) resDP (s a(n_{e,j}^{SG}) a(N_{SG}))</td>
</tr>
<tr>
<td>get_subedge_length</td>
<td>arg ((j, j_{SG}) ) resDP (s a(n_{e,j}^{SG}) a(N_{SG}))</td>
</tr>
<tr>
<td>get_subedge_total_depth</td>
<td>arg ((j, j_{SG}) ) resDP (s a(n_{e,j}^{SG}) a(N_{SG}))</td>
</tr>
<tr>
<td>get_subpolygon_area</td>
<td>arg ((i, i_{SG}) ) resDP (s a(n_{p,i}^{SG}) a(N_{p,SG}))</td>
</tr>
<tr>
<td>get_subpolygon_depth</td>
<td>arg ((i, i_{SG}) ) resDP (s a(n_{p,i}^{SG}) a(N_{SG}))</td>
</tr>
<tr>
<td>get_subpolygon_total_depth</td>
<td>arg ((i, i_{SG}) ) resDP (s a(n_{p,i}^{SG}) a(N_{SG}))</td>
</tr>
<tr>
<td>get_surface_alpha</td>
<td>arg ((m, i) ) resDP (s a(N_{p}) a(N_{p,N}))</td>
</tr>
<tr>
<td>get_surface_beta</td>
<td>arg ((m, i) ) resDP (s a(N_{p}) a(N_{p,N}))</td>
</tr>
<tr>
<td>get_surface_concentration</td>
<td>arg ((m, i) ) resDP (s a(N_{p}) a(N_{p,N}))</td>
</tr>
<tr>
<td>get_surface_flux</td>
<td>arg ((m, i) ) resDP (s a(N_{c}))</td>
</tr>
</tbody>
</table>

Table 18: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_subedge_depth – get_surface_flux). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>Interface Name</th>
<th>Allowed Sets of Arguments and Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_theta</td>
<td>arg (θ) resDP θ</td>
</tr>
<tr>
<td>get_time</td>
<td>arg () resDP t</td>
</tr>
<tr>
<td>get_time_step</td>
<td>arg () resDP Δt</td>
</tr>
<tr>
<td>get_top_face</td>
<td>arg (j) resINT k a(Nsf)</td>
</tr>
<tr>
<td>get_top_prism</td>
<td>arg (i) resINT k a(Np)</td>
</tr>
<tr>
<td>get_total_mass</td>
<td>arg (m,i,k) (m,i) (m) () resDP s s s a(Nc)</td>
</tr>
<tr>
<td>get_total_volume</td>
<td>arg (i,k) (i) () resDP s s s</td>
</tr>
<tr>
<td>get_turbulent_h_diffusivity</td>
<td>arg (m,j,k) (m,j) (m) () resDP s a(Nc) a(Ji) b(Ji,Np)</td>
</tr>
<tr>
<td>get_turbulent_h_viscosity</td>
<td>arg (j,k) (j) () resDP s a(Nc) a(Ji)</td>
</tr>
<tr>
<td>get_turbulent_v_diffusivity</td>
<td>arg (m,i,k) (m,i) (m) () resDP s a(Nc) a(I3) b(I3,Np)</td>
</tr>
</tbody>
</table>

Table 19: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_theta – get_turbulent_v_diffusivity). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_turbulent_viscosity</td>
<td>arg ( (i,k) ) ( (i) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(N_z) ) ( a(I_z) )</td>
</tr>
<tr>
<td>get_unerodible_bottom_face</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resINT ( k ) ( a(N_{sy}) )</td>
</tr>
<tr>
<td>get_unerodible_bottom_prism</td>
<td>arg ( (i) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resINT ( k ) ( a(N_p) )</td>
</tr>
<tr>
<td>get_velocity</td>
<td>arg ( (j,k) ) ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(N_z) ) ( a(I_z) )</td>
</tr>
<tr>
<td>get_velocity_courant</td>
<td>arg ( (j,k) ) ( (j) )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(N_z) )</td>
</tr>
<tr>
<td>get_vertex_coordinates</td>
<td>arg ( (n) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( a(2) ) ( b(2,N_z) )</td>
</tr>
<tr>
<td>get_vertical_diffusivity</td>
<td>arg ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s )</td>
</tr>
<tr>
<td>get_vertical_velocity</td>
<td>arg ( (i,k) ) ( (i,z) ) ( (i) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( s ) ( a(0:N_z) ) ( a(I_z) )</td>
</tr>
<tr>
<td>get_vertical viscosity</td>
<td>arg ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s )</td>
</tr>
<tr>
<td>get_vertically_avr_velocity</td>
<td>arg ( (j) ) ( () )</td>
</tr>
<tr>
<td></td>
<td>resDP ( s ) ( a(N_{sy}) )</td>
</tr>
</tbody>
</table>

