TC-Toolbox for MATLAB® Documentation

Release 2024a

Thermo-Calc Software AB

Dec 08, 2023

CONTENTS

QUICK INSTALLATION GUIDE

This guide helps you to get a working TC-Toolbox for MATLAB® installation. It is only a short guideline, please refer to the **Thermo-Calc Installation Guides** for more details if required.

The present documentation is also included in your installation as a PDF-file. In the Thermo-Calc menu, select **Help → Manuals Folder**. Then double-click to open the **Software Development Kits (SDKs)** folder.

Note: A license is required to run TC-Toolbox for MATLAB®.

Note: TC-Toolbox for MATLAB® is available for Windows.

1.1 Installing TC-Toolbox for MATLAB®

1.1.1 Automatic Installation of TC-Toolbox

When the following conditions are met, Thermo-Calc automatically installs the TC-Toolbox for MATLAB® component on your computer.

1. MATLAB® is already installed.

2. There is only one Windows user on the computer where TC-Toolbox is being installed.

> **Note:** Administrator privileges are needed when you start the Thermo-Calc installer.

Then follow the regular installation instructions for Thermo-Calc, choosing whether you use a Standalone or Network installation.

1.1.2 Manual Installation of TC-Toolbox

The installation is not automatic if:

- There are multiple Windows users on the same machine, or

- The installer cannot find the directory path to the MATLAB® installation.

1.1.2.1 Multiple Windows Users on Same Machine

If there are multiple Windows users on the same machine, then the following manual steps are done at the end of the automatic installation.

1. A message at the end of the Thermo-Calc installation process displays with instructions.

2. An Explorer window automatically opens to this folder `C:\Users\<user>\Documents\Thermo-Calc\2024a\SDK\TC-Toolbox-MATLAB`.

3. Double-click the `InstallTCToolboxMATLAB.cmd` file to finalize the process. This briefly launches MATLAB® and installs TC-Toolbox.

1.1.2.2 Installer Cannot Find the MATLAB® Installation Directory

1. Start the version of MATLAB® that you want to install TC-Toolbox in.

2. Open and run the script `C:\Users\<user>\Documents\Thermo-Calc\2024a\SDK\TC-Toolbox-MATLAB\setupTCToolbox.m`.

For more information, se the more detailed instructions given in the **Thermo-Calc Installation Guides**.

1.1.3 Check the Installation

To check if the installation was successful start MATLAB® and run, for example, the diagnostics script, which is located in the folder `C:\Users\<user>\Documents\Thermo-Calc\2024a\SDK\TC-Toolbox-MATLAB\Examples\Miscellaneous`.

Alternatively open the **Add-Ons** menu (in the **HOME** tab) in MATLAB® and choose **Manage Add-Ons**. If the toolbox is installed it will be included in this list.

1.2 Uninstalling TC-Toolbox for MATLAB®

If you are logged in as a **user with administrator rights, and have Thermo-Calc installed for this user**, then the uninstallation is automatically done at the same time as a full Thermo-Calc uninstallation. Otherwise a manual step is required:

Note: To uninstall the TC-Toolbox for MATLAB® if it is not uninstalled by the Thermo-Calc uninstaller, start MATLAB® and select (in the **HOME** tab) **Manage Add-Ons** from the **Add-Ons** menu. Right-click TC-Toolbox and choose **Uninstall**.

ARCHITECTURE OVERVIEW

TC-Toolbox contains classes of these types:

- `TCToolbox` – this is where you start with general settings.
- `SystemBuilder` and `System` – where you choose database and elements etc.
- `Calculation` – where you choose and configure the calculation.
- `Result` – where you get the results from a calculation you have run.

2.1 TCToolbox

This is the starting point for all TC-Toolbox usage.

You can think of this as the start of a "wizard".

You use it to select databases and elements and then in the next step, configure the system.

Example:

```
import tc_toolbox.*

session = TCToolbox();
session.select_database_and_elements(...
% e.t.c.
% ...
```

Note: When your MATLAB® script runs a row like this:

```
session = TCToolbox();
```

a process running a calculation server starts. Your code, via TC-Toolbox, uses socket communication to send and receive messages to and from that server.

When you remove the variable *session* from the MATLAB® workspace, the calculation server automatically shuts down, and all temporary files are deleted.

Note: You can set up a folder location to re-use results from saved calculations. This folder can be a network folder and shared by many users. This is done using the method `set_cache_folder()`.

```
import tc_toolbox.*

session = TCToolbox();
session.set_cache_folder("cache")
```

Once the cache folder is created, if a previous TC-Toolbox calculation has run with the same cache folder and **exactly** the same system and calculation settings, the calculation is not re-run. Instead the result is automatically loaded from disk.

It is also possible to explicitly save and load results.

Example:

```
import tc_toolbox.*

session = TCToolbox();
%... diffusion calculation (could be any calculation type)
calculation_result.save_to_disk('path to folder')
%...
loaded_result = start.load_result_from_disk().diffusion('path to folder')
```

2.2 SystemBuilder and System

A `SystemBuilder` is returned when you have selected your database and elements in `TCToolbox`.

The `SystemBuilder` lets you further specify your system, for example the phases that should be part of your system.

Example:

```
import tc_toolbox.*

session = TCToolbox();
start.select_database_and_elements("ALDEMO", ["Al", "Sc"]).select_phase("FCC_A1")
% e.t.c
```

When all configuration is done, you call `get_system()` which returns an instance of a `System` class. The `System` class is immutable and cannot be changed. If you want to change the database, elements, or something else, you can:

- change the `SystemBuilder` and call `get_system()` again, or
- create a new `SystemBuilder` and call `get_system()`.

From the `System` you can create one or more calculations, which is the next step in the "wizard".

Note: You can use the same `System` object to create several calculations.

2.3 Calculation

All available calculation types are set up in a similar way, some calculations have many settings. But default values are used where it is applicable, and are overridden if you specify something different.

Tip: Review the TC-Toolbox examples included with the Thermo-Calc installation to see how calculations are used for various solutions.

When you have configured your calculation you call `calculate()` to start the actual calculation. That returns a `Result`, which is the next step.

2.3.1 Single Equilibrium Calculations

In single equilibrium calculations you need to specify the correct number of conditions, depending on how many elements your `System` contains.

This is done by calling `set_condition()`.

An important difference from other calculations is that single equilibrium calculations have two functions to get result values.

The `calculate()` method, which gives a `SingleEquilibriumTempResult`, is used to get actual values. This result is *temporary*, meaning that if you run other calculations or rerun the current one, the resulting object no longer gives values corresponding to the first calculation.

This is different from how other calculations work. If you want a `Result` that you can use *after* running other calculations, you need to call `calculate_with_state()`, which returns a `SingleEquilibriumResult`.

Note: `calculate()` is the recommended function and works in almost all situations. Also it has *significantly* better performance than `calculate_with_state()`.

Example:

```
import tc_toolbox.*

session = TCToolbox();

sys = session.select_database_and_elements("FEDEMO", ["Fe", "Cr", "C"]).get_system();
calculation = sys.with_single_equilibrium_calculation()...
                .set_condition(ThermodynamicQuantity.temperature(), 2000.0)...
                .set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("Cr"),
↪ 0.1)...
                .set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"),␣
↪0.01)...
                .calculate();

gibbs_energy = calculation.get_value_of("G")
```

2.3.2 Batch Equilibrium Calculations

Batch equilibrium calculations are used when you want to do many single equilibrium calculations and it is known from the beginning which result values are required from the equilibrium. This is a vectorized type of calculation that can reduce the overhead from MATLAB® and TC-Toolbox.

Tip: The performance of batch equilibrium calculations can be significantly better than looping and using single equilibrium calculations **if the actual Thermo-Calc calculation is fast**. There is little advantage if the Thermo-Calc equilibrium calculations take a long time (typically for large systems and databases).

Example:

```
import tc_toolbox.*

session = TCToolbox();
session.set_cache_folder("_cache");

system_builder = session.select_database_and_elements("NIDEMO", ["Ni", "Al", "Cr"]);
system_builder.without_default_phases();
system_builder.select_phase('BCC_A2');
sys = system_builder.get_system();
batch_calculation = sys.with_batch_equilibrium_calculation();

batch_calculation.set_condition("T", 800);
batch_calculation.set_condition("X(Al)", 1E-2);
batch_calculation.set_condition("X(Cr)", 1E-2);
batch_calculation.disable_global_minimization();

list_of_x_Al = linspace(1e-4, 10e-2, 10);
list_of_x_Cr = linspace(1e-4, 15e-2, 10);
list_of_density = [];
equilibria = {};

i = 1;
for x_Al = list_of_x_Al
    for x_Cr = list_of_x_Cr
        equilibria{i} = {{"X(Al)", x_Al} {"X(Cr)", x_Cr}};
        i = i+1;
    end
end

batch_calculation.set_conditions_for_equilibria(equilibria);

results = batch_calculation.calculate(["BM", "VM"], 100);

masses = results.get_values_of("BM");
volumes = results.get_values_of('VM');
density = 1e-3 * masses ./ volumes
```

2.3.3 Precipitation Calculations

All the configuration settings for the *Precipitation Calculator* in Graphical Mode are available for this calculation. However, you must at least enter a matrix phase, a precipitate phase, temperature, simulation time, and compositions.

Example:

```
import tc_toolbox.precipitation.*
import tc_toolbox.*

session = TCToolbox();
session.set_cache_folder("_cache");

system_builder = session.select_thermodynamic_and_kinetic_databases_with_elements("ALDEMO
↪", "MALDEMO", ["Al","Sc"]);
sys = system_builder.get_system();

precipitationCalculation = sys.with_isothermal_precipitation_calculation();
precipitationCalculation.set_composition("Sc", 0.18);
precipitationCalculation.set_temperature(623.15);
precipitationCalculation.set_simulation_time(1e5);
precipitationCalculation.with_matrix_phase(MatrixPhase("FCC_A1")...
                                    .add_precipitate_phase(PrecipitatePhase("AL3SC
↪")));

result = precipitationCalculation.calculate();
[time, meanRadius] = result.get_mean_radius_of("AL3SC");
```

2.3.4 Scheil Calculations

All Scheil calculation settings available in Graphical Mode (using the *Scheil Calculator*) or Console Mode (using the Scheil module) are available for this calculation. The minimum you need to specify are the elements and compositions. Everything else is set to a default value.

Example:

```
import tc_toolbox.*

session = TCToolbox();
sys = session.select_database_and_elements("FEDEMO", ["Fe", "C"]).get_system();
temperature_vs_mole_fraction_of_solid = sys.with_scheil_calculation()...
                                    .set_composition("C", 0.3)...
                                    .calculate()...
                                    .get_values_of(ScheilQuantity.temperature(),..
↪.
                                            ScheilQuantity.mole_fraction_
↪of_all_solid_phases());
```

2.3.5 Property Diagram Calculations

For the property diagram (step) calculation, everything that you can configure in the *Equilibrium Calculator* when choosing *One axis* in Graphical Mode can also be configured in this calculation. In Console Mode the property diagram is created using the Step command. The minimum you need to specify are elements, conditions, and the calculation axis. All other settings use the default values unless specified otherwise.

Example:

```
import tc_toolbox.*
import tc_toolbox.step_or_map_diagrams.*

session = TCToolbox();
    property_diagram = session...
            .select_database_and_elements("FEDEMO", ["Fe", "C"])...
            .get_system()...
            .with_property_diagram_calculation()...
                .with_axis(CalculationAxis(ThermodynamicQuantity.temperature())...
                            .set_min(500)...
                            .set_max(3000))...
                .set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"),␣
→0.01)...
                .calculate()...
                .get_values_grouped_by_stable_phases_of(ThermodynamicQuantity.
→temperature(),...
                                            ThermodynamicQuantity.volume_
→fraction_of_a_phase("ALL"));
```

2.3.6 Phase Diagram Calculations

For the phase diagram (map) calculation, everything that you can configure in the *Equilibrium Calculator* when choosing *Phase diagram* in Graphical Mode can also be configured in this calculation. In Console Mode the phase diagram is created using the Map command. The minimum you need to specify are elements, conditions, and two calculation axes. All other settings use the default values unless specified otherwise.

Example:

```
import tc_toolbox.*
import tc_toolbox.step_or_map_diagrams.*

session = TCToolbox();
    property_diagram = session...
            .select_database_and_elements("FEDEMO", ["Fe", "C"])...
            .get_system()...
            .with_phase_diagram_calculation()...
                .with_first_axis(CalculationAxis(ThermodynamicQuantity.temperature())...
                            .set_min(500)...
                            .set_max(3000))...
                .with_second_axis(CalculationAxis(ThermodynamicQuantity.mole_fraction_of_
→a_component("C"))...
                            .set_min(0)...
                            .set_max(1))...
                .set_condition(ThermodynamicQuantity.mole_fraction_of_a_component("C"),␣
```

(continues on next page)

```
↪0.01)...
                .calculate()...
                .get_values_grouped_by_stable_phases_of(ThermodynamicQuantity.mass_
↪fraction_of_a_component("C"),...
                                                    ThermodynamicQuantity.
↪temperature());
```

2.3.7 Diffusion Calculations

For diffusion calculations, everything that you can configure in the *Diffusion Calculator* can also be configured in this calculation. The minimum you need to specify are elements, temperature, simulation time, a region with a grid and width, a phase, and an initial composition.

Example:

```
import tc_toolbox.diffusion.*
import tc_toolbox.*

session = TCToolbox();

tc_system = session...
    .select_thermodynamic_and_kinetic_databases_with_elements("FEDEMO", "MFEDEMO", ["Fe",
↪ "Ni"])...
    .get_system();

calculator = tc_system...
                .with_isothermal_diffusion_calculation()...
                .set_temperature(1400.0)...
                .set_simulation_time(108000.0)...
                .add_region(Region("Austenite")...
                        .set_width(100E-6)...
                        .with_grid(CalculatedGrid.linear()...
                                .set_no_of_points(50))...
                        .with_composition_profile(CompositionProfile()...
                                            .add("Ni",␣
↪ElementProfile.linear(10.0, 50.0)))...
                        .add_phase("FCC_A1"));

results = calculator.calculate();

[distance, mass_frac_ni] = results.get_mass_fraction_of_component_at_time("Ni",␣
↪SimulationTime.LAST);
```

2.3.8 Property Model Calculations

For Property Model calculations, all the configuration settings for the *Property Model Calculator* in Graphical Mode are available for this calculation. The minimum you need to specify are elements, composition, and which Property Model you want to use.

Example:

```
import tc_toolbox.*

session = TCToolbox();
"Available property models: " + session.get_property_models()

property_model = session...
                    .select_database_and_elements("FEDEMO", ["Fe", "C"])...
                    .get_system()...
                    .with_property_model_calculation("Driving Force")...
                    .set_composition("C", 1.0)...
                    .set_argument("matrix", "LIQUID")...
                    .set_argument("precipitate", "GRAPHITE");

"Available arguments: " + property_model.get_arguments()
result = property_model.calculate();

"Available result quantities: " + result.get_result_quantities()
driving_force = result.get_value_of("normalizedDrivingForce")
```

2.3.9 Material to Material Calculations

Material to Material calculations are generally regular single equilibrium, property diagram or phase diagram calculations but they are specialised to handle the mixture of two materials A and B. Everything that you can configure in the Material to Material Calculator in Graphical Mode can also be configured in this calculation. The minimum required configuration is shown below for a Property diagram calculation for varying amount of material B. The other calculators (single fraction of material B and phase diagram calculations) are configured in a similar way.

Example:

```
import tc_toolbox.*
import tc_toolbox.material_to_material.*;

independent_elements = ["Cr", "Ni"];
a_comp = [10.0, 15.0];
b_comp = [15.0, 10.0];

activity_elements = ["C"];
activities = [0.1];

session = TCToolbox();

material_to_material_property_diagram = session...
    .select_database_and_elements("FEDEMO", ["Fe", "Cr", "Ni", "C"])...
    .get_system()...
    .with_material_to_material()...
```

(continues on next page)

```matlab
        .with_property_diagram_calculation()...
        .set_material_a(containers.Map(independent_elements, a_comp), "Fe")...
        .set_material_b(containers.Map(independent_elements, b_comp), "Fe")...
        .set_activities(containers.Map(activity_elements, activities))...
        .with_constant_condition(ConstantCondition.temperature(800 + 273.15))...
        .with_axis(MaterialToMaterialCalculationAxis.fraction_of_material_b());

result = material_to_material_property_diagram.calculate();
data = result.get_values_grouped_by_quantity_of(...
    Constants.MATERIAL_B_FRACTION,...
    ThermodynamicQuantity.volume_fraction_of_a_phase(Constants.ALL_PHASES));

for k = data.keys()
    group = data(k{1});
    fractions_of_b = group.get_x();
    volume_fraction_of_phase = group.get_y();
    phase_name = group.get_label();
end
```

2.3.10 Process Metallurgy Calculations

Process Metallurgy calculations are specialized to support the convenient handling of component-based additions (i.e., slag compositions such as *50% Al2O3 - 30% CaO - 20% SiO2*), provide tailor-made result quantities, a framework for developing kinetic process simulations, and more useful features.

There are two distinct type of calculations:

- *+tc_toolbox.+process_metallurgy.+equilibrium.EquilibriumCalculation*: isothermal and adiabatic equilibrium calculations

- *+tc_toolbox.+process_metallurgy.+process.ProcessSimulationCalculation*: a kinetic process simulation framework, based an *Effective Equilibrium Reaction Zone* (EERZ) approach

Equilibrium calculation example:

Equilibrium calculations are useful in a large large of situations when considering the kinetics of a process is unnecessary.

```matlab
import tc_toolbox.process_metallurgy.base.*;
import tc_toolbox.process_metallurgy.equilibrium.*;
import tc_toolbox.*

session = tc_toolbox.TCToolbox();

metal = EquilibriumAddition(containers.Map(["Fe", "C", "Si"], {NaN, 4.5, 1.0}), 100e3,␣
↪1650 + 273.15);
slag = EquilibriumAddition(containers.Map(["CaO", "Al2O3"], {75, 25}), 3e3, 1600 + 273.
↪15);
gas = EquilibriumGasAddition(containers.Map({'O2'}, {100}), 1000, GasAmountUnit.NORM_
↪CUBIC_METER);
calc = session.with_metallurgy().with_adiabatic_equilibrium_calculation(ProcessDatabase.
↪OXDEMO);
```

```
(calc...
 .add_addition(metal)...
 .add_addition(slag)...
 .add_addition(gas));

result = calc.calculate();

disp("Stable phases:")
disp(result.get_stable_phases())
disp("Temperature: " + result.get_temperature() + " K");
```

Process simulation example:

TC-Toolbox is providing a *framework* for modelling in principle any process in metallurgy, especially steel-making. It is up to the user to actually develop a concrete model for the process in question. The framework is in the current release limited to one reaction zone connecting two bulk zones. These bulk zones are typically the steel melt and the top slag, but not limited to that. The framework in its current version has proven to be useful to model industrial ladle furnaces, AOD- and VOD-converters and more. Process features such as heating and cooling, heat transfer between the bulk zones, inclusion formation and their flotation, etc., can be modelled.

This is a very simplified minimal but complete model mimicking a BOF process:

```
import tc_toolbox.process_metallurgy.base.*;
import tc_toolbox.process_metallurgy.process.*;
import tc_toolbox.*

session = tc_toolbox.TCToolbox();


calc = (session.with_metallurgy()...
        .with_adiabatic_process_calculation(ProcessDatabase.OXDEMO)...
        .set_end_time(15 * 60));

steel_zone = MetalBulkZone(7800);
slag_zone = SlagBulkZone(4500);

steel_zone.add_addition(SingleTimeAddition(containers.Map(["Fe", "C", "Si"], {NaN, 4.5,␣
→1.0}), 120e3,...
                                          1600 + 273.15), 0);
slag_zone.add_addition(SingleTimeAddition(containers.Map(["CaO", "SiO2"], {75, 25}), 1.
→2e3,...
                                          1500 + 273.15,...
                                          CompositionUnit.MOLE_PERCENT), 0);

steel_zone.add_continuous_addition(ContinuousGasAddition(containers.Map({'O2'}, {100}),␣
→1,...
                                          GasRateUnit.NORM_CUBIC_METER_
→PER_SEC));

calc.with_reaction_zone(ReactionZone(10.0,...
                                     steel_zone, 1.0e-5,...
                                     slag_zone, 1.0e-6));
```

```
result = calc.calculate();

disp("Stable phases in the steel melt:")
disp(result.get_stable_phases('metal'))
disp("C-content in steel vs. time:")
compositions = result.get_composition_of_phase_group('metal', PhaseGroup.ALL_METAL);
disp(compositions('C'))
```

2.4 Result

All calculations have a method called `calculate()` that starts the calculations and when finished, returns a `Result`.

The `Result` classes have very different methods, depending on the type of calculation.

The `Result` is used to get numerical values from a calculation that has run.

The `Result` can be saved to disk by the method `save_to_disk()`.

Previously saved results can be loaded by the method `load_result_from_disk()` on the `SetUp` class.

Example:

```
% code above sets up the calculation
r = calculation.calculate()
time, meanRadius = r.get_mean_radius_of("AL3SC")
```

The `Result` objects are completely independent from calculations done before or after they are created. The objects return valid values corresponding to the calculation they were created from, for their lifetime. The only exception is if you call `calculate()` and not `calculate_with_state()` on a single equilibrium calculation.

As in the following example you can mix different calculations and results, and use old results after another calculation has run.

Example:

```
% ...
% some code to set up a single equilibrium calculation
% ...

single_eq_result = single_eq_calculation.calculate_with_state()

% ...
% some code to set up a precipitation calculation
% ...

prec_result = precipitation_calculation.calculate()

% ...
% some code to set up a Scheil calculation
% ...

scheil_result = scheil_calculations.calculate()

% now it is possible to get results from the single equilibrium calculation,
```

```
% without having to re-run it (because it has been calculated with saving of the state)

gibbs = single_eq_result.get_value_of("G")
```
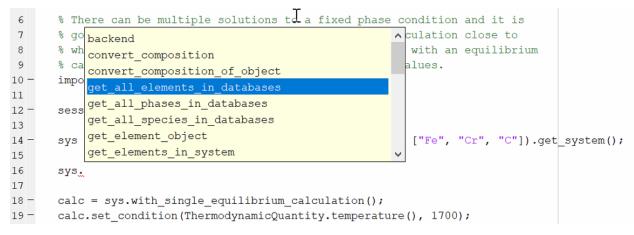
BEST PRACTICES

3.1 Using Tab-Completion and the Integrated documentation

TC-Toolbox contains over 1000 functions and more than 200 classes. These functions are available for use in different contexts, as described in the *Architecture overview*.

In order to know which functions and classes are available for you at a given time and how they can be used, we encourage you to use MATLAB® tab completion and the MATLAB® help.

This is a feature of MATLAB® and the exact functionality can vary depending on the version of MATLAB® and if you use MATLAB® live scripts, classic MATLAB® scripts or the interactive console.

