BUNDESANSTALT FÜR WASSERBAU
Karlsruhe • Hamburg • Ilmenau

Technical Report

Mathematical Model UnTRIM

User Interface Description

– Version June 2004 (1.0) –

BUNDESANSTALT FÜR WASSERBAU
Karlsruhe • Hamburg • Ilmenau

Technical Report

Mathematical Model UnTRIM

User Interface Description

– Version June 2004 (1.0) –

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

Version-Date: *June* 2004

Version-no.: 1.0

Trento · Hamburg — *August* 2004

The Federal Waterways Engineering and Research Institute · Wedeler Landstr. 157 · D-22559 Hamburg · ([+49] (0)40 81908 - 0

Summary

This document is the user interface description for the mathematical model UnTRIM. This document can be regarded as an appendix to *Mathematical Model* UnTRIM – *Validation Document*, which is available as a separate report.

Organization of this document

Chapter B contains a short overview for all user interface routines (see Section B.4 on page 66) together with a detailed description of all Set-routines (see Section B.1 on page 1) as well as Get-functions (please refer to Section B.2 on page 24).

Chapter C summarizes the available Set-routines and Get-functions in tabular form. This may help the experienced user of UnTRIM 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.

Acknowledgements

Ralph T. Cheng is responsible for the development of a version of the UnTRIM at the U.S. Geological Survey (USGS). Continuing discussions and exchanging of modeling ideas with regards to user interface description of UnTRIM with Ralph T. Cheng are acknowledged.

During the first *International* UnTRIM *Users Workshop* in Verona, June 7–9, 2004, it was decided to update this document, to conform to the actual version of the software (June 2004). Positive and stimulating feedback from the users to continue working on this type of document is herewidth gratefully acknowledged too.

Contents

List of Figures

List of Tables

B User interface

B.1 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 file
    - `set_grid_file`
  - sources and sinks file
    - `set_source_file`
- steering data
  - transport algorithm for scalar species
    - `set_flux_limiter`
  - scalar species
    - `set_sediment`
- initial data and (state) variables
  - hydrodynamics
    - `set_density`
    - `set_elevation`
    - `set_horizontal_velocity`
    - `set_pressure`
    - `set_turbulent_h_viscosity`
    - `set_turbulent_v_viscosity`
    - `set_velocity`
  - scalar species
    - `set_concentration`
    - `set_settling_velocity`
    - `set_turbulent_h_diffusivity`
    - `set_turbulent_v_diffusivity`
- data at lateral open boundaries with *prescribed water level*
  - hydrodynamics

        - `set_elevation_bc`
        - `set_pressure_bc`
    - scalar species
        - `set_concentration_bc`

- data at lateral open boundaries with *prescribed flow*
    - hydrodynamics
        - `set_inflow_bc`
    - scalar species
        - `set_inflow_cc`

- boundary data at the free surface
    - atmospheric data near the free surface
        - `set_atmospheric_pressure`
        - `set_wind_friction`
        - `set_wind_velocity`
    - 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`

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

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

B.1.1 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 [real]` 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 [real]`;
3. `CALL set_atmospheric_pressure(a(:))`: a spatially varying normalized atmospheric pressure `a(`N_p`) [real]` is assigned to all polygons N_p.

To determine N_p use `get_nof_polygons()`. To obtain the normalized atmospheric pressure p_a, the (real physical) atmospheric pressure has to be divided by ρ_0.

B.1.2 Interface `set_bottom_alpha`

physical unit: m / s

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

Assign bottom flux parameter α_B at polygon centers:

1. `CALL set_bottom_alpha(s)`: a constant `s [real]` is assigned to the bottom flux parameter α_B for all scalar species N_c and all polygons N_p;
2. `CALL set_bottom_alpha(m,s)`: a constant `s [real]` is assigned to the bottom flux parameter α_B for the m-th scalar specie ($1 \leq m \leq N_c$) and all polygons N_p;
3. `CALL set_bottom_alpha(m,i,s)`: a constant `s [real]` is assigned to the bottom flux parameter α_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_alpha(m,a(:))`: spatially varying data `a(`N_p`) [real]` are assigned to the bottom flux parameter α_B 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_c`) [real]`, different for each of the N_c scalar species, are assigned to the bottom flux parameter α_B for all N_c species and all N_p polygons.

$\alpha_B \geq 0.0$ must be fulfilled. N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the bottom boundary the settings for α_B, β_B (`set_bottom_beta`) as well as C_B (`set_bottom_concentration`) must be chosen in a coordinated way.

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

B.1.3 Interface `set_bottom_beta`

physical unit: m / s

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

Assign bottom flux parameter β_B at polygon centers:

1. `CALL set_bottom_beta(s)`: a constant `s [real]` is assigned to the bottom flux parameter β_B for all scalar species N_c and all polygons N_p;
2. `CALL set_bottom_beta(m,s)`: a constant `s [real]` is assigned to the bottom flux parameter β_B for the m-th scalar specie ($1 \leq m \leq N_c$) and all polygons N_p;
3. `CALL set_bottom_beta(m,i,s)`: a constant `s [real]` is assigned to the bottom flux parameter β_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`) [real]` are assigned to the bottom flux parameter β_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_c`) [real]`, different for each of the N_c scalar species, are assigned to the bottom flux parameter β_B for all N_c species and all N_p polygons.

$\beta_B \geq 0.0$ must be fulfilled. N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the bottom boundary the settings for β_B, α_B (`set_bottom_alpha`) as well as C_B (`set_bottom_concentration`) must be chosen in a coordinated way.

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

B.1.4 Interface `set_bottom_concentration`

physical unit: identical to physical unit of specie used

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

Assign bottom concentration C_B at polygon centers:

1. `CALL set_bottom_concentration(s)`: a constant `s [real]` 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 [real]` 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 [real]` 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`) [real]` 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_c`) [real]`, different for each of the N_c scalar species, are assigned to the bottom concentration C_B for all N_c species and all N_p polygons.

N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the bottom boundary the settings for C_B, α_B (`set_bottom_alpha`) as well as β_B (`set_bottom_beta`) must be chosen in a coordinated way.

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

B.1.5 Interface `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 `"untrim.inp"`

Assign bottom friction coefficient r_B at sides / edges:

1. `CALL set_bottom_friction(s)`: a constant bottom friction `s [real]` is assigned to all sides N_s;
2. `CALL set_bottom_friction(j,s)`: bottom friction is set for the j-th side ($1 \leq j \leq N_s$) to a constant value `s [real]`;
3. `CALL set_bottom_friction(a(:))`: a spatially varying bottom friction `a(`N_s`) [real]` is assigned to all sides N_s.

$r_B \geq 0.0$ must be fulfilled. N_s can be determined by means of `get_nof_edges()`. In dependence on the computation of r_B, different types of bottom friction laws can be applied. In general H in the aforementioned formulas should be replaced by $\Delta z_{j,k_b(j)}$. In a two-dimensional depth-averaged simulation this is equivalent to the water depth, whereas in the three-dimensional case this corresponds to the height of the lowermost face. $k_b(j)$ is obtained from `get_bottom_face(j)`.

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

B.1.6 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 [real]` is assigned to the specie concentration C for all scalar species N_c and all prisms I_3;
2. `CALL set_concentration(m,s)`: a constant `s [real]` is assigned to the specie concentration C for the m-th scalar specie ($1 \leq m \leq N_c$) and all prisms I_3;
3. `CALL set_concentration(m,i,s)`: a constant `s [real]` 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 [real]` is assigned to the specie concentration C for the m-th scalar specie ($1 \leq m \leq N_c$) in the k-th layer ($k_b(i) \leq k \leq k_t(i)$) above the i-th polygon ($1 \leq i \leq N_p$);
5. `CALL set_concentration(m,a(:))`: spatially varying data `a(`I_3`) [real]` are assigned to the specie concentration C for the m-th scalar specie ($1 \leq m \leq N_c$) at all I_3 prisms;
6. `CALL set_concentration(m,i,a(:))`: depth varying data `a(`N_z`) [real]` 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_3, N_c`) [real]`, different for each of the N_c scalar species, are assigned to the concentration C for all N_c species and all I_3 prisms.

The total number of prisms I_3 is given by `get_nof_prisms()`. N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`.

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.1.7 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 [real]` is assigned to the specie concentration at lateral open boundaries for all scalar species N_c and all prisms I_3^* above open boundary polygons;
2. `CALL set_concentration_bc(m,s)`: a constant `s [real]` 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_3^* above open boundary polygons;
3. `CALL set_concentration_bc(m,i,s)`: a constant `s [real]` 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 ($1 \leq i \leq N_p^*$) open boundary polygon;
4. `CALL set_concentration_bc(m,i,k,s)`: a constant `s [real]` 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 i ($1 \leq i \leq N_p^*$);
5. `CALL set_concentration_bc(m,a(:))`: spatially varying data `a(`I_3^*`) [real]` 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_3^* prisms above open boundary polygons;
6. `CALL set_concentration_bc(m,i,a(:))`: depth varying data `a(`N_z`) [real]` 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(`I_3^*, N_c`) [real]`, different for each of the N_c scalar species, are assigned to the concentration at lateral open boundaries for all N_c species and all I_3^* prisms above open boundary polygons.

The total number of open boundary prisms is given by $I_3^* = \sum_{i=1}^{N_p^*} (\text{get\_nof\_prisms(i)} + 1)$. N_p^* can be determined from `get_nof_boundary_polygons()`, whereas N_c can be obtained from `get_nof_species()`. 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.1.8 Interface `set_density`

physical unit: —

default value: 1.0 (if interface is *not* used)

Assign normalized water density ρ/ρ_0 at prism centers:

1. `CALL set_density(s)`: a constant `s [real]` is assigned to the normalized water density ρ/ρ_0 for all prisms I_3;
2. `CALL set_density(i,s)`: a constant `s [real]` is assigned to the normalized water density ρ/ρ_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 [real]` is assigned to the normalized water density ρ/ρ_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`) [real]` are assigned to the normalized water density ρ/ρ_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_3`) [real]` are assigned to the normalized water density ρ/ρ_0 for all prisms I_3.

$\rho > 0$ must be fulfilled. The total number of prisms I_3 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.1.9 Interface `set_elevation`

physical unit: m [reference level]

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

Assign water surface elevation η 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 [real]` is assigned to the water surface elevation η for all polygons N_p;

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

N_p is given by `get_nof_polygons()`.

B.1.10 Interface `set_elevation_bc`

physical unit: m [reference level]

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

Assign water surface elevation η at polygon centers along open boundaries with *prescribed water level*:

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

N_p^* is given by `get_nof_boundary_polygons()`.

B.1.11 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; the following options are available at the moment:
   (a) `"Minmod"`;
   (b) `"van Leer"`;
   (c) `"Superbee"`.

Default is `"0(upwind)"`, which corresponds to the classical method used in previous UnTRIM–versions.

B.1.12 Interface `set_grid_file`

physical unit: text string, no unit

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

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.

B.1.13 Interface `set_horizontal_velocity`

physical unit: m / s

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 [real]` is assigned to the normal velocity component u for all computational faces J_3;
2. `CALL set_horizontal_velocity(j,uu,vv)`: a constant horizontal velocity with given x-y-components `uu, vv [real]` is assigned to the normal velocity component u for all computational faces lying above the j-th side ($1 \leq j \leq N_s$);
3. `CALL set_horizontal_velocity(j,k,uu,vv)`: a constant horizontal velocity with given x-y-components `uu, vv [real]` is assigned to the normal velocity component u for the computational face located in the k-th layer ($k_b(j) \leq k \leq k_t(j)$) above the j-th side ($1 \leq j \leq N_s$);
4. `CALL set_horizontal_velocity(j,uu(:),vv)`: a depth-varying horizontal velocity with given variable x- and constant y-component `uu`(N_z)`, vv [real]` is assigned to the normal velocity component u for the computational faces lying above the j-th side ($1 \leq j \leq N_s$);
5. `CALL set_horizontal_velocity(j,uu,vv(:))`: a depth-varying horizontal velocity with given variable y- and constant x-component `uu, vv`(N_z)` [real]` is assigned to the normal velocity component u for the computational faces lying above the j-th side ($1 \leq j \leq N_s$);
6. `CALL set_horizontal_velocity(j,uu(:),vv(:))`: a depth-varying horizontal velocity with given x-y-components `uu`(N_z)`, vv`(N_z)` [real]` is assigned to the normal velocity component u for the computational faces lying above the j-th side ($1 \leq j \leq N_s$);

7. `CALL set_horizontal_velocity(uu(:),vv)`: a spatially varying horizontal velocity in *x*-direction and constant in *y*-direction with `uu(`J_3`), vv [real]` is assigned to the normal velocity component *u* at all the J_3 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_3`) [real]` is assigned to the normal velocity component *u* at all the J_3 computational faces;

9. `CALL set_horizontal_velocity(uu(:),vv(:))`: a spatially varying horizontal velocity with given *x*-*y*-components `uu(`J_3`), vv(`J_3`) [real]` is assigned to the normal velocity component *u* at all the J_3 computational faces.

The total number of faces J_3 is determined by means of `get_nof_faces()`, whereas N_s is obtained from `get_nof_egdes()`. 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 $k_t(j)$ can be obtained from `get_top_face(j)`.

B.1.14 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 [real]` is equally distributed to all N_{s_f} boundary edges with prescribed flow boundary condition;

2. `CALL set_inflow_bc(j,s)`: a constant integral discharge `s [real]` is equally distributed to all faces above edge j ($N_{s_i}+1 \leq j \leq N_{s_f}$) where the prescribed flow boundary condition shall be applied;

3. `CALL set_inflow_bc(j,k,s)`: a constant `s [real]` is assigned to the flow within the *k*-th layer ($k_b(j) \leq k \leq k_t(j)$) at open boundary edge j ($N_{s_i}+1 \leq j \leq N_{s_f}$) where the prescribed flow boundary condition shall be applied.