Table 20: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_turbulent_viscosity – get_vertically_avr_velocity). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_volume</td>
<td>arg ( i,k ) ( i ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{DP} ) ( s ) ( a(N_c) ) ( a(I_i) )</td>
</tr>
<tr>
<td>get_wet_area</td>
<td>arg ( i,k ) ( i ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{DP} ) ( s ) ( s ) ( s )</td>
</tr>
<tr>
<td>get_wet_length</td>
<td>arg ( j,k ) ( j ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{DP} ) ( s ) ( s ) ( a(N_r) )</td>
</tr>
<tr>
<td>get_which_bottom_flux</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( s ) ( a(N_c) )</td>
</tr>
<tr>
<td>get_which_diffusivity</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( a(2) ) ( b(N_r,2) )</td>
</tr>
<tr>
<td>get_which_flux</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( a(2) ) ( b(N_r,2) )</td>
</tr>
<tr>
<td>get_which_h_diffusivity</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( s ) ( a(N_c) )</td>
</tr>
<tr>
<td>get_which_settling_velocity</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( s ) ( a(N_c) )</td>
</tr>
<tr>
<td>get_which_surface_flux</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( s ) ( a(N_c) )</td>
</tr>
<tr>
<td>get_which_v_diffusivity</td>
<td>arg ( m ) ()</td>
</tr>
<tr>
<td></td>
<td>res( \text{INT} ) ( s ) ( a(N_c) )</td>
</tr>
</tbody>
</table>

Table 21: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get\_volume – get\_which\_v\_diffusivity). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
### interface name  
allowed sets of arguments and results

<table>
<thead>
<tr>
<th>interface name</th>
<th>arg</th>
<th>resDP</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_wind_friction</td>
<td>(j)</td>
<td>(a(N_i))</td>
</tr>
<tr>
<td>get_wind_stress</td>
<td>(j)</td>
<td>(a(N_i))</td>
</tr>
<tr>
<td>get_wind_velocity</td>
<td>(j)</td>
<td>(a(N_i))</td>
</tr>
<tr>
<td>get_wind_velocity_x</td>
<td>(j)</td>
<td>(a(2), b(2,N_i))</td>
</tr>
<tr>
<td>get_wind_velocity_y</td>
<td>(j)</td>
<td>(a(N_i))</td>
</tr>
<tr>
<td>get_x_slope</td>
<td>()</td>
<td>(a(N_v))</td>
</tr>
<tr>
<td>get_x_vertex_coordinate</td>
<td>(n)</td>
<td>(a(N_v))</td>
</tr>
<tr>
<td>get_y_slope</td>
<td>()</td>
<td>(a(N_v))</td>
</tr>
<tr>
<td>get_y_vertex_coordinate</td>
<td>(n)</td>
<td>(a(N_v))</td>
</tr>
<tr>
<td>is_dry_edge</td>
<td>(j)</td>
<td>(resLOG)</td>
</tr>
</tbody>
</table>

Table 22: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (get_wind_friction – is_dry_edge). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
<table>
<thead>
<tr>
<th>interface name</th>
<th>allowed sets of arguments and results</th>
</tr>
</thead>
<tbody>
<tr>
<td>is_dry_polygon</td>
<td>arg (i)</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
<tr>
<td>is_wet_edge</td>
<td>arg (j)</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
<tr>
<td>is_wet_polygon</td>
<td>arg (i)</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
<tr>
<td>sediment</td>
<td>arg (m)</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
<tr>
<td>with_hydrodynamic_pressure</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
<tr>
<td>with_scalar_transport</td>
<td>arg ()</td>
</tr>
<tr>
<td></td>
<td>resLOG s</td>
</tr>
</tbody>
</table>