To access tab completion, press the **dot (.)** key then **Tab**. Use the **up/down arrow** keys to scroll through the list.

```
 6      % There can be multiple solutions to a fixed phase condition and it is
 7      % go  backend                          ^ culation close to
 8      % wh  convert_composition                with an equilibrium
 9      % ca  convert_composition_of_object     alues.
10  −   impo get_all_elements_in_databases
11           get_all_phases_in_databases
12  −   sess get_all_species_in_databases
13           get_element_object
14  −   sys  get_elements_in_system           ["Fe", "Cr", "C"]).get_system();
15
16      sys.
17
18  −   calc = sys.with_single_equilibrium_calculation();
19  −   calc.set_condition(ThermodynamicQuantity.temperature(), 1700);
```

To open the help for a specific function or class, click to place the cursor on the function or object and press the **F1** key.

```
sys = session.select_database_and_elements("FEDEMO", ["Fe", "Cr", "C"]).get_system();

calc = sys.with_single_e
calc.set_condition(Therm
calc.set_condition(Therm
calc.set_condition(Therm

% calculate equilibrium
result = calc.calculate(
disp("Equilibrium at tem
listStablePhases(result)

% calculate liquidus tem
calc.remove_condition(Th
```

select_database_and_elements
(tc_toolbox.TCToolbox) - MATLAB File
Help

View code for
tc_toolbox.TCToolbox/select_database_and_elements

tc_toolbox.TCToolbox/select_database_and_elem

Selects a first thermodynamic or kinetic database and selects the element

Args:
        database_name: The name of the database, for example "FEDEMO"
        list_of_elements: The list of the selected elements in that databas

Open Help Browser   F1 to toggle focus; Escape to close

The built-in help for parameters of a specific function can be reached by placing the cursor within the parentheses of the function and pressing **CTRL + F1**.

```
session = TCToolbox();

sys = session.select_database_and_elements("FEDEMO", ["Fe", "Cr", "C"]).get_system();

                            select_database_and_elements(tc_toolbox.TCToolbox object...)
calc = sys.with_single_equilibrium_calculat                                  More Help...
calc.set_condition(ThermodynamicQuantity.temperature(), 1700);
```

Click **More Help…** to view the corresponding help text.

Note: The MATLAB® script first needs to be run before you can view help text when **More Help…** is clicked. Once the script is run, the respective object is present in the workspace and the help is available.

3.2 Re-use of the Single Equilibrium Calculation State

The Thermo-Calc core keeps an internal state containing the data from previously performed calculations (such as composition of sublattices, previously formed phases, etc.). This is used for start values of future calculations (if not explicitly overwritten) and can strongly influence their convergence and calculation time. It can be useful to save and restore later the core-state **in advanced use cases**, these include:

- Improving the convergence speed in case of very complicated equilibria if a similar equilibrium has already been calculated. "Similar" refers here primarily to composition, temperature, and entered phase set. This case can occur, for example, with the Thermo-Calc nickel-based superalloys database, TCNi.

- Convenient and fast switching between states that have changed a lot (for example regarding suspended phases, numerical settings, etc.)

The mechanism of saving and restoring the state is called bookmarking and is controlled with the two methods `bookmark_state()` and `set_state_to_bookmark()`. The following short example shows how to switch between two different states:

```
import tc_toolbox.*
session = TCToolbox();

calc = session...
        .select_database_and_elements("FEDEMO", ["Fe", "C"])...
        .get_system()...
```

(continues on next page)

```matlab
        .with_single_equilibrium_calculation()...
        .set_condition(ThermodynamicQuantity.temperature(), 2000.0)...
        .set_condition("X(C)", 0.01);

calc.calculate();
bookmark_temp_condition = calc.bookmark_state();

calc.set_phase_to_fixed("BCC", 0.5);
calc.remove_condition(ThermodynamicQuantity.temperature());
bookmark_fixed_phase_condition = calc.bookmark_state();

result_temp = calc.set_state_to_bookmark(bookmark_temp_condition);
disp("Conditions do contain temperature:")
disp(result_temp.get_conditions())
% this calculation had already been performed
disp("Stable phases (do not contain BCC):")
disp(result_temp.get_stable_phases())

result_fixed_phase = calc.set_state_to_bookmark(bookmark_fixed_phase_condition);
disp("Conditions do not contain temperature:")
disp(result_fixed_phase.get_conditions())
% this calculation had **not yet** been performed
disp("Stable phases (do contain BCC):")
disp(calc.calculate().get_stable_phases())
```

3.3 Re-use and Saving Results

Before a calculation is run in MATLAB®, a check is made to see if the exact same calculation has run before, and if that is the case, the result from the calculation can be loaded from disk instead of being re-calculated.

This functionality is always enabled within a script running MATLAB®, but you can make it work the same way when re-running a script, or even when running a completely different script.

You can set up a folder location to re-use results from saved calculations. This folder can be a network folder and shared by many users. This is done using the method `set_cache_folder()`.

```matlab
import tc_toolbox.*

session = TCToolbox();
session.set_cache_folder("cache")
```

The calculation is not re-run if there is a previous MATLAB® calculation with the same cache folder and exactly the same settings; the result is instead loaded from disk.

Another possibility is to explicitly save the result to disk and reload it later:

```matlab
import tc_toolbox.*

session = TCToolbox();
% ... the system and calculator are set up and the calculation is performed
result = calculator.calculate()
```

```
result.save_to_disk("./result_dir")
```

You can then load the result again in another session, for example:

```
import tc_toolbox.*

session = TCToolbox();
result = session.load_result_from_disk().diffusion("./result_dir")
[x, frac] = result.get_mole_fraction_of_component_at_time("Cr", 1000.0)
```

3.4 Using the TCToolbox class efficiently

Normally you should only create *one* `TCToolbox()` variable.

Note: When a `TCToolbox()` variable is deleted, the Java backend engine process is stopped and all temporary data is deleted. When creating a new `TCToolbox()` variable, a new Java process is started. This can take several seconds.

If appropriate, it is safe to create a `TCToolbox()` variable in a loop. **Due to the time it takes this only makes sense if the calculation time per iteration is longer than a minute**.

To prevent creating a `TCToolbox()` variable multiple times, you can use the following pattern.

Example:

```
import tc_toolbox.*

session = tc_toolbox.TCToolbox();
system = session.select_database_and_elements("FEDEMO", ["Fe", "Cr"]).get_system();
calculation = system.with_single_equilibrium_calculation();
calculation.set_condition("T", 1000);

for i = 0:50
    calculate(calculation)
end

function calculate(calculator)
    % you could also pass the `session` or `system` object if more appropriate
    calculator.set_condition("W(Cr)", 0.1);
    % further configuration ...

    result = calculator.calculate();
    % ...
    result.invalidate();  % if the temporary data needs to be cleaned up immediately
end
```

3.5 Parallel Calculations

It is possible to perform parallel calculations with TC-Toolbox using the Parallel Computing Toolbox™ of MATLAB®. This is a separate toolbox that can be purchased for MATLAB®, it is not part of the standard configuration of MATLAB®.

A general pattern that can be applied is shown below. This code snippet shows how to perform single equilibrium calculations for different compositions in parallel. In the same way all other calculators of Thermo-Calc can be used or combined.

Example:

```
num_processes = 2;
min_cr = 10;  % in wt-%
max_cr = 19;  % in wt-%
delta_cr = 1;  % in wt-%
chunk_size = 5;  % this simple code expects that the Cr-range can be exactly divided␣
→into such chunks


if (isempty(gcp('nocreate')))
    parpool("local", num_processes);
end


num_points = 1 + (max_cr - min_cr) / delta_cr;
total_cr_range = linspace(min_cr, max_cr, num_points);
chunked_cr_ranges = num2cell(reshape(total_cr_range, chunk_size, []), 1);

% this requires the Parallel Computing Toolbox(TM), can be run with "for" instead␣
→without parallelization
num_chunks = ceil(num_points / chunk_size);
bcc_fraction_results = cell(num_chunks, 1);
parfor chunk_index = 1 : num_chunks
    bcc_fraction_results{chunk_index} = do_perform(chunked_cr_ranges{chunk_index});
end

bcc_phase_fraction = cell2mat(bcc_fraction_results);
% ... use the result in `bcc_phase_fraction`, for example for plotting


function phase_fractions = do_perform(cr_range)
    % this function is running in a subprocess
    import tc_toolbox.step_or_map_diagrams.*
    import tc_toolbox.*

    elements = ["Fe", "Cr", "Ni", "C"];

    session = TCToolbox();

    sys = session.select_database_and_elements("FEDEMO", elements).get_system();

    calc = sys.with_single_equilibrium_calculation();
    calc.set_condition(ThermodynamicQuantity.temperature(), 1100.0); % in K
    calc.set_condition(ThermodynamicQuantity.mass_fraction_of_a_component("C"), 0.1 /␣
→100);
```

```matlab
    calc.set_condition(ThermodynamicQuantity.mass_fraction_of_a_component("Ni"), 2.0 /␣
→100);

    phase_fractions = zeros(size(cr_range, 1));
    for cr_index = 1 : size(cr_range, 1)
        cr = cr_range(cr_index);
        calc.set_condition("W(Cr)", cr / 100);
        result = calc.calculate();
        phase_fractions(cr_index) = result.get_value_of("NPM(BCC_A2)");
    end
end
```

3.6 Handling Calculation Engine Crashes

In some cases the Thermo-Calc calculation engine can crash. If batch calculations are performed, this brings down the complete batch. To handle this situation there is an error you can use: `UnrecoverableCalculationException()`.

That error is raised if the calculation server enters a state where no further calculations are possible. You should catch that exception and create a new instance of `TCToolbox()`, which you use from that point.

Example:

```matlab
import tc_toolbox.*
import tc_toolbox.diffusion.*

temperatures = linspace(900,1100,10);
session = TCToolbox();
for i = 1:length(temperatures)
    temperature = temperatures(i);
    try
        diffusion_result = session...
                .select_thermodynamic_and_kinetic_databases_with_elements("FEDEMO",
→"MFEDEMO", ["Fe", "Ni"])...
                .get_system()...
                .with_isothermal_diffusion_calculation()...
                    .set_temperature(temperature)...
                    .set_simulation_time(108000.0)...
                    .add_region(Region("Austenite")...
                        .set_width(1E-4)...
                        .with_grid(CalculatedGrid.linear().set_no_of_points(50))...
                        .with_composition_profile(CompositionProfile()...
                            .add("Ni", ElementProfile.linear(10.0, 50.0))...
                        )...
                    .add_phase("FCC_A1"))...
            .calculate();

        [distance, ni_fraction] = diffusion_result.get_mass_fraction_of_component_at_
→time("Ni", 108000.0);
        disp("Succeeded!")

    catch e
```

```
        if contains(e.message, 'UnrecoverableCalculationException')
            disp('Could not calculate. Creating a new TCToolbox and continuing with next␣
→calculation...')
            session = TCToolbox();
        else
            disp('Could not calculate. Using the previous TCToolbox and continuing with␣
→next calculation...')
        end
    end
end
```

3.7 Process Metallurgy Calculations

3.7.1 Equilibrium calculations with changing elements between calculations

It is possible to add, change or remove additions after performing an equilibrium calculation using *+tc_toolbox.*
+process_metallurgy.+equilibrium.EquilibriumCalculation.calculate(). This will change the elements being present in the system if the elements of the additions are differing. The Process Metallurgy Module will handle this situation by reloading the database with the latest set of elements. While this is an appropriate approach in most cases, there can be some disadvantages: reloading the database takes some time and the internal engine state is lost, which may lead to successive calculations failures in some situations.

To avoid the database reload, it is possible **to add the respective elements to additions being present in all calculations (with a zero-fraction)**:

```
import tc_toolbox.process_metallurgy.base.*;
import tc_toolbox.process_metallurgy.equilibrium.*;
import tc_toolbox.*

session = tc_toolbox.TCToolbox();

calc = session.with_metallurgy().with_adiabatic_equilibrium_calculation(ProcessDatabase.
→OXDEMO);

% add the element Al with zero-fraction already
steel = EquilibriumAddition(containers.Map(["Fe", "C", "Al"]), {NaN, 4.5, 0.0}), 100.0e3,
→ 1700 + 273.15);
slag = EquilibriumAddition(containers.Map(["CaO", "SiO2"], {70.0, 30.0}), 3.0e3, 1700 +␣
→273.15);

al_addition = EquilibriumAddition(containers.Map(["Al"], {100}), 1.0e3);

calc...
 .add_addition(steel)...
 .add_addition(slag);

result_1 = calc.calculate();

calc.add_addition(al_addition);
```

```matlab
result_2 = calc.calculate();
% evaluate the result as required ...
```

Or **to add a later addition already before the first call to** `calculate()` **with a zero amount**:

```matlab
import tc_toolbox.process_metallurgy.base.*;
import tc_toolbox.process_metallurgy.equilibrium.*;
import tc_toolbox.*

session = tc_toolbox.TCToolbox();

calc = session.with_metallurgy().with_adiabatic_equilibrium_calculation(ProcessDatabase.
↪OXDEMO);

steel = EquilibriumAddition(containers.Map(["Fe", "C"]), {NaN, 4.5}), 100.0e3, 1700 +␣
↪273.15);
slag = EquilibriumAddition(containers.Map(["CaO", "SiO2"], {70.0, 30.0}), 3.0e3, 1700 +␣
↪273.15);

% add the addition for now with zero-amount
al_addition = EquilibriumAddition(containers.Map(["Al"], {100}), 0);

calc...
 .add_addition(al_addition)...
 .add_addition(steel)...
 .add_addition(slag);

result_1 = calc.calculate();

calc.update_addition(al_addition.set_amount(1.0e3));

result_2 = calc.calculate();
% evaluate the result as required ...
```

3.7.2 Zones

TC-Toolbox is providing a *framework* for building time-dependent kinetic simulations of industrial and academic metallurgical processes where liquid phases are important. It is based on an *Effective Equilibrium Reaction Zone* (EERZ) approach which is separating a process into different zones. These zones have identical temperature and composition and are called *bulk zones*. Such zones can be in contact and react with each other by *reaction zones*. That means a *reaction zone* is modelling the interface between two *bulk zones*. One bulk zone is typically the steel melt and another bulk zone the top slag.

3.7.3 Applications

While this approach can in principle be extended to any number of zones, in the current release TC-Toolbox is providing **only one reaction zone**. Practical work has however proven that this limitation is not critical for a lot of industrial processes, including ladle furnaces, AOD- and VOD-converters. Even more processes can be modelled with some limit of accuracy.

The reason for the power of the current implementation is that a number of important process features can be included:

- heating (`+tc_toolbox.+process_metallurgy.+process.Zone.add_power()`)

- cooling (`+tc_toolbox.+process_metallurgy.+process.Zone.add_power()`)

- heat transfer between bulk zones (`+tc_toolbox.+process_metallurgy.+process.ReactionZone.add_heat_transfer()`)

- inclusion formation

- inclusion flotation and other transfer of phase groups between bulk zones (`+tc_toolbox.+process_metallurgy.+process.ReactionZone.add_transfer_of_phase_group()`)

- addition of material and gas at any time in any zone (`+tc_toolbox.+process_metallurgy.+process.Zone.add_addition()` / `+tc_toolbox.+process_metallurgy.+process.Zone.add_continuous_addition()`)

- an exhaust gas zone collecting all formed gas (`+tc_toolbox.+process_metallurgy.+process.ProcessSimulationResult.get_exhaust_gas()`)

- time-dependent definition of most parameters (e.g., mass transfer coefficient, transfer of phase group, heating, etc.)

Please note that many of these features are called as well a *reaction zone* in other EERZ model implementations.

3.7.4 Implementation of practical process models

The Process Metallurgy Module has been successfully applied to a number of industrial processes.

Due to the broad range of industrial metallurgical processes, TC-Toolbox is not providing ready-to-use models for certain processes. There are however examples available for common processes and this collection will be extended over time. The implementation of a model is an abstraction of the real process and should always be kept as simple as possible. Practical experience has proven that in many situations not more than one *reaction zone* is required.

The mass transfer coefficient is a fundamental parameter describing the kinetics in a *reaction zone* and is generally an empirical parameter. It depends however mostly on the geometry and stirring conditions in the process and not on the material compositions. Further on, the mass transfer coefficient has usually typical values for a given process - regardless of the actual furnace. That means that existing suggestions from the literature can be used as a starting point to derive the actual mass transfer coefficient for the process of interest.

API REFERENCE

4.1 Calculations

4.1.1 Package "single_equilibrium"

class +tc_toolbox.+single_equilibrium.**AbstractSingleEquilibriumCalculation**(*back*)

Abstract configuration required for a single equilibrium calculation.

Note: This is an abstract class that cannot be used directly.

AbstractSingleEquilibriumCalculation(*back*)

Call base constructor: tc_toolbox.AbstractCalculation.

disable_global_minimization()

Turns the global minimization completely off.

> **Returns**
>
>   This *SingleEquilibriumCalculation* object

enable_global_minimization()

Turns the global minimization on (using the default settings).

> **Returns**
>
>   This *SingleEquilibriumCalculation* object

get_components()

Returns a list of components in the system (including all components auto-selected by the database(s)).

> **Returns**
>
>   The components

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added

to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
> > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>
> **Returns**
> > Gibbs energy addition to G per mole formula unit.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
> > The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
> > **command** – The Thermo-Calc Console Mode command
>
> **Returns**
> > This *SingleEquilibriumCalculation* object

set_component_to_entered(*component*)

Sets the specified component to the status ENTERED, that is the default state.

> **Parameters**
> > **component** – The component name or *ALL_COMPONENTS*
>
> **Returns**
> > This *SingleEquilibriumCalculation* object

set_component_to_suspended(*component*, *reset_conditions*)

Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

> **Parameters**

- **reset_conditions** – if 'True' also remove composition conditions for the component if they are defined

- **component** – The component name or *ALL_COMPONENTS*

> **Returns**
>> This *SingleEquilibriumCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

- **gibbs_energy** – Addition to G per mole formula unit

> **Returns**
>> This *SingleEquilibriumCalculation* object

set_phase_to_dormant(*phase*)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
>> This *SingleEquilibriumCalculation* object

set_phase_to_entered(*phase*, *amount*)

Sets the phase to the status ENTERED, that is the default state.

> **Parameters**

- **phase** – The phase name or *ALL_PHASES* for all phases

- **amount** – The phase fraction (between 0.0 and 1.0)

> **Returns**
>> This *SingleEquilibriumCalculation* object

set_phase_to_fixed(*phase*, *amount*)

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

> **Parameters**

- **phase** – The phase name

- **amount** – The fixed phase fraction (between 0.0 and 1.0)

> **Returns**
>> This *SingleEquilibriumCalculation* object

set_phase_to_suspended(*phase*)

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
>> This *SingleEquilibriumCalculation* object

with_options(*options*)

> Sets the simulation options.

>> **Parameters**
>>> **options** – The simulation options

>> **Returns**
>>> This *SingleEquilibriumCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

> For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

> If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

>> **Parameters**

>>> • **component** – The name of the element must be given.

>>> • **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.

>>> • **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

>>> • **pressure** – The Pressure (in Pa) for the reference state.

>> **Returns**
>>> This *SingleEquilibriumCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

>> **Parameters**
>>> **system_modifications** – The system modification to be performed

---

> **Returns**
>> This *SingleEquilibriumCalculation* object

class +tc_toolbox.+single_equilibrium.**SingleEquilibriumCalculation**(*back*)

> Configuration for a single equilibrium calculation.

> **Note:** Specify the conditions and possibly other settings, the calculation is performed with *calculate()*.

> **SingleEquilibriumCalculation**(*back*)
>> Call base constructor: tc_toolbox.single_equilibrium.AbstractSingleEquilibriumCalculation.

> **bookmark_state**(*bookmark_id*)
>> Puts a "bookmark" on the current calculation-state of the calculator allowing the program to return to this state later as needed.
>>
>> By bookmarking a state, you can simplify the convergence of equilibria when they strongly depend on the starting conditions (i.e. the state). Also use it to improve performance by running a calculation, then bookmarking it, and later returning to it for other equilibria whose conditions are "close" to the bookmarked equilibrium.
>>
>> This method is used in combination with the method *set_state_to_bookmark()*.
>>
>>> **Parameters**
>>>> **bookmark_id** – The bookmark id. If omitted a generated id is used and returned
>>>
>>> **Returns**
>>>> The bookmark id

> **calculate**()
>> Performs the calculation and provides a temporary result object that is only valid until something gets changed in the calculation state. The method *calculate()* is the default approach and should be used in most cases.
>>
>> ---
>>
>> **Warning:** If the result object should be valid for the whole program lifetime, use *calculate_with_state()* instead.
>>
>> ---
>>
>>> **Returns**
>>>> A new *SingleEquilibriumTempResult* object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has been changed.

> **calculate_with_state**(*timeout_in_minutes*)
>> Performs the calculation and provides a result object that reflects the present state of the calculation during the whole lifetime of the object.
>>
>> ---
>>
>> **Note:** Because this method has performance and temporary disk space overhead (i.e. it is resource heavy), only use it when it is necessary to access the result object after the state is changed. In most cases you should use the method *calculate()*.
>>
>> ---
>>
>>> **Parameters**
>>>> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is

---

wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a Unrecoverable CalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

> **Returns**
> A new `SingleEquilibriumResult` object which can be used later at any time to get specific values from the calculated result.

disable_global_minimization()

> Turns the global minimization completely off.

> > **Returns**
> > This `SingleEquilibriumCalculation` object

enable_global_minimization()

> Turns the global minimization on (using the default settings).

> > **Returns**
> > This `SingleEquilibriumCalculation` object

get_components()

> Returns a list of components in the system (including all components auto-selected by the database(s)).

> > **Returns**
> > The components

get_configuration_as_string()

> Returns detailed information about the current state of the calculation object.

> > **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_gibbs_energy_addition_for(*phase*)

> Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

> It is not composition-, temperature- or pressure-dependent.

> > **Parameters**
> > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

> > **Returns**
> > Gibbs energy addition to G per mole formula unit.

get_interfacial_energy(*matrix_phase*, *precipitate_phases*, *zero_volume_elements*)

> Estimates the interfacial energy between a matrix phase and a precipitate phase using thermodynamic data from a CALPHAD database. The approximation model is based on Becker's bond energy approach.

> **Default**: elements with no contribution to volume are C and N.

> > **Parameters**
> > - **matrix_phase** – The matrix phase.
> > - **precipitate_phases** – The list of precipitate phases for which interfacial energy between them and the matrix phase is to be calculated.
> > - **zero_volume_elements** – The elements that are assumed to not contribute to the volume.

Returns

A dictionary containing interfacial energy per precipitate phase.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

Returns

The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

remove_all_conditions()

Removes all set conditions.

Returns

This *SingleEquilibriumCalculation* object

remove_condition(*quantity*)

Removes the specified condition.

Parameters

quantity – the thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example "X(Cr)")

Returns

This *SingleEquilibriumCalculation* object

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

Parameters

command – The Thermo-Calc Console Mode command

Returns

This *SingleEquilibriumCalculation* object

set_component_to_entered(*component*)

Sets the specified component to the status ENTERED, that is the default state.