N_{s_i} is given by `get_nof_internal_edges()` and N_{s_f} by N_{s_i}+`get_nof_inflow_edges()`, while the bottom layer index above edges $k_b(j)$ is given by `get_bottom_face(j)` and $k_t(j)$ can be obtained from `get_top_face(j)`.

B.1.15 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 [real]` is assigned to the specie concentration for all open boundary edges with prescribed flow boundary condition for all scalar species N_c;
2. `CALL set_inflow_cc(m,s)`: a constant `s [real]` is assigned to the specie concentration for all open boundary edges 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 [real]` is assigned to the specie concentration at the j-th edge ($N_{s_i}+1 \leq j \leq N_{s_f}$) with prescribed flow boundary condition for the m-th scalar specie ($1 \leq m \leq N_c$);
4. `CALL set_inflow_cc(m,j,k,s)`: a constant `s [real]` is assigned to the m-th specie concentration ($1 \leq m \leq N_c$) at the j-th edge ($N_{s_i}+1 \leq j \leq N_{s_f}$) with prescribed flow boundary condition in the k-th layer ($k_b(j) \leq k \leq k_t(j)$).

N_{s_i} is given by `get_nof_internal_edges()` and N_{s_f} by N_{s_i}+`get_nof_inflow_edges()`, while the bottom layer index above edges $k_b(j)$ is given by `get_bottom_face(j)` and $k_t(j)$ can be obtained from `get_top_face(j)`, whereas N_c can be retrieved using `get_nof_species()`.

B.1.16 Interface `set_input_file`

physical unit: text string, no unit

default value: `"untrim.inp"` (if interface is *not* used)

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.1.17 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 [real]` is assigned to all source / sink locations N_d;
2. `CALL set_point_sources_discharge(n,s)`: the source / sink discharge `s [real]` 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`) [real]` is assigned to all the N_d source / sink locations.

N_d is given by `get_nof_point_sources()`.

B.1.18 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 [real]` 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 [real]` 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 [real]` 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`) [real]` are assigned for the m-th specie ($1 \leq m \leq N_c$) to the N_d source / sink locations;
5. `CALL set_point_sources_concentration(b(:,:))`: spatially varying data `b(`N_d`,`N_c`) [real]` are assigned for all species N_c and all source / sink locations N_d.

$C \geq 0.0$ must be fulfilled. N_d is given by `get_nof_point_sources()` whereas N_c can be retrieved using `get_nof_species()`. During outflow, application of these routines has no effect on computed results.

B.1.19 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 [real]` is assigned to the normalized pressure component q for all computational prisms I_3;
2. `CALL set_pressure(i,s)`: a constant `s [real]` 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 [real]` 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`) [real]` 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`) [real]` are assigned to the normalized pressure q for all I_3 computational prisms.

The total number of prisms I_3 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)`. To obtain the normalized pressure p, the (real physical) pressure has to be divided by ρ_0.

B.1.20 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 [real]` 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 [real]` 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 [real]` 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`) [real]` 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^*`) [real]` 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 $I_3^* = \sum_{i=1}^{N_p^*} (\text{get\_nof\_prisms(i)} + 1)$. N_p^* can be determined from `get_nof_boundary_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)`. To obtain the normalized pressure p, the (real physical) pressure has to be divided by ρ_0.

B.1.21 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) or passive non-settling ($= 0.0$) behaviour:

1. `CALL set_sediment(s)`: a constant factor `s [real]` is assigned to all species N_c;

2. `CALL set_sediment(a(:))`: a species-dependent constant factor `a(`N_c`) [real]` is assigned to all species N_c;

3. `CALL set_sediment(m,s)`: a constant factor `s [real]` is assigned to the m-th scalar specie ($1 \leq m \leq N_c$).

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.

B.1.22 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 [real]` is assigned to the settling velocity w^s for all computational prisms I_3;
2. `CALL set_settling_velocity(i,s)`: a constant `s [real]` is assigned to the settling velocity w^s for all computational prisms located above polygon i ($1 \leq i \leq N_p$);
3. `CALL set_settling_velocity(i,k,s)`: a constant `s [real]` is assigned to the settling velocity w^s for the computational prism located in layer k ($k_b(i) \leq k \leq k_t(i)$) above polygon i ($1 \leq i \leq N_p$);
4. `CALL set_settling_velocity(i,a(:))`: depth varying data `a(`N_z`) [real]` are assigned to the settling velocity w^s for all prisms above polygon i ($1 \leq i \leq N_p$);
5. `CALL set_settling_velocity(a(:))`: spatially varying data `a(`I_3`) [real]` are assigned to the settling velocity w^s for all prisms I_3.

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`). $w^s \geq 0.0$ must be fulfilled. The total number of prisms I_3 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.1.23 Interface `set_source_file`

physical unit: text string, no unit

default value: `"untrim.srs"` (if interface is *not* used)

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.1.24 Interface `set_surface_alpha`

physical unit: m / s

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

Assign surface flux parameter α_T at polygon centers:

1. `CALL set_surface_alpha(s)`: a constant `s [real]` is assigned to the surface flux parameter α_T for all scalar species N_c and all polygons N_p;

2. `CALL set_surface_alpha(m,s)`: a constant `s [real]` is assigned to the surface flux parameter α_T for the m-th scalar specie ($1 \leq m \leq N_c$) and all polygons N_p;

3. `CALL set_surface_alpha(m,i,s)`: a constant `s [real]` is assigned to the surface flux parameter α_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. `CALL set_surface_alpha(m,a(:))`: spatially varying data `a(`N_p`) [real]` are assigned to the surface flux parameter α_T 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_c`) [real]`, different for each of the N_c scalar species, are assigned to the surface flux parameter α_T for all N_c species and all N_p polygons.

$\alpha_T \geq 0.0$ must be fulfilled. N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the water surface boundary the settings for α_T, β_T (`set_surface_beta`) as well as C_T (`set_surface_concentration`) must be chosen in a coordinated way.

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

B.1.25 Interface `set_surface_beta`

physical unit: m / s

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

Assign surface flux parameter β_T at polygon centers:

1. `CALL set_surface_beta(s)`: a constant `s [real]` is assigned to the surface flux parameter β_T for all scalar species N_c and polygons N_p;

2. `CALL set_surface_beta(m,s)`: a constant `s [real]` is assigned to the surface flux parameter β_T 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)`: a constant `s [real]` is assigned to the surface flux parameter β_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. `CALL set_surface_beta(m,a(:))`: spatially varying data `a(`N_p`) [real]` are assigned to the surface flux parameter β_T 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_c`) [real]`, different for each of the N_c scalar species, are assigned to the surface flux parameter β_T for all N_c species and all N_p polygons.

$\beta_T \geq 0.0$ must be fulfilled. N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the water surface boundary the settings for β_T, α_T (`set_surface_alpha`) as well as C_T (`set_surface_concentration`) must be chosen in a coordinated way.

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

B.1.26 Interface `set_surface_concentration`

physical unit: identical to physical unit of specie used

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

Assign surface concentration C_T at polygon centers:

1. `CALL set_surface_concentration(s)`: a constant `s [real]` is assigned to the surface concentration C_T for all scalar species N_c and all polygons N_p;
2. `CALL set_surface_concentration(m,s)`: a constant `s [real]` 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. `CALL set_surface_concentration(m,i,s)`: a constant `s [real]` 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. `CALL set_surface_concentration(m,a(:))`: spatially varying data `a(`N_p`) [real]` 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. `CALL set_surface_concentration(b(:,:))`: spatially varying data `b(`N_p, N_c`) [real]`, different for each of the N_c scalar species, are assigned to the surface concentration C_T for all N_c species and all N_p polygons.

N_p can be determined by means of `get_nof_polygons()`, whereas N_c can be obtained from `get_nof_species()`. In dependence on the situation at the water surface boundary the settings for C_T, α_T (`set_surface_alpha`) as well as β_T (`set_surface_beta`) must be chosen in a coordinated way.

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

B.1.27 Interface `set_turbulent_h_diffusivity`

physical unit: m^2/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 [real]` is assigned to the horizontal turbulent diffusivity K^h for all computational faces J_3;
2. `CALL set_turbulent_h_diffusivity(j,s)`: a constant `s [real]` is assigned to the horizontal turbulent diffusivity K^h for all computational faces above the j-th side ($1 \leq j \leq N_s$);
3. `CALL set_turbulent_h_diffusivity(j,k,s)`: a constant `s [real]` is assigned to the horizontal turbulent diffusivity K^h for the computational face within the k-th layer ($k_b(j) \leq k \leq k_t(j)$) above the j-th side ($1 \leq j \leq N_s$);
4. `CALL set_turbulent_h_diffusivity(j,a(:))`: depth varying data `a(`N_z`) [real]` are assigned to the horizontal turbulent diffusivity K^h for all computational faces above side j ($1 \leq j \leq N_s$);
5. `CALL set_turbulent_h_diffusivity(a(:))`: spatially varying data `a(`J_3`) [real]` are assigned to the horizontal turbulent diffusivity K^h for each of the J_3 computational faces.

$K^h \geq 0.0$ must be fulfilled. The total number of faces J_3 is determined by means of `get_nof_faces()`, whereas N_s is obtained from `get_nof_egdes()`. 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 $k_t(j)$ can be obtained from `get_top_face(j)`.

B.1.28 Interface `set_turbulent_h_viscosity`

physical unit: m^2/s

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

Assign horizontal turbulent viscosity ν^h at face centers:

1. `CALL set_turbulent_h_viscosity(s)`: a constant `s [real]` is assigned to the horizontal turbulent viscosity ν^h for all computational faces J_3;
2. `CALL set_turbulent_h_viscosity(j,s)`: a constant `s [real]` is assigned to the horizontal turbulent viscosity ν^h for all computational faces located above the j-th edge ($1 \leq j \leq N_s$);
3. `CALL set_turbulent_h_viscosity(j,k,s)`: a constant `s [real]` is assigned to the horizontal turbulent viscosity ν^h for computational face within the k-th layer ($k_b(j) \leq k \leq k_t(j)$) above the j-th edge ($1 \leq j \leq N_s$);

4. `CALL set_turbulent_h_viscosity(j,a(:))`: depth varying data `a(`N_z`) [real]` are assigned to the horizontal turbulent viscosity ν^h for all computational faces above edge j ($1 \leq j \leq N_p$);

5. `CALL set_turbulent_h_viscosity(a(:))`: spatially varying data `a(`J_3`) [real]` are assigned to the horizontal turbulent viscosity ν^h for each of the J_3 computational faces.

$\nu^h \geq 0.0$ must be fulfilled. The total number of faces J_3 is determined by means of `get_nof_faces()`, whereas N_s is obtained from `get_nof_egdes()`. 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 $k_t(j)$ can be obtained from `get_top_face(j)`.

B.1.29 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 [real]` is assigned to the vertical turbulent diffusivity K^v for all computational prisms I_3;

2. `CALL set_turbulent_v_diffusivity(i,s)`: a constant `s [real]` is assigned to the vertical turbulent diffusivity K^v for all computational prisms above the i-th polygon ($1 \leq i \leq N_p$);

3. `CALL set_turbulent_v_diffusivity(i,k,s)`: a constant `s [real]` is assigned to the vertical turbulent diffusivity K^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_diffusivity(i,a(:))`: depth varying data `a(`N_z`) [real]` are assigned to the vertical turbulent diffusivity K^v for all computational prisms above polygon i ($1 \leq i \leq N_p$);

5. `CALL set_turbulent_v_diffusivity(a(:))`: spatially varying data `a(`I_3`) [real]` are assigned to the vertical turbulent diffusivity K^v for each of the I_3 computational prisms.

$K^v \geq 0.0$ must be fulfilled. The total number of prisms I_3 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.1.30 Interface `set_turbulent_v_viscosity`

physical unit: m^2/s

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

Assign vertical turbulent viscosity ν^v at the top of prisms:

1. `CALL set_turbulent_v_viscosity(s)`: a constant `s [real]` is assigned to the vertical turbulent viscosity ν^v for all computational prisms I_3;
2. `CALL set_turbulent_v_viscosity(i,s)`: a constant `s [real]` is assigned to the vertical turbulent viscosity ν^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)`: a constant `s [real]` is assigned to the vertical turbulent viscosity ν^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`) [real]` are assigned to the vertical turbulent viscosity ν^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_3`) [real]` are assigned to the vertical turbulent viscosity ν^v for each of the I_3 computational prisms.

$\nu^v \geq 0.0$ must be fulfilled. The total number of prisms I_3 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.1.31 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 [real]` is assigned to the normal velocity component u for all computational faces J_3;
2. `CALL set_velocity(j,s)`: a constant `s [real]` is assigned to the normal velocity component u for all computational faces above the j-th side ($1 \leq j \leq N_s$);

3. `CALL set_velocity(j,k,s)`: a constant `s [real]` 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 ($1 \leq j \leq N_s$);

4. `CALL set_velocity(j,a(:))`: depth varying data `a(`N_z`) [real]` are assigned to the normal velocity component u for the computational faces lying above the j-th side ($1 \leq j \leq N_s$);

5. `CALL set_velocity(a(:))`: spatially varying data `a(`J_3`) [real]` are assigned to the normal velocity component u at all the J_3 computational faces.

The total number of faces J_3 is determined by means of `get_nof_faces()`, whereas N_s is obtained from `get_nof_egdes()`. 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 $k_t(j)$ can be obtained from `get_top_face(j)`.

B.1.32 Interface `set_wind_friction`

physical unit: —

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

Assign wind friction coefficient r_T at sides / edges:

1. `CALL set_wind_friction(s)`: a constant wind friction `s [real]` is assigned to all sides N_s;

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 [real]`;

3. `CALL set_wind_friction(a(:))`: a spatially varying wind friction `a(`N_s`) [real]` is assigned to all sides N_s.