Table 23: Available interfaces for get-functions together with allowed arguments (arg) and returned results (res) (is_dry_polygon – with_scalar_transport). For details about the nomenclature please refer to Table 1 on page 1 and Table 2 on page 2.
D Example input data files

D.1 Grid file

D.1.1 Grid sorting

Polygons as well as edges/sides of a grid stored in file "untrim.grd" must be sorted in an appropriate way. This is required for reasons of efficiency and correctness. The following orderings must be fulfilled (tasks for the grid generator):

- vertices:
  1. for each polygon the vertices must be given in counter-clockwise orientation (program will stop if this is not the case);
  2. no sorting is required for the global vertex numbers.

- polygons:
  1. for reasons of efficiency and accuracy a red-black-ordering of the polygons is recommended:
     (a) the red polygons must become the first \([1,N_{p_r}]\) ones out of the \(N_p\) polygons;
     (b) the black polygons must be numbered \([N_{p_r}+1,N_p]\);
     Notice: If no red-black-ordering is used, \(N_{p_r} = N_p\) must hold true.
  2. if there are \(N^*_p\) polygons along the open boundary where the water level shall be prescribed, they must be assigned the lowermost numbers \([1,N^*_p]\).

- sides/edges:
  1. for reasons of efficiency all internal sides (sharing two polygons) must be assigned the lowermost numbers \([1,N_{s_i}]\);
  2. all (closed or open) boundary edges must be moved to the tail \([N_{s_i}+1,N_{s_f}]\);
  3. if some open boundary edges are sides with prescribed flow, these edges must be brought to the head of the tail \([N_{s_i}+1,N_{s_f}]\);
  4. all open boundary edges \([N_{s_i}+1,N_{s_f}]\) are not allowed to have right neighbour polygons.
     Notice: If a Dirichlet boundary condition is not used at all, \(N_{s_f} = N_{s_i}\) must hold true.

- sub polygon scale information:
  1. sub polygon scale bathymetry can be described by means of subpolygons of arbitrary shape;
  2. each polygon can contain an arbitrary number of subpolygons \(n_{p,i}^{SG} \geq 1\);
  3. for each subpolygon \(i^{SG}\) within polygon \(i\) size (area) \(a_{i,j,p,i}^{SG}\) as well as depth \(h_{i,j,p,i}^{SG}\) must be given;
4. the subpolygons may not fully cover their hosting polygon, i.e. \( \sum_{SG}^{nSG} a_{iSG} \leq A(i) \) must hold, where \( A(i) \) corresponds to the area of the polygon.

- sub edge scale information:
  1. sub edge scale bathymetry can be similarly described by means of subedges;
  2. each edge can consist out of an arbitrary number of subedges \( n_{SG}^{j} \geq 1 \);
  3. for each subedge \( j^{SG} \) along edge \( j \) length \( l_{j}^{SG} \) as well as depth \( h_{j}^{SG} \) must be given;
  4. the subedges may not fully cover their hosting edge, i.e. \( \sum_{SG}^{nSG} l_{j}^{SG} \leq l(j) \) must hold, where \( l(j) \) is the length of the edge.

Please notice: subgrid scale information can be used to describe bathymetry with subgrid scale resolution. The computational core of UnTRIM\(^2\) takes this information into account in integral form. Therefore the precise location of the subpolygons as well as the subedges is unimportant for the core. Any distribution of subpolygons and subedges which results in the same depth distribution will give the same computational result.