Parameters

component – The component name or *ALL_COMPONENTS*

> **Returns**
> > This *SingleEquilibriumCalculation* object

set_component_to_suspended(*component*, *reset_conditions*)

> Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

> > **Parameters**
> >
> > - **reset_conditions** – if 'True' also remove composition conditions for the component if they are defined
> >
> > - **component** – The component name or *ALL_COMPONENTS*
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

set_condition(*quantity*, *value*)

> Sets the specified condition.

> > **Parameters**
> >
> > - **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example "X(Cr)")
> >
> > - **value** – The value of the condition
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

> Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

> It is not composition-, temperature- or pressure-dependent.

> > **Parameters**
> >
> > - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
> >
> > - **gibbs_energy** – Addition to G per mole formula unit
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

> > **Parameters**
> > > **phase** – The phase name or *ALL_PHASES* for all phases

> > **Returns**
> > > This *SingleEquilibriumCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.

> > **Parameters**
> >
> > - **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > - **amount** – The phase fraction (between 0.0 and 1.0)

> **Returns**
> This *SingleEquilibriumCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
>
> > **Parameters**
> >
> > - **phase** – The phase name
> >
> > - **amount** – The fixed phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > This *SingleEquilibriumCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> > **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > This *SingleEquilibriumCalculation* object

set_state_to_bookmark(*bookmark_id*)

> Resets the calculation state to a previously bookmarked state.
>
> After calling this method, the calculation behaves exactly as it would after the bookmarked calculation ran.
>
> This method is used in combination with the method *bookmark_state()*.
>
> > **Parameters**
> > **bookmark_id** – The bookmark id of the state to return to.
> >
> > **Returns**
> > A new *SingleEquilibriumTempResult* object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has been changed.

with_options(*options*)

> Sets the simulation options.
>
> > **Parameters**
> > **options** – The simulation options
> >
> > **Returns**
> > This *SingleEquilibriumCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
>
> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.
>
> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set

---

as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**
> - **component** – The name of the element must be given.
> - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> - **pressure** – The Pressure (in Pa) for the reference state.
>
> **Returns**
> This *SingleEquilibriumCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed
>
> **Returns**
> This *SingleEquilibriumCalculation* object

class +tc_toolbox.+single_equilibrium.**SingleEquilibriumOptions**

General simulation conditions for the thermodynamic calculations.

SingleEquilibriumOptions()

General simulation conditions for thermodynamic calculations. Constructs an instance of *SingleEquilibriumOptions*.

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
> This *SingleEquilibriumOptions* object

---

disable_control_step_size_during_minimization()

>   Disables stepsize control during minimization (non-global).

>   **Default**: Enabled

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

disable_force_positive_definite_phase_hessian()

>   Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

>   **Default**: Enabled

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

enable_approximate_driving_force_for_metastable_phases()

>   Enables the approximation of the driving force for metastable phases.

>   **Default**: Enabled

>   **Note:**   When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

enable_control_step_size_during_minimization()

>   Enables stepsize control during normal minimization (non-global).

>   **Default**: Enabled

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

enable_force_positive_definite_phase_hessian()

>   Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

>   **Default**: Enabled

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

set_global_minimization_max_grid_points(*max_grid_points*)

>   Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used**.

>   **Default**: 2000 points

>>   **Parameters**
>>>   **max_grid_points** – The maximum number of grid points

>>   **Returns**
>>>   This *SingleEquilibriumOptions* object

set_max_no_of_iterations(*max_no_of_iterations*)

Set the maximum number of iterations.

Default: max. 500 iterations

> **Note:** As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

> **Parameters**
>> **max_no_of_iterations** – The max. number of iterations

> **Returns**
>> This *SingleEquilibriumOptions* object

set_required_accuracy(*accuracy*)

Sets the required relative accuracy.

Default: 1.0E-6

> **Note:** This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

> **Parameters**
>> **accuracy** – The required relative accuracy

> **Returns**
>> This *SingleEquilibriumOptions* object

set_smallest_fraction(*smallest_fraction*)

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The **default value** for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

> **Parameters**
>> **smallest_fraction** – The smallest fraction for constituents that are unstable

> **Returns**
>> This *SingleEquilibriumOptions* object

class +tc_toolbox.+single_equilibrium.**SingleEquilibriumResult**(*back*)

Result of a single equilibrium calculation, it can be evaluated using a Quantity or Console Mode syntax.

SingleEquilibriumResult(*back*)

Call base constructor: `tc_toolbox.AbstractResult`.

change_pressure(*pressure*)

Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.

> **Parameters**
> > **pressure** – The pressure [Pa]
>
> **Returns**
> > This *SingleEquilibriumCalculation* object

change_temperature(*temperature*)

> Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.
>
> ---
>
> **Note:** This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature during cooling.
>
> ---
>
> **Parameters**
> > **temperature** – The temperature [K]
>
> **Returns**
> > This *SingleEquilibriumCalculation* object

get_components()

> Returns the names of the components selected in the system (including any components auto-selected by the database(s)).
>
> **Returns**
> > The names of the selected components

get_conditions()

> Returns the conditions.
>
> **Returns**
> > The selected conditions

get_phases()

> Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method System. get_phases_in_system().
>
> **Returns**
> > The names of the phases in the system including automatically added phases

get_stable_phases()

> Returns the stable phases (i.e. the phases present in the current equilibrium).
>
> **Returns**
> > The names of the stable phases

get_value_of(*quantity*)

> Returns a value from a single equilibrium calculation.
>
> **Parameters**
> > **quantity** – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example "NPM(FCC_A1)")
>
> **Returns**
> > The requested value

---

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

> Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine. This affects only the state of the result object.

> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> > **Parameters**
> > > **command** – The Thermo-Calc Console Mode command
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

save_to_disk(*path*)

> Saves the result to disk. Note that the result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`

> > **Parameters**
> > > **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
> >
> > **Returns**
> > > this *SingleEquilibriumResult* object

class +tc_toolbox.+single_equilibrium.**SingleEquilibriumTempResult**(*back*)

Result of a single equilibrium calculation that is only valid until something gets changed in the calculation state. It can be evaluated using a Quantity or Console Mode syntax.

> **Warning:** Note that it is undefined behavior to use that object after something has been changed in the state of the calculation, this will result in an `InvalidResultStateException` exception being raised.

SingleEquilibriumTempResult(*back*)

> Call base constructor: `tc_toolbox.AbstractResult`.

change_pressure(*pressure*)

> Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.

> > **Parameters**
> > > **pressure** – The pressure [Pa]
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

---

change_temperature(*temperature*)

> Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.

> ---
> **Note:** This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature during cooling.
> ---

> > **Parameters**
> > > **temperature** – The temperature [K]
> >
> > **Returns**
> > > This *SingleEquilibriumCalculation* object

get_components()

> Returns the names of the components selected in the system (including any components auto-selected by the database(s)).
>
> > **Returns**
> > > If something has been changed in the state of the calculation since that result object has been created

get_conditions()

> Returns the conditions.
>
> > **Returns**
> > > If something has been changed in the state of the calculation since that result object has been created

get_phases()

> Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method `System.get_phases_in_system()`.
>
> > **Returns**
> > > If something has been changed in the state of the calculation since that result object has been created

get_stable_phases()

> Returns the stable phases (i.e. the phases present in the current equilibrium).
>
> > **Returns**
> > > If something has been changed in the state of the calculation since that result object has been created

get_value_of(*quantity*)

> Returns a value from a single equilibrium calculation.
>
> > **Parameters**
> > > **quantity** – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example "NPM(FCC_A1)")
> >
> > **Returns**
> > > If something has been changed in the state of the calculation since that result object has been created

invalidate()

>   Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

run_poly_command(*command*)

>   Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

>   **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

>   **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

>   **Parameters**
>   **command** – The Thermo-Calc Console Mode command

>   **Returns**
>   This *SingleEquilibriumCalculation* object

4.1.2 Package "batch_equilibrium"

class +tc_toolbox.+batch_equilibrium.**BatchEquilibriumCalculation**(*back*)

>   Configuration for a series of single equilibrium calculations performed in a vectorized fashion.

>   **Tip:** The performance of batch equilibrium calculations can be significantly better than looping and using SingleEquilibriumCalculation **if the actual Thermo-Calc calculation is fast**. There is little advantage if the Thermo-Calc equilibrium calculations take a long time (typically for large systems and databases).

>   **Note:** Specify the conditions and call *calculate()*.

BatchEquilibriumCalculation(*back*)

>   Call base constructor: tc_toolbox.AbstractCalculation.

calculate(*quantities*, *logging_frequency*, *timeout_in_minutes*)

>   Runs the batch equilibrium calculation. The calculated *BatchEquilibriumResult* can then be queried for the values of the quantities specified.

>   Example:

```
>>> quantities = ['G', 'X(BCC)']
```

>   **Parameters**
>   - **quantities** – A list of the quantities to be calculated.
>   - **logging_frequency** – Determines how often logging should be done.
>   - **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a

UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

> **Returns**
> A `BatchEquilibriumResult` which later can be used to get specific values from the calculated result.

disable_global_minimization()

Turns the global minimization completely off.

> **Returns**
> This `BatchEquilibriumCalculation` object

enable_global_minimization()

Turns the global minimization on (using the default settings).

> **Returns**
> This `BatchEquilibriumCalculation` object

get_components()

Returns a list of components in the system (including all components auto-selected by the database(s)).

> **Returns**
> The components

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
> **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>
> **Returns**
> Gibbs energy addition to G per mole formula unit.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as *.tdb*-file.

> **Returns**
> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

remove_all_conditions()

> Removes all set conditions.
>
> > **Returns**
> >
> > > This *BatchEquilibriumCalculation* object

remove_condition(*quantity*)

> Removes the specified condition.
>
> > **Parameters**
> >
> > > **quantity** – the thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example "X(Cr)")
> >
> > **Returns**
> >
> > > This *BatchEquilibriumCalculation* object

run_poly_command(*command*)

> Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.
>
> ---
>
> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.
>
> ---
>
> > **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).
>
> > **Parameters**
> >
> > > **command** – The Thermo-Calc Console Mode command
> >
> > **Returns**
> >
> > > This *BatchEquilibriumCalculation* object

set_component_to_entered(*component*)

> Sets the specified component to the status ENTERED, that is the default state.
>
> > **Parameters**
> >
> > > **component** – The component name or *ALL_COMPONENTS*
> >
> > **Returns**
> >
> > > This *BatchEquilibriumCalculation* object

set_component_to_suspended(*component*, *reset_conditions*)

> Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> > **Parameters**
> >
> > > - **reset_conditions** – if 'True' also remove composition conditions for the component if they are defined
> > >
> > > - **component** – The component name or *ALL_COMPONENTS*
> >
> > **Returns**
> >
> > > This *BatchEquilibriumCalculation* object

set_condition(*quantity*, *value*)

> Sets the specified condition.
>
> > **Parameters**
> >
> > - **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example "X(Cr)")
> >
> > - **value** – The value of the condition
> >
> > **Returns**
> >
> > This *BatchEquilibriumCalculation* object

set_conditions_for_equilibria(*equilibria*)

> Set the conditions of the equilibria to be calculated.
>
> This is done by sending a list of equilibria at once.
>
> Each equilibrium itself is a list of conditions that will be changed for that equilibrium.
>
> A condition is described by a tuple containing:
>
> - A Console Mode syntax string or a `ThermodynamicQuantity` instance,
>
> - A float value specifying the value of the condition.
>
> Example:

```
>>> [[('T', 800), ('X(Cr)', 0.1)], [('T', 850), ('X(Cr)', 0.11)]]
```

> You can use `ThermodynamicQuantity` instead of a Console Mode syntax string when specifying type of condition.
>
> Example:

```
>>> [[(ThermodynamicQuantity.temperature(), 800), (ThermodynamicQuantity.mole_
→fraction_of_a_component('Cr'), 0.1)], [(ThermodynamicQuantity.temperature(),
→850), (ThermodynamicQuantity.mole_fraction_of_a_component('Cr'), 0.15)]]
```

> > **Parameters**
> >
> > **equilibria** – The list of equilibria
> >
> > **Returns**
> >
> > This *BatchEquilibriumCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

> Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
> > **Parameters**
> >
> > - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
> >
> > - **gibbs_energy** – Addition to G per mole formula unit
> >
> > **Returns**
> >
> > This *BatchEquilibriumCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.
>
> > **Parameters**
> > > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > > This *BatchEquilibriumCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.
>
> > **Parameters**
> >
> > > - **phase** – The phase name or *ALL_PHASES* for all phases
> > >
> > > - **amount** – The phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > > This *BatchEquilibriumCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
>
> > **Parameters**
> >
> > > - **phase** – The phase name
> > >
> > > - **amount** – The fixed phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > > This *BatchEquilibriumCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> > **Parameters**
> > > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > > This *BatchEquilibriumCalculation* object

with_options(*options*)

> Sets the simulation options.
>
> > **Parameters**
> > > **options** – The simulation options
> >
> > **Returns**
> > > This *BatchEquilibriumCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
>
> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**
> - **component** – The name of the element must be given.
> - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> - **pressure** – The Pressure (in Pa) for the reference state.
>
> **Returns**
> This *BatchEquilibriumCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed
>
> **Returns**
> This *BatchEquilibriumCalculation* object

class +tc_toolbox.+batch_equilibrium.**BatchEquilibriumResult**(*back*)

Result of a batch equilibrium calculation. This can be used to query for specific values.

BatchEquilibriumResult(*back*)

Constructs an instance of *BatchEquilibriumResult*.

get_values_of(*quantity*)

Returns values from a batch equilibrium calculation.

Example:

```
>>> batch_result = batch_calculation.calculate(quantities = ['G', 'X(BCC)'])
>>> batch_result.get_values_of('G')
```

> **Warning:** The quantity must be one of the quantities specified for the
> `BatchEquilibriumCalculation` object that created the result object.

> **Parameters**
> **quantity** – the thermodynamic quantity to get the value of; a Console Mode syntax strings
> can be used as an alternative (for example "NPM(FCC_A1)")

invalidate()

Invalidates the object and frees the disk space used by it.

> **Note:** This is only required if the disk space occupied by the object needs to be released during the
> calculation. No data can be retrieved from the object afterwards.

4.1.3 Package "precipitation"

class +tc_toolbox.+precipitation.**FixedGrainSize**(*grain_radius*)

Factory class providing objects representing a grain growth model.

FixedGrainSize(*grain_radius*)

Fixed grain radius size. **Default**: 1.0E-4 m

> **Parameters**
> **grain_radius** – The grain radius / size [m]

static fixed_grain_size(*grain_radius*)

Fixed grain radius size. **Default**: 1.0E-4 m

> **Parameters**
> **grain_radius** – The grain radius / size [m]

static grain_growth(*grain_size_distribution*)

Sets the initial grain size distribution for the matrix. **Default**: If the initial grain size distribution is not
explicitly provided, a constant average grains size will be used and no grain growth evaluated during the
simulation.

> **Tip:** Use this option if you want to study the further evolution of an existing microstructure.

> **Parameters**
> **grain_size_distribution** – grain size distribution

set_grain_aspect_ratio(*grain_aspect_ratio*)

Enter a numerical value. **Default**: 1.0.

> **Parameters**
> **grain_aspect_ratio** – The grain aspect ratio [-]

class +tc_toolbox.+precipitation.**GrainGrowth**(*grain_size_distribution*)

Factory class providing objects representing a grain growth model.

GrainGrowth(*grain_size_distribution*)

Sets the initial grain size distribution for the matrix. **Default**: If the initial grain size distribution is not explicitly provided, a constant average grains size will be used and no grain growth evaluated during the simulation.

Tip: Use this option if you want to study the further evolution of an existing microstructure.

> **Parameters**
> **grain_size_distribution** – grain size distribution

disable_zener_pinning()

Disable Zener pinning to ignore the particle pinning effect on the grain growth. Zener pinning is by default disabled when no grain size distribution is defined, i.e. a single constant grain size is used. The setting is by default enabled when a grain size distribution is defined.

> **Returns**
> This *GrainSizeDistribution* object

enable_zener_pinning()

Enable Zener pinning to simulate the particle pinning effect on the grain growth. The setting is by default enabled when a grain size distribution is defined.

> **Returns**
> This *GrainSizeDistribution* object

static fixed_grain_size(*grain_radius*)

Fixed grain radius size. **Default**: 1.0E-4 m

> **Parameters**
> **grain_radius** – The grain radius / size [m]

static grain_growth(*grain_size_distribution*)

Sets the initial grain size distribution for the matrix. **Default**: If the initial grain size distribution is not explicitly provided, a constant average grains size will be used and no grain growth evaluated during the simulation.

Tip: Use this option if you want to study the further evolution of an existing microstructure.

> **Parameters**
> **grain_size_distribution** – grain size distribution

set_grain_boundary_energy(*energy*)

Set the energy of the grain boundary.

> **Parameters**
> **energy** – The grain boundary energy [J/m2]

> **Returns**
> This *GrainSizeDistribution* object

set_grain_boundary_mobility_activation_energy(*activation_energy*)

Set the grain boundary mobility activation energy where the mobility is defined by an Arrhenius type of equation.

>   >   **Parameters**
>   >   >   `activation_energy` – The mobility activation energy [J/mol]
>
>   >   **Returns**
>   >   >   This *GrainSizeDistribution* object
>
>   **set_grain_boundary_mobility_pre_factor**(*pre_factor*)
>
>   >   Set the grain boundary mobility prefactor where the mobility is defined by an Arrhenius type of equation.
>
>   >   **Parameters**
>   >   >   `pre_factor` – The grain boundary mobility pre factor [m^4/(J s)]
>
>   >   **Returns**
>   >   >   This *GrainSizeDistribution* object

class +tc_toolbox.+precipitation.**GrainGrowthModel**

>   Factory class providing objects representing a grain growth model.
>
>   **static fixed_grain_size**(*grain_radius*)
>
>   >   Fixed grain radius size. **Default**: 1.0E-4 m
>
>   >   **Parameters**
>   >   >   `grain_radius` – The grain radius / size [m]
>
>   **static grain_growth**(*grain_size_distribution*)
>
>   >   Sets the initial grain size distribution for the matrix. **Default**: If the initial grain size distribution is not explicitly provided, a constant average grains size will be used and no grain growth evaluated during the simulation.
>
>   >   ---
>
>   >   **Tip:** Use this option if you want to study the further evolution of an existing microstructure.
>
>   >   ---
>
>   >   **Parameters**
>   >   >   `grain_size_distribution` – grain size distribution

class +tc_toolbox.+precipitation.**GrainSizeDistribution**

>   Represents the grain size distribution at a certain time.
>
>   **GrainSizeDistribution**()
>
>   >   Constructs an instance of *GrainSizeDistribution*.
>
>   **add_radius_and_number_density**(*radius*, *number_density*)
>
>   >   Adds a radius and number density pair to the grain size distribution.
>
>   >   **Parameters**
>   >
>   >   >   - `radius` – The radius [m]
>   >
>   >   >   - `number_density` – The number of grains per unit volume per unit length [m^-4]
>
>   >   **Returns**
>   >   >   This *GrainSizeDistribution* object

class +tc_toolbox.+precipitation.**GrowthRateModel**

>   Choice of the used **growth rate model** for a precipitate.
>
>   The most efficient model is the *Simplified model*, which is the default and applicable to most alloy systems under the assumption that either the supersaturation is small, or the alloying elements have comparable diffusivity. If all alloying elements are substitutional but they have remarkable diffusivity difference, e.g. in Al-Zr system, or if the diffusivity is strongly composition-dependent, the *General model* is preferred. If the supersaturation is high,

and meanwhile there are fast-diffusing interstitial elements such as C, the *Advanced model* is more appropriate to capture the NPLE mechanism.

class +tc_toolbox.+precipitation.**MatrixPhase**(*matrix_phase_name*)

> The matrix phase in a precipitation calculation

> **MatrixPhase**(*matrix_phase_name*)

> **add_precipitate_phase**(*precipitate_phase*)

>> Adds a precipitate phase.

>> **Parameters**
>>> **precipitate_phase** – The precipitate phase

> **set_dislocation_density**(*dislocation_density*)

>> Enter a numerical value. **Default**: 5.0E12 m^-2.

>> **Parameters**
>>> **dislocation_density** – The dislocation density [m^-2]

> **set_mobility_adjustment**(*element*, *prefactor*, *activation_energy*)

>> A value that adds to the activation energy of mobility data from the database.

>> **Parameters**

>>> • **element** – The alement to apply the adjustment for. If "all" is given, adjustment will apply to all elements.

>>> • **prefactor** – A parameter that multiplies to the mobility data from a database. This value scales the mobility by a constant amount. This can be useful, for example, when the material has a higher than normal vacancy concentration at the start of the precipitation simulation (e.g. from a prior solutionizing and quenching treatment).

>>> • **activation_energy** – A value that adds to the activation energy of mobility data from a database. It scales the mobility by a temperature dependent amount. Similar usage as mobility adjustment prefactor. [J/mol]

> **set_mobility_enhancement_activation_energy**(*mobility_enhancement_activation_energy*)

>> A value that adds to the activation energy of mobility data from the database. **Default**: 0.0 J/mol

>> **Parameters**
>>> **mobility_enhancement_activation_energy** – The value that adds to the activation energy of mobility data from the database [J/mol].

> **set_mobility_enhancement_prefactor**(*mobility_enhancement_prefactor*)

>> A parameter that multiplies to the mobility data from database. **Default**: 1.0

>> **Parameters**
>>> **mobility_enhancement_prefactor** – The mobility enhancement factor [-]

> **set_molar_volume**(*volume*)

>> Sets the molar volume of the phase.

>> **Default**: If not set, the molar volume is taken from the thermodynamic database (or set to 7.0e-6 m^3/mol if the database contains no molar volume information).