$r_T \geq 0.0$ must be fulfilled. N_s is obtained from `get_nof_egdes()`. In dependence on the computation of r_T, different types of wind friction laws can be applied.

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

B.1.33 Interface `set_wind_velocity`

physical unit: m / s

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

Assign components of the wind velocity (u_a,v_a) at sides / edges:

1. `CALL set_wind_velocity(wx,wy)`: constant x- and y-components `wx, wy [real]` are assigned to the wind velocity (u_a,v_a) for all sides N_s;
2. `CALL set_wind_velocity(j,wx,wy)`: x- and y-components `wx, wy [real]` are assigned to the wind velocity (u_a,v_a) 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 [real]` are assigned to the wind velocity (u_a,v_a) for all sides N_s;
4. `CALL set_wind_velocity(wx,wy(:))`: spatially varying y- and constant x-component `wx, wy(`N_s`) [real]` are assigned to the wind velocity (u_a,v_a) for all sides N_s;
5. `CALL set_wind_velocity(wx(:),wy(:))`: spatially varying x- and y-components `wx(`N_s`), wy(`N_s`) [real]` are assigned to the wind velocity (u_a,v_a) for all sides N_s.

N_s is obtained from `get_nof_egdes()`. It is normally assumed, that the wind velocity is known 10 m above the free water surface.

B.2 Get-functions

The get-functions are grouped according to their functionality:

- unstructured orthogonal grid data (horizontal structure)
  - static size
    - `get_nof_boundary_polygons`
    - `get_nof_inflow_edges`
    - `get_nof_internal_edges`
    - `get_nof_edges`
    - `get_nof_polygons`
    - `get_nof_vertices`
  - dynamic size
    - `get_nof_wet_edges`
    - `get_nof_wet_polygons`
  - static variables
    - `get_dx_min`
    - `get_hland`
    - `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_dx`
    - `get_dy`
    - `get_edge_center`
    - `get_polygon_area`
    - `get_polygon_center`
    - `get_vertex_coordinates`
    - `get_x_vertex_coordinate`
    - `get_y_vertex_coordinate`
  - dynamic geometry

        - get_depth
        - get_depth_at_edge

- computational grid
    - static size
        - get_nof_faces
        - get_nof_layers
        - get_nof_prisms
    - dynamic size
        - get_nof_wet_faces
        - get_nof_wet_prisms
    - static variables
        - get_dz_min
    - static structure
        - get_layer
    - dynamic structure
        - get_bottom_face
        - get_bottom_prism
        - get_top_face
        - get_top_prism
    - static geometry
        - get_layer_interface
    - dynamic geometry
        - get_face_height
        - get_prism_height
        - get_volume
- computational steering data
    - static variables
        - get_flux_limiter
        - get_nof_point_sources
        - get_nof_species
    - logical data
        - sediment
        - with_hydrodynamic_pressure
        - with_scalar_transport

  - time stepping information
    - `get_time_step`
  - iterative solvers
    - `get_elevation_tolerance`
    - `get_pressure_tolerance`
    - `get_iterations`
    - `get_maximum_iteration`

- computational results
  - hydrodynamics
    - `get_bottom_stress`
    - `get_bottom_flux`
    - `get_elevation`
    - `get_horizontal_velocity`
    - `get_horizontal_velocity_x`
    - `get_horizontal_velocity_y`
    - `get_pressure`
    - `get_surface_flux`
    - `get_velocity`
    - `get_vertical_velocity`
    - `get_wind_stress`
  - species
    - `get_concentration`
    - `get_mass`

- computational parameters and coefficients
  - scalars
    - `get_dt_min`
    - `get_gravity`
    - `get_chezy`
    - `get_turbulent_h_diffusivity`
    - `get_turbulent_h_viscosity`
    - `get_radiation_time`
    - `get_theta`
    - `get_turbulent_v_diffusivity`
    - `get_turbulent_v_viscosity`
  - equation of state data

    - `get_density`
  - turbulence related data
    - `get_turbulent_h_diffusivity`
    - `get_turbulent_h_viscosity`
    - `get_turbulent_v_diffusivity`
    - `get_turbulent_v_viscosity`
  - data related to sources and sinks
    - `get_source_concentration`
    - `get_source_discharge`
    - `get_source_layer`
    - `get_source_polygon`
  - hydrodynamic boundary data
    - `get_bottom_friction`
    - `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_surface_alpha`
    - `get_surface_beta`
    - `get_surface_concentration`
  - sediment data
    - `get_settling_velocity`

All get routines available to the user are subsequently described and listed in alphabetical order.

B.2.1 Interface `get_adjacent_polygon`

physical unit: —

Evaluate indices of adjacent polygons:

1. `get_adjacent_polygon(i,l)`: returns scalar `s[int]` with index of adjacent polygon for the l-th ($1 \leq l \leq S_i$) side of polygon i ($1 \leq i \leq N_p$);

2. `get_adjacent_polygon(i)`: array `a(`S_i`) [int]` of adjacent polygons for the i-th polygon ($1 \leq i \leq N_p$) is returned, where `a(l)` is the adjacent polygon index for the l-th side;

3. `get_adjacent_polygon()`: array `b(`N_p`,4) [int]` of adjacent polygons for all N_p polygons is returned, where `b(i,l)` represents the index of the polygon which is adjacent to the l-th ($1 \leq l \leq S_i$) side of polygon i.

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 `get_nof_edges(i)`, whereas the total number of polygons N_p can be retrieved from `get_nof_polygons()`.

B.2.2 Interface `get_atmospheric_pressure`

physical unit: m^2 / s^2

Retrieve normalized atmospheric pressure p_a at polygon centers:

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

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

3. `get_atmospheric_pressure(x,y)`: returns scalar `s [real]` with p_a at position `x,y [real]`.

To determine N_p use `get_nof_polygons()`. Please notice that to obtain the (real physical) atmospheric pressure the normalized atmospheric pressure p_a has to be multiplied by ρ_0.

B.2.3 Interface `get_bottom_alpha`

physical unit: m / s

Retrieve bottom flux parameter α_B at polygon centers:

1. `get_bottom_alpha(m,i)`: returns scalar `s [real]` with α_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_alpha(m)`: array `a(`N_p`) [real]` is returned with α_B for the m-th specie ($1 \leq m \leq N_c$) at all N_p polygons, where `a(i)` contains α_B for the i-th polygon;

3. `get_bottom_alpha()`: array `b(`N_p, N_c`) [real]` is returned, where `b(i,m)` contains α_B for specie m at polygon i;

4. `get_bottom_alpha(m,x,y)`: returns scalar `s [real]` with α_B for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p is obtained from `get_nof_polygons()`.

B.2.4 Interface `get_bottom_beta`

physical unit: m / s

Retrieve bottom flux parameter β_B at polygon centers:

1. `get_bottom_beta(m,i)`: returns scalar `s [real]` with β_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_beta(m)`: array `a(`N_p`) [real]` is returned with β_B for the m-th specie ($1 \leq m \leq N_c$) at all N_p polygons, where `a(i)` contains β_B for the i-th polygon;
3. `get_bottom_beta()`: array `b(`N_p, N_c`) [real]` is returned, where `b(i,m)` contains β_B for specie m at polygon i;
4. `get_bottom_beta(m,x,y)`: returns scalar `s [real]` with β_B for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p is obtained from `get_nof_polygons()`.

B.2.5 Interface `get_bottom_concentration`

physical unit: identical to physical unit of specie used

Retrieve bottom concentration C_B at polygon centers:

1. `get_bottom_concentration(m,i)`: returns scalar `s [real]` 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`) [real]` 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_c`) [real]` is returned, where `b(i,m)` contains C_B for specie m at polygon i;
4. `get_bottom_concentration(m,x,y)`: returns scalar `s [real]` with C_B for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p is obtained from `get_nof_polygons()`.

B.2.6 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 [int]` with bottom face layer index k for side j ($1 \leq j \leq N_s$);
2. `get_bottom_face()`: array `a(`N_s`) [int]` is returned with bottom face layer indices for all N_s sides, where `a(j)` represents the bottom layer index for the j-th side of the grid.

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

B.2.7 Interface `get_bottom_flux`

physical unit: "physical unit of specie used"$*\mathrm{m}^3$

Retrieve computed bottom flux q_B at polygon centers:

1. `get_bottom_flux(m,i)`: returns scalar `s [real]` 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 [real]` is returned with integral bottom flux q_B for the m-th specie ($1 \leq m \leq N_c$);
3. `get_bottom_flux()`: array `a(`N_c`) [real]` is returned, where `a(m)` is the integral bottom flux q_B for specie m.

N_c can be retrieved using `get_nof_species()`, N_p is given by `get_nof_polygons()`.

Notice: Flux data returned by this function are not precisely equal to the simulated flux, due to the sub-stepping algorithm used in the solver for the transport equation.

B.2.8 Interface `get_bottom_friction`

physical unit: —

Retrieve bottom friction coefficient r_B at sides / edges:

1. `CALL get_bottom_friction(j)`: returns scalar `s [real]` with r_B at the j-th side ($1 \leq j \leq N_s$);
2. `CALL get_bottom_friction()`: array `a(`N_s`) [real]` is returned with r_B at all N_s sides where `a(j)` contains r_B for the j-th side.

N_s can be determined by means of `get_nof_edges()`.

B.2.9 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;
3. `get_bottom_prism(x,y)`: returns scalar `s [int]` with bottom prism layer index k for position `x,y [real]` (if the position is out of the model area, $k = 0$ will be returned instead).

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

B.2.10 Interface `get_bottom_stress`

physical unit: $\mathrm{m}^2/\mathrm{s}^2$

Retrieve (normalized) normal component of bottom shear stress at sides / edges:

1. `get_bottom_stress(j)`: return scalar `s [real]` for the j-th side (edge) ($1 \leq j \leq N_s$);
2. `get_bottom_stress()`: array `a(`N_s`) [real]` is returned with normal component of bottom stress at all N_s sides (edges), where `a(j)` is the respective value for the j-th side.

N_s can be obtained from `get_nof_edges()`. To obtain the (real physical) normal component of the bottom shear stress τ_B, the retrieved normalized one has to be multiplied by ρ_0.

B.2.11 Interface `get_chezy`

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

Retrieve constant Chezy friction factor (prescribed in steering data file `untrim.inp`):

1. `get_chezy()`: scalar `s [real]` is returned with actual value used for the Chezy coefficient.

Normally the bottom friction coefficient is varying in space and time. Use `get_bottom_friction` under these circumstances.

B.2.12 Interface `get_concentration`

physical unit: identical to physical unit of specie used

Retrieve specie concentration C at prism centers:

1. `get_concentration(m,i,k)`: returns scalar `s [real]` 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. `get_concentration(m,i)`: array `a(`N_z`) [real]` 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. `get_concentration(m)`: array `a(`I_3`) [real]` 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_3, N_c`) [real]` is returned with C for all species N_c at all computational prisms I_3.
5. `get_concentration(m,x,y,z)`: returns scalar `s [real]` with C for the m-th specie ($1 \leq m \leq N_c$) at position `x,y,z [real]`;
6. `get_concentration(m,x,y)`: array `a(`N_z`) [real]` is returned with C for the m-th specie ($1 \leq m \leq N_c$) for all computational prisms above position `x,y [real]`.

N_c is retrieved from `get_nof_species()`, N_p from `get_nof_polygons()` and N_z from `get_nof_layers()`. I_3 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.2.13 Interface `get_density`

physical unit: —

Retrieve normalized water density ρ/ρ_0 at prism centers:

1. `get_density(i,k)`: returns scalar `s [real]` with ρ/ρ_0 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_density(i)`: array `a(`N_z`)` is returned with ρ/ρ_0 for all computational prisms above polygon i ($1 \leq i \leq N_p$), where `a(k)` corresponds to the normalized density within layer k ($k_b(i) \leq k \leq k_t(i)$);
3. `get_density()`: array `a(`I_3`)` is returned with ρ/ρ_0 for all computational prisms I_3;
4. `get_density(x,y,z)`: returns scalar `s [real]` with ρ/ρ_0 at position `x,y,z [real]`;

5. `get_density(x,y)`: array `a(`N_z`)` is returned with ρ/ρ_0 for all computational prisms above position `x,y`.

The total number of prisms I_3 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.2.14 Interface `get_depth`

physical unit: m

Extract depth at polygons:

1. `get_depth(i)`: returns scalar `s [real]` with depth at polygon i ($1 \leq i \leq N_p$);
2. `get_depth()`: array `a(`N_p`) [real]` is returned with depth data for all N_p polygons, `a(i)` is equivalent to the depth at the i-th polygon;
3. `get_depth(x,y)`: returns scalar `s [real]` with (polygon) depth at position `x,y [real]` (if the position is out of the model area, h_L will be returned).

The total number of polygons N_p can be determined by means of `get_nof_polygons()`. The depth for permanently dry land h_L can be obtained from `get_hland()`.

B.2.15 Interface `get_depth_at_edge`

physical unit: m

Determine bathymetric depth h at sides / edges:

1. `get_depth_at_edge(j)`: returns scalar `s [real]` with the bathymetric depth h_j for the j-th side ($1 \leq j \leq N_s$);
2. `get_depth_at_edge()`: array `a(`N_s`) [real]` is returned with bathymetric depth for all N_s sides, where `a(j)` represents the bathymetric depth for the j-th side.

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

B.2.16 Interface `get_dt_min`

physical unit: s

Extract minimum allowed time substep size $\Delta\tau$:

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

B.2.17 Interface `get_dx`

physical unit: m

Retrieve distance between adjacent polygon centers δ:

1. `get_dx(j)`: returns scalar `s [real]` with distance between the polygon centers adjacent to the j-th side ($1 \leq j \leq N_s$);
2. `get_dx()`: array `a(`N_s`) [real]` is returned with δ for all N_s sides, where `a(j)` corresponds to the center distance between polygons adjacent to the j-th side.