D.1.2 Short description

The grid file is sub-divided into the following sections:

1. Fortran NAMELIST-data section GRD_2008:
   (a) \( ne \): number of polygons \( N_p \);
   (b) \( ns \): number of sides \( N_s \);
   (c) \( nv \): number of vertices \( N_v \);
   (d) \( tne \): number of subpolygons \( N_{SG}^{p} \);
   (e) \( tns \): number of subedges \( N_{SG}^{s} \);
   (f) \( nbc \): number of boundary polygons \( N_{p}^{*} \);
   (g) \( nr \): number of red polygons \( N_{pr} \) (red-black-sorting);
   (h) \( nsi \): number of internal sides \( N_{si} \);
   (i) \( nsf \): number of last side with prescribed flow \( N_{sf} \);
   (j) \( angle \): (mean) geographic latitude \( \Phi \);
   (k) \( location \): text string to describe the model domain.

2. list of cartesian coordinates for all vertices of the grid; each line of input contains the following data:
   (a) \( n \): vertex number (informational use only);
   (b) \( x \): \( x \)-coordinate of vertex \( n \);
3. list of data related to the $N_p$ polygons in the grid; each input line contains the following data for one polygon:

(a) $i$: polygon number (informational use only);
(b) $ks$: number of edges/vertices $S_i$ ($3 \leq S_i \leq 4$) of polygon $i$;
(c) $xc$: $x$-coordinate for the $i$-th polygon center;
(d) $yc$: $y$-coordinate for the $i$-th polygon center;
(e) nen(1:$ks$): vertex indices for polygon $i$;
(f) is(1:$ks$): edge indices for polygon $i$;

Please notice that vertices as well as sides must be prescribed consecutively in counter-clockwise orientation for each polygon.

4. list of data related to the $N_s$ edges/sides in the grid; each input line contains the following data for one side/edge:

(a) $j$: edge number (informational use only);
(b) $jb$: index for the first vertex of edge $j$;
(c) $jt$: index for the second vertex of edge $j$;
(d) je(2): indices for the left ($i(j,1)$) as well as the right ($i(j,2)$) polygon adjacent to the actual edge; zero must be prescribed if the respective neighbour polygon is not present (e. g. along boundaries), but only right polygons are allowed to be not present.

5. list of data related to the subgrid scale bathymetry information for polygons; for each polygon $(i = 1,N_p)$ the following data have to be prescribed:

(a) $i,nis$: polygon number $i$ (informational use only) and number of subpolygons $n_{SG,i}$ for polygon $i$;
(b) $aaai(1:nis)$: supolygon area $a_{SG,iSG}$ for all subpolygons ($iSG = 1,n_{SG,i}$) of polygon $i$;
(c) $hwi(1:nis)$: subpolygon depth $h_{SG,iSG}$ for all subpolygons ($iSG = 1,n_{SG,i}$) of polygon $i$.

6. list of data related to the subgrid scale bathymetry information for edges with flow; for each edge $(j = 1,N_s)$ the following data have to be prescribed:

(a) $j,njs$: edge number $j$ (informational use only) and number of subedges $n_{SG,j}$ for edge $j$;
(b) $dyj(1:njs)$: subedge length $l_{SG,jSG}$ for all subedges ($jSG = 1,n_{SG,j}$) of edge $j$;
(c) $huj(1:njs)$: subedge depth $h_{SG,jSG}$ for all subedges ($jSG = 1,n_{SG,j}$) of edge $j$.

Please notice: no subedge data have to be prescribed for all edges along solid boundaries.
D.1.3 Example file

The following lines show an (abbreviated) example for the grid data file "untrim.grd". The line-numbers l???? indicated are not part of the grid input file.

```
l001 &GRD_2008
l002 NV =8923,
l003 NE =8668,
l004 NR =4374,
l005 NS =17592,
l006 NSI =17040,
l007 NSF =17120,
l008 NBC =80,
l009 TNE =30726,
l010 TNS =35176,
l011 ANGLE =0.0000,
l012 LOCATION='vonKarman - quads with subgrid details'
l013/
lxxx vertices
l101 1 -14.0000 -10.0000
l102 2 -14.0000 -9.7500
lxxx polygons
l201 1 4 14.3750 4.8750 8843 1804 1722 8844 162 17121 160 161
l202 2 4 14.3750 5.1250 8844 1722 1640 8845 160 17122 322 323
lxxx edges
l301 1 83 84 8668 161
l302 2 84 2 8668 82
l...
l303 17592 8784 8866 22 0
lxxx subpolygons
l401 1 4
l402 0.0156 0.0156 0.0156 0.0156
l403 3.3400 3.3700 3.4000 3.4300
l404 2 4
l405 0.0156 0.0156 0.0156 0.0156
l406 3.2300 3.2600 3.2900 3.3200
lxxx subedges
l501 1 4
l502 0.0625 0.0625 0.0625 0.0625
l503 1.1100 1.1400 1.1700 1.2000
l504 2 1
l505 0.2500
l506 1.2000
```