>> **Parameters**
>>> **volume** – The molar volume [m^3/mol]

---

with_elastic_properties_cubic(*c11*, *c12*, *c44*)

Sets the elastic properties to "cubic" and specifies the elastic stiffness tensor components. **Default**: if not chosen, the default is DISREGARD

> **Parameters**
>
> > • **c11** – The stiffness tensor component c11 [GPa]
> >
> > • **c12** – The stiffness tensor component c12 [GPa]
> >
> > • **c44** – The stiffness tensor component c44 [GPa]

with_elastic_properties_disregard()

Set to disregard to ignore the elastic properties. **Default**: This is the default option

with_elastic_properties_isotropic(*shear_modulus*, *poisson_ratio*)

Sets elastic properties to isotropic. **Default**: if not chosen, the default is DISREGARD

> **Parameters**
>
> > • **shear_modulus** – The shear modulus [GPa]
> >
> > • **poisson_ratio** – The Poisson's ratio [-]

with_grain_growth_model(*grain_growth_model*)

Sets the model for grain growth. Either fixed size or with a starting distribution

Default: Fixed grain radius size 1.0E-4 m

> **Parameters**
> **grain_growth_model** – the grain growth model

class +tc_toolbox.+precipitation.**NumericalParameters**

Numerical parameters

NumericalParameters()

Constructs an instance of *NumericalParameters*.

set_max_overall_volume_change(*max_overall_volume_change*)

This defines the maximum absolute (not ratio) change of the volume fraction allowed during one time step. **Default**: 0.001

> **Parameters**
> **max_overall_volume_change** – The maximum absolute (not ratio) change of the volume fraction allowed during one time step [-]

set_max_radius_points_per_magnitude(*max_radius_points_per_magnitude*)

Sets the maximum number of grid points over one order of magnitude in radius. **Default**: 200.0

> **Parameters**
> **max_radius_points_per_magnitude** – The maximum number of grid points over one order of magnitude in radius [-]

set_max_rel_change_critical_radius(*max_rel_change_critical_radius*)

Used to place a constraint on how fast the critical radium can vary, and thus put a limit on time step. **Default**: 0.1

> **Parameters**
> **max_rel_change_critical_radius** – The maximum relative change of the critical radius [-]

set_max_rel_change_nucleation_rate_log(*max_rel_change_nucleation_rate_log*)

This parameter ensures accuracy for the evolution of effective nucleation rate. **Default**: 0.5

> **Parameters**
> **max_rel_change_nucleation_rate_log** – The maximum logarithmic relative change of the nucleation rate [-]

set_max_rel_radius_change(*max_rel_radius_change*)

The maximum value allowed for relative radius change in one time step. **Default**: 0.01

> **Parameters**
> **max_rel_radius_change** – The maximum relative radius change in one time step [-]

set_max_rel_solute_composition_change(*max_rel_solute_composition_change*)

Set a limit on the time step by controlling solute depletion or saturation, especially at isothermal stage. **Default**: 0.01

> **Parameters**
> **max_rel_solute_composition_change** – The limit for the relative solute composition change [-]

set_max_time_step(*max_time_step*)

The maximum time step allowed for time integration as fraction of the simulation time. **Default**: 0.1

> **Parameters**
> **max_time_step** – The maximum time step as fraction of the simulation time [-]

set_max_time_step_during_heating(*max_time_step_during_heating*)

The upper limit of the time step that has been enforced in the heating stages. **Default**: 1.0 s

> **Parameters**
> **max_time_step_during_heating** – The maximum time step during heating [s]

set_max_volume_fraction_dissolve_time_step(*max_volume_fraction_dissolve_time_step*)

Sets the maximum volume fraction of subcritical particles allowed to dissolve in one time step. **Default**: 0.01

> **Parameters**
> **max_volume_fraction_dissolve_time_step** – The maximum volume fraction of subcritical particles allowed to dissolve in one time step [-]

set_min_radius_nucleus_as_particle(*min_radius_nucleus_as_particle*)

The cut-off lower limit of precipitate radius. **Default**: 5.0E-10 m

> **Parameters**
> **min_radius_nucleus_as_particle** – The minimum radius of a nucleus to be considered as a particle [m]

set_min_radius_points_per_magnitude(*min_radius_points_per_magnitude*)

Sets the minimum number of grid points over one order of magnitude in radius. **Default**: 100.0

> **Parameters**
> **min_radius_points_per_magnitude** – The minimum number of grid points over one order of magnitude in radius [-]

set_radius_points_per_magnitude(*radius_points_per_magnitude*)

Sets the number of grid points over one order of magnitude in radius. **Default**: 150.0

> **Parameters**
> **radius_points_per_magnitude** – The number of grid points over one order of magnitude in radius [-]

set_rel_radius_change_class_collision(*rel_radius_change_class_collision*)

> Sets the relative radius change for avoiding class collision. **Default**: 0.5

>> **Parameters**
>>> **rel_radius_change_class_collision** – The relative radius change for avoiding class collision [-]

class +tc_toolbox.+precipitation.**ParticleSizeDistribution**

> Represents the state of a microstructure evolution at a certain time including its particle size distribution, composition and overall phase fraction.

> **ParticleSizeDistribution**()

>> Constructs an instance of *ParticleSizeDistribution*.

> **add_radius_and_number_density**(*radius*, *number_density*)

>> Adds a radius and number density pair to the particle size distribution.

>>> **Parameters**

>>> - **radius** – The radius [m]

>>> - **number_density** – The number of particles per unit volume per unit length [m^-4]

>>> **Returns**
>>>> This *ParticleSizeDistribution* object

> **set_initial_composition**(*element_name*, *composition_value*)

>> Sets the initial precipitate composition.

>>> **Parameters**

>>> - **element_name** – The name of the element

>>> - **composition_value** – The composition value [composition unit defined for the calculation]

>>> **Returns**
>>>> This *ParticleSizeDistribution* object

> **set_volume_fraction_of_phase_type**(*volume_fraction_of_phase_type_enum*)

>> Sets the type of the phase fraction or percentage. **Default**: By default volume fraction is used.

>>> **Parameters**
>>>> **volume_fraction_of_phase_type_enum** – Specifies if volume percent or fraction is used

>>> **Returns**
>>>> This *ParticleSizeDistribution* object

> **set_volume_fraction_of_phase_value**(*value*)

>> Sets the overall volume fraction of the phase (unit based on the setting of *set_volume_fraction_of_phase_type()*).

>>> **Parameters**
>>>> **value** – The volume fraction 0.0 - 1.0 or percent value 0 - 100

>>> **Returns**
>>>> This *ParticleSizeDistribution* object

class +tc_toolbox.+precipitation.**PrecipitateElasticProperties**

> Represents the elastic transformation strain of a certain precipitate class.

Note: This class is only relevant if the option TransformationStrainCalculationOption.USER_DEFINED has been chosen using *PrecipitatePhase.set_transformation_strain_calculation_option()*. The elastic strain can only be considered for non-spherical precipitates.

PrecipitateElasticProperties()

> Constructs an instance of *PrecipitateElasticProperties*.

set_e11(*e11*)

> Sets the elastic strain tensor component e11. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e11** – The elastic strain tensor component e11
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

set_e12(*e12*)

> Sets the strain tensor component e12. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e12** – The elastic strain tensor component e12
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

set_e13(*e13*)

> Sets the elastic strain tensor component e13. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e13** – The elastic strain tensor component e13
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

set_e22(*e22*)

> Sets the elastic strain tensor component e22. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e22** – The elastic strain tensor component e22
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

set_e23(*e23*)

> Sets the elastic strain tensor component e23. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e23** – The elastic strain tensor component e23
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

set_e33(*e33*)

> Sets the elastic strain tensor component e33. **Default**: 0.0
>
> > **Parameters**
> >
> > > **e33** – The elastic strain tensor component e33
> >
> > **Returns**
> >
> > > This *PrecipitateElasticProperties* object

---

class +tc_toolbox.+precipitation.**PrecipitateMorphology**

    Available precipitate morphologies.

class +tc_toolbox.+precipitation.**PrecipitatePhase**(*precipitate_phase_name*)

    Represents a certain precipitate class (i.e. a group of precipitates with the same phase and settings).

    PrecipitatePhase(*precipitate_phase_name*)

    disable_calculate_aspect_ratio_from_elastic_energy()

        Disables the automatic calculation of the aspect ratio from the elastic energy of the phase.

        Default: This is the default setting (with an aspect ratio of 1.0).

        Note: If you use this method, you are required to set the aspect ratio explicitly using the method *set_aspect_ratio_value()*.

        Returns

            This *PrecipitatePhase* object

    disable_driving_force_approximation()

        Disables driving force approximation for this precipitate class. **Default**: Driving force approximation is disabled.

        Returns

            This *PrecipitatePhase* object

    enable_calculate_aspect_ratio_from_elastic_energy()

        Enables the automatic calculation of the aspect ratio from the elastic energy of the phase. **Default**: The aspect ratio is set to a value of 1.0.

        Returns

            This *PrecipitatePhase* object

    enable_driving_force_approximation()

        Enables driving force approximation for this precipitate class. This approximation is often required when simulating precipitation of multiple particles that use the same phase description. E.g. simultaneous precipitation of a Metal-Carbide(MC) and Metal-Nitride(MN) if configured as different composition sets of the same phase FCC_A1. **Default**: Driving force approximation is disabled.

        Tip: Use this if simulations with several compositions sets of the same phase cause problems.

        Returns

            This *PrecipitatePhase* object

    set_alias(*alias*)

        Sets an alias string that can later be used to get values from a calculated result. Typically used when having the same phase for several precipitates, but with different nucleation sites. For example two precipitates of the phase M7C3 with nucleation sites in 'Bulk' and at 'Dislocations'. The alias can be used instead of the phase name when retrieving simulated results.

        Note: Typically used when having using the same precipitate phase, but with different settings in the same calculation.

> **Parameters**
>   `alias` – The alias string for this class of precipitates
>
> **Returns**
>   This *PrecipitatePhase* object

set_aspect_ratio_value(*aspect_ratio_value*)

Sets the aspect ratio of the phase. **Default**: An aspect ratio of 1.0.

Note: Only relevant if `disable_calculate_aspect_ratio_from_elastic_energy()` is used (which is the default).

> **Parameters**
>   `aspect_ratio_value` – The aspect ratio value
>
> **Returns**
>   This *PrecipitatePhase* object

set_gibbs_energy_addition(*gibbs_energy_addition*)

Sets a Gibbs energy addition to the Gibbs energy of the phase. **Default**: 0,0 J/mol

> **Parameters**
>   `gibbs_energy_addition` – The Gibbs energy addition [J/mol]
>
> **Returns**
>   This *PrecipitatePhase* object

set_interfacial_energy(*interfacial_energy*)

Sets the interfacial energy. **Default**: If the interfacial energy is not set, it is automatically calculated using a broken-bond model.

Note: The calculation of the interfacial energy using a broken-bond model is based on the assumption of an interface between a bcc- and a fcc-crystal structure with (110) and (111) lattice planes regardless of the actual phases.

> **Parameters**
>   `interfacial_energy` – The interfacial energy [J/m^2]
>
> **Returns**
>   This *PrecipitatePhase* object

set_interfacial_energy_estimation_prefactor(*interfacial_energy_estimation_prefactor*)

Sets the interfacial energy prefactor. **Default**: Prefactor of 1.0 (only relevant if the interfacial energy is automatically calculated).

Note: The interfacial energy prefactor is an amplification factor for the automatically calculated interfacial energy. Example: *interfacial_energy_estimation_prefactor = 2.5 => 2.5 * calculated interfacial energy*

> **Parameters**
>   `interfacial_energy_estimation_prefactor` – The prefactor for the calculated interfacial energy

> **Returns**
> This *PrecipitatePhase* object

set_molar_volume(*volume*)

> Sets the molar volume of the precipitate phase. **Default**: The molar volume obtained from the database. If no molar volume information is present in the database, a value of 7.0e-6 m^3/mol is used.
>
> > **Parameters**
> > **volume** – The molar volume [m^3/mol]
> >
> > **Returns**
> > This *PrecipitatePhase* object

set_nucleation_at_dislocations(*number_density*)

> Activates nucleation at dislocations for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default**: If not set, by default bulk nucleation is chosen.
>
> > **Parameters**
> > **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size, dislocation density) [m^-3].
> >
> > **Returns**
> > This *PrecipitatePhase* object

set_nucleation_at_grain_boundaries(*wetting_angle*, *number_density*)

> Activates nucleation at grain boundaries for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default**: If not set, by default bulk nucleation is chosen.
>
> > **Parameters**
> > - **wetting_angle** – If not set, a default value of 90 degrees is used
> > - **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) [m^-3].
> >
> > **Returns**
> > This *PrecipitatePhase* object

set_nucleation_at_grain_corners(*wetting_angle*, *number_density*)

> Activates nucleation at grain corners for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default**: If not set, by default bulk nucleation is chosen.
>
> > **Parameters**
> > - **wetting_angle** – If not set, a default value of 90 degrees is used]
> > - **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) [m^-3].
> >
> > **Returns**
> > This *PrecipitatePhase* object

set_nucleation_at_grain_edges(*wetting_angle*, *number_density*)

> Activates nucleation at the grain edges for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default**: If not set, by default bulk nucleation is chosen.
>
> > **Parameters**
> > - **wetting_angle** – If not set, a default value of 90 degrees is used

- **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (grain size) [m^-3].

> **Returns**
>> This *PrecipitatePhase* object

set_nucleation_in_bulk(*number_density*)

> Activates nucleation in the bulk for this class of precipitates. Calling the method overrides any other nucleation setting for this class of precipitates. **Default**: This is the default setting (with an automatically calculated number density).

>> **Parameters**
>>> **number_density** – Number density of nucleation sites. If not set, the value is calculated based on the matrix settings (molar volume) [m^-3]

>> **Returns**
>>> This *PrecipitatePhase* object

set_phase_boundary_mobility(*phase_boundary_mobility*)

> Sets the phase boundary mobility. **Default**: 10.0 m^4/(Js).

>> **Parameters**
>>> **phase_boundary_mobility** – The phase boundary mobility [m^4/(Js)]

>> **Returns**
>>> This *PrecipitatePhase* object

set_precipitate_morphology(*precipitate_morphology_enum*)

> Sets the precipitate morphology. **Default**: `PrecipitateMorphology.SPHERE`

>> **Parameters**
>>> **precipitate_morphology_enum** – The precipitate morphology

>> **Returns**
>>> This *PrecipitatePhase* object

set_trans_interface_mobility_adjustment(*element*, *prefactor*, *activation_energy*)

> Trans-interface mobility adjustment Only relevant when growth rate model is PE Automatic A value that adds to the activation energy of mobility data from the database.

>> **Parameters**

>>> - **element** – The element to apply the adjustment for. If "all" is given, adjustment will apply to all elements.

>>> - **prefactor** – A parameter that multiplies to the mobility data from a database. The value scales the mobility by a constant amount. This results in the trans-interface mobility that controls the kinetics of Para-Equilibrium to Ortho-Equilibrium transition.

>>> - **activation_energy** – A value that adds to the activation energy of mobility data from a database.It scales the mobility by a temperature dependent amount. Similar usage as trans-interface mobility adjustment prefactor. [J/mol]

set_transformation_strain_calculation_option(*transformation_strain_calculation_option_enum*)

> Sets the transformation strain calculation option. **Default**: `TransformationStrainCalculationOption.DISREGARD`.

>> **Parameters**
>>> **transformation_strain_calculation_option_enum** – The chosen option

>> **Returns**
>>> This *PrecipitatePhase* object

---

set_zener_pinning_parameters(*cutoff_size*, *kinetic_prefactor*, *exponent*)

These parameters are only relevant when zener pinning is enabled in the matrix phase

> **Parameters**
>> • **cutoff_size** – Precipitates with radius smaller than this value are neglected in pinning force calculation.
>>
>> • **kinetic_prefactor** – Dimensionless kinetic coefficient in Zener equation.
>>
>> • **exponent** – Exponent of precipitate volume fraction in Zener equation.
>
> **Returns**
>> This *PrecipitatePhase* object

with_elastic_properties(*elastic_properties*)

Sets the elastic properties. **Default**: The elastic transformation strain is disregarded by default.

Note: This method has only an effect if the option TransformationStrainCalculationOption. USER_DEFINED is chosen using the method *set_transformation_strain_calculation_option()*.

> **Parameters**
>> **elastic_properties** – The elastic properties object
>
> **Returns**
>> This *PrecipitatePhase* object

with_growth_rate_model(*growth_rate_model_enum*)

Sets the growth rate model for the class of precipitates. **Default**: GrowthRateModel.SIMPLIFIED

> **Parameters**
>> **growth_rate_model_enum** – The growth rate model
>
> **Returns**
>> This *PrecipitatePhase* object

with_particle_size_distribution(*particle_size_distribution*)

Sets the initial particle size distribution for this class of precipitates. **Default**: If the initial particle size distribution is not explicitly provided, the simulation will start from a supersaturated matrix.

Tip: Use this option if you want to study the further evolution of an existing microstructure.

> **Parameters**
>> **particle_size_distribution** – The initial particle size distribution object
>
> **Returns**
>> This *PrecipitatePhase* object

class +tc_toolbox.+precipitation.**PrecipitationCCTCalculation**(*back*)

Configuration for a Continuous-Cooling-Time (CCT) precipitation calculation.

PrecipitationCCTCalculation(*back*)

Call base constructor: tc_toolbox.AbstractCalculation.

calculate(*timeout_in_minutes*)

Runs the CCT diagram calculation.

> **Parameters**
>> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

> **Returns**
>> A *PrecipitationCalculationTTTorCCTResult* which later can be used to get specific values from the calculated result

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
>> The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

set_composition(*element_name*, *value*)

Sets the composition of the elements. The unit for the composition can be changed using *set_composition_unit()*. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT)

> **Parameters**

>> • **element_name** – The element

>> • **value** – The composition (fraction or percent depending on the composition unit)

> **Returns**
>> This *PrecipitationCCTCalculation* object

set_composition_unit(*unit_enum*)

Sets the composition unit. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT).

> **Parameters**
>> **unit_enum** – The new composition unit

> **Returns**
>> This *PrecipitationCCTCalculation* object

set_cooling_rates(*cooling_rates*)

> Sets all cooling rates for which the CCT diagram should be calculated.

> > **Parameters**
> > > **cooling_rates** – A list of cooling rates [K/s]

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

set_max_temperature(*max_temperature*)

> Sets maximum temperature of the CCT diagram.

> > **Parameters**
> > > **max_temperature** – the maximum temperature [K]

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

set_min_temperature(*min_temperature*)

> Sets the minimum temperature of the CCT diagram.

> > **Parameters**
> > > **min_temperature** – the minimum temperature [K]

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

stop_at_volume_fraction_of_phase(*stop_criterion_value*)

> Sets the stop criterion as a volume fraction of the phase. This setting is applied to all phases.

> > **Parameters**
> > > **stop_criterion_value** – the volume fraction of the phase (a value between 0 and 1)

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

with_matrix_phase(*matrix_phase*)

> Sets the matrix phase.

> > **Parameters**
> > > **matrix_phase** – The matrix phase

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

with_numerical_parameters(*numerical_parameters*)

> Sets the numerical parameters. If not specified, reasonable defaults are be used.

> > **Parameters**
> > > **numerical_parameters** – The parameters

> > **Returns**
> > > This *PrecipitationCCTCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
>> **system_modifications** – The system modification to be performed
>
> **Returns**
>> This *PrecipitationCCTCalculation* object

class +tc_toolbox.+precipitation.**PrecipitationCalculationResult**(*back*)

> Result of a precipitation calculation. This can be used to query for specific values.

> **PrecipitationCalculationResult**(*back*)
>> Call base constructor: tc_toolbox.AbstractResult.

> **invalidate**()
>> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

> **save_to_disk**(*path*)
>> Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()
>>
>> **Parameters**
>>> **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
>>
>> **Returns**
>>> this *PrecipitationCalculationResult* object

class +tc_toolbox.+precipitation.**PrecipitationCalculationSingleResult**(*back*)

> Result of a isothermal or non-isothermal precipitation calculation. This can be used to query for specific values.

> Search the Thermo-Calc help for definitions of the axis variables, e.g. search *isothermal variables* or *non-isothermal variables*.

> **PrecipitationCalculationSingleResult**(*back*)
>> Call base constructor: tc_toolbox.precipitation.PrecipitationCalculationResult.

> **get_aspect_ratio_distribution_for_particle_length_of**(*precipitate_id*, *time*)
>> Returns the aspect ratio distribution of a precipitate in dependency of its mean particle length at a certain time.
>>
>> Only available if the morphology is set to PrecipitateMorphology.NEEDLE or PrecipitateMorphology.PLATE.
>>
>> **Parameters**
>>> - **time** – The time [s]
>>>
>>> - **precipitate_id** – The id of a precipitate can either be the phase name or an alias
>>
>> **Returns**
>>> A tuple of two lists of floats (mean particle length [m], aspect ratio)

> **get_aspect_ratio_distribution_for_radius_of**(*precipitate_id*, *time*)
>> Returns the aspect ratio distribution of a precipitate in dependency of its mean radius at a certain time.
>>
>> Only available if the morphology is set to PrecipitateMorphology.NEEDLE or PrecipitateMorphology.PLATE.
>>
>> **Parameters**
>>> - **time** – The time [s]
>>>
>>> - **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns
A tuple of two lists of floats (mean radius [m], aspect ratio)

get_critical_radius_of(*precipitate_id*)

Returns the critical radius of a precipitate in dependency of the time.

Parameters
precipitate_id – The id of a precipitate can either be phase name or alias

Returns
A tuple of two lists of floats (time [s], critical radius [m])

get_cubic_factor_distribution_for_particle_length_of(*precipitate_id*, *time*)

Returns the cubic factor distribution of a precipitate in dependency of its mean particle length at a certain time.

Only available if the morphology is set to `PrecipitateMorphology.CUBOID`.

Parameters

- **time** – The time in seconds

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns
A tuple of two lists of floats (particle length [m], cubic factor)

get_cubic_factor_distribution_for_radius_of(*precipitate_id*, *time*)

Returns the cubic factor distribution of a precipitate in dependency of its mean radius at a certain time. Only available if the morphology is set to `PrecipitateMorphology.CUBOID`.

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns
A tuple of two lists of floats (radius [m], cubic factor)

get_driving_force_of(*precipitate_id*)

Returns the (by R * T) normalized driving force of a precipitate in dependency of the time.

Parameters
precipitate_id – The id of a precipitate can either be the phase name or an alias

Returns
A tuple of two lists of floats (time [s], normalized driving force)

get_grain_critical_radius()

Returns the critical radius of grains in dependency of the time.

Returns
A tuple of two lists of floats (time [s], critical radius [m])

get_grain_mean_radius()

Returns the mean grain size of the matrix phase in dependency of the time.

Returns
A tuple of two lists of floats (time [s], mean radius [m])

get_grain_number_density()

> Returns the grain number density (concentration) in dependency of the time.
>
> > **Returns**
> >
> > A tuple of two lists of floats (time [s], grain number density [m^-3])

get_grain_number_density_distribution_for_length(*time*)

> Returns the number density distribution of grains in terms of length at the requested time(s).
>
> > **Parameters**
> >
> > **time** – The time [s]
> >
> > **Returns**
> >
> > A tuple of two lists of floats (grain length[m], number of grains per unit volume in a size class [m^-3])

get_grain_number_density_distribution_for_radius(*time*)

> Returns the number density distribution of a grains in terms of grain radius at the requested time(s).
>
> > **Parameters**
> >
> > **time** – The time [s]
> >
> > **Returns**
> >
> > A tuple of two lists of floats (radius [m], number of grains per unit volume in a size class [m^-3])

get_grain_size_distribution(*time*)

> Returns the size distribution of the matrix phase in dependency of its grain radius at the requested time(s).
>
> > **Parameters**
> >
> > **time** – The time [s]
> >
> > **Returns**
> >
> > A tuple of two lists of floats (grain radius[m], number density of grains[m^-4])

get_matrix_composition_in_mole_fraction_of(*element_name*)

> Returns the matrix composition (as mole fractions) of a certain element in dependency of the time.
>
> > **Parameters**
> >
> > **element_name** – The element
> >
> > **Returns**
> >
> > A tuple of two lists of floats (time [s], mole fraction)

get_matrix_composition_in_weight_fraction_of(*element_name*)

> Returns the matrix composition (as weight fraction) of a certain element in dependency of the time.
>
> > **Parameters**
> >
> > **element_name** – The element
> >
> > **Returns**
> >
> > A tuple of two lists of floats (time [s], weight fraction)

get_mean_aspect_ratio_of(*precipitate_id*)

> Returns the mean aspect ratio of a precipitate in dependency of the time.
>
> Only available if the morphology is set to `PrecipitateMorphology.NEEDLE` or `PrecipitateMorphology.PLATE`.
>
> > **Parameters**
> >
> > **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns
> A tuple of two lists of floats (time [s], mean aspect ratio)

get_mean_cubic_factor_of(*precipitate_id*)

> Returns the mean cubic factor of a precipitate in dependency of the time. Only available if the morphology is set to `PrecipitateMorphology.CUBOID`.