The total number of sides (edges) N_s can be determined calling `get_nof_edges()`. The condition $\delta \geq \delta_{min}$ always holds true. The minimum allowed distance between polygon centers δ_{min} can be obtained from `get_dx_min()`.

B.2.18 Interface `get_dx_min`

physical unit: m

Return minimum allowed distance δ_{min} between adjacent polygon centers:

1. `get_dx_min()`: returns scalar `s [real]` with minimum allowed distance δ_{min} between adjacent polygon centers.

The distance δ between polygon centers adjacent to sides is limited by δ_{min} for reasons of numerical stability. δ_{min} must be prescribed by the user in the input data file.

B.2.19 Interface `get_dy`

physical unit: m

Extract side length λ:

1. `get_dy(j)`: returns scalar `s [real]` with the length of the j-th side ($1 \leq j \leq N_s$);

2. `get_dy()`: array `a(`N_s`) [real]` is returned with λ for all N_s sides, where `a(j)` corresponds to the length of the j-th side.

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

B.2.20 Interface `get_dz_min`

physical unit: m

Return minimum allowed water depth $H_{\min}$:

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

A computational point is considered to be *wet* (or active) if $H \geq H_{\min}$ holds true. Otherwise the computational point is treated as *dry* (or passive). Due to reasons of numerical stability the minimum allowed water depth must be always larger than zero. $H_{\min}$ is prescribed by the user in the input data file.

B.2.21 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_v instead.

B.2.22 Interface `get_edge_center`

physical unit: m [origin]

Retrieve edge (side) center coordinates x_j, y_j:

1. `get_edge_center(j)`: array `a(2) [real]` is returned with the center coordinates for the j-th edge ($1 \leq j \leq N_s$), where `a(1)` corresponds to x_j and `a(2)` to y_j;

2. `get_edge_center()`: array `b(`N_s`,2) [real]` is returned with the center coordinates for all N_s sides, wherein `b(j,1)` and `b(j,2)` are the center coordinates x_j, y_j for the j-th edge.

The total number of sides (edges) N_s can be determined calling `get_nof_edges()`. The edge center is located in the middle of an edge.

B.2.23 Interface `get_edge_end`

physical unit: —

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

1. `get_edge_end(j)`: returns scalar `s [int]` with the second vertex index for the j-th side ($1 \leq j \leq N_s$);
2. `get_edge_end()`: array `a(`N_s`) [int]` is returned with all indices of the second vertex for all N_s sides, where `a(j)` corresponds to the index of the second 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_v instead.

B.2.24 Interface `get_elevation`

physical unit: m [reference level]

Retrieve water surface elevation η at polygon centers:

1. `get_elevation(i)`: returns scalar `s [real]` with water surface elevation η_i at the i-th polygon ($1 \leq i \leq N_p$);
2. `get_elevation()`: array `a(`N_p`) [real]` is returned with η for all polygons N_p, where `a(i)` corresponds the water surface elevation at polygon i;
3. `get_elevation(x,y)`: returns scalar `s [real]` with water surface elevation η at `x,y [real]` for positions inside the modelling domain, or h_L otherwise.

N_p can be obtained from `get_nof_polygons()`. h_L can be retrieved using `get_hland()`.

B.2.25 Interface `get_elevation_tolerance`

physical unit: —

Retrieve tolerance ε_η for free-surface iterative solver:

1. `get_elevation_tolerance()`: returns scalar `s[real]` with actual ε_η used in the program.

ε_η is specified by the user in the input data file `"untrim.inp"`.

B.2.26 Interface `get_face_height`

physical unit: m

Extract height Δz for faces above sides / edges:

1. `get_face_height(j,k)`: returns scalar `s[real]` with the actual height $\Delta z_{j,k}$ for j-th side ($1 \leq j \leq N_s$) and k-th layer ($k_b(j) \leq k \leq k_t(j)$);
2. `get_face_height(j)`: array `a(`N_z`)[real]` is returned, wherein `a(k)` corresponds to the actual face height of the k-th layer for side j;
3. `get_face_height()`: array `b(`J_3`)[real]` with the actual height for all faces J_3 is retrieved.

$\Delta z_{j,k}$ is zero if the respective face is either a solid wall (boundary) or a *dry* (or passive) computational point. The total number of sides (edges) N_s can be determined calling `get_nof_edges()`, whereas the number of level surfaces (layers) N_z can be obtained from `get_nof_layers()`. J_3 can be obtained from `get_nof_faces()`, while the bottom layer index $k_b(j)$ is given by `get_bottom_face(j)` and the respective surface layer index results from `get_top_face(j)`.

B.2.27 Interface `get_flux_limiter`

physical unit: text string, no unit

Retrieve name of flux limiter used to compute advection for scalar species:

1. `CALL get_flux_limiter( )`: returns scalar `ch[character,len=80]` with the name of the flux limiter actually used:
   (a) `"0(upwind)"`;
   (b) `"Minmod"`;
   (c) `"van Leer"`;
   (d) `"Superbee"`.

B.2.28 Interface `get_gravity`

physical unit: m / s^2

Extract gravitational acceleration *g*:

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

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

B.2.29 Interface `get_hland`

physical unit: m [reference level]

Retrieve depth for permanently dry land h_L:

1. `get_hland()`: returns scalar `s [real]` with the value for the bathymetric depth which is used to characterize permanently dry land h_L.

Locations with bathymetric depth $h > h_L$ are considered to become eventually *wet* (or active). All others are considered to be permanently *dry* land.

B.2.30 Interface `get_horizontal_diffusivity`

physical unit: m^2 / s

Retrieve scalar constant for horizontal diffusivity:

1. `get_horizontal_diffusivity()`: returns scalar `s [real]` with scalar constant horizontal 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 its spatially varying contribution set using `CALL set_turbulent_h_diffusivity`.

B.2.31 Interface `get_horizontal_velocity`

physical unit: m / s

Retrieve horizontally interpolated velocity components *u* (*x*-direction) and *v* (*y*-direction):

1. `get_horizontal_velocity(x,y,z)`: returns array `a(2)[real]` with horizontal velocity components at position `x,y,z[real]`, where `a(1)` and `a(2)` correspond to u and v respectively;

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

3. `get_horizontal_velocity(x,y,z,i)`: returns array `a(2)[real]` with u and v at position `x,y,z[real]`, which is assumed to be located in a computational prism above the i-th polygon ($1 \leq i \leq N_p$);

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

5. `get_horizontal_velocity(x,y,i)`: returns array `b(`N_z`,2)[real]` with u and v for all layers above position `x,y[real]`, which is assumed to lie inside polygon i ($1 \leq i \leq N_p$), where `b(`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 from the projected horizontal velocities extrapolated to the vertices. In z-direction no interpolation scheme is applied.

B.2.32 Interface `get_horizontal_velocity_x`

physical unit: m / s

Retrieve horizontally interpolated velocity u (x-direction):

1. `get_horizontal_velocity_x(x,y,z)`: scalar `s[real]` is returned with horizontal velocity component u at position `x,y,z[real]`;

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

3. `get_horizontal_velocity_x(x,y,z,i)`: scalar `s[real]` is returned with u at position `x,y,z[real]`, which is assumed to be located in a computational prism above the i-th polygon ($1 \leq i \leq N_p$);

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

5. `get_horizontal_velocity_x(x,y,i)`: returns array `a(`N_z`) [real]` with u for all layers above position `x,y [real]`, 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 from the projected horizontal velocities extrapolated to the vertices. In z-direction no interpolation scheme is applied.

B.2.33 Interface `get_horizontal_velocity_y`

physical unit: m / s

Retrieve horizontally interpolated velocity v (y-direction):

1. `get_horizontal_velocity_y(x,y,z)`: scalar `s [real]` is returned with horizontal velocity component v at position `x,y,z [real]`;
2. `get_horizontal_velocity_y(x,y)`: returns array `a(`N_z`) [real]` with v for all layers above position `x,y [real]`, where `a(`k`)` corresponds to the v-component within the k-th layer ($k_b(i) \leq k \leq k_t(i)$);
3. `get_horizontal_velocity_y(x,y,z,i)`: scalar `s [real]` is returned with v at position `x,y,z [real]`, which is assumed to be located in a computational prism above the i-th polygon ($1 \leq i \leq N_p$);
4. `get_horizontal_velocity_y(x,y,i,k)`: scalar `s [real]` is returned with v at position `x,y [real]`, which is assumed to lie inside polygon i ($1 \leq i \leq N_p$) within layer k ($k_b(i) \leq k \leq k_t(i)$);
5. `get_horizontal_velocity_y(x,y,i)`: returns array `a(`N_z`) [real]` with v for all layers above position `x,y [real]`, 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 ($k_b(i) \leq k \leq k_t(i)$).

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 v is interpolated in horizontal direction from the projected horizontal velocities extrapolated to the vertices. In z-direction no interpolation scheme is applied.

B.2.34 Interface `get_horizontal_viscosity`

physical unit: m^2 / s

Retrieve scalar constant for horizontal viscosity:

1. `get_horizontal_viscosity()`: returns scalar `s [real]` with scalar constant horizontal 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 horizontal viscosity applied during computation is the sum of this value plus its spatially varying contribution set using `CALL set_turbulent_h_viscosity`.

B.2.35 Interface `get_iterations`

physical unit: —

Retrieve actual number of iterations done by the iterative solvers:

1. `get_iterations()`: returns array `a(1|2) [real]` with the actual number of iterations used in the program:

   (a) `a(1)` contains the number of iterations used to compute the *free surface*;

   (b) `a(2)` contains the number of iterations used to compute the *hydrodynamic pressure*.

   In situations when the pressure is assumed to by *hydrostatic* an array of length 1 is returned, whereas its length is 2 in *non-hydrostatic* situations!

Which approximation for the pressure is used in the computation can be inquired by means of `with_hydrodynamic_pressure`.

B.2.36 Interface `get_layer`

physical unit: —

Return layer index k for a given depth z:

1. `get_layer(z)`: for a given depth `z [real]` a scalar `s [int]` is returned, which represents the actual layer index k ($1 \leq k \leq N_z$) for a given depth z; if $k = 0$ is returned z lies below the lowermost (deepest) layer.

The number of level surfaces (layers) N_z can be obtained from `get_nof_layers()`.

B.2.37 Interface `get_layer_interface`

physical unit: m [reference level]

Retrieve *z*-coordinate of level surface:

1. `get_layer_interface(k)`: returns scalar `s [real]` with *z*-coordinate $z_{k+\frac{1}{2}}$ for the *k*-th level surface ($1 \leq k \leq N_z$);
2. `get_layer_interface()`: array `a(0:`N_z`) [real]` is returned, wherein `a(0)` corresponds to the maximum bathymetric depth and `a(k)` is the *z*-coordinate $z_{k+\frac{1}{2}}$ for the *k*-th level surface, i. e. $a(N_z)$ is equivalent to h_L.

The number of level surfaces (layers) N_z can be obtained from `get_nof_layers()`.

B.2.38 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 [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`) [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 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.2.39 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.

B.2.40 Interface `get_mass`

physical unit: "physical unit of specie used"$*\mathrm{m}^3$

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

1. `get_mass(m,i,k)`: returns scalar `s[real]` with 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_mass(m,i)`: returns scalar `s[real]` with accumulated 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_mass(m)`: returns scalar `s[real]` with accumulated specie mass for the m-th specie ($1 \leq m \leq N_c$) within the overall computational domain;
4. `get_mass()`: array `a(`N_c`)[real]` is returned with accumulated 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)`.

B.2.41 Interface `get_maximum_iteration`

physical unit: —

Retrieve maximum number of iterations n_ε for iterative solvers:

1. `get_maximum_iteration()`: returns scalar `s[int]` with n_ε used in the program.

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

B.2.42 Interface `get_nof_boundary_polygons`

physical unit: —

Retrieve the number of polygons N_p^* located along the open boundary of the modelling domain with *prescribed water level*.

1. `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 `ün trim.grd¨`.

B.2.43 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 S_i for polygon i ($1 \leq i \leq N_p$);
2. `get_nof_edges()`: array `a(`N_p`)[int]` with actual number of sides for all polygons N_p is returned, where `a(i)` contains the number of sides for the i-th polygon.

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

B.2.44 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 ($1 \leq j \leq N_s$);
2. `get_nof_faces()`: a scalar `s[int]` with the total number of faces J_3 above all N_s sides is returned.

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

B.2.45 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_internal_edges()`: returns scalar `s[int]` with number of edges n_{s_f} with prescribed flow.

This value is (indirectly) set by the user in the grid file `"untrim.grd"`. Please notice that $N_{s_f} = N_{s_i} + n_{s_f}$ holds.

B.2.46 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.2.47 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.2.48 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.2.49 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.2.50 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;
3. `get_nof_prisms(x,y)`: returns scalar `s [int]` with the number of computational prisms I_i above position `x,y [real]` (if the position is out of the model area, $I_i = 0$ will be returned instead).

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

B.2.51 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.2.52 Interface `get_nof_vertices`

physical unit: —

Retrieve number of vertices:

1. `get_nof_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_nof_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_nof_polygons()`.

B.2.53 Interface `get_nof_wet_edges`

physical unit: —

Determine the actual number of *wet* sides / edges:

1. `get_nof_wet_edges(i)`: returns scalar `s [int]` with number of wet sides (edges) for the *i*-th polygon ($1 \leq i \leq N_p$);
2. `get_nof_wet_edges()`: returns scalar `s [int]` with the overall number of wet edges.

The total number of polygons N_p can be retrieved calling `get_nof_polygons()`. An edge j is considered *wet* if $H_j \geq H_{\min}$ holds true.

B.2.54 Interface `get_nof_wet_faces`

physical unit: —

Determine the actual number of *wet* faces:

1. `get_nof_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_nof_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_nof_edges()`. A face (edge) j at layer k is considered *wet* if $\Delta z_{j,k} \geq H_{\min}$ holds true.