A few comments on the short example shown above:

**l001 – l011**: contents of namelist GRD_2008;

**l003 – l004**: red black sorting of grid, 4374 red polygons;
l006 – l007: there are 80 edges with prescribed flow;
l008 – l008: there are 80 polygons with prescribed water level;
l009 – l010: subgrid bathymetry details for polygons and sides available;
l011 – l011: use \texttt{set\_constant\_coriolis(0.0)} in e.g. \texttt{user\_set\_initial\_conditions} — see Subsection B.6.2 on page 114 — in case no or a constant Coriolis force shall be applied, otherwise the beta-plane approximation will be used;
l012 – l012: title of project;
l013 – l013: terminator “/” for the namelist section;
l101 – l102: x- and y-coordinate of the first two vertices;
l201 – l202: polygon-oriented data for the first two polygon (two quads);
l301 – l302: edge-oriented data for the first two edges;
l303: edge-oriented data for the last edge, which is an edge at the (solid) boundary (missing right neighbour polygon);
l401 & l404: polygon numbers and number of subpolygons for the first two polygons;
l402 & l405: subpolygon area data for the first two polygons;
l403 & l406: subpolygon depth data for the first two polygons;
l501 & l504: edge numbers and number of subedges for the first two edges, where the second edge is not subdivided into subedges.
l502 & l505: subedge length data for the first two edges;
l503 & l506: subedge depth data for the first two edges.

D.2 Input data file

D.2.1 Short description

The input data file is sub-divided into the following sections:

1. Fortran NAMELIST-data section \texttt{LISTINP}: noli = 0 slip = 1.0
   
   (a) \( n_k \): number of level surfaces \( N_z \) \((N_z \geq 1)\); if one layer is chosen the depth averaged velocities will be computed; if more than one layer is chosen the velocity will be computed for several layers (vertical velocity profile);
   
   (b) \( n_q \): steering parameter \((0 = \text{use hydrostatic pressure approximation}, 1 = \text{hydrodynamic pressure})\);
(c) nsp: number of dissolved species (e.g. salinity, temperature, suspended sediment);

(d) epsi: tolerance $\varepsilon^n$ for the free-surface iterative solver (problem dependent, choose with care, for control use \texttt{check\_continuity});

(e) qpsi: tolerance $\varepsilon_q$ for the non-hydrostatic pressure iterative solver (problem dependent, choose with care);

(f) maxiter: maximum number of iterations $n_e$ for the iterative solvers;

(g) theta: implicitness factor $\theta$ ($0.5 < \theta \leq 1.0$); recommended value $\theta = 0.6$;

(h) delt: numerical time step $\Delta t$;

(i) dxmin: minimum allowed distance $\delta_{\text{min}}$ ($\delta_{\text{min}} > 0.0$) between polygon centers;

(j) dzmin: minimum allowed water depth $H_{\text{min}}$ ($H_{\text{min}} > 0.0$);

(k) dtmin: to solve for tracer transport, the time step $\Delta t$ is automatically broken into a smaller or larger number of substeps $\Delta t_i$; if $M_s$ is the current number of substeps, the substepping procedure will continue as long as $\Delta t - \sum_i^{M_s} \Delta t_i > \text{dtmin}$ holds;

(l) hvis: horizontal molecular viscosity $\nu^h_m$ ($\nu^h_m \geq 0.0$);

(m) vvis: vertical molecular viscosity $\nu^v_m$ ($\nu^v_m \geq 0.0$);

(n) hdif: horizontal molecular diffusivity $K^h_m$ ($K^h_m \geq 0.0$);

(o) vdif: vertical molecular diffusivity $K^v_m$ ($K^v_m \geq 0.0$);

(p) trx: inverse of the relaxation time at open boundaries;

(q) cz: Chezy bottom friction coefficient $C_z$ ($C_z > 0.0$);

(r) noli: choose advective scheme,
  i. noli$= 0$ (no advection),
  ii. noli$= 1$ (with advection, diffusive scheme), and
  iii. noli$= 2$ (with advection, recommended less diffusive scheme).