> **Parameters**
> > **precipitate_id** – The id of a precipitate can either be the phase name or an alias

> **Returns**
> > A tuple of two lists of floats (time [s], mean cubic factor)

get_mean_particle_length_of(*precipitate_id*)

> Returns the mean particle length of a precipitate in dependency of the time.

> Only available if the morphology is set to `PrecipitateMorphology.NEEDLE` or `PrecipitateMorphology.PLATE`.

> **Parameters**
> > **precipitate_id** – The id of a precipitate can either be the phase name or an alias

> **Returns**
> > A tuple of two lists of floats (time [s], mean particle length [m])

get_mean_radius_2d_of(*precipitate_id*)

> Returns the mean radius of cross-sections taken through the dispersion in dependency of the time.

> **Parameters**
> > **precipitate_id** – The id of a precipitate can either be phase name or alias

> **Returns**
> > A tuple of two lists of floats (time [s], mean radius [m])

get_mean_radius_of(*precipitate_id*)

> Returns the mean radius of a precipitate in dependency of the time.

> **Parameters**
> > **precipitate_id** – The id of a precipitate can either be phase name or alias

> **Returns**
> > A tuple of two lists of floats (time [s], mean radius [m])

get_normalized_grain_size_distribution(*time*)

> Returns the normalized number density distribution with the grain radius normalized by the mean radius, for the requested time(s).

> **Parameters**
> > **time** – The time [s]

> **Returns**
> > A tuple of two lists of floats (Normalized size, Probability)

get_normalized_number_density_distribution_2d_of(*precipitate_id*, *time*)

> Returns the normalized number density distribution of a precipitate in terms of the radius of cross-sections created by taking a plane through the dispersion, normalized by the mean radius of the cross-section, for the requested time(s).

> **Parameters**
> > - **time** – The time [s]
> > - **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (Normalized size, Probability)

get_normalized_number_density_distribution_of(*precipitate_id*, *time*)

Returns the normalized number density distribution with the particle radius normalized by the mean radius, for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (Normalized size, Probability)

get_nucleation_rate_of(*precipitate_id*)

Returns the nucleation rate of a precipitate in dependency of the time.

Parameters

precipitate_id – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (time [s], nucleation rate [m^-3 s^-1])

get_number_density_distribution_2d_for_particle_length_of(*precipitate_id*, *time*)

Returns the number density distribution of a precipitate considering the radius of cross-sections created by taking a plane through the dispersion, approximating the particles as spherical, for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (particle length[m], number of particles per unit area within a size class [m^-2])

get_number_density_distribution_2d_for_radius_of(*precipitate_id*, *time*)

Returns the number density distribution of a precipitate considering the radius of cross-sections created by taking a plane through the dispersion for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (radius [m], number of particles per unit area within a size class [m^-2])

get_number_density_distribution_for_particle_length_of(*precipitate_id*, *time*)

Returns the number density distribution of a precipitate in terms of length for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

> **Returns**
> A tuple of two lists of floats (particle length[m], number of particles in the size class per unit volume [m^-3])

get_number_density_distribution_for_radius_of(*precipitate_id*, *time*)

Returns the number density distribution of a precipitate in terms of radius for the requested time(s).

> **Parameters**
>
> - **time** – The time [s]
>
> - **precipitate_id** – The id of a precipitate can either be the phase name or an alias
>
> **Returns**
> A tuple of two lists of floats (radius [m], number of particles in the size class per unit volume [m^-3])

get_number_density_of(*precipitate_id*)

Returns the particle number density (concentration) of a precipitate in dependency of the time.

> **Parameters**
> **precipitate_id** – The id of a precipitate can either be phase name or alias
>
> **Returns**
> A tuple of two lists of floats (time [s], particle number density [m^-3])

get_precipitate_composition_in_mole_fraction_of(*precipitate_id*, *element_name*)

Returns the precipitate composition (as mole fractions) of a certain element in dependency of the time.

> **Parameters**
>
> - **precipitate_id** – The id of a precipitate can either be phase name or alias
>
> - **element_name** – The element
>
> **Returns**
> A tuple of two lists of floats (time [s], mole fraction)

get_precipitate_composition_in_weight_fraction_of(*precipitate_id*, *element_name*)

Returns the precipitate composition (as weight fraction) of a certain element in dependency of the time.

> **Parameters**
>
> - **precipitate_id** – The id of a precipitate can either be phase name or alias
>
> - **element_name** – The element
>
> **Returns**
> A tuple of two lists of floats (time [s], weight fraction)

get_size_distribution_2d_for_particle_length_of(*precipitate_id*, *time*)

Returns the size distribution of a precipitate considering the radius of cross-sections created by taking a plane through the dispersion, approximating the particles as spherical, for the requested time(s).

> **Parameters**
>
> - **time** – The time [s]
>
> - **precipitate_id** – The id of a precipitate can either be the phase name or an alias
>
> **Returns**
> A tuple of two lists of floats (particle length[m], number of particles per unit area per unit length [m^-3])

get_size_distribution_2d_for_radius_of(*precipitate_id*, *time*)

Returns the size distribution of a precipitate considering the radius of cross-sections created by taking a plane through the dispersion for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (radius [m], number of particles per unit area per unit length [m^-3])

get_size_distribution_for_particle_length_of(*precipitate_id*, *time*)

Returns the size distribution of a precipitate in terms of length for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (particle length[m], number of particles per unit volume per unit length [m^-4])

get_size_distribution_for_radius_of(*precipitate_id*, *time*)

Returns the size distribution of a precipitate in terms of radius for the requested time(s).

Parameters

- **time** – The time [s]

- **precipitate_id** – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (radius [m], number of particles per unit volume per unit length [m^-4])

get_volume_fraction_of(*precipitate_id*)

Returns the volume fraction of a precipitate in dependency of the time.

Parameters

precipitate_id – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (time [s], volume fraction)

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

save_to_disk(*path*)

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()

Parameters

path – the path to the folder you want the result to be saved in. It can be relative or absolute.

Returns

this *PrecipitationCalculationResult* object

---

class +tc_toolbox.+precipitation.**PrecipitationCalculationTTTorCCTResult**(*back*)

Result of a TTT or CCT precipitation calculation.

PrecipitationCalculationTTTorCCTResult(*back*)

Call base constructor: tc_toolbox.precipitation.PrecipitationCalculationResult.

get_result_for_precipitate(*precipitate_id*)

Returns the calculated data of a TTT or CCT diagram for a certain precipitate.

Parameters

precipitate_id – The id of a precipitate can either be the phase name or an alias

Returns

A tuple of two lists of floats (time [s], temp [K])

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with load_result_from_disk()

Parameters

path – the path to the folder you want the result to be saved in. It can be relative or absolute.

Returns

this *PrecipitationCalculationResult* object

class +tc_toolbox.+precipitation.**PrecipitationIsoThermalCalculation**(*back*)

Configuration for an isothermal precipitation calculation.

PrecipitationIsoThermalCalculation(*back*)

Call base constructor: tc_toolbox.AbstractCalculation.

calculate(*timeout_in_minutes*)

Runs the isothermal precipitation calculation.

Parameters

timeout_in_minutes – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns

A *PrecipitationCalculationSingleResult* which later can be used to get specific values from the calculated result

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
> > The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

set_composition(*element_name*, *value*)

> Sets the composition of the elements. The unit for the composition can be changed using *set_composition_unit()*. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT)

> > **Parameters**
> >
> > - **element_name** – The element
> >
> > - **value** – The composition (fraction or percent depending on the composition unit)
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

set_composition_unit(*unit_enum*)

> Sets the composition unit. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT).

> > **Parameters**
> > > **unit_enum** – The new composition unit
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

set_simulation_time(*simulation_time*)

> Sets the simulation time.

> > **Parameters**
> > > **simulation_time** – The simulation time [s]
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

set_temperature(*temperature*)

> Sets the temperature for the isothermal simulation.

> > **Parameters**
> > > **temperature** – the temperature [K]
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

with_matrix_phase(*matrix_phase*)

> Sets the matrix phase.

> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

with_numerical_parameters(*numerical_parameters*)

> Sets the numerical parameters. If not specified, reasonable defaults are be used.
>
> > **Parameters**
> > > **numerical_parameters** – The parameters
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).
>
> ---
>
> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.
>
> ---
>
> > **Parameters**
> > > **system_modifications** – The system modification to be performed
> >
> > **Returns**
> > > This *PrecipitationIsoThermalCalculation* object

class +tc_toolbox.+precipitation.**PrecipitationNonIsoThermalCalculation**(*back*)

> Configuration for a non-isothermal precipitation calculation.

> **PrecipitationNonIsoThermalCalculation**(*back*)
>
> > Call base constructor: `tc_toolbox.AbstractCalculation`.

> **calculate**(*timeout_in_minutes*)
>
> > Runs the non-isothermal precipitation calculation.
> >
> > > **Parameters**
> > > > **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
> > >
> > > **Returns**
> > > > A *PrecipitationCalculationSingleResult* which later can be used to get specific values from the calculated result

> **get_configuration_as_string**()
>
> > Returns detailed information about the current state of the calculation object.
> >
> > ---
> >
> > **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.
> >
> > ---

> **get_system_data**()
>
> > Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.
> >
> > ---
> >
> > **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.
> >
> > ---

> **Returns**
> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

set_composition(*element_name*, *value*)

> Sets the composition of the elements. The unit for the composition can be changed using *set_composition_unit()*. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT)
>
> > **Parameters**
> >
> > - **element_name** – The element
> >
> > - **value** – The composition (fraction or percent depending on the composition unit)
> >
> > **Returns**
> > This *PrecipitationIsoThermalCalculation* object

set_composition_unit(*unit_enum*)

> Sets the composition unit. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT).
>
> > **Parameters**
> > **unit_enum** – The new composition unit
> >
> > **Returns**
> > This *PrecipitationIsoThermalCalculation* object

set_simulation_time(*simulation_time*)

> Sets the simulation time.
>
> > **Parameters**
> > **simulation_time** – The simulation time [s]
> >
> > **Returns**
> > This PrecipitationNonThermalCalculation object

with_matrix_phase(*matrix_phase*)

> Sets the matrix phase.
>
> > **Parameters**
> > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > This *PrecipitationIsoThermalCalculation* object

with_numerical_parameters(*numerical_parameters*)

> Sets the numerical parameters. If not specified, reasonable defaults are be used.
>
> > **Parameters**
> > **numerical_parameters** – The parameters
> >
> > **Returns**
> > This *PrecipitationIsoThermalCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
>> **system_modifications** – The system modification to be performed
>
> **Returns**
>> This `PrecipitationNonThermalCalculation` object

with_temperature_profile(*temperature_profile*)

> Sets the temperature profile to use with this calculation.
>
> **Parameters**
>> **temperature_profile** – the temperature profile object (specifying time / temperature points)
>
> **Returns**
>> This `PrecipitationNonThermalCalculation` object

class +tc_toolbox.+precipitation.**PrecipitationTTTCalculation**(*back*)

> Configuration for a TTT (Time-Temperature-Transformation) precipitation calculation.
>
> **PrecipitationTTTCalculation**(*back*)
>> Call base constructor: `tc_toolbox.AbstractCalculation`.
>
> **calculate**(*timeout_in_minutes*)
>> Runs the TTT diagram calculation.
>>
>> **Parameters**
>>> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
>>
>> **Returns**
>>> A *PrecipitationCalculationTTTorCCTResult* which later can be used to get specific values from the calculated result.
>
> **get_configuration_as_string**()
>> Returns detailed information about the current state of the calculation object.
>>
>> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.
>
> **get_system_data**()
>> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.
>>
>> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as *.tdb*-file.
>>
>> **Returns**
>>> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

set_composition(*element_name*, *value*)

> Sets the composition of the elements. The unit for the composition can be changed using *set_composition_unit()*. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT)
>
> > **Parameters**
> >
> > - **element_name** – The element
> >
> > - **value** – The composition (fraction or percent depending on the composition unit)
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

set_composition_unit(*unit_enum*)

> Sets the composition unit. **Default**: Mole percent (CompositionUnit.MOLE_PERCENT).
>
> > **Parameters**
> > > **unit_enum** – The new composition unit
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

set_max_annealing_time(*max_annealing_time*)

> Sets the maximum annealing time, i.e. the maximum time of the simulation if the stopping criterion is not reached.
>
> > **Parameters**
> > > **max_annealing_time** – the maximum annealing time [s]
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

set_max_temperature(*max_temperature*)

> Sets the maximum temperature for the TTT diagram.
>
> > **Parameters**
> > > **max_temperature** – the maximum temperature [K]
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

set_min_temperature(*min_temperature*)

> Sets the minimum temperature for the TTT diagram.
>
> > **Parameters**
> > > **min_temperature** – the minimum temperature [K]
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

set_temperature_step(*temperature_step*)

> Sets the temperature step for the TTT diagram. If not set, the default value is 10 K.
>
> > **Parameters**
> > > **temperature_step** – the temperature step [K]
> >
> > **Returns**
> > > This *PrecipitationTTTCalculation* object

stop_at_percent_of_equilibrium_fraction(*percentage*)

> Sets the stop criterion to a percentage of the overall equilibrium phase fraction, alternatively a required volume fraction can be specified (using *stop_at_volume_fraction_of_phase()*).

> > **Parameters**
> > > **percentage** – the percentage to stop at (value between 0 and 100)

> > **Returns**
> > > This *PrecipitationTTTCalculation* object

stop_at_volume_fraction_of_phase(*volume_fraction*)

> Sets the stop criterion as a volume fraction of the phase, alternatively a required percentage of the equilibrium phase fraction can be specified (using stop_at_percent_of_equilibria_fraction()). Stopping at a specified volume fraction is the default setting.

> This setting is applied to all phases.

> > **Parameters**
> > > **volume_fraction** – the volume fraction to stop at (a value between 0 and 1)

> > **Returns**
> > > This *PrecipitationTTTCalculation* object

with_matrix_phase(*matrix_phase*)

> Sets the matrix phase.

> > **Parameters**
> > > **matrix_phase** – The matrix phase

> > **Returns**
> > > This *PrecipitationTTTCalculation* object

with_numerical_parameters(*numerical_parameters*)

> Sets the numerical parameters. If not specified, reasonable defaults are be used.

> > **Parameters**
> > > **numerical_parameters** – The parameters

> > **Returns**
> > > This *PrecipitationTTTCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

> ---
> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a *.tdb-file.
> ---

> > **Parameters**
> > > **system_modifications** – The system modification to be performed

> > **Returns**
> > > This *PrecipitationTTTCalculation* object

class +tc_toolbox.+precipitation.**TransformationStrainCalculationOption**

> Options for calculating the transformation strain.

class +tc_toolbox.+precipitation.**VolumeFractionOfPhaseType**

> Unit of the volume fraction of a phase.

4.1.4 Package "scheil"

class +tc_toolbox.+scheil.**CalculateSecondaryDendriteArmSpacing**

Configures a secondary dendrite arm spacing calculation used by Scheil *with back diffusion*. The used equation is `c * cooling_rate^(-n)` with `c` and `n` being provided either by the user or taken from the defaults.

CalculateSecondaryDendriteArmSpacing()

Configures a secondary dendrite arm spacing calculation used by Scheil *with back diffusion*. The used equation is `c * cooling_rate^(-n)` with `c` and `n` being provided either by the user or taken from the defaults. Constructs an instance of *CalculateSecondaryDendriteArmSpacing*.

static calculate_secondary_dendrite_arm_spacing()

Calculate the secondary dendrite arm spacing based on the following equation: `c * cooling_rate^(-n)` with `c` and `n` being provided either by the user or taken from the defaults.

Use the methods provide by *CalculateSecondaryDendriteArmSpacing* to configure the parameters.

> **Returns**
> A *CalculateSecondaryDendriteArmSpacing*

static constant_secondary_dendrite_arm_spacing(*secondary_dendrite_arm_spacing*)

Assuming constant secondary dendrite arm spacing, provided either by the user or taken from the defaults.

Default: 50 μm

> **Parameters**
> **secondary_dendrite_arm_spacing** – The dendrite arm spacing [m]
>
> **Returns**
> A *ConstantSecondaryDendriteArmSpacing*

disable_delta_ferrite_to_austenite_transition()

Turns off the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off. :return: This *CalculateSecondaryDendriteArmSpacing* object

enable_delta_ferrite_to_austenite_transition()

Turns on the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off. :return: This *CalculateSecondaryDendriteArmSpacing* object

static scheil_back_diffusion()

Configuration for *back diffusion in the solid primary phase*.

> **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done. :return: A *ScheilBackDiffusion*

static scheil_classic()

Configuration for Classic Scheil with fast diffusers. :return: A *ScheilClassic*

static scheil_solute_trapping()

Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A *ScheilSoluteTrapping*

set_c(*c*)

Sets the scaling factor `c` in the governing equation `c * cooling_rate^(-n)`.

Default: 50 μm

> **Parameters**
> > `c` – The scaling factor [m]
>
> **Returns**
> > This *CalculateSecondaryDendriteArmSpacing* object

set_cooling_rate(*cooling_rate*)

Sets the cooling rate.

Default: 1.0 K/s

An increased value moves the result from equilibrium toward a Scheil-Gulliver calculation.

> **Parameters**
> > `cooling_rate` – The cooling rate [K/s]
>
> **Returns**
> > This *CalculateSecondaryDendriteArmSpacing* object

set_fast_diffusing_elements(*element_names*)

Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.

Default: No fast-diffusing elements.

> **Parameters**
> > `element_names` – The elements
>
> **Returns**
> > This *CalculateSecondaryDendriteArmSpacing* object

set_n(*n*)

Sets the exponent `n` in the governing equation `c * cooling_rate^(-n)`.

Default: 0.33

> **Parameters**
> > `n` – The exponent [-]
>
> **Returns**
> > This *CalculateSecondaryDendriteArmSpacing* object

set_primary_phasename(*primary_phase_name*)

Sets the name of the primary phase.

The primary phase is the phase where the back diffusion takes place. If *AUTOMATIC* is selected, the program tries to find the phase which will give the most back diffusion. That behavior can be overridden by selecting a specific primary phase.

Default: *AUTOMATIC*

> **Parameters**
> > `primary_phase_name` – The phase name (or *AUTOMATIC*)
>
> **Returns**
> > This *CalculateSecondaryDendriteArmSpacing* object

class `+tc_toolbox.+scheil.`**`ConstantSecondaryDendriteArmSpacing`**(*secondary_dendrite_arm_spacing*)

Configures a constant secondary dendrite arm spacing used by Scheil *with back diffusion*. The secondary dendrite arm spacing can either be provided by the user or taken from the defaults.

`ConstantSecondaryDendriteArmSpacing`(*secondary_dendrite_arm_spacing*)

Configures a constant secondary dendrite arm spacing used by Scheil *with back diffusion*. The secondary dendrite arm spacing can either be provided by the user or taken from the defaults.

Default: 50 μm

> **Parameters**
> **`secondary_dendrite_arm_spacing`** – The dendrite arm spacing [m]

static **`calculate_secondary_dendrite_arm_spacing`**()

Calculate the secondary dendrite arm spacing based on the following equation: `c * cooling_rate^(-n)` with `c` and `n` being provided either by the user or taken from the defaults.

Use the methods provide by *`CalculateSecondaryDendriteArmSpacing`* to configure the parameters.

> **Returns**
> A *`CalculateSecondaryDendriteArmSpacing`*

static **`constant_secondary_dendrite_arm_spacing`**(*secondary_dendrite_arm_spacing*)

Assuming constant secondary dendrite arm spacing, provided either by the user or taken from the defaults.

Default: 50 μm

> **Parameters**
> **`secondary_dendrite_arm_spacing`** – The dendrite arm spacing [m]

> **Returns**
> A *`ConstantSecondaryDendriteArmSpacing`*

`disable_delta_ferrite_to_austenite_transition`()

Turns off the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off.   :return: This *`ConstantSecondaryDendriteArmSpacing`* object

`enable_delta_ferrite_to_austenite_transition`()

Turns on the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off.   :return: This *`ConstantSecondaryDendriteArmSpacing`* object

static **`scheil_back_diffusion`**()

Configuration for *back diffusion in the solid primary phase*.

> **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done.   :return: A *`ScheilBackDiffusion`*

static **`scheil_classic`**()

Configuration for Classic Scheil with fast diffusers. :return: A *`ScheilClassic`*

static **`scheil_solute_trapping`**()

Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A *`ScheilSoluteTrapping`*

set_cooling_rate(*cooling_rate*)

> Sets the cooling rate.
>
> **Default**: 1.0 K/s
>
> An increased value moves the result from equilibrium toward a Scheil-Gulliver calculation.
>
> > **Parameters**
> > > **cooling_rate** – The cooling rate [K/s]
> >
> > **Returns**
> > > This *ConstantSecondaryDendriteArmSpacing* object

set_fast_diffusing_elements(*element_names*)

> Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.
>
> **Default**: No fast-diffusing elements.
>
> > **Parameters**
> > > **element_names** – The elements
> >
> > **Returns**
> > > This *ConstantSecondaryDendriteArmSpacing* object

set_primary_phasename(*primary_phase_name*)

> Sets the name of the primary phase.
>
> The primary phase is the phase where the back diffusion takes place. If *AUTOMATIC* is selected, the program tries to find the phase which will give the most back diffusion. That behavior can be overridden by selecting a specific primary phase.
>
> **Default**: *AUTOMATIC*
>
> > **Parameters**
> > > **primary_phase_name** – The phase name (or *AUTOMATIC*)
> >
> > **Returns**
> > > This *ConstantSecondaryDendriteArmSpacing* object

class +tc_toolbox.+scheil.**ScheilBackDiffusion**

> Configuration for *back diffusion in the solid primary phase*.
>
> > **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done.
>
> **static calculate_secondary_dendrite_arm_spacing**()
>
> > Calculate the secondary dendrite arm spacing based on the following equation: `c * cooling_rate^(-n)` with `c` and `n` being provided either by the user or taken from the defaults.
> >
> > Use the methods provide by *CalculateSecondaryDendriteArmSpacing* to configure the parameters.
> >
> > > **Returns**
> > > > A *CalculateSecondaryDendriteArmSpacing*
>
> **static constant_secondary_dendrite_arm_spacing**(*secondary_dendrite_arm_spacing*)
>
> > Assuming constant secondary dendrite arm spacing, provided either by the user or taken from the defaults.
> >
> > **Default**: 50 μm

>   **Parameters**
>   **secondary_dendrite_arm_spacing** – The dendrite arm spacing [m]
>
>   **Returns**
>   A *ConstantSecondaryDendriteArmSpacing*

static scheil_back_diffusion()

Configuration for *back diffusion in the solid primary phase*.