B.2.55 Interface `get_nof_wet_polygons`

physical unit: —

Determine the actual number of *wet* polygons:

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

An polygon i is considered *wet* if $H_i \geq H_{\min}$ holds true.

B.2.56 Interface `get_nof_wet_prisms`

physical unit: —

Determine the actual number of *wet* prisms:

1. `get_nof_wet_prisms(i)`: returns scalar `s[int]` with number of wet prisms above polygon i ($1 \leq i \leq N_p$);
2. `get_nof_wet_prisms()`: returns scalar `s[int]` with the overall number of wet prisms;
3. `get_nof_wet_prisms(x,y)`: returns scalar `s[int]` with the number of wet prisms above position `x,y[real]` (if the position is out of the model area, zero will be returned instead).

The total number of polygons N_p can be retrieved calling `get_nof_polygons()`. A prism is considered being *wet* if $\Delta z_{i,k} \geq H_{\min}$ holds true.

B.2.57 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[real]` is located (if the position is out of the model area, $i = 0$ will be returned instead).

B.2.58 Interface `get_polygon_area`

physical unit: m^2

Determine polygon area P:

1. `get_polygon_area(i)`: returns scalar `s[real]` with area P_i for the i-th polygon ($1 \leq i \leq N_p$);
2. `get_polygon_area()`: array `a(`N_p`)[real]` 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()`.

B.2.59 Interface `get_polygon_center`

physical unit: m [origin]

Retrieve center coordinates x_i, y_i for polygons:

1. `get_polygon_center(i)`: array `a(2)[real]` 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. `get_polygon_center()`: array `b(`N_p`,2)[real]` is returned with the center coordinates for all N_p polygons, wherein `b(i,1)` and `b(i,2)` are the center coordinates x_i, y_i for the i-th polygon;
3. `get_polygon_center(x,y)`: returns array `a(2)[real]` with the center coordinates of the polygon wherein `x,y[real]` is located.

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

B.2.60 Interface `get_polygon_edge`

physical unit: —

Evaluate indices for sides (edges) of polygons:

1. `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. `get_polygon_edge(i)`: array `a(`S_i`)[int]` of side indices $j(i,:)$ is returned for the the i-th polygon, where `a(l)` is the side index j for the l-th side of the i-th polygon;
3. `get_polygon_edge()`: array `b(`N_p`,4)[int]` of side indices for all N_p polygons is returned, where `b(i,l)` 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 `get_nof_edges(i)`, whereas the total number of polygons N_p can be retrieved from `get_nof_polygons()`.

B.2.61 Interface `get_polygon_vertex`

physical unit: —

Evaluate indices for vertices (nodes) of polygons:

1. `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. `get_polygon_vertex(i)`: array `a(4)[int]` of vertex indices is returned for the the *i*-th polygon, where `a(l)` is the vertex index *n* for the *l*-th node of the *i*-th polygon;

The total number of polygons N_p can be retrieved from `get_nof_polygons()` whereas the actual number of edges (=vertices) S_i for polygon *i* is obtained by means of `get_nof_edges(i)`.

B.2.62 Interface `get_pressure`

physical unit: m^2 / s^2

Retrieve normalized non-hydrostatic pressure component *q* at prism centers:

1. `get_pressure(i,k)`: returns scalar `s[real]` with *q* at polygon *i* ($1 \leq i \leq N_p$) within layer *k* ($k_b(i) \leq k \leq k_t(i)$);
2. `get_pressure(i)`: array `a(`N_z`)[real]` is returned with *q* at all prisms above polygon *i* ($1 \leq i \leq N_p$), where `a(k)` corresponds to *q* within the *k*-th layer ($k_b(i) \leq k \leq k_t(i)$);
3. `get_pressure()`: array `a(`I_3`)[real]` is returned with *q* at all I_3 computational prisms;
4. `get_pressure(x,y,z)`: returns scalar `s[real]` with *q* at position `x,y,z[real]`;
5. `get_pressure(x,y)`: array `a(`N_z`)[real]` is returned with *q* for all computational prisms above position `x,y[real]`.

N_p can be retrieved from `get_nof_polygons()` and N_z from `get_nof_layers()`. I_3 can be determined using `get_nof_prisms()`. For the *i*-th polygon the bottom and top layer indices $k_b(i)$ and $k_t(i)$ can be obtained from `get_bottom_prism(i)` and `get_top_prism(i)` respectively. Multiplication of *q* with ρ_0 gives the (real physical) non-hydrostatic pressure component.

B.2.63 Interface `get_pressure_tolerance`

physical unit: —

Retrieve tolerance ε_q for non-hydrostatic pressure iterative solver:

1. `get_pressure_tolerance()`: returns scalar `s[real]` with actual ε_q used in the program.

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

B.2.64 Interface `get_prism_height`

physical unit: m

Retrieve prism height $\Delta z_{i,k}$:

1. `get_prism_height(i,k)`: returns scalar `s [real]` with the actual height $\Delta z_{i,k}$ at polygon i ($1 \leq i \leq N_p$) for layer k ($k_b(i) \leq k \leq k_t(i)$);
2. `get_prism_height(i)`: array `a(`N_z`) [real]` is returned, wherein `a(k)` corresponds to the actual prism height for the k-th layer of polygon i ($1 \leq i \leq N_p$);
3. `get_prism_height()`: array `b(`I_3`) [real]` with the actual height for all prisms I_3 is retrieved;
4. `get_prism_height(x,y,z)`: returns scalar `s [real]` with the actual height $\Delta z_{i,k}$ for position `x,y,z [real]` (if position is not located within the computational volume $\Delta z_{i,k} = 0.0$ is returned);
5. `get_prism_height(x,y)`: array `a(`N_z`) [real]` is returned for the column of computational prisms above position `x,y [real]`.

The total number of polygons N_p can be retrieved calling `get_nof_polygons()`, whereas the number of level surfaces (layers) N_z is available from `get_nof_layers()`. $k_b(i)$ results from `get_bottom_prism(i)` and $k_t(i)$ from `get_top_prism(i)`. The total number of prisms I_3 can be determined using `get_nof_prisms()`.

B.2.65 Interface `get_radiation_time`

physical unit: 1 / s

Retrieve radiation time t_r:

1. `get_radiation_time()`: returns scalar `s [real]` with actual t_r used in the program.

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

B.2.66 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.2.67 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(i,k)`: returns scalar `s[real]` with w^s at polygon i ($1 \leq i \leq N_p$) for layer k ($k_b(i) \leq k \leq k_t(i)$);

2. `get_settling_velocity(i)`: array `a(0:`N_z`)[real]` is returned 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 k-th layer ($k_b(i) \leq k \leq k_t(i)$);

3. `get_settling_velocity()`: array `a(`I_3`)[real]` is returned with w^s for all computational prisms;

4. `get_settling_velocity(x,y,z)`: returns scalar `s[real]` with vertically interpolated w^s at position `x,y,z[real]`;

5. `get_settling_velocity(z,i)`: returns scalar `s[real]` with vertically interpolated w^s at position `z[real]` above polygon i ($k_b(i) \leq k \leq k_t(i)$);

6. `get_settling_velocity(x,y)`: array `a(0:`N_z`)[real]` is returned with w^s for the computational column above position `x,y[real]`, where `a(k)` is the settling velocity for the k-th layer ($k_b(i) \leq k \leq k_t(i)$);

7. `get_settling_velocity(x,y,k)`: returns scalar `s[real]` with w^s for the k-th layer ($k_b(i) \leq k \leq k_t(i)$) at position `x,y[real]`.

I_3 can be retrieved using `get_nof_prisms()`, whereas N_z 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)`. 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.

Notice: 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.2.68 Interface `get_source_concentration`

physical unit: identical to physical unit of specie used

Retrieve source concentration C at prism centers:

1. `get_source_concentration(n,m)`: returns scalar `s[real]` 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`)[real]` 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`)[real]` 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_nof_point_sources()` and N_c from `get_nof_species()`.

B.2.69 Interface `get_source_discharge`

physical unit: m^3/s

Extract discharge for sources and sinks:

1. `get_source_discharge(n)`: returns scalar `s[real]` with actual discharge of the n-th source ($1 \leq n \leq N_d$);
2. `get_source_discharge()`: array `a(`N_d`)[real]` 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_nof_point_sources()`.

B.2.70 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.2.71 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.2.72 Interface `get_surface_alpha`

physical unit: m / s

Retrieve surface flux parameter α_T at polygon centers:

1. `get_surface_alpha(m,i)`: returns scalar `s [real]` with α_T 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`) [real]` is returned with α_T for the m-th specie ($1 \leq m \leq N_c$) at all N_p polygons, where `a(i)` contains α_T for the i-th polygon;
3. `get_surface_alpha()`: array `b(`N_p, N_c`) [real]` is returned, where `b(i,m)` contains α_T for specie m at polygon i;
4. `get_surface_alpha(m,x,y)`: returns scalar `s [real]` with α_T for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p is given by `get_nof_polygons()`.

B.2.73 Interface `get_surface_beta`

physical unit: m / s

Retrieve surface flux parameter β_T at polygon centers:

1. `get_surface_beta(m,i)`: returns scalar `s [real]` with β_T 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_beta(m)`: array `a(`N_p`) [real]` is returned with β_T for the m-th specie ($1 \leq m \leq N_c$) at all N_p polygons, where `a(i)` contains β_T for the i-th polygon;
3. `get_surface_beta()`: array `b(`N_p, N_c`) [real]` is returned, where `b(i,m)` contains β_T for specie m at polygon i;
4. `get_surface_beta(m,x,y)`: returns scalar `s [real]` with β_T for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p from `get_nof_polygons()`.

B.2.74 Interface `get_surface_concentration`

physical unit: identical to physical unit of specie used

Retrieve surface concentration C_T at polygon centers:

1. `get_surface_concentration(m,i)`: returns scalar `s [real]` 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. `get_surface_concentration(m)`: array `a(`N_p`) [real]` 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. `get_surface_concentration()`: array `b(`N_p, N_c`) [real]` is returned, where `b(i,m)` contains C_T for specie m at polygon i;
4. `get_surface_concentration(m,x,y)`: returns scalar `s [real]` with C_T for the m-th specie ($1 \leq m \leq N_c$) at position `x,y [real]`.

N_c can be retrieved using `get_nof_species()`, N_p is given by `get_nof_polygons()`.

B.2.75 Interface `get_surface_flux`

physical unit: "physical unit of specie used"$*\mathrm{m}^3$

Retrieve computed free surface flux q_T at polygon centers:

1. `get_surface_flux(m,i)`: returns scalar `s [real]` 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. `get_surface_flux(m)`: scalar `s [real]` is returned with integral free surface flux q_T for the m-th specie ($1 \leq m \leq N_c$);
3. `get_surface_flux()`: array `a(`N_c`) [real]` is returned, where `a(m)` is the integral free surface flux q_T for specie m.

N_c can be retrieved using `get_nof_species()`, N_p is given by `get_nof_polygons()`.

Notice: Flux data returned by this function are not precisely equal to the simulated flux, due to the sub-stepping algorithm used in the solver for the transport equation.

B.2.76 Interface `get_theta`

physical unit: —

Retrieve impliciteness factor θ:

1. `get_theta()`: returns scalar `s [real]` with actual θ used in the program.

θ is prescribed by the user in the input data file `"untrim.inp"`.

B.2.77 Interface `get_time_step`

physical unit: s

Extract time step Δt (in seconds):

1. `get_time_step()`: returns scalar `s [real]` with actual Δt used in the program.

Δt is prescribed by the user in the input data file `"untrim.inp"`.

B.2.78 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 ($1 \leq j \leq N_s$);
2. `get_top_face()`: array `a(`N_s`) [int]` is returned with surface face layer indices for all N_s sides, where `a(j)` represents the surface layer index for the j-th side of the grid.

A surface layer index $k = 0$ will be returned if no (active) computational face is available (dry side). The total number of sides (edges) N_s can be determined calling `get_nof_edges()`.

B.2.79 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;
3. `get_top_prism(x,y)`: returns scalar `s [int]` with surface prism layer index k for position `x,y [real]` (if the position is out of the model area, $k = 0$ will be returned instead).

A surface layer index $k = 0$ will be returned if no (active) computational prism is available (dry polygon). The total number of polygons N_p can be determined by means of `get_nof_polygons()`.

B.2.80 Interface `get_turbulent_h_diffusivity`

physical unit: m^2/s

Retrieve horizontal turbulent diffusivity K^h at faces:

1. `get_turbulent_h_diffusivity(j,k)`: returns scalar `s [real]` with K^h at the j-th side ($1 \leq j \leq N_s$) within layer k ($k_b(j) \leq k \leq k_t(j)$);
2. `get_turbulent_h_diffusivity(j)`: array `a(`N_z`) [real]` is returned with K^h for all computational faces above side j ($1 \leq j \leq N_s$), where `a(k)` is the corresponding value for the k-th layer ($k_b(j) \leq k \leq k_t(j)$);
3. `get_turbulent_h_diffusivity()`: array `a(`J_3`) [real]` is returned with K^h for all computational faces.

J_3 is retrieved from `get_nof_faces()`. N_s can be obtained from `get_nof_edges()` and N_z from `get_nof_layers()`. 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.2.81 Interface `get_turbulent_v_diffusivity`

physical unit: m^2/s

Retrieve vertical turbulent diffusivity K^v at top of prisms:

1. `get_turbulent_v_diffusivity(i,k)`: returns scalar `s [real]` with K^v at the i-th polygon ($1 \leq i \leq N_p$) for layer k ($k_b(i) \leq k \leq k_t(i)$);
2. `get_turbulent_v_diffusivity(i)`: array `a(`N_z`) [real]` is returned 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 k-th layer ($k_b(i) \leq k \leq k_t(i)$);
3. `get_turbulent_v_diffusivity()`: array `a(`I_3`) [real]` is returned with K^v for all computational prisms.