(s) slip: slip parameter for velocity boundary condition at solid vertical walls, e.g.:
  i. free slip: 1.0;
  ii. partial slip: 0.0;
  iii. no slip: -1.0.

2. if more than one layer was chosen, $nk-1$ level surfaces must be prescribed from top to bottom.

\textit{Notice:} The use of parameter epsi can be optimized by checking the influence on results from function \texttt{check\_continuity}.
D.2.2 Example file

The following lines show an example for the input data file `untrim.inp`. The line-numbers ?? indicated are not part of the input data file.

```
101 &LISTINP
102 NK = 10
103 NQ = 1
104 NSP = 1
105 EPSI = 1.00E-04
106 QPSI = 1.00E-04
107 MAXITER = 1000
108 THETA = 0.6
109 DELT = 0.25
110 DXMIN = 5.00000E-02
111 DZMIN = 5.00000E-03
112 DTMIN = 1.00000E-04
113 HVIS = 0.0
114 VVIS = 0.0
115 HDIF = 0.0
116 VDIF = 0.0
117 TRX = 1000000000.0
118 CZ = 50.0
119 NOLI=1
120 SLIP=1.0
121 /
122   -2.00000
123    -1.00000
124        0.00000
125         1.00000
126            2.00000
127               4.00000
128                  6.00000
129                     8.00000
130                          10.00000
```

A few comments on the short example shown above:

**102–102:** ten vertical layers are prescribed in this case, which means that the vertical flow profile can be resolved.

**103–103:** the pressure will be computed during the simulation because the pressure is assumed to be non-hydrostatic.

**104–104:** one dissolved specie (e.g. temperature) is taken into account.

**105–106:** tolerances for the iterative solvers.

**107–107:** maximum number of inner PCG iterations for the iterative solvers.
108–108: an implicitness factor of 0.6 guarantees numerical stability.

109–110: small time steps are necessary if the time scales of the physical processes are also small.

110–110: minimum horizontal distance between polygon centers.

111–111: minimum allowed water depth.

112–112: minimum substep size.

113–116: molecular viscosities are set to zero in this situation (inviscid case).

117–117: the inverse of the relaxation time is set to a high number at open boundaries; as a consequence of this outgoing waves may be reflected (back) at open boundaries.

118–118: default bottom friction coefficient.

119–119: non-linear advective terms are switched on;

120–120: free slip condition for velocities at lateral boundaries;

121–121: terminator “/” for the namelist section.

122–130: level surfaces between vertical layers.

D.3 Source and sink data file

D.3.1 Short description

The input file for sources and sinks is sub-divided into the following sections:

1. number of sources and sinks \( N_d \)

   (a) \( ncs \): number of sources and sinks \( N_d \) (\( N_d \geq 0 \)).

2. list of data related to the \( N_d \) sources and sinks; each line contains the following data for one source/sink:

   (a) \( ics \): polygon index \( i \) for the horizontal location of the source/sink;

   (b) \( kcs \): layer index \( k \) for the vertical position of the source/sink;

If this type of file is absent or empty no sources/sinks will be assumed (\( N_d = 0 \)).
D.3.2 Example file

The following lines show an example for the source/sink data file untrim.srs. The line-numbers !? indicated are not part of the source/sink data file.

```
 101 3
 102 1254 2
 103 1254 3
 104 3140 3
```

A few comments on the short example shown above:

**101 – 101:** three sources/sinks are prescribed;

**102 – 103:** the first two sources/sinks are located in the same polygon but at different depth (within layers two and three);

**104 – 104:** the third source/sink is located in a different polygon (within layer three).