> **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done.   :return: A *ScheilBackDiffusion*

static scheil_classic()

Configuration for Classic Scheil with fast diffusers. :return: A *ScheilClassic*

static scheil_solute_trapping()

Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A *ScheilSoluteTrapping*

class +tc_toolbox.+scheil.**ScheilCalculation**(*back*)

Configuration for a Scheil solidification calculation.

> **Note:** Specify the settings, the calculation is performed with *calculate()*.

ScheilCalculation(*back*)

Call base constructor: `tc_toolbox.AbstractCalculation`.

calculate(*timeout_in_minutes*)

Runs the Scheil calculation.

>   **Parameters**
>   **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
>
>   **Returns**
>   A *ScheilCalculationResult* which later can be used to get specific values from the simulation.

disable_global_minimization()

Disables global minimization.

Default: Enabled

> **Note:** When enabled, a global minimization test is performed when an equilibrium is reached. This costs more computer time but the calculations are more robust.

>   **Returns**
>   This *ScheilCalculation* object

---

enable_global_minimization()

Enables global minimization.

Default: Enabled

Note: When enabled, a global minimization test is performed when an equilibrium is reached. This costs more computer time but the calculations are more robust.

> **Returns**
>> This *ScheilCalculation* object

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as *.tdb*-file.

> **Returns**
>> The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

set_composition(*component_name*, *value*)

Sets the composition of a component. The unit for the composition can be changed using *set_composition_unit()*.

Default: Mole percent (CompositionUnit.MOLE_PERCENT)

> **Parameters**
>> • **component_name** – The component
>>
>> • **value** – The composition value [composition unit defined for the calculation]

> **Returns**
>> This *ScheilCalculation* object

set_composition_unit(*unit_enum*)

Sets the composition unit.

Default: Mole percent (CompositionUnit.MOLE_PERCENT).

> **Parameters**
>> **unit_enum** – The new composition unit

> **Returns**
>> This *ScheilCalculation* object

set_start_temperature(*temperature_in_kelvin*)

> Sets the start temperature.

> **Default**: 2500.0 K

> ---
> **Warning:** The start temperature needs to be higher than the liquidus temperature of the alloy.
> ---

>> **Parameters**
>>> **temperature_in_kelvin** – The temperature [K]

>> **Returns**
>>> This *ScheilCalculation* object

with_calculation_type(*scheil_calculation_type*)

> Chooses a specific Scheil calculation. ClassicScheil for only setting fast diffusers, ScheilBackDiffusion enables back diffusion in the solid primary phase and optionally fast diffusers in all solid phases, and ScheilSoluteTrapping enables solute trapping in the solid primary phase. :param scheil_type: Type of Scheil calculation, either ScheilClassic, ScheilBackDiffusion or ScheilSoluteTrapping :return: This *ScheilCalculation* object

with_options(*options*)

> Sets the Scheil simulation options.

>> **Parameters**
>>> **options** – The Scheil simulation options

>> **Returns**
>>> This *ScheilCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

> ---
> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.
> ---

>> **Parameters**
>>> **system_modifications** – The system modification to be performed

>> **Returns**
>>> This *ScheilCalculation* object

class +tc_toolbox.+scheil.**ScheilCalculationResult**(*back*)

> Result of a Scheil calculation.

ScheilCalculationResult(*back*)

> Call base constructor: `tc_toolbox.AbstractResult`.

get_solid_phase_with_largest_mole_fraction()

> Returns the name of the solid phase with the largest amount in terms of mole fraction at the end of the Scheil simulation.

Returns
Phase name

get_stable_phases()

Returns all phases that were stable during a Scheil simulation.

Returns
The list of stable phases

get_values_grouped_by_quantity_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example in dependency of phases or components). Use *get_values_of()* instead if you need no separation. The available quantities can be found in the documentation of the factory class `ScheilQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Parameters

- **x_quantity** – The first Scheil quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example "T")

- **y_quantity** – The second Scheil quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example "NV")

- **sort_and_merge** – If *True*, the data is sorted and merged into as few subsections as possible (divided by *NaN*)

Returns
Containing the `ResultValueGroup` dataset objects with their *quantity labels* as keys

get_values_grouped_by_stable_phases_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the sets of "stable phases" (for example "LIQUID" or "LIQUID + FCC_A1"). Use *get_values_of()* instead if you need no separation. The available quantities can be found in the documentation of the factory class `ScheilQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Parameters

- **x_quantity** – The first Scheil quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example "T")

- **y_quantity** – The second Scheil quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example "NV")

- **sort_and_merge** – If *True*, the data will be sorted and merged into as few subsections as possible (divided by *NaN*)

Returns
Containing the `ResultValueGroup` dataset objects with their *"stable phases" labels* as keys

get_values_of(*x_quantity*, *y_quantity*)

> Returns sorted x-y-line data without any separation. Use `get_values_grouped_by_quantity_of()` or `get_values_grouped_by_stable_phases_of()` instead if you need such a separation. The available quantities can be found in the documentation of the factory class `ScheilQuantity`.

> ---
> **Note:** This method will always return sorted data without any *NaN*-values. In case of ambiguous quantities (for example: CompositionOfPhaseAsWeightFraction("FCC_A1", "All")) that can give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods.
> ---

> **Parameters**
> - **x_quantity** – The first Scheil quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example "T")
> - **y_quantity** – The second Scheil quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example "NV")

> **Returns**
> A tuple containing the x- and y-data in lists

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

> Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`

> **Parameters**
> **path** – the path to the folder you want the result to be saved in.

> **Returns**
> this `ScheilCalculationResult` object

class +tc_toolbox.+scheil.**ScheilCalculationType**

> Specific configuration for the different Scheil calculation types

static scheil_back_diffusion()

> Configuration for *back diffusion in the solid primary phase*.

> ---
> **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done.   :return: A `ScheilBackDiffusion`
> ---

static scheil_classic()

> Configuration for Classic Scheil with fast diffusers. :return: A `ScheilClassic`

static scheil_solute_trapping()

> Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A `ScheilSoluteTrapping`

class +tc_toolbox.+scheil.**ScheilClassic**

> Configuration for Classic Scheil with fast diffusers.

ScheilClassic()

Configuration for Classic Scheil when fast diffusers are included. Constructs an instance of *ScheilClassic*.

disable_delta_ferrite_to_austenite_transition()

Turns off the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off. :return: This *ScheilClassic* object

enable_delta_ferrite_to_austenite_transition()

Turns on the delta ferrite BCC to austenite FCC transition.

Default: Delta ferrite to austenite transition is off. :return: This *ScheilClassic* object

static scheil_back_diffusion()

Configuration for *back diffusion in the solid primary phase*.

> **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done. :return: A *ScheilBackDiffusion*

static scheil_classic()

Configuration for Classic Scheil with fast diffusers. :return: A *ScheilClassic*

static scheil_solute_trapping()

Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A *ScheilSoluteTrapping*

set_fast_diffusing_elements(*element_names*)

Sets elements as fast diffusing. This allows redistribution of these elements in both the solid and liquid parts of the alloy.

Default: No fast-diffusing elements.

> **Parameters**
> **element_names** – The elements
>
> **Returns**
> This *ScheilClassic* object

class +tc_toolbox.+scheil.**ScheilOptions**

Options for the Scheil simulation.

ScheilOptions()

Options for the Scheil simulation. Constructs an instance of *ScheilOptions*.

calculate_from_gas()

Calculates the evaporation temperature if a gas phase is selected in the system, and then calculates equilibria in the gas+liquid and liquid regions until liquidus temperature is reached.

Default: Calculation starts from liquidus temperature.

> **Returns**
> This *ScheilOptions* object

calculate_from_liquidus()

Solidification calculation starting from the liquidus temperature. Liquid properties between start temperature and liquidus are not obtainable.

Default: Calculation starts from liquidus temperature.

> **Returns**
> > This *ScheilOptions* object

calculate_from_start_temperature()

Calculation of equilibria from start temperature at 50 K intervals until liquidus temperature is reached. This option makes it possible to obtain properties of the liquid phase before the solidification starts.

Default: Calculation starts from liquidus temperature.

> **Returns**
> > This *ScheilOptions* object

calculate_to_end_of_scheil()

Stops the calculation when the Scheil calculation is finished.

Default: Calculation stops when the Scheil calculation is finished.

> **Returns**
> > This *ScheilOptions* object

calculate_to_temperature_below_solidus(*number_of_steps*, *final_temperature*)

Calculates properties in the solid state, for the phase compositions and fractions at the end of the Scheil calculation.

Default: Calculation stops when the Scheil calculation is finished.

> **Parameters**
>
> - **number_of_steps** – Calculates properties for the given number of temperatures, down to the final temperature.
>
> - **final_temperature** – The final (lowest) temperature where the calculation is performed.
>
> **Returns**
> > This *ScheilOptions* object

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
> > This *ScheilOptions* object

disable_control_step_size_during_minimization()

Disables stepsize control during minimization (non-global).

Default: Enabled

> **Returns**
> > This *ScheilOptions* object

disable_equilibrium_solidification_calculation()

Skips the property (one axis) diagram calculation of solidification under equilibrium conditions, before the Scheil solidification calculation starts.

In general it is not necessary to perform this calculation.

Default: Disabled. The equilibrium solidification calculation is skipped.

> **Returns**
> This *ScheilOptions* object

disable_evaporation_property_calculation()

Disables calculation of evaporation properties.

Default: Disabled. The evaporation properties are not calculated.

> **Returns**
> This *ScheilOptions* object

disable_force_positive_definite_phase_hessian()

Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

Default: Enabled

> **Returns**
> This *ScheilOptions* object

enable_approximate_driving_force_for_metastable_phases()

Enables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
> This *ScheilOptions* object

enable_control_step_size_during_minimization()

Enables stepsize control during normal minimization (non-global).

Default: Enabled

> **Returns**
> This *ScheilOptions* object

enable_equilibrium_solidification_calculation()

Performs a property (one axis) diagram calculation of solidification under equilibrium conditions, before the Scheil solidification calculation starts, in the same way as is typically done in graphical and console mode.

In general it is not necessary to perform this calculation.

Default: Disabled. The equilibrium solidification calculation is skipped.

> **Returns**
> This *ScheilOptions* object

enable_evaporation_property_calculation()

> Enables calculation of the properties molar mass of gas, driving force for evaporation and evaporation enthalpy. The calculation requires the gas phase to be selected.

> **Default**: Disabled. The evaporation properties are not calculated.

> > **Returns**
> > > This *ScheilOptions* object

enable_force_positive_definite_phase_hessian()

> Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

> **Default**: Enabled

> > **Returns**
> > > This *ScheilOptions* object

set_gas_phase(*phase_name*)

> Sets the phase used as the gas phase.

> **Default**: The phase "GAS".

> > **Parameters**
> > > **phase_name** – The phase name

> > **Returns**
> > > This *ScheilOptions* object

set_global_minimization_max_grid_points(*max_grid_points*)

> Sets the maximum number of grid points in global minimization. ** Only applicable if global minimization is actually used**.

> **Default**: 2000 points

> > **Parameters**
> > > **max_grid_points** – The maximum number of grid points

> > **Returns**
> > > This *ScheilOptions* object

set_global_minimization_test_interval(*global_test_interval*)

> Sets the interval for the global test.

> **Default**: 10

> > **Parameters**
> > > **global_test_interval** – The global test interval

> > **Returns**
> > > This *ScheilOptions* object

set_liquid_phase(*phase_name*)

> Sets the phase used as the liquid phase.

> **Default**: The phase "LIQUID".

> > **Parameters**
> > > **phase_name** – The phase name

> > **Returns**
> > > This *ScheilOptions* object

set_max_no_of_iterations(*max_no_of_iterations*)

> Set the maximum number of iterations.
>
> **Default**: max. 500 iterations
>
> ---
>
> **Note:** As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.
>
> ---
>
> > **Parameters**
> > > **max_no_of_iterations** – The max. number of iterations
> >
> > **Returns**
> > > This *ScheilOptions* object

set_required_accuracy(*accuracy*)

> Sets the required relative accuracy.
>
> **Default**: 1.0E-6
>
> ---
>
> **Note:** This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.
>
> ---
>
> > **Parameters**
> > > **accuracy** – The required relative accuracy
> >
> > **Returns**
> > > This *ScheilOptions* object

set_smallest_fraction(*smallest_fraction*)

> Sets the smallest fraction for constituents that are unstable.
>
> It is normally only in the gas phase that you can find such low fractions.
>
> The **default value** for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.
>
> > **Parameters**
> > > **smallest_fraction** – The smallest fraction for constituents that are unstable
> >
> > **Returns**
> > > This *ScheilOptions* object

set_temperature_step(*temperature_step_in_kelvin*)

> Sets the temperature step. Decreasing the temperature step increases the accuracy, but the default value is usually adequate.
>
> **Default step**: 1.0 K
>
> > **Parameters**
> > > **temperature_step_in_kelvin** – The temperature step [K]
> >
> > **Returns**
> > > This *ScheilOptions* object

terminate_on_fraction_of_liquid_phase(*fraction_to_terminate_at*)

> Sets the termination condition to a specified remaining fraction of liquid phase.
>
> **Default**: Terminates at 0.01 fraction of liquid phase.
>
> ---
>
> **Note:** Either the termination criterion is set to a temperature or fraction of liquid limit, both together are not possible.
>
> ---
>
> > **Parameters**
> >
> > > **fraction_to_terminate_at** – the termination fraction of liquid phase (value between 0 and 1)
> >
> > **Returns**
> >
> > > This *ScheilOptions* object

terminate_on_temperature(*temperature_in_kelvin*)

> Sets the termination condition to a specified temperature.
>
> **Default**: Terminates at 0.01 fraction of liquid phase, i.e. not at a specified temperature.
>
> ---
>
> **Note:** Either the termination criterion is set to a temperature or fraction of liquid limit, both together are not possible.
>
> ---
>
> > **Parameters**
> >
> > > **temperature_in_kelvin** – the termination temperature [K]
> >
> > **Returns**
> >
> > > This *ScheilOptions* object

class +tc_toolbox.+scheil.**ScheilSoluteTrapping**

> Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults.

ScheilSoluteTrapping()

> Configures the Scheil solute trapping settings. The used solification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. Constructs an instance of *ScheilSoluteTrapping*.

static scheil_back_diffusion()

> Configuration for *back diffusion in the solid primary phase*.
>
> > **Warning:** This feature has only effect on systems with diffusion data (typically a mobility database). If used for a system without diffusion data, a normal Scheil calculation is done. :return: A *ScheilBackDiffusion*

static scheil_classic()

> Configuration for Classic Scheil with fast diffusers. :return: A *ScheilClassic*

static scheil_solute_trapping()

> Configures the Scheil solute trapping settings. The used solidification speed equation is *Scanning speed * cos(angle)* with *Scanning speed* and *angle* being provided either by the user or taken from the defaults. :return: A *ScheilSoluteTrapping*

set_angle(*alpha*)

> Sets the transformation angle alpha between the solid/liquid boundary and laser scanning direction.
>
> **Default**: 45.0
>
> > **Parameters**
> > > **alpha** – The transformation angle [degree]
> >
> > **Returns**
> > > This *ScheilSoluteTrapping* object

set_primary_phasename(*primary_phase_name*)

> Sets the name of the primary phase.
>
> The primary phase is the phase where solute trapping takes place. A necessary condition for this phase is that the phase definition contains all of the elements that are chosen in the system. When *AUTOMATIC* is selected, the program tries to find a suitable primary phase that fills this condition.
>
> **Default**: *AUTOMATIC*
>
> > **Parameters**
> > > **primary_phase_name** – The phase name (or *AUTOMATIC*)
> >
> > **Returns**
> > > This *ScheilSoluteTrapping* object

set_scanning_speed(*scanning_speed*)

> Sets the scanning speed.
>
> **Default**: 1 m/s
>
> > **Parameters**
> > > **scanning_speed** – The scaling factor [m/s]
> >
> > **Returns**
> > > This *ScheilSoluteTrapping* object

4.1.5 Package "step_or_map_diagrams"

class +tc_toolbox.+step_or_map_diagrams.**AbstractAxisType**

> The abstract base class for all axis types.

class +tc_toolbox.+step_or_map_diagrams.**AbstractPhaseDiagramCalculation**(*back*)

> Abstract configuration required for a property diagram calculation.

> **Note:** This is an abstract class that cannot be used directly.

AbstractPhaseDiagramCalculation(*back*)

> Call base constructor: tc_toolbox.AbstractCalculation.

add_initial_equilibrium(*initial_equilibrium*)

> Add initial equilibrium start points from which a phase diagram is calculated.
>
> Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary.
>
> It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.

It is also possible to create a sequence of start points from one initial equilibria.

> **Parameters**
> > **initial_equilibrium** – The initial equilibrium
>
> **Returns**
> > This *PhaseDiagramCalculation* object

calculate(*keep_previous_results*, *timeout_in_minutes*)

disable_global_minimization()

> Disables global minimization.
>
> **Default**: Enabled
>
> > **Returns**
> > > This *PhaseDiagramCalculation* object

dont_keep_default_equilibria()

> Do not keep the initial equilibria added by default.
>
> This is only relevant in combination with *add_initial_equilibrium()*.
>
> This is the default behavior.
>
> > **Returns**
> > > This *PhaseDiagramCalculation* object

enable_global_minimization()

> Enables global minimization.
>
> **Default**: Enabled
>
> > **Returns**
> > > This *PhaseDiagramCalculation* object

get_components()

> Returns the names of the components in the system (including all components auto-selected by the database(s)).
>
> > **Returns**
> > > The component names

get_configuration_as_string()

> Returns detailed information about the current state of the calculation object.
>
> > **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_gibbs_energy_addition_for(*phase*)

> Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
> > **Parameters**
> > > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
> >
> > **Returns**
> > > Gibbs energy addition to G per mole formula unit.

get_system_data()

> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> > **Returns**
> > > The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

keep_default_equilibria()

> Keep the initial equilibria added by default. This is only relevant in combination with `add_initial_equilibrium()`.

> Default behavior is to not keep default equilibria.

> > **Returns**
> > > This *PhaseDiagramCalculation* object

remove_all_initial_equilibria()

> Removes all previously added initial equilibria.

> > **Returns**
> > > This *PhaseDiagramCalculation* object

run_poly_command(*command*)

> Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> > **Parameters**
> > > **command** – The Thermo-Calc Console Mode command

> > **Returns**
> > > This *PhaseDiagramCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

> Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

> It is not composition-, temperature- or pressure-dependent.

> > **Parameters**
> > > - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

> • **gibbs_energy** – Addition to G per mole formula unit

> **Returns**
>> This *PhaseDiagramCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
>> This *PhaseDiagramCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.

> **Parameters**

>> • **phase** – The phase name or *ALL_PHASES* for all phases

>> • **amount** – The phase fraction (between 0.0 and 1.0)

> **Returns**
>> This *PhaseDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

> **Parameters**

>> • **phase** – The phase name

>> • **amount** – The fixed phase fraction (between 0.0 and 1.0)

> **Returns**
>> This *PhaseDiagramCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
>> This *PhaseDiagramCalculation* object

with_options(*options*)

> Sets the simulation options.

> **Parameters**
>> **options** – The simulation options

> **Returns**
>> This *PhaseDiagramCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**
> - **component** – The name of the element must be given.
> - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> - **pressure** – The Pressure (in Pa) for the reference state.
>
> **Returns**
> This *PhaseDiagramCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed
>
> **Returns**
> This *PhaseDiagramCalculation* object

class +tc_toolbox.+step_or_map_diagrams.**AbstractPropertyDiagramCalculation**(*back*)

Abstract configuration required for a property diagram calculation.

Note: This is an abstract class that cannot be used directly.

AbstractPropertyDiagramCalculation(*back*)

Call base constructor: `tc_toolbox.AbstractCalculation`.

calculate(*keep_previous_results*, *timeout_in_minutes*)

disable_global_minimization()

Disables global minimization.

Default: Enabled

> **Returns**
>> This *PropertyDiagramCalculation* object

disable_step_separate_phases()

Disables *step separate phases*. This is the **default** setting.

> **Returns**
>> This *PropertyDiagramCalculation* object

enable_global_minimization()

Enables global minimization.

Default: Enabled

> **Returns**
>> This *PropertyDiagramCalculation* object

enable_step_separate_phases()

Enables *step separate phases*.

Default: By default separate phase stepping is *disabled*

Note: This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

> **Returns**
>> This *PropertyDiagramCalculation* object

get_components()

Returns the names of the components in the system (including all components auto-selected by the database(s)).

> **Returns**
>> The component names

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
>> **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>
> **Returns**
>> Gibbs energy addition to G per mole formula unit.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
>> The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
>> **command** – The Thermo-Calc Console Mode command
>
> **Returns**
>> This `PropertyDiagramCalculation` object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
>
>> - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>>
>> - **gibbs_energy** – Addition to G per mole formula unit
>
> **Returns**
>> This `PropertyDiagramCalculation` object

set_phase_to_dormant(*phase*)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases
>
> **Returns**
>> This *PropertyDiagramCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.
>
> **Parameters**
>
> - **phase** – The phase name or *ALL_PHASES* for all phases
>
> - **amount** – The phase fraction (between 0.0 and 1.0)
>
> **Returns**
>> This *PropertyDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
>
> **Parameters**
>
> - **phase** – The phase name
>
> - **amount** – The fixed phase fraction (between 0.0 and 1.0)
>
> **Returns**
>> This *PropertyDiagramCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> **Parameters**
>> **phase** – The phase name or *ALL_PHASES* for all phases
>
> **Returns**
>> This *PropertyDiagramCalculation* object

with_options(*options*)

> Sets the simulation options.
>
> **Parameters**
>> **options** – The simulation options
>
> **Returns**
>> This *PropertyDiagramCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
>
> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.
>
> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**
> - **component** – The name of the element must be given.
>
> - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
>
> - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
>
> - **pressure** – The Pressure (in Pa) for the reference state.
>
> **Returns**
> This *PropertyDiagramCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed
>
> **Returns**
> This *PropertyDiagramCalculation* object

class +tc_toolbox.+step_or_map_diagrams.**AxisType**

Factory class providing objects for configuring a logarithmic or linear axis by using *AxisType.linear()* or *AxisType.logarithmic()*.

static linear()

Creates an object for configuring a linear calculation axis.

Default: A minimum number of 40 steps.

Note: The returned object can be configured regarding the maximum step size *or* the minimum number of steps on the axis.

> **Returns**
> A new *Linear* object

static logarithmic()

Creates an object for configuring a logarithmic calculation axis.

Default: A scale factor of 1.1

Note: The returned object can be configured regarding the scale factor.

> **Returns**
>> A new *Logarithmic* object

class +tc_toolbox.+step_or_map_diagrams.**CalculationAxis**(*quantity*)

> A calculation axis used for property and phase diagram calculations.