I_3 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.2.82 Interface `get_turbulent_v_viscosity`

physical unit: m^2/s

Retrieve vertical turbulent viscosity ν^v at top of prisms:

1. `get_turbulent_v_viscosity(i,k)`: returns scalar `s [real]` with ν^v at the i-th polygon ($1 \leq i \leq N_p$) for layer k ($k_b(i) \leq k \leq k_t(i)$);
2. `get_turbulent_v_viscosity(i)`: array `a(`N_z`) [real]` is returned with ν^v for all computational prisms above polygon i ($1 \leq i \leq N_p$), where `a(k)` is the corresponding value for the k-th layer ($k_b(i) \leq k \leq k_t(i)$);
3. `get_turbulent_v_viscosity()`: array `a(`I_3`) [real]` is returned with ν^v for all computational prisms.

I_3 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.2.83 Interface `get_velocity`

physical unit: m/s

Retrieve horizontal normal velocity u at computational faces:

1. `get_velocity(j,k)`: returns scalar `s [real]` with u at the j-th side (edge) ($1 \leq j \leq N_s$) within layer k ($k_b(j) \leq k \leq k_t(j)$);
2. `get_velocity(j)`: array `a(`N_z`) [real]` is returned with u for all computational faces above side j ($1 \leq j \leq N_s$), where `a(k)` contains u for the k-th layer ($k_b(j) \leq k \leq k_t(j)$);

3. `get_velocity()`: array `a(`J_3`) [real]` is returned with u at all J_3 faces.

N_s can be retrieved using `get_nof_edges()` and N_z is obtained from `get_nof_layers()`. The bottom face layer index $k_b(j)$ can be determined using `get_bottom_face(j)` whereas $k_t(j)$ is given by `get_top_face(j)`. J_3 is retrieved from `get_nof_faces()`.

B.2.84 Interface `get_vertex_coordinates`

physical unit: m [origin]

Retrieve coordinates (x,y) for vertices:

1. `get_vertex_coordinates(i)`: array `a(2) [real]` is returned with coordinates of the i-th vertex ($1 \leq i \leq N_v$), where `a(1)` and `a(2)` are the respective x- and y-coordinates of the i-th vertex;

2. `get_vertex_coordinates()`: array `b(`N_v`,2) [real]` ist returned with the coordinates for all N_v vertices, wherein `b(i,1)` and `b(i,2)` are the respective x- and y-coordinates of the i-th vertex.

The overall number of vertices N_v can be obtained from `get_nof_vertices()`.

B.2.85 Interface `get_vertical_diffusivity`

physical unit: m^2/s

Retrieve scalar constant for vertical diffusivity:

1. `get_vertical_diffusivity()`: returns scalar `s [real]` with scalar constant vertical 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 its spatially varying contribution set using `CALL set_turbulent_v_diffusivity`.

B.2.86 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 [real]` with w at the i-th polygon ($1 \leq i \leq N_p$) for layer k ($k_b(i) \leq k \leq k_t(i)$);

2. `get_vertical_velocity(i)`: array `a(0:`N_z`)` `[real]` is returned with w for all computational faces above polygon i ($1 \leq i \leq N_p$), where `a(k)` contains w for the k-th layer ($k_b(i) \leq k \leq k_t(i)$);

3. `get_vertical_velocity()`: array `a(`I_3`)` `[real]` is returned with w for all I_3 computational prisms;

4. `get_vertical_velocity(x,y,z)`: returns scalar `s [real]` with w vertically interpolated to position `x,y,z [real]`;

5. `get_vertical_velocity(x,y,k)`: returns scalar `s [real]` with w at position `x,y [real]` for layer k ($k_b(i) \leq k \leq k_t(i)$);

6. `get_vertical_velocity(z,i)`: returns scalar `s [real]` with w vertically interpolated to position `z [real]` above polygon i ($1 \leq i \leq N_p$);

7. `get_vertical_velocity(x,y)`: array `a(0:`N_z`)` `[real]` is returned with w for all computational faces above position `x,y [real]`, where `a(k)` contains w for the k-th layer ($k_b(i) \leq k \leq k_t(i)$).

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.2.87 Interface `get_vertical_viscosity`

physical unit: m^2/s

Retrieve scalar constant for vertical viscosity:

1. `get_vertical_viscosity()`: returns scalar `s [real]` with scalar constant vertical 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 its spatially varying contribution set using `CALL set_turbulent_v_viscosity`.

B.2.88 Interface `get_volume`

physical unit: m^3

Computes the volume for computational prisms:

1. `get_volume(i,k)`: returns scalar `s [real]` 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 scalar `s [real]` with the fluid volume above polygon i ($1 \leq i \leq N_p$);
3. `get_volume()`: returns scalar `s [real]` with the overall fluid volume.

The total number of polygons N_p can be retrieved from `get_nof_polygons`. $k_b(i)$ can be obtained from `get_bottom_prism(i)` and $k_t(i)$ from `get_top_prism(i)`.

B.2.89 Interface `get_wind_friction`

physical unit: —

Retrieve wind friction coefficient r_T at sides / edges:

1. `CALL get_wind_friction(j)`: returns scalar `s [real]` with r_T at the j-th side ($1 \leq j \leq N_s$);
2. `CALL get_wind_friction()`: array `a(`N_s`) [real]` is returned with r_T at all N_s sides, where `a(j)` contains r_T for the j-th side.

N_s can be determined by means of `get_nof_edges()`.

B.2.90 Interface `get_wind_stress`

physical unit: m^2 / s^2

Retrieve the normalized normal component of wind shear stress at sides / edges:

1. `get_wind_stress(j)`: return scalar `s [real]` for the j-th side ($1 \leq j \leq N_s$);
2. `get_wind_stress()`: array `a(`N_s`) [real]` is returned for all all N_s sides, where `a(j)` is the respective value for the j-th side.

N_s can be obtained from `get_nof_edges()`. The (real physical) normal component of the actual wind stress τ_T is obtained by multiplication with ρ_0.

B.2.91 Interface `get_wind_velocity`

physical unit: m / s

Retrieve components of the wind velocity (u_a,v_a) at sides / edges:

1. `get_wind_velocity(j)`: array `a(2) [real]` is returned for the j-th side ($1 \leq j \leq N_s$), where `a(1)` and `a(2)` correspond to the x- and y-components of the wind velocity (u_a,v_a);
2. `get_wind_velocity()`: array `b(`N_s`,2) [real]` is returned with wind velocity at all N_s sides, where `b(j,:)` gives the respective value for the j-th side.

N_s is obtained from `get_nof_egdes()`.

B.2.92 Interface `get_wind_velocity_x`

physical unit: m / s

Retrieve x-component u_a of the wind velocity at sides / edges:

1. `get_wind_velocity_x(j)`: returns scalar `s [real]` with the x-component u_a for the j-th side ($1 \leq j \leq N_s$);
2. `get_wind_velocity_x()`: array `a(`N_s`) [real]` is returned with x-component u_a at all N_s sides.

N_s is obtained from `get_nof_egdes()`.

B.2.93 Interface `get_wind_velocity_y`

physical unit: m / s

Retrieve y-component v_a of the wind velocity at sides / edges:

1. `get_wind_velocity_y(j)`: returns scalar `s [real]` with the y-component v_a for the j-th side ($1 \leq j \leq N_s$);
2. `get_wind_velocity_y()`: array `a(`N_s`) [real]` is returned with y-component v_a at all N_s sides.

N_s is obtained from `get_nof_egdes()`.

B.2.94 Interface `get_x_vertex_coordinate`

physical unit: m [origin]

Retrieve *x*-coordinate for vertices:

1. `get_x_vertex_coordinate(i)`: scalar `s [real]` is returned with the *x*-coordinate for the *i*-th vertex ($1 \leq i \leq N_v$);
2. `get_x_vertex_coordinate()`: array `a(`N_v`) [real]` 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.2.95 Interface `get_y_vertex_coordinate`

physical unit: m [origin]

Retrieve *y*-coordinate for vertices:

1. `get_y_vertex_coordinate(i)`: scalar `s [real]` is returned with the *y*-coordinate for the *i*-th vertex ($1 \leq i \leq N_v$);
2. `get_y_vertex_coordinate()`: array `a(`N_v`) [real]` 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.2.96 Interface `sediment`

physical unit: logical, no unit

Check if a specie is a sediment with a settling velocity w^s:

1. `sediment(m)`: returns scalar `s [log]` which is *true* if the *m*-th specie ($1 \leq m \leq N_c$) is a sediment with settling velocity w^s or *false* otherwise.

N_c can be determined from `get_nof_species()`.

Notice: Currently factors set using `CALL set_sediment` cannot be retrieved by an appropriate Get-function.

B.2.97 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[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"`.

B.2.98 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[log]` which is *true* if the ongoing simulation is performed with transport of scalar species or *false* otherwise.

The result of this function is dependent on user's input data in file `"untrim.inp"`.

B.3 Check routines

- check routines
  - grid structure and grid quality
    - `check_grid`
  - continuity
    - `check_continuity`

All check routines available to the user are subsequently described and listed in alphabetical order.

B.3.1 Subroutine `check_continuity`

This subroutine has no parameters. The routine can be called during each time step. The following checks are performed:

1. determine maximum error for computed water levels;
2. determine maximum volume error within a polygon;
3. determine 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.3.2 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 follwing checks are carried through:

1. determine smallest / largest area for all polygons;
2. determine smallest / largest length for all sides (edges);
3. determine smallest / largest distance between adjacent polygon centers for all edges;
4. compute statistics for all distances between adjacent polygon centers;
5. determine the number of centers which are not contained in their polygon;
6. compute maximum violation of orthogonality condition;
7. determine the number of edges with incorrect orientation (edges with *negative* distance between adjacent polygon centers);
8. stop execution if there are *negative* polygon areas;
9. stop execution in case of invalid sources and / or sinks locations.

B.4 User interface routines

In the following sections for each of the typical operations within the user interface routines the most important get- and set-functions are listed. These listings are in general not comprehensive, and the user may find that additional functions out of the vast amount of available functions (see section B.1 on page 1 and section B.2 on page 24) might be useful in case.

B.4.1 Subroutine `user_set_input_files`

This routine is called once during the initialization phase by the computational core of the software. The user has the possibility to set user-specific paths and names for the three different input files. The following routines and / or functions may be useful in this respect:

- path and name for the steering file (default is `"untrim.inp"`)
  - `set_input_file`
- path and name for the grid file (default is `"untrim.grd"`)
  - `set_grid_file`
- path and name for the sources and sinks file (default is `"untrim.srs"`)
  - `set_source_file`

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.4.2 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 inital conditions as required. The following routines and / or functions may be useful in this context:

- auxiliary functions
  - `get_bottom_face`
  - `get_bottom_prism`
  - `get_nof_faces`
  - `get_nof_layers`
  - `get_nof_polygons`
  - `get_nof_prisms`
  - `get_nof_species`
  - `get_top_face`
  - `get_top_prism`

- inquiry functions
  - `with_hydrodynamic_pressure`
  - `with_scalar_transport`
- hydrodynamic initial data
  - `CALL set_elevation`
  - `CALL set_horizontal_velocity`
  - `CALL set_velocity`
  - `CALL set_pressure`
- species initial data
  - `CALL set_concentration`
  - `CALL set_sediment`
- check routines for the grid
  - `CALL check_grid`

Notice: if one or several of the before mentioned set-routines are not used, the respective data will be initialized to zero.

B.4.3 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.

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:

- auxiliary functions
  - `get_bottom_face`
  - `get_bottom_prism`
  - `get_nof_layers`
  - `get_nof_boundary_polygons`
  - `get_nof_inflow_edges`
  - `get_time_step`
  - `get_top_face`
  - `get_top_prism`

- inquiry functions
  - `with_hydrodynamic_pressure`
  - `with_scalar_transport`
  - `get_face_height`
  - `get_prism_height`
  - `get_edge_center`
  - `get_polygon_center`
- hydrodynamic open boundary data (with *prescribed water level*)
  - `CALL set_elevation_bc`
  - `CALL set_pressure_bc`
- 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.

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:

- auxiliary functions
  - `get_nof_layers`
  - `get_nof_point_sources`
  - `get_nof_species`
  - `get_time_step`
- inquiry functions
  - `get_concentration`
  - `get_source_concentration`

  - `get_source_discharge`
  - `get_source_layer`
  - `get_source_polygon`
  - `with_scalar_transport`

- strength of sources and sinks
  - `CALL set_point_sources_discharge`
- concentration for sources and sinks
  - `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 surface. 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:

- auxiliary functions
  - `get_nof_edges`
  - `get_nof_polygons`
  - `get_time`
- inquiry functions
  - `get_depth_at_edge`
  - `get_edge_center`
  - `get_elevation`
- 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:

- auxiliary functions

  - get_nof_edges
  - get_nof_polygons
  - get_time_step

- inquiry functions
  - get_edge_center
  - get_polygon_center

- wind friction coefficient
  - CALL set_wind_friction

- wind speed
  - CALL set_wind_velocity

- atmospheric pressure
  - CALL set_atmospheric_pressure

Notice: if some of the aforementioned set-routines are never used respective atmospheric data will be initialized to default values (zero).