> **Default**: A *Linear* axis with a *minimum number of 40 steps*

Note: A calculation axis is defining the varied condition and the range of variation. It is the same concept as in Thermo-Calc *Graphical Mode* or *Console Mode*.

> **CalculationAxis**(*quantity*)
>> **Default**: A *Linear* axis with a minimum number of 40 steps
>>
>>> **Parameters**
>>>> **quantity** – The ThermodynamicQuantity to set as axis variable; a Console Mode syntax string can be used as an alternative (for example "X(Cr)")

> **set_max**(*max*)
>> Sets the maximum quantity value of the calculation axis.
>>
>> **There is no default value set, it always needs to be defined.**
>>
>>> **Parameters**
>>>> **max** – The maximum quantity value of the axis [unit according to the axis quantity]
>>>
>>> **Returns**
>>>> This *CalculationAxis* object

> **set_min**(*min*)
>> Sets the minimum quantity value of the calculation axis.
>>
>> **There is no default value set, it always needs to be defined.**
>>
>>> **Parameters**
>>>> **min** – The minimum quantity value of the axis [unit according to the axis quantity]
>>>
>>> **Returns**
>>>> This *CalculationAxis* object

> **set_start_at**(*at*)
>> Sets the starting point of the calculation on the axis.
>>
>> **Default**: The default starting point is the center between the minimum and maximum quantity value
>>
>>> **Parameters**
>>>> **at** – The starting point on the axis [unit according to the axis quantity]
>>>
>>> **Returns**
>>>> This *CalculationAxis* object

> **with_axis_type**(*axis_type*)
>> Sets the type of the axis.
>>
>> **Default**: A *Linear* axis with a *minimum number of 40 steps*

> **Parameters**
> > **axis_type** – The axis type (linear or logarithmic)
>
> **Returns**
> > This *CalculationAxis* object

class +tc_toolbox.+step_or_map_diagrams.**Direction**

> An enumeration.

class +tc_toolbox.+step_or_map_diagrams.**InitialEquilibrium**(*first_axis*, *second_axis*)

> **InitialEquilibrium**(*first_axis*, *second_axis*)
>
> **add_equilibria_at_all_phase_changes**()
> > This generates one start point for each set of phase change in the chosen direction of the specified axis This ensures finding all possible phase boundary lines (not just the first one) along such an axis direction.
> >
> > Default behavior is to only generate one start point at the first phase change.
> >
> > > **Returns**
> > > > This *InitialEquilibrium* object
>
> **add_equilibria_at_first_phase_change**()
> > This generates one start point at the first phase change.
> >
> > This is the default behavior.
> >
> > > **Returns**
> > > > This *InitialEquilibrium* object
>
> **set_direction**(*direction_enum*)
> > Specifies along which axes the initial equilibria should be added.
> >
> > The default direction is INCREASE_FIRST_AXIS.
> >
> > > **Parameters**
> > > > **direction_enum** –
> > >
> > > **Returns**
> > > > This *InitialEquilibrium* object

class +tc_toolbox.+step_or_map_diagrams.**Linear**

> Represents a linear axis.
>
> **Linear**()
> > Creates an object representing a linear axis. Constructs an instance of *Linear*.
>
> **get_type**()
> > Convenience method for getting axis type.
> >
> > > **Returns**
> > > > The type
>
> **static linear**()
> > Creates an object for configuring a linear calculation axis.
> >
> > **Default**: A minimum number of 40 steps.
> >
> > ---
> >
> > **Note:** The returned object can be configured regarding the maximum step size *or* the minimum number of steps on the axis.
> >
> > ---

> **Returns**
>> A new *Linear* object

static logarithmic()

>Creates an object for configuring a logarithmic calculation axis.
>
>**Default**: A scale factor of 1.1
>
>---
>
>**Note:** The returned object can be configured regarding the scale factor.
>
>---
>
>> **Returns**
>>> A new *Logarithmic* object

set_max_step_size(*max_step_size*)

>Sets the axis to use the *maximum step size* configuration.
>
>**Default**: This is not the default which is *minimum number of steps*
>
>---
>
>**Note:** Either *maximum step size* or *minimum number of steps* can be used but not both at the same time.
>
>---
>
>> **Parameters**
>>> **max_step_size** – The maximum step size [unit according to the axis quantity]
>>
>> **Returns**
>>> This *Linear* object

set_min_nr_of_steps(*min_nr_of_steps*)

>Sets the axis to use the *minimum number of steps* configuration.
>
>**Default**: This is the default option (with a *minimum number of steps* of 40)
>
>---
>
>**Note:** Either *maximum step size* or *minimum number of steps* can be used but not both at the same time.
>
>---
>
>> **Parameters**
>>> **min_nr_of_steps** – The minimum number of steps
>>
>> **Returns**
>>> This *Linear* object

class +tc_toolbox.+step_or_map_diagrams.**Logarithmic**(*scale_factor*)

>Represents a logarithmic axis.
>
>---
>
>**Note:** A logarithmic axis is useful for low fractions like in a gas phase where 1E-7 to 1E-2 might be an interesting range. For the pressure a logarithmic axis is often also useful.
>
>---
>
>**Logarithmic**(*scale_factor*)
>
>>Creates an object representing a logarithmic axis.
>>
>>**Default**: 1.1

> **Parameters**
> **scale_factor** – The scale factor setting the maximum factor between two calculated values, must be larger than 1.0.

get_type()

Convenience method for getting axis type.

> **Returns**
> The type

static linear()

Creates an object for configuring a linear calculation axis.

Default: A minimum number of 40 steps.

Note: The returned object can be configured regarding the maximum step size *or* the minimum number of steps on the axis.

> **Returns**
> A new *Linear* object

static logarithmic()

Creates an object for configuring a logarithmic calculation axis.

Default: A scale factor of 1.1

Note: The returned object can be configured regarding the scale factor.

> **Returns**
> A new *Logarithmic* object

set_scale_factor(*scale_factor*)

Sets the scale factor.

Default: 1.1

> **Parameters**
> **scale_factor** – The scale factor setting the maximum factor between two calculated values, must be larger than 1.0

> **Returns**
> This *Logarithmic* object

class +tc_toolbox.+step_or_map_diagrams.**PhaseDiagramCalculation**(*back*)

Configuration for a phase diagram calculation.

Note: Specify the conditions, the calculation is performed with *calculate()*.

PhaseDiagramCalculation(*back*)

Call base constructor: tc_toolbox.step_or_map_diagrams.AbstractPhaseDiagramCalculation.

add_initial_equilibrium(*initial_equilibrium*)

> Add initial equilibrium start points from which a phase diagram is calculated.
>
> Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary.
>
> It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.
>
> It is also possible to create a sequence of start points from one initial equilibria.
>
> > **Parameters**
> > > **initial_equilibrium** – The initial equilibrium
> >
> > **Returns**
> > > This *PhaseDiagramCalculation* object

calculate(*keep_previous_results*, *timeout_in_minutes*)

> Performs the phase diagram calculation.
>
> > **Warning:** If you use *keep_previous_results=True*, you must not use another calculator or even get results in between the calculations using *calculate()*. Then the previous results will actually be lost.
>
> > **Parameters**
> > - **keep_previous_results** – If True, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the mapping fails at a certain condition.
> > - **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
> >
> > **Returns**
> > > A new *PhaseDiagramResult* object which later can be used to get specific values from the calculated result.

disable_global_minimization()

> Disables global minimization.
>
> **Default**: Enabled
>
> > **Returns**
> > > This *PhaseDiagramCalculation* object

dont_keep_default_equilibria()

> Do not keep the initial equilibria added by default.
>
> This is only relevant in combination with *add_initial_equilibrium()*.
>
> This is the default behavior.
>
> > **Returns**
> > > This *PhaseDiagramCalculation* object

enable_global_minimization()

> Enables global minimization.
>
> **Default**: Enabled

> **Returns**
>> This *PhaseDiagramCalculation* object

get_components()

> Returns the names of the components in the system (including all components auto-selected by the database(s)).
>
>> **Returns**
>>> The component names

get_configuration_as_string()

> Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

> Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
>> **Parameters**
>>> **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>>
>> **Returns**
>>> Gibbs energy addition to G per mole formula unit.

get_system_data()

> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

>> **Returns**
>>> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

keep_default_equilibria()

> Keep the initial equilibria added by default. This is only relevant in combination with *add_initial_equilibrium()*.
>
> Default behavior is to not keep default equilibria.
>
>> **Returns**
>>> This *PhaseDiagramCalculation* object

remove_all_conditions()

> Removes all set conditions.
>
>> **Returns**
>>> This *PhaseDiagramCalculation* object

---

remove_all_initial_equilibria()

>   Removes all previously added initial equilibria.

>   **Returns**
>   >   This *PhaseDiagramCalculation* object

remove_condition(*quantity*)

>   Removes the specified condition.

>   **Parameters**
>   >   **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string
>   >   can be used as an alternative (for example *X(Cr)*)

>   **Returns**
>   >   This `ThermodynamicCalculation` object

run_poly_command(*command*)

>   Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

> ---

> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method
> implemented in the API instead.

> ---

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the
> program in case of spelling mistakes (e.g. forgotten equals sign).

>   **Parameters**
>   >   **command** – The Thermo-Calc Console Mode command

>   **Returns**
>   >   This *PhaseDiagramCalculation* object

set_condition(*quantity*, *value*)

>   Sets the specified condition.

>   **Parameters**
>   >   • **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string
>   >   can be used as an alternative (for example *X(Cr)*)
>   >
>   >   • **value** – The value of the condition

>   **Returns**
>   >   This *PhaseDiagramCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

>   Used to specify the additional energy term (always being a constant) of a given phase. The value
>   (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can rep-
>   resent a nucleation barrier, surface tension, elastic energy, etc.

>   It is not composition-, temperature- or pressure-dependent.

>   **Parameters**
>   >   • **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>   >
>   >   • **gibbs_energy** – Addition to G per mole formula unit

> **Returns**
> This *PhaseDiagramCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.
>
> > **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.
>
> > **Parameters**
> >
> > - **phase** – The phase name or *ALL_PHASES* for all phases
> > - **amount** – The phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
>
> > **Parameters**
> >
> > - **phase** – The phase name
> > - **amount** – The fixed phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> > **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

with_first_axis(*axis*)

> Sets the first calculation axis.
>
> > **Parameters**
> > **axis** – The axis
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

with_options(*options*)

> Sets the simulation options.
>
> > **Parameters**
> > **options** – The simulation options
> >
> > **Returns**
> > This *PhaseDiagramCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

>**Parameters**
>
>- **component** – The name of the element must be given.
>
>- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
>
>- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
>
>- **pressure** – The Pressure (in Pa) for the reference state.
>
>**Returns**
>   This *PhaseDiagramCalculation* object

with_second_axis(*axis*)

Sets the second calculation axis.

>**Parameters**
>   **axis** – The axis
>
>**Returns**
>   This *PhaseDiagramCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

>**Parameters**
>   **system_modifications** – The system modification to be performed
>
>**Returns**
>   This *PhaseDiagramCalculation* object

class +tc_toolbox.+step_or_map_diagrams.**PhaseDiagramOptions**

Simulation options for phase diagram calculations.

PhaseDiagramOptions()

Simulation options for the phase diagram calculations. Constructs an instance of *PhaseDiagramOptions*.

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
>> This *PhaseDiagramOptions* object

disable_control_step_size_during_minimization()

Disables stepsize control during minimization (non-global).

Default: Enabled

> **Returns**
>> This *PhaseDiagramOptions* object

disable_force_positive_definite_phase_hessian()

Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

Default: Enabled

> **Returns**
>> This *PhaseDiagramOptions* object

dont_use_auto_start_points()

Switches the usage of automatic starting points for the mapping off.

Default: Switched on

> **Returns**
>> This *PhaseDiagramOptions* object

dont_use_inside_mesh_points()

Switches the usage of inside meshing points for the mapping off.

Default: Switched off

> **Returns**
>> This *PhaseDiagramOptions* object

enable_approximate_driving_force_for_metastable_phases()

Enables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
> > This *PhaseDiagramOptions* object

enable_control_step_size_during_minimization()

Enables stepsize control during normal minimization (non-global).

Default: Enabled

> **Returns**
> > This *PhaseDiagramOptions* object

enable_force_positive_definite_phase_hessian()

Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

Default: Enabled

> **Returns**
> > This *PhaseDiagramOptions* object

set_global_minimization_max_grid_points(*max_grid_points*)

Sets the maximum number of grid points in global minimization. ** Only applicable if global minimization is actually used**.

Default: 2000 points

> **Parameters**
> > **max_grid_points** – The maximum number of grid points
>
> **Returns**
> > This *PhaseDiagramOptions* object

set_global_minimization_test_interval(*global_test_interval*)

Sets the interval for the global test.

Default: 0

> **Parameters**
> > **global_test_interval** – The global test interval
>
> **Returns**
> > This *PhaseDiagramOptions* object

set_max_no_of_iterations(*max_no_of_iterations*)

Set the maximum number of iterations.

Default: max. 500 iterations

Note: As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

> **Parameters**
> > **max_no_of_iterations** – The max. number of iterations
>
> **Returns**
> > This *PhaseDiagramOptions* object

set_no_of_mesh_along_axis(*no_of_mesh_along_axis*)

    Sets the number of meshes along an axis for the mapping.

    Default: 3

        Parameters

            no_of_mesh_along_axis – The number of meshes

        Returns

            This *PhaseDiagramOptions* object

set_required_accuracy(*accuracy*)

    Sets the required relative accuracy.

    Default: 1.0E-6

Note: This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

        Parameters

            accuracy – The required relative accuracy

        Returns

            This *PhaseDiagramOptions* object

set_smallest_fraction(*smallest_fraction*)

    Sets the smallest fraction for constituents that are unstable.

    It is normally only in the gas phase that you can find such low fractions.

    The **default value** for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

        Parameters

            smallest_fraction – The smallest fraction for constituents that are unstable

        Returns

            This *PhaseDiagramOptions* object

use_auto_start_points()

    Switches the usage of automatic starting points for the mapping on.

    Default: Switched on

        Returns

            This *PhaseDiagramOptions* object

use_inside_mesh_points()

    Switches the usage of inside meshing points for the mapping off.

    Default: Switched off

        Returns

            This *PhaseDiagramOptions* object

class +tc_toolbox.+step_or_map_diagrams.**PhaseDiagramResult**(*back*)

    Result of a phase diagram calculation, it can be evaluated using quantities or Console Mode syntax.

PhaseDiagramResult(*back*)

> Call base constructor: `tc_toolbox.AbstractResult`.

add_coordinate_for_phase_label(*x*, *y*)

> Sets a coordinate in the result plot for which the stable phases will be evaluated and provided in the result data object. This can be used to plot the phases of a region into the phase diagram or just to programmatically evaluate the phases in certain regions.

> > **Warning:** This method takes coordinates of the **plot** axes and not of the calculation axis.

> > **Parameters**
> >
> > - **x** – The coordinate of the first **plot** axis ("x-axis") [unit of the **plot** axis]
> >
> > - **y** – The coordinate of the second **plot** axis ("y-axis") [unit of the **plot** axis]
> >
> > **Returns**
> >   This *PhaseDiagramResult* object

get_values_grouped_by_quantity_of(*x_quantity*, *y_quantity*)

> Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example in dependency of components). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

> > **Note:** The different datasets will contain *NaN*-values between different subsections and are not sorted (because they are unsortable due to their nature).

> > **Note:** Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains '='.

> > **Parameters**
> >
> > - **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), or even a function (for example *'f=T*1.01'*)
> >
> > - **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), or even a function (for example *'CP=HM.T'*)
> >
> > **Returns**
> >   The phase diagram data

get_values_grouped_by_stable_phases_of(*x_quantity*, *y_quantity*)

> Returns x-y-line data grouped by the sets of "stable phases" (for example "LIQUID" or "LIQUID + FCC_A1"). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

> > **Note:** The different datasets will contain *NaN*-values between different subsections and are not sorted (because they are unsortable due to their nature).

> **Note:** Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains *'='*.

> **Parameters**
> - **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), or even a function (for example *'f=T*1.01'*)
> - **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), or even a function (for example *'CP=HM.T'*)
>
> **Returns**
>   The phase diagram data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

remove_phase_labels()

> Erases all added coordinates for phase labels.
>
> **Returns**
>   This *PhaseDiagramResult* object

save_to_disk(*path*)

> Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`
>
> **Parameters**
>   **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
>
> **Returns**
>   this *PhaseDiagramResult* object

set_phase_name_style(*phase_name_style_enum*)

> Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, . . .).
>
> **Default**: PhaseNameStyle.NONE
>
> **Parameters**
>   **phase_name_style_enum** – The phase name style
>
> **Returns**
>   This *PhaseDiagramResult* object

class +tc_toolbox.+step_or_map_diagrams.**PhaseDiagramResultValues**(*back*)

> Represents the data of a phase diagram.

PhaseDiagramResultValues(*back*)

> Constructs an instance of *PhaseDiagramResultValues*.

get_invariants()

> Returns the x- and y-datasets of all invariants in the phase diagram.

> **Note:** The datasets will normally contain different sections separated by *NaN*-values.

Returns
The invariants dataset object

get_lines()
Returns the x- and y-datasets of all phase boundaries in the phase diagram.

Note: The datasets will normally contain different sections separated by *NaN*-values.

Returns
Containing the phase boundary datasets with the *quantities* or *stable phases* as keys (depending on the used method to get the values)

get_phase_labels()
Returns the phase labels added for certain coordinates using *PhaseDiagramResult.* *add_coordinate_for_phase_label()*.

Returns
The list with the phase label data (that contains plot coordinates and stable phases)

get_tie_lines()
Returns the x- and y-datasets of all tie-lines in the phase diagram.

Note: The datasets will normally contain different sections separated by *NaN*-values.

Returns
The tie-line dataset object

class +tc_toolbox.+step_or_map_diagrams.**PhaseLabel**(*back*)
Represents a *phase label at a plot coordinate*, i.e. the stable phases that are present at that *plot* coordinate.

PhaseLabel(*back*)
Constructs an instance of *PhaseLabel*.

get_text()
Accessor for the phase label :return: the phase label

get_x()
Accessor for the x-value :return: the x value

get_y()
Accessor for the y-value :return: the y value

class +tc_toolbox.+step_or_map_diagrams.**PhaseNameStyle**
The style of the phase names used in the labels.

class +tc_toolbox.+step_or_map_diagrams.**PropertyDiagramCalculation**(*back*)
Abstract configuration required for a property diagram calculation.

Note: This is an abstract class that cannot be used directly.

PropertyDiagramCalculation(*back*)

Call base constructor: `tc_toolbox.step_or_map_diagrams.AbstractPropertyDiagramCalculation`.

calculate(*keep_previous_results*, *timeout_in_minutes*)

Performs the property diagram calculation.

> **Warning:** If you use *keep_previous_results=True*, you must not use another calculator or even get results in between the calculations using `calculate()`. Then the previous results will actually be lost.

> **Parameters**
> - **keep_previous_results** – If *True*, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the stepping fails at a certain condition.
> - **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
>
> **Returns**
> A new *PropertyDiagramResult* object which later can be used to get specific values from the calculated result

disable_global_minimization()

Disables global minimization.

Default: Enabled

> **Returns**
> This *PropertyDiagramCalculation* object

disable_step_separate_phases()

Disables *step separate phases*. This is the **default** setting.

> **Returns**
> This *PropertyDiagramCalculation* object

enable_global_minimization()

Enables global minimization.

Default: Enabled

> **Returns**
> This *PropertyDiagramCalculation* object

enable_step_separate_phases()

Enables *step separate phases*.

Default: By default separate phase stepping is *disabled*

> **Note:** This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

> **Returns**
>> This `PropertyDiagramCalculation` object

get_components()

> Returns the names of the components in the system (including all components auto-selected by the database(s)).
>
>> **Returns**
>>> The component names

get_configuration_as_string()

> Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

> Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
>> **Parameters**
>>> **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>>
>> **Returns**
>>> Gibbs energy addition to G per mole formula unit.

get_system_data()

> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

>> **Returns**
>>> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

remove_all_conditions()

> Removes all set conditions.
>
>> **Returns**
>>> This `PropertyDiagramCalculation` object

remove_condition(*quantity*)

> Removes the specified condition.
>
>> **Parameters**
>>> **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example *X(Cr)*)

Returns
> This *PropertyDiagramCalculation* object

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

Parameters
> **command** – The Thermo-Calc Console Mode command

Returns
> This *PropertyDiagramCalculation* object

set_condition(*quantity*, *value*)

Sets the specified condition.

Parameters

- **quantity** – The thermodynamic quantity to set as condition; a Console Mode syntax string can be used as an alternative (for example *X(Cr)*)

- **value** – The value of the condition

Returns
> This *PropertyDiagramCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

- **gibbs_energy** – Addition to G per mole formula unit

Returns
> This *PropertyDiagramCalculation* object

set_phase_to_dormant(*phase*)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters
> **phase** – The phase name or *ALL_PHASES* for all phases

Returns
> This *PropertyDiagramCalculation* object

---

set_phase_to_entered(*phase*, *amount*)

Sets the phase to the status ENTERED, that is the default state.

> **Parameters**
>
> - **phase** – The phase name or *ALL_PHASES* for all phases
>
> - **amount** – The phase fraction (between 0.0 and 1.0)
>
> **Returns**
> This *PropertyDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

> **Parameters**
>
> - **phase** – The phase name
>
> - **amount** – The fixed phase fraction (between 0.0 and 1.0)
>
> **Returns**
> This *PropertyDiagramCalculation* object

set_phase_to_suspended(*phase*)

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

> **Parameters**
> **phase** – The phase name or *ALL_PHASES* for all phases
>
> **Returns**
> This *PropertyDiagramCalculation* object

with_axis(*axis*)

Sets the calculation axis.

> **Parameters**
> **axis** – The axis
>
> **Returns**
> This *PropertyDiagramCalculation* object

with_options(*options*)

Sets the simulation options.

> **Parameters**
> **options** – The simulation options
>
> **Returns**
> This *PropertyDiagramCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to

set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**
> - **component** – The name of the element must be given.
> - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> - **pressure** – The Pressure (in Pa) for the reference state.
>
> **Returns**
> This *PropertyDiagramCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed
>
> **Returns**
> This *PropertyDiagramCalculation* object

class +tc_toolbox.+step_or_map_diagrams.**PropertyDiagramOptions**

Simulation options for the property diagram calculations.

PropertyDiagramOptions()

Simulation options for property diagram calculations. Constructs an instance of *PropertyDiagramOptions*.

disable_approximate_driving_force_for_metastable_phases()

Disables the approximation of the driving force for metastable phases.

Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
> This *PropertyDiagramOptions* object

---

disable_control_step_size_during_minimization()

> Disables stepsize control during minimization (non-global).

> **Default**: Enabled

>> **Returns**
>>> This *PropertyDiagramOptions* object

disable_force_positive_definite_phase_hessian()

> Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

> **Default**: Enabled

>> **Returns**
>>> This *PropertyDiagramOptions* object

enable_approximate_driving_force_for_metastable_phases()

> Enables the approximation of the driving force for metastable phases.

> **Default**: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

>> **Returns**
>>> This *PropertyDiagramOptions* object

enable_control_step_size_during_minimization()

> Enables stepsize control during normal minimization (non-global).

> **Default**: Enabled

>> **Returns**
>>> This *PropertyDiagramOptions* object

enable_force_positive_definite_phase_hessian()

> Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

> **Default**: Enabled

>> **Returns**
>>> This *PropertyDiagramOptions* object

set_global_minimization_max_grid_points(*max_grid_points*)

> Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used**.

> **Default**: 2000 points

>> **Parameters**
>>> **max_grid_points** – The maximum number of grid points

>> **Returns**
>>> This *PropertyDiagramOptions* object

set_global_minimization_test_interval(*global_test_interval*)

Sets the interval for the global test.

Default: 0

> **Parameters**
> > **global_test_interval** – The global test interval
>
> **Returns**
> > This *PropertyDiagramOptions* object

set_max_no_of_iterations(*max_no_of_iterations*)

Set the maximum number of iterations.

Default: max. 500 iterations

Note: As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

> **Parameters**
> > **max_no_of_iterations** – The max. number of iterations
>
> **Returns**
> > This *PropertyDiagramOptions* object

set_required_accuracy(*accuracy*)

Sets the required relative accuracy.

Default: 1.0E-6

Note: This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

> **Parameters**
> > **accuracy** – The required relative accuracy
>
> **Returns**
> > This *PropertyDiagramOptions* object

set_smallest_fraction(*smallest_fraction*)

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The **default value** for the smallest site-fractions is 1E-12 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

> **Parameters**
> > **smallest_fraction** – The smallest fraction for constituents that are unstable
>
> **Returns**
> > This *PropertyDiagramOptions* object

---

class +tc_toolbox.+step_or_map_diagrams.**PropertyDiagramResult**(*back*)

Result of a property diagram. This can be used to query for specific values.

PropertyDiagramResult(*back*)

Call base constructor: `tc_toolbox.AbstractResult`.

get_values_grouped_by_quantity_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the multiple datasets of the specified quantities (typically the phases). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), or even a function (for example *'CP=HM.T'*)

- **sort_and_merge** – If *True*, the data is sorted and merged into as few subsections as possible (divided by *NaN*)

Returns

Containing the datasets with the quantities as their keys

get_values_grouped_by_stable_phases_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the sets of "stable phases" (for example "LIQUID" or "LIQUID + FCC_A1"). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and different lines of an ambiguous dataset. They might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), or even a function (for example *'CP=HM.T'*)

- **sort_and_merge** – If *True*, the data will be sorted and merged into as few subsections as possible (divided by *NaN*)

> **Returns**
> Containing the datasets with the quantities as their keys

get_values_of(*x_quantity*, *y_quantity*)

Returns sorted x-y-line data without any separation. Use `get_values_grouped_by_quantity_of()` or `get_values_grouped_by_stable_phases_of()` instead if you need such a separation. The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: This method will always return sorted data without any *NaN*-values. If it is unsortable that might give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods. One example of this is to use quantities with *All*-markers, for example *MassFractionOfAComponent("All")*.

Note: Its possible to use functions as axis variables, either by using *ThermodynamicQuantity.user_defined_function*, or by using an expression that contains '='.

> **Parameters**
> - **x_quantity** – The first Thermodynamic quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*) or even a function (for example *'f=T*1.01'*)
> - **y_quantity** – The second Thermodynamic quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), or even a function (for example *'CP=HM.T'*)
>
> **Returns**
> A tuple containing the x- and y-data in lists

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`

> **Parameters**
> **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
>
> **Returns**
> this `PropertyDiagramResult` object

set_phase_name_style(*phase_name_style_enum*)

Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, . . .).

Default: PhaseNameStyle.NONE

> **Parameters**
> **phase_name_style_enum** – The phase name style
>
> **Returns**
> This `PropertyDiagramResult` object

4.1.6 Package "diffusion"

class +tc_toolbox.+diffusion.**AbstractBoundaryCondition**

>   The abstract base class for all boundary conditions.

class +tc_toolbox.+diffusion.**AbstractCalculatedGrid**

>   The abstract base class for calculated grids.

class +tc_toolbox.+diffusion.**AbstractElementProfile**

>   The abstract base class for all initial composition profile types.

class +tc_toolbox.+diffusion.**AbstractGrid**

>   The abstract base class for all grids.

class +tc_toolbox.+diffusion.**AbstractSolver**

>   Abstract base class for the solvers (Classic, Homogenization and Automatic).

class +tc_toolbox.+diffusion.**ActivityFluxFunction**

>   Contains factory methods for the the different boundary conditions available.

>   **ActivityFluxFunction()**

>   >   Represents a boundary having a activity flux function.

>   >   This types of boundary conditions is used to take into account the finite rate of a surface reaction.

>   >   The flux for the independent components must be given in the format:

>   >   J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

>   >   where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

>   >   f and g must be expressed in DICTRA Console Mode syntax.

>   >   ---

>   >   **Note:** The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

>   >   ---

>   >   ---

>   >   **Note:** For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

>   >   ---

>   >   Constructs an instance of *ActivityFluxFunction*.

>   **static activity_flux_function()**

>   >   Factory method that creates a **new** activity-flux-function boundary condition.

>   >   This type of boundary condition is used to take into account the finite rate of a surface reaction.

>   >   The flux for the independent components must be given in the format:

>   >   J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

>   >   where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

>   >   f and g must be expressed in DICTRA Console Mode syntax.

Note: The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

> **Returns**
> > A new *ActivityFluxFunction* object

static closed_system()

> Factory method that creates a **new** closed-system boundary condition.
>
> > **Returns**
> > > A new *ClosedSystem* object

static fix_flux_value()

> Factory method that creates a **new** fix-flux-value boundary condition.
>
> This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.
>
> > **Returns**
> > > A new *FixFluxValue* object

static fixed_compositions(*unit_enum*)

> Factory method that creates a **new** fixed-composition boundary condition.
>
> > **Parameters**
> > > **unit_enum** – The composition unit
> >
> > **Returns**
> > > A new *FixedCompositions* object

get_type()

> The type of the boundary condition.
>
> > **Returns**
> > > The type

static mixed_zero_flux_and_activity()

> Factory method that creates a **new** mixed zero-flux and activity boundary condition
>
> > **Returns**
> > > A new *MixedZeroFluxAndActivity* object

set_flux_function(*element_name*, *f*, *g*, *n*, *to_time*)

> The flux for the independent components must be given in the format:
>
> J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))
>
> where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.
>
> f and g must be expressed in DICTRA Console Mode syntax.
>
> > **Parameters**
> > > • **element_name** – The name of the element

- **f** – the function f in the formula above

- **g** – the function g in the formula above

- **n** – the constant N in the formula above

- **to_time** – The max-time for which the flux function is used.

class +tc_toolbox.+diffusion.AutomaticSolver

Solver using the *homogenization model* if any region has more than one phase, otherwise using the *classic model*.

Note: This is the **default solver** and recommended for most applications.

AutomaticSolver()

Solver using the *homogenization model* if any region has more than one phase, otherwise using the *classic model*.

Note: This is the **default solver** and recommended for most applications.

Constructs an instance of *AutomaticSolver*.

static automatic()

Factory method that creates a **new** *automatic solver*. **This is the default solver and recommended for most applications**.

Note: This solver uses the homogenization model if any region has more than one phase, otherwise it uses the classic model.

> **Returns**
> A new *AutomaticSolver* object

static classic()

Factory method that creates a **new** *classic solver*.

Note: This solver never switches to the homogenization model even if the solver fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

> **Returns**
> A new *ClassicSolver* object

get_type()

The type of the solver.

> **Returns**
> The type

static homogenization()

Factory method that creates a **new** *homogenization solver*.

> **Note:** This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the `+tc_toolbox.diffusion.AutomaticSolver` instead if you do not need that behavior.

> **Returns**
>> A new *HomogenizationSolver* object

set_flux_balance_equation_accuracy(*accuracy*)

> **Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).**

> Sets the required accuracy during the solution of the flux balance equations. **Default**: 1.0e-16

>> **Parameters**
>>> **accuracy** – The required accuracy

>> **Returns**
>>> A new *AutomaticSolver* object

set_tieline_search_variable_to_activity()

> **Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).**

> Configures the solver to use the *activity of a component* to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. **Default**: This is the default setting

>> **Returns**
>>> A new *AutomaticSolver* object

set_tieline_search_variable_to_potential()

> **Only valid if the :class:`ClassicSolver` is actually used (i.e. not more than one phase in each region).**

> Configures the solver to use the *chemical potential of a component* to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. **Default**: To use the activity

>> **Returns**
>>> A new *AutomaticSolver* object

class +tc_toolbox.+diffusion.**BoundaryCondition**

> Contains factory methods for the the different boundary conditions available.

static activity_flux_function()

> Factory method that creates a **new** activity-flux-function boundary condition.

> This type of boundary condition is used to take into account the finite rate of a surface reaction.

> The flux for the independent components must be given in the format:

> J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

> where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

> f and g must be expressed in DICTRA Console Mode syntax.

> **Note:** The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

Returns
    A new *ActivityFluxFunction* object

static closed_system()

Factory method that creates a **new** closed-system boundary condition.

Returns
    A new *ClosedSystem* object

static fix_flux_value()

Factory method that creates a **new** fix-flux-value boundary condition.

This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.

Returns
    A new *FixFluxValue* object

static fixed_compositions(*unit_enum*)

Factory method that creates a **new** fixed-composition boundary condition.

Parameters
    unit_enum – The composition unit

Returns
    A new *FixedCompositions* object

static mixed_zero_flux_and_activity()

Factory method that creates a **new** mixed zero-flux and activity boundary condition

Returns
    A new *MixedZeroFluxAndActivity* object

class +tc_toolbox.+diffusion.CalculatedGrid

Factory class for grids generated by a mathematical series (linear, geometric, ...). Use `+tc_toolbox.diffusion.PointByPointGrid` instead if you want to use an existing grid from experimental data or a previous calculation.

Note: A region must contain a number of grid points. The composition is only known at these grid points and the software assumes that the composition varies linearly between them. The amount and composition of all the phases present at a single grid point in a certain region are those given by thermodynamic equilibrium keeping the over-all composition at the grid point fixed.

---

static double_geometric(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

Factory method that creates a **new** double geometric grid.

Note: Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

Parameters
- **no_of_points** – The number of points
- **lower_geometrical_factor** – The geometrical factor for the left half
- **upper_geometrical_factor** – The geometrical factor for the right half

Returns
A new *DoubleGeometricGrid* object

static **geometric**(*no_of_points*, *geometrical_factor*)

Factory method that creates a **new** geometric grid.

Note: A grid that yields a varying density of grid points in the region. A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

Parameters
- **no_of_points** – The number of points
- **geometrical_factor** – The geometrical factor

Returns
A new *GeometricGrid* object

static **linear**(*no_of_points*)

Factory method that creates a **new** equally spaced grid.

Parameters
no_of_points – The number of points

Returns
A new *LinearGrid* object

class +tc_toolbox.+diffusion.**ClassicSolver**

Solver using the *Classic model*.

Note: This solver **never switches** to the homogenization model even if it fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

ClassicSolver()

Solver using the *Classic model*.

Note: This solver **never switches** to the homogenization model even though the solver fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

Constructs an instance of *ClassicSolver*.

static **automatic**()

Factory method that creates a **new** *automatic solver*. **This is the default solver and recommended for most applications**.

> **Note:** This solver uses the homogenization model if any region has more than one phase, otherwise it uses the classic model.

> **Returns**
> > A new *AutomaticSolver* object

static classic()

Factory method that creates a **new** *classic solver*.

> **Note:** This solver never switches to the homogenization model even if the solver fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

> **Returns**
> > A new *ClassicSolver* object

get_type()

Convenience method for getting the type of the solver.

> **Returns**
> > The type of the solver

static homogenization()

Factory method that creates a **new** *homogenization solver*.

> **Note:** This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the +tc_toolbox.diffusion. AutomaticSolver instead if you do not need that behavior.

> **Returns**
> > A new *HomogenizationSolver* object

set_flux_balance_equation_accuracy(*accuracy*)

Sets the required accuracy during the solution of the flux balance equations. **Default**: 1.0e-16

> **Parameters**
> > **accuracy** – The required accuracy

> **Returns**
> > A new *ClassicSolver* object

set_tieline_search_variable_to_activity()

Configures the solver to use the *activity of a component* to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. **Default**: This is the default setting

set_tieline_search_variable_to_potential()

Configures the solver to use the *chemical potential of a component* to find the correct tie-line at the phase interface. Either activity or chemical potential are applied to reduce the degrees of freedom at the local equilibrium. **Default**: To use the activity

> **Returns**
>> A new *ClassicSolver* object

class +tc_toolbox.+diffusion.**ClosedSystem**

Represents a boundary for a closed system.

ClosedSystem()

Represents a boundary for a closed system. Constructs an instance of *ClosedSystem*.

static activity_flux_function()

Factory method that creates a **new** activity-flux-function boundary condition.

This type of boundary condition is used to take into account the finite rate of a surface reaction.

The flux for the independent components must be given in the format:

J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

f and g must be expressed in DICTRA Console Mode syntax.

Note: The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

> **Returns**
>> A new *ActivityFluxFunction* object

static closed_system()

Factory method that creates a **new** closed-system boundary condition.

> **Returns**
>> A new *ClosedSystem* object

static fix_flux_value()

Factory method that creates a **new** fix-flux-value boundary condition.

This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.

> **Returns**
>> A new *FixFluxValue* object

static fixed_compositions(*unit_enum*)

Factory method that creates a **new** fixed-composition boundary condition.

> **Parameters**
>> **unit_enum** – The composition unit

> **Returns**
>> A new *FixedCompositions* object

get_type()

> Convenience method for getting the type of the boundary condition.

> > **Returns**
> >
> > > The type of the boundary condition

static mixed_zero_flux_and_activity()

> Factory method that creates a **new** mixed zero-flux and activity boundary condition

> > **Returns**
> >
> > > A new *MixedZeroFluxAndActivity* object

class +tc_toolbox.+diffusion.**CompositionProfile**(*unit_enum*)

> Contains initial concentration profiles for the elements.

> **CompositionProfile**(*unit_enum*)

> > Contains initial concentration profiles for the elements.

> > > **Parameters**
> > >
> > > > **unit_enum** – The unit of the compositions

> **add**(*element_name*, *profile*)

> > Adds a concentration profile for the specified element.

> > > **Parameters**
> > >
> > > > • **element_name** – The name of the element
> > > >
> > > > • **profile** – The initial concentration profile

> > > **Returns**
> > >
> > > > A *CompositionProfile* object

class +tc_toolbox.+diffusion.**ConstantProfile**(*value*)

> Represents a constant initial concentration profile.

> **ConstantProfile**(*value*)

> > Represents a constant initial concentration profile.

> > > **Parameters**
> > >
> > > > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].

> **static constant**(*value*)

> > Factory method that creates a **new** constant initial concentration profile.

> > > **Parameters**
> > >
> > > > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].

> > > **Returns**
> > >
> > > > A new *ConstantProfile* object

> **static funct**(*dictra_console_mode_function*)

> > Factory method that creates a **new** initial concentration profile defined by a function in DICTRA Console Mode syntax.

> > ---

> > **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.

> > ---

> **Parameters**
>> **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.
>
> **Returns**
>> A new *FunctionProfile* object

get_type()

> The type of the element profile.
>
> **Returns**
>> The type

static linear(*start_value*, *end_value*)

> Factory method that creates a **new** linear initial concentration profile.
>
> **Parameters**
>> - **start_value** – Composition at the left side of the region [unit as defined in *CompositionProfile*].
>>
>> - **end_value** – Composition at the right side of the region [unit as defined in *CompositionProfile*].
>
> **Returns**
>> A new *LinearProfile* object

static step(*lower_boundary*, *upper_boundary*, *step_at*)

> Factory method that creates a **new** initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.
>
> **Parameters**
>> - **lower_boundary** – Composition before the step [unit as defined in *CompositionProfile*].
>>
>> - **upper_boundary** – Composition after the step [unit as defined in *CompositionProfile*].
>>
>> - **step_at** – The distance where the step should be [m].
>
> **Returns**
>> A new *StepProfile* object

class +tc_toolbox.+diffusion.**ContinuedDiffusionCalculation**(*back*)

> Configuration for a diffusion calculation that is a continuation of a previous isothermal or non-isothermal diffusion calculation. It contains a subset of the settings possible in the original calculation.
>
> Use *set_simulation_time()* to set a simulation time that is higher than the original calculation.

ContinuedDiffusionCalculation(*back*)

> Call base constructor: tc_toolbox.AbstractCalculation.

calculate(*timeout_in_minutes*)

> Runs the diffusion calculation.
>
> **Parameters**
>> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns

A *DiffusionCalculationResult* which later can be used to get specific values from the calculated result

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

Returns

The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

set_simulation_time(*simulation_time*)

Sets the simulation time.

Parameters

simulation_time – The simulation time [s]

Returns

This *DiffusionIsoThermalCalculation* object

with_left_boundary_condition(*boundary_condition*, *to*)

Defines the boundary condition on the left edge of the system.

Default: A closed-system boundary condition.

It is possible specify the upper time-point for which this setting is valid using the parameter "to".

Default: The end of the simulation.

Examples

- *with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)*

- *with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*

- *with_left_boundary_condition(BoundaryCondition.closed_system())*

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

> **Note:** You can specify time-dependent boundary conditions by calling
> `with_left_boundary_condition()` many times, with different values of the "to" parameter.

> **Parameters**
>> • **boundary_condition** – The boundary condition
>>
>> • **to** – The upper time-limit for boundary_condition.
>
> **Returns**
>> This *DiffusionIsoThermalCalculation* object

with_options(*options*, *to*)

> Sets the general simulation conditions.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
>> **Parameters**
>>> • **options** – The general simulation conditions
>>>
>>> • **to** – The upper time-limit for options.
>>
>> **Returns**
>>> This *DiffusionIsoThermalCalculation* object

with_right_boundary_condition(*boundary_condition*, *to*)

> Defines the boundary condition on the right edge of the system.
>
> **Default**: A closed-system boundary condition
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.

Examples

- *with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)*

- *with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*

- *with_right_boundary_condition(BoundaryCondition.closed_system())*

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

> **Note:** You can specify time-dependent boundary conditions by calling
> `with_right_boundary_condition()` many times, with different values of the "to" parameter.

> **Parameters**
>> • **boundary_condition** – The boundary condition
>>
>> • **to** – The upper time-limit for boundary_condition.

> **Returns**
>> This *DiffusionIsoThermalCalculation* object

with_solver(*solver*, *to*)

> Sets the solver to use (*Classic*, *Homogenization* or *Automatic*). **Default is Automatic**.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
>> **Parameters**
>>
>> • **solver** – The solver to use
>>
>> • **to** – The upper time-limit for solver.
>>
>> **Returns**
>>> This *DiffusionIsoThermalCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

>> **Parameters**
>>> **system_modifications** – The system modification to be performed

with_timestep_control(*timestep_control*, *to*)

> Sets the timestep control options.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
>> **Parameters**
>>
>> • **timestep_control** – The new timestep control options
>>
>> • **to** – The upper time-limit for timestep_control.
>>
>> **Returns**
>>> This *DiffusionIsoThermalCalculation* object

class +tc_toolbox.+diffusion.**DiffusionCalculationResult**(*back*)

> Result of a diffusion calculation. This can be used to query for specific values. For details of the axis variables, search the Thermo-Calc help.

DiffusionCalculationResult(*back*)

> Call base constructor: `tc_toolbox.AbstractResult`.

get_mass_fraction_at_lower_interface(*region*, *component*)

> Returns the mass fraction of the specified component at the lower boundary of the specified region, in dependency of time.
>
>> **Parameters**
>>
>> • **region** – The name of the region
>>
>> • **component** – The name of the component

Returns
A tuple of two lists of floats (time [s], mass fraction of the specified component)

get_mass_fraction_at_upper_interface(*region*, *component*)

Returns the mass fraction of the specified component at the upper boundary of the specified region, in dependency of time.

Parameters

- **region** – The name of the region

- **component** – The name of the component

Returns
A tuple of two lists of floats (time [s], mass fraction of the specified component)

get_mass_fraction_of_component_at_time(*component*, *time*)

Returns the mass fraction of the specified component at the specified time.

Note: Use the enum +tc_toolbox.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

Parameters

- **component** – The name of the component

- **time** – The time [s]

Returns
A tuple of two lists of floats (distance [m], mass fraction of component at the specified time)

get_mass_fraction_of_phase_at_time(*phase*, *time*)

Returns the mass fraction of the specified phase.

Note: Use the enum +tc_toolbox.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

Parameters

- **phase** – The name of the phase

- **time** – The time [s]

Returns
A tuple of two lists of floats (distance [m], mass fraction of hte phase at the specified time)

get_mole_fraction_at_lower_interface(*region*, *component*)

Returns the mole fraction of the specified component at the lower boundary of the specified region, in dependency of time.

Parameters

- **region** – The name of the region

- **component** – The name of the component

> **Returns**
>> A tuple of two lists of floats (time [s], mole fraction of the specified component)

get_mole_fraction_at_upper_interface(*region*, *component*)

> Returns the mole fraction of the specified component at the upper boundary of the specified region, in dependency of time.

> **Parameters**
>> - **region** – The name of the region
>>
>> - **component** – The name of the component

> **Returns**
>> A tuple of two lists of floats (time [s], mole fraction of the specified component)

get_mole_fraction_of_component_at_time(*component*, *time*)

> Returns the mole fraction of the specified component at the specified time.

Note: Use the enum +tc_toolbox.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

> **Parameters**
>> - **component** – The name of the component
>>
>> - **time** – The time [s]

> **Returns**
>> A tuple of two lists of floats (distance [m], mole fraction of component at the specified time)

get_mole_fraction_of_phase_at_time(*phase*, *time*)

> Returns the mole fraction of the specified phase.

Note: Use the enum +tc_toolbox.diffusion.SimulationTime to choose the first or the last timepoint of the simulation. A timepoint close to the last one should never be specified manually because the actual end of the simulation can slightly deviate.

> **Parameters**
>> - **phase** – The name of the phase
>>
>> - **time** – The time [s]

> **Returns**
>> A tuple of two lists of floats (distance [m], mole fraction of the phase at the specified time)

get_position_of_lower_boundary_of_region(*region*)

> Returns the position of the lower boundary of the specified region in dependency of time.

> **Parameters**
>> **region** – The name of the region

> **Returns**
>> A tuple of two lists of floats (time [s], position of lower boundary of region [m])

`get_position_of_upper_boundary_of_region`(*region*)

Returns the position of the upper boundary of the specified region in dependency of time.

> **Parameters**
>> **`region`** – The name of the region
>
> **Returns**
>> A tuple of two lists of floats (time [s], position of upper boundary of region [m])

`get_regions`()

Returns the regions of the diffusion simulation.

Note: Automatically generated regions (*R_###*) are included in the list.

> **Returns**
>> The region names

`get_time_steps`()

Returns the timesteps of the diffusion