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:

- auxiliary functions
  - get_bottom_prism
  - get_nof_layers
  - get_nof_prisms
  - get_nof_species
  - get_time_step
  - get_top_prism

- inquiry functions
  - with_scalar_transport
  - get_concentration

- density
  - 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 (ν^v, ν^h) and diffusivities (K^v, K^h). They normally strongly depend on space and time. The following routines and / or functions may be useful in this respect:

- auxiliary functions
  - `get_bottom_face`
  - `get_bottom_prism`
  - `get_nof_layers`
  - `get_nof_polygons`
  - `get_nof_prisms`
  - `get_nof_species`
  - `get_top_face`
  - `get_top_prism`
- inquiry functions
  - `get_density`
  - `get_edge_center`
  - `get_face_height`
  - `get_gravity`
  - `get_horizontal_velocity`
  - `get_horizontal_velocity_x`
  - `get_horizontal_velocity_y`
  - `get_polygon_center`
  - `get_prism_height`
  - `with_scalar_transport`
- 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:

- auxiliary functions
  - `get_bottom_prism`
  - `get_nof_polygons`
  - `get_nof_species`
  - `get_time_step`
  - `get_top_prism`
- inquiry functions
  - `get_concentration`
  - `get_prism_height`
  - `sediment`
  - `get_settling_velocity`
  - `with_scalar_transport`
- 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:

- auxiliary functions

  - `get_bottom_prism`
  - `get_nof_layers`
  - `get_nof_polygons`
  - `get_nof_prisms`
  - `get_nof_species`
  - `get_top_prism`

- inquiry functions
  - `get_concentration`
  - `get_gravity`
  - `get_turbulent_v_viscosity`
  - `get_turbulent_v_diffusivity`
  - `sediment`

- settling velocity
  - `CALL set_sediment`
  - `CALL set_settling_velocity`

The settling velocity (set by means of `CALL set_settling_velocity`) will be muliplied 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_iterations`
- water volume
  - `get_volume`
- tracer mass
  - `get_mass`
- fluxes of tracer at the free surface and at the bottom
  - `get_surface_flux`
  - `get_bottom_flux`

- continuity of water volume
  - `CALL check_continuity`
- drying and wetting
  - `get_nof_wet_edges`
  - `get_nof_wet_faces`
  - `get_nof_wet_polygons`
  - `get_nof_wet_prisms`

B.4.4 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:

- auxiliary functions
  - `get_nof_faces`
  - `get_nof_layers`
  - `get_nof_point_sources`
  - `get_nof_polygons`
  - `get_nof_prisms`
  - `get_nof_species`
  - `get_time_step`
- inquiry functions
  - `with_hydrodynamic_pressure`
  - `with_scalar_transport`
- computational results
  - `get_bottom_stress`
  - `get_concentration`
  - `get_elevation`
  - `get_horizontal_velocity`
  - `get_horizontal_velocity_x`
  - `get_horizontal_velocity_y`
  - `get_mass`
  - `get_pressure`

  - get_velocity
  - get_vertical_velocity
  - get_wind_stress

- parameters and coefficients
  - get_bottom_alpha
  - get_bottom_beta
  - get_bottom_concentration
  - get_bottom_friction
  - get_density
  - get_horizontal_diffusivity
  - get_horizontal_viscosity
  - get_settling_velocity
  - get_surface_alpha
  - get_surface_beta
  - get_surface_concentration
  - get_turbulent_h_diffusivity
  - get_turbulent_h_viscosity
  - get_turbulent_v_diffusivity
  - get_turbulent_v_viscosity
  - get_vertical_diffusivity
  - get_vertical_viscosity
  - get_wind_friction

- boundary data
  - get_atmospheric_pressure
  - get_bottom_flux
  - get_surface_flux
  - get_wind_velocity
  - get_wind_velocity_x
  - get_wind_velocity_y

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 througout Table 2 on page 78 to Table 13 on page 89 the nomenclature shown in Table 1 on the next page applies.

argument	*description*
`i`	polygon index
`j`	edge (side) index
`k`	layer index
`l`	side / vertex index within polygon
`m`	specie index
`n`	source / sink location index
`n12(:)`	integer array of size 1 (hydrostatic) or 2 (non-hydrostatic)
`s,t`	scalar variables
`ac(:)`	data `ac(`N_z`)` for all layers N_z
`ac0(:)`	data `ac0(`$0:N_z$`)` for all layer interfaces N_z
`af(:),bf(:)`	data `af(`J_3`),bf(`J_3`)` for all faces J_3
`ai(:)`	data `ai(`N_p`)` for all polygons N_p
`ail(:,:)`	data `ail(`$N_p,4$`)` for all polygons N_p and all sides
`ai_b(:)`	data `ai_b(`N_p^*`)` for all boundary polygons N_p^*
`aim(:,:)`	data `aim(`N_p,N_c`)` for all polygons N_p and all species N_c
`aj(:),bj(:)`	data `aj(`N_s`),bj(`N_s`)` for all edges (sides) N_s
`al(:)`	data `al(`S_i`)` for all sides S_i of a polygon
`am(:)`	data `am(`N_c`)` for all species N_c
`an(:)`	data `an(`N_d`)` for all source / sink locations N_d
`anm(:,:)`	data `anm(`N_d,N_c`)` for all source / sink locations N_d and all species N_c
`ap(:)`	data `ap(`I_3`)` for all prisms I_3
`ap_b(:)`	data `ap_b(`I_3^*`)` for all boundary prisms I_3^*
`apm(:,:)`	data `apm(`I_3,N_c`)` for all prisms I_3 and all species N_c
`apm_b(:,:)`	data `apm_b(`I_3^*,N_c`)` for all boundary prisms I_3^* and all species N_c
`av(:)`	data `av(`N_v`)` for all vertices N_v
`c(:)`	coordinates `c(2)` for a single point
`ci(:,:)`	coordinates `ci(`$N_p,2$`)` for all polygons
`cj(:,:)`	coordinates `cj(`$N_s,2$`)` for all edges
`cv(:,:)`	coordinates `cv(`$N_v,2$`)` for all vertices
`st`	character string `st(LEN=80)`
`v(:)`	vector data `v(2)` for a single point
`vj(:,:)`	vector data `v(`$N_s,2$`)` for all edges
`vc(:,:)`	vector data `v(`$N_z,2$`)` for all layers N_z
`x,y,z`	coordinates

Table 1: Nomenclature for set-routines and get-functions.

interface name	*allowed sets of arguments*
set_atmospheric_pressure	(s|i,s|ai(:))
set_bottom_alpha	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_bottom_beta	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_bottom_concentration	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_surface_alpha	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_surface_beta	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_surface_concentration	(s|m,s|m,i,s|m,ai(:)|aim(:,:))
set_bottom_friction	(s|j,s|aj(:))
set_concentration	(s|m,s|m,i,s|m,i,k,s|m,ap(:)|m,i,ac(:)|apm(:,:))
set_concentration_bc	(s|m,s|m,i,s|m,i,k,s|m,ap_b(:)|m,i,ac(:)|apm_b(:,:))
set_density	(s|i,s|i,k,s|i,ac(:)|ap(:))
set_elevation	(s|i,s|ai(:))
set_elevation_bc	(s|i,s|ai_b(:))
set_flux_limiter	(st)
set_grid_file	(st)
set_horizontal_velocity	(s,t|j,s,t|j,k,s,t|j,ac(:),t|j,s,bc(:)|j,ac(:),bc(:)|... ...af(:),s|s,bf(:)|af(:),bf(:))
set_input_file	(st)

Table 2: Available interfaces for set-routines together with allowed arguments (set_atmospheric_pressure – set_input_file). For details about the nomenclature please refer to Table 1 on the page before.

interface name	*allowed sets of arguments*
set_inflow_bc	(s|j,s|j,k,s)
set_inflow_cc	(s|m,s|m,j,s|m,j,k,s)
set_point_sources_discharge	(s|n,s|an(:))
set_point_sources_concentration	(s|n,s|n,m,s|m,an(:)|anm(:,:))
set_pressure	(s|i,s|i,k,s|i,ac(:)|ap(:))
set_pressure_bc	(s|i,s|i,k,s|i,ac(:)|ap_b(:))
set_sediment	(s|am(:)|m,s)
set_settling_velocity	(s|i,s|i,k,s|i,ac(:)|ap(:))
set_source_file	(st)
set_turbulent_h_diffusivity	(s|j,s|j,k,s|j,ac(:)|af(:))
set_turbulent_h_viscosity	(s|j,s|j,k,s|j,ac(:)|af(:))
set_turbulent_v_diffusivity	(s|i,s|i,k,s|i,ac(:)|ap(:))
set_turbulent_v_viscosity	(s|i,s|i,k,s|i,ac(:)|ap(:))
set_velocity	(s|j,s|j,k,s|j,ac(:)|af(:))
set_wind_friction	(s|j,s|aj(:))
set_wind_velocity	(s,t|j,s,t|aj(:),t|s,bj(:)|aj(:),bj(:))

Table 3: Available interfaces for set-routines together with allowed arguments (set_inflow_bc – set_wind_velocity). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*				
get_adjacent_polygon	*arg*	(i,l)	(i)	()	
	resINT	i	al(:)	ail(:,:)	
get_atmospheric_pressure	*arg*	(i)	()	(x,y)	
	resREAL	s	ai(:)	s	
get_bottom_alpha	*arg*	(m,i)	(m)	()	m,x,y
	resREAL	s	ai(:)	aim(:,:)	s
get_bottom_beta	*arg*	(m,i)	(m)	()	m,x,y
	resREAL	s	ai(:)	aim(:,:)	s
get_bottom_concentration	*arg*	(m,i)	(m)	()	m,x,y
	resREAL	s	ai(:)	aim(:,:)	s
get_bottom_face	*arg*	(j)	()		
	resINT	s	aj(:)		
get_bottom_flux	*arg*	(m,i)	m	()	
	resREAL	s		am(:)	
get_bottom_friction	*arg*	(j)	()		
	resREAL	s	aj(:)		
get_bottom_prism	*arg*	(i)	()	x,y	
	resINT	s	ai(:)	s	
get_bottom_stress	*arg*	(j)	()		
	resREAL	s	aj(:)		

Table 4: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_adjacent_polygon – get_bottom_stress). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*						
get_concentration	*arg*	(m,i,k)	(m,i)	(m)	()	(m,x,y,z)	(m,x,y)
	resREAL	s	ac(:)	ap(:)	apm(:,:)	s	ac(:)
get_chezy	*arg*	()					
	resREAL	s					
get_density	*arg*	(i,k)	(i)	()	(x,y,z)	(x,y)	
	resREAL	s	ac(:)	ap(:)	s	ac(:)	
get_depth	*arg*	(i)	()	(x,y)			
	resREAL	s	ai(:)	s			
get_depth_at_edge	*arg*	(j)	()				
	resREAL	s	aj(:)				
get_dt_min	*arg*	()					
	resREAL	s					
get_dx	*arg*	()					
	resREAL	s					
get_dx_min	*arg*	()					
	resREAL	s					
get_dy	*arg*	(j)	()				
	resREAL	s	aj(:)				
get_dz_min	*arg*	()					
	resREAL	s					

Table 5: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_concentration – get_dzmin). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*			
get_edge_begin	*arg*	(j)	()	
	resINT	s	aj(:)	
get_edge_center	*arg*	(j)	()	
	resREAL	c(:)	cj(:,:)	
get_edge_end	*arg*	(j)	()	
	resINT	s	aj(:)	
get_elevation	*arg*	(i)	()	(x,y)
	resREAL	s	ai(:)	s
get_elevation_tolerance	*arg*	()		
	resREAL	s		
get_face_height	*arg*	(j,k)	(j)	()
	resREAL	s	ac(:)	af(:)
get_flux_limiter	*arg*	()		
	resCHAR	st		
get_gravity	*arg*	()		
	resREAL	s		
get_hland	*arg*	()		
	resREAL	s		
get_horizontal_diffusivity	*arg*	()		
	resREAL	s		

Table 6: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_edge_begin – get_horizontal_diffusivity). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*					
get_horizontal_velocity	*arg*	(x,y,z)	(x,y)	(x,y,z,i)	(x,y,i,k)	(x,y,i)
	resREAL	v(:)	vc(:,:)	v(:)	v(:)	vc(:,:)
get_horizontal_velocity_x	*arg*	(x,y,z)	(x,y)	(x,y,z,i)	(x,y,i,k)	(x,y,i)
	resREAL	s	ac(:)	s	s	ac(:)
get_horizontal_velocity_y	*arg*	(x,y,z)	(x,y)	(x,y,z,i)	(x,y,i,k)	(x,y,i)
	resREAL	s	ac(:)	s	s	ac(:)
get_horizontal_viscosity	*arg*	()				
	resREAL	s				
get_layer	*arg*	(z)				
	resINT	k				
get_iterations	*arg*	()				
	resINT	n12(:)				
get_layer_interface	*arg*	(k)	()			
	resREAL	s	ac0(:)			
get_left_polygon	*arg*	(j)	()			
	resINT	s	ai(:)			
get_location	*arg*	()				
	resCHAR	st				
get_mass	*arg*	(m,i,k)	(m,i)	(m)	()	
	resREAL	s	s	s	am(:)	

Table 7: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_horizontal_velocity – get_mass). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*			
get_maximum_iteration	*arg*	()		
	resINT	s		
get_nof_boundary_polygons	*arg*	()		
	resINT	s		
get_nof_edges	*arg*	(i)	()	
	resINT	s	ai(:)	
get_nof_faces	*arg*	(j)	()	
	resINT	s	s	
get_nof_inflow_edges	*arg*	()		
	resINT	s		
get_nof_internal_edges	*arg*	()		
	resINT	s		
get_nof_layers	*arg*	()		
	resINT	s		
get_nof_point_sources	*arg*	()		
	resINT	s		
get_nof_polygons	*arg*	()		
	resINT	s		
get_nof_prisms	*arg*	(i)	()	(x,y)
	resINT	s	s	s

Table 8: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_maximum_iteration – get_nof_prisms). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*			
get_nof_species	*arg*	()		
	resINT	s		
get_nof_vertices	*arg*	(i)	()	
	resINT	s	s	
get_nof_wet_edges	*arg*	(i)	()	
	resINT	s	s	
get_nof_wet_faces	*arg*	(j)	()	
	resINT	s	s	
get_nof_wet_polygons	*arg*	()		
	resINT	s		
get_nof_wet_prisms	*arg*	(i)	()	(x,y)
	resINT	s	s	s
get_polygon	*arg*	(x,y)		
	resINT	s		
get_polygon_area	*arg*	(i)	()	
	resREAL	s	ai(:)	
get_polygon_center	*arg*	(i)	()	(x,y)
	resREAL	c(:)	ci(:,:)	c
get_polygon_edge	*arg*	(i,l)	(i)	()
	resINT	s	al(:)	ail(:,:)

Table 9: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_nof_species – get_polygon_edge). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*					
get_polygon_vertex	*arg*	(i,l)	(i)	()		
	resINT	s	al(:)	ail(:,:)		
get_pressure	*arg*	(i,k)	(i)	()	(x,y,z)	(x,y)
	resREAL	s	ac(:)	ap(:)	s	ac(:)
get_pressure_tolerance	*arg*	()				
	resREAL	s				
get_prism_height	*arg*	(i,k)	(i)	()	(x,y,z)	(x,y)
	resREAL	s	ac(:)	ap(:)	s	ac(:)
get_radiation_time	*arg*	()				
	resREAL	s				
get_right_polygon	*arg*	(j)	()			
	resINT	s	aj(:)			
get_source_concentration	*arg*	(n,m)	(m)	()		
	resREAL	s	an(:)	anm(:)		
get_source_discharge	*arg*	(n)	()			
	resREAL	s	an(:)			
get_source_layer	*arg*	(n)	()			
	resINT	s	an(:)			
get_surface_flux	*arg*	(m,i)	m	()		
	resREAL	s		am(:)		

Table 10: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_polygon_vertex – get_surface_flux). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*							
get_source_polygon	*arg*	(n)	()					
	resINT	s	an(:)					
get_surface_alpha	*arg*	(m,i)	(m)	()	(m,x,y)			
	resREAL	s	ai(:)	aim(:,:)	s			
get_surface_beta	*arg*	(m,i)	(m)	()	(m,x,y)			
	resREAL	s	ai(:)	aim(:,:)	s			
get_surface_concentration	*arg*	(m,i)	(m)	()	(m,x,y)			
	resREAL	s	ai(:)	aim(:,:)	s			
get_settling_velocity	*arg*	(i,k)	(i)	()	(x,y,z)	(z,i)	(x,y)	(x,y,k)
	resREAL	s	ac0(:)	ap(:)	s	s	ac0(:)	s
get_theta	*arg*	()						
	resREAL	s						
get_time_step	*arg*	()						
	resREAL	s						
get_top_face	*arg*	(j)	()					
	resINT	s	aj(:)					
get_top_prism	*arg*	(i)	()	(x,y)				
	resINT	s	ai(:)	s				
get_turbulent_h_diffusivity	*arg*	(j,k)	(j)	()				
	resREAL	s	ac(:)	af(:)				

Table 11: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_source_polygon – get_turbulent_h_diffusivity). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*							
get_turbulent_h_viscosity	*arg*	(j,k)	(j)	()				
	resREAL	s	ac(:)	af(:)				
get_turbulent_v_diffusivity	*arg*	(i,k)	(i)	()				
	resREAL	s	ac(:)	ap(:)				
get_turbulent_v_viscosity	*arg*	(i,k)	(i)	()				
	resREAL	s	ac(:)	ap(:)				
get_velocity	*arg*	(j,k)	(j)	()				
	resREAL	s	ac(:)	af(:)				
get_vertex_coordinates	*arg*	(i)	()					
	resREAL	c(:)	cv(:,:)					
get_vertical_diffusivity	*arg*	()						
	resREAL	s						
get_vertical_velocity	*arg*	(i,k)	(i)	()	(x,y,z)	(x,y,k)	(z,i)	(x,y)
	resREAL	s	ac(:)	ap(:)	s	s	s	ac(:)
get_vertical_viscosity	*arg*	()						
	resREAL	s						
get_volume	*arg*	(i,k)	(i)	()				
	resREAL	s	s	s				
get_wind_friction	*arg*	(j)	()					
	resREAL	s	aj(:)					

Table 12: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_turbulent_v_diffusivity – get_wind_friction). For details about the nomenclature please refer to Table 1 on page 77.

interface name	*allowed sets of arguments and results*		
`get_wind_stress`	*arg*	`(j)`	`()`
	resREAL	`s`	`aj(:)`
`get_wind_velocity`	*arg*	`(j)`	`()`
	resREAL	`v(:)`	`vj(:,:)`
`get_wind_velocity_x`	*arg*	`(j)`	`()`
	resREAL	`s`	`aj(:)`
`get_wind_velocity_y`	*arg*	`(j)`	`()`
	resREAL	`s`	`aj(:)`
`get_x_vertex_coordinate`	*arg*	`(l)`	`()`
	resREAL	`s`	`av(:)`
`get_y_vertex_coordinate`	*arg*	`(l)`	`()`
	resREAL	`s`	`av(:)`
`sediment`	*arg*	`(m)`	
	resLOG	`s`	
`with_hydrodynamic_pressure`	*arg*	`()`	
	resLOG	`s`	
`with_scalar_transport`	*arg*	`()`	
	resLOG	`s`	

Table 13: Available interfaces for get-functions together with allowed arguments (*arg*) and returned results (*res*) (get_wind_stress–with_scalar_transport). For details about the nomenclature please refer to Table 1 on page 77.

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 $\left[1, N_p^*\right]$.
- 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 $[N_{s_i}+1, N_s]$;
  3. if some open boundary edges are sides with prescribed flow (Dirichlet boundary condition), these edges must be brought to the head of the tail $\left[N_{s_i}+1, N_{s_f}\right]$.

  Notice: If a Dirichlet boundary condition is not used at all, $N_{s_f} = N_{s_i}$ must hold true.

D.1.2 Short description

The grid file is sub-divided into the following sections:

1. Fortran `NAMELIST`-data section `LISTGRD`:
   (a) `ne`: number of polygons N_p;
   (b) `ns`: number of sides N_s;
   (c) `nv`: number of vertices N_v;

(d) `nbc`: number of boundary polygons N_p^*;

(e) `nr`: number of *red* polygons N_{p_r} (red-black-sorting);

(f) `nsi`: number of internal sides N_{s_i};

(g) `nsf`: number of last side with prescribed flow N_{s_f};

(h) `angle`: (mean) geographic latitude Φ;

(i) `hland`: depth for permanently dry land h_L;

(j) `location`: text string to describe the model domain.

Please notice that h_L must be chosen in such a way that for the water level $\max \eta_i < -h_L$ always holds true.

2. list of cartesian coordinates x_l, y_l ($l = 1, N_v$) for all vertices of the grid; each line of input contains the following data for a vertex:

   (a) `x`: x-coordinate;

   (b) `y`: y-coordinate.

3. list of data related to the N_p polygons in the grid; each input line contains the following data for one polygon:

   (a) `ks`: number of sides / vertices S_i ($3 \leq S_i \leq 4$) of the polygon;

   (b) `xc`: x-coordinate for the center of the polygon;

   (c) `yc`: y-coordinate for the center of the polygon;

   (d) list of pairs (`nen`,`is`) for each vertex / side ($l = 1, S_i$) of the polygon:

      i. `nen`: index for the l-th vertex;

      ii. `is`: index for the l-th side / edge.

   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) `hu`: depth h;

   (b) `jb`: index for the first vertex;

   (c) `jb`: index for the second vertex;

   (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).

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.

```
l01 &LISTGRD
l02 NE  = 5154
l03 NS  = 8166
l04 NV  = 3013
l05 NBC = 6
l06 NR  = 1621
l07 NSI = 6572
l08 NSF = 6578
l09 ANGLE = 0
l10 HLAND = -10000
l11 LOCATION = "channel turn with rectangular cross section"
l12 /
l13 4500.0 440.0
l14 4500.0 460.0
l15 4500.0 480.0
l16 ... cont.
l17 3 4495.0 450.0 2  4 2549 2    1 3
l18 3 4495.0 470.0 2  5    3 7 2549 4
l19 3 4495.0 490.0 4 10 1460 8    3 9
l20 ... cont.
l21 -10001.0 1  870    0 5143
l22     10.0 1 2549 5143    1
l23 -10001.0 1    2    1    0
l24     10.0 2 2549    1    2
l25 -10001.0 2    3    2    0
l26     10.0 3 1304 5144 5145
l27     10.0 3 2549    2 5144
l28     10.0 3 1460 5145    3
l29 -10001.0 3    4    3    0
l30     10.0 4 1460    3    4
l31 ... cont.
```

A few comments on the short example shown above:

l01 – l11: contents of the namelist LISTGRD; the ANGLE was set to be 0, which also means that no Coriolis effects will be taken into account in the simulation.

l05 – l05: there are six polygons with prescribed water level;

l07 – l08: there are six sides with prescribed flow;

l12 – l12: terminator "/"for the namelist section.

l13 – l13: *x*- and *y*-coordinate of the first vertex.

l14 – l16: coordinates for the remaining vertices.

l17 – l17: polygon-oriented data for the first polygon (triangle);

l18 – l20: polygon-oriented data for the remaining polygons.

l21 – l21: edge-oriented data for the first edge; the depth of the first edge (−10001.0) is smaller than HLAND, which means that this edge represents permanently dry land (e. g. true for solid walls at the boundaries); on the other hand the left neighbour polygon is not available for this edge, whereas the right polygon index corresponds to 5143.

l22 – l31: edge-oriented data for the remaining 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 LISTINP: noli = 0 slip = 1.0
   (a) nk: 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) nq: steering parameter (0 = use hydrostatic pressure approximation, 1 = hydrodynamic pressure);
   (c) nsp: number of dissolved species (e. g. salinity, temperature, suspended sediment);
   (d) epsi: tolerance ε_η for the free-surface iterative solver ($10^{-8} \leq \varepsilon_\eta \leq 0.1$);
   (e) qpsi: tolerance ε_q for the non-hydrostatic pressure iterative solver ($10^{-8} \leq \varepsilon_\eta \leq 0.1$);
   (f) maxiter: maximum number of iterations n_ε for the iterative solvers;
   (g) theta: impliciteness factor θ ($0.5 < \theta \leq 1.0$); recommended value $\theta = 0.6$;
   (h) delt: numerical time step Δt;
   (i) dxmin: minimum allowed distance $\delta_{\min}$ ($\delta_{\min} > 0.0$) between polygon centers;
   (j) dzmin: minimum allowed water depth $H_{\min}$ ($H_{\min} > 0.0$);
   (k) dtmin: to solve for tracer transport, the time step Δt is automatically broken into a smaller or larger number of substeps $\Delta\tau^i$; if M_s is the current number of substeps, the substepping procedure will continue as long as $\Delta t - \sum_1^{M_s} \Delta\tau^i >$ dtmin holds;
   (l) hvis: horizontal molecular viscosity ν_m^h ($\nu_m^h \geq 0.0$);
   (m) vvis: vertical molecular viscosity ν_m^v ($\nu_m^v \geq 0.0$);
   (n) hdif: horizontal molecular diffusivity K_m^h ($K_m^h \geq 0.0$);
   (o) vdif: vertical molecular diffusivity K_m^v ($K_m^v \geq 0.0$);

(p) `trx`: inverse of the relaxation time at open boundaries;

(q) `cz`: Chezy bottom friction coeffizcient C_z ($C_z > 0.0$);

(r) `noli`: switch non-linear advective terms on (noli= 1) or off (noli= 0);

(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 `nk` was chosen, `nk-1` level surfaces must be prescribed from top to bottom.

Notice: The use of parameter `epsi` can be optimized by checking the influence on results from `CALL check_continuity`.

D.2.2 Example file

The following lines show an example for the input data file `untrim.inp`. The line-numbers `l??` indicated are not part of the input data file.

```
l01 &LISTINP
l02 NK = 10
l03 NQ = 1
l04 NSP = 1
l05 EPSI = 1.00E-04
l06 QPSI = 1.00E-04
l07 MAXITER = 1000
l08 THETA = 0.6
l09 DELT = 0.25
l10 DXMIN = 5.00000E-02
l11 DZMIN = 5.00000E-03
l12 DTMIN = 1.00000E-04
l13 HVIS = 0.0
l14 VVIS = 0.0
l15 HDIF = 0.0
l16 VDIF = 0.0
l17 TRX = 1000000000.0
l18 CZ = 50.0
l19 NOLI=1
l20 SLIP=1.0
l21 /
l22       -2.00000
l23       -1.00000
l24        0.00000
l25        1.00000
l26        2.00000
l27        4.00000
```

```
l28          6.00000
l29          8.00000
l30         10.00000
```

A few comments on the short example shown above:

l02 – l02: ten vertical layers are prescribed in this case, which means that the vertical flow profile can be resolved.

l03 – l03: the pressure will be computed during the simulation because the pressure is assumed to be non-hydrostatic.

l04 – l04: one dissolved specie (e. g. temperature) is taken into account.

l05 – l06: tolerances for the iterative solvers.

l07 – l07: maximum number of iterations for the iterative solvers.

l08 – l08: an impliciteness factor of 0.6 guarantees numerical stability.

l09 – l09: small time steps are necessary if the time scales of the physical processes are also small.

l10 – l10: minimum horizontal distance between polygon centers.

l11 – l11: minimum allowed water depth.

l12 – l12: minimum substep size.

l13 – l16: molecular viscosities are set to zero in this situation (inviscid case).

l17 – l17: 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.

l18 – l18: default bottom friction coefficient.

l19 – l19: non-linear advective terms are switched on;

l20 – l20: free slip condition for velocities at lateral boundaries;

l21 – l21: terminator ”/”for the namelist section.

l22 – l30: 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 assumend ($N_d = 0$).

D.3.2 Example file

The following lines show an example for the source / sink data file `untrim.srs`. The line-numbers `l??` indicated are not part of the source / sink data file.

```
l01 3
l02 1254 2
l03 1254 3
l04 3140 3
```

A few comments on the short example shown above:

l01 – l01: three sources / sinks are prescribed;

l02 – l03: the first two sources / sinks are located in the same polygon but at different depth (within layers two and three);

l04 – l04: the third source / sink is located in a different polygon (within layer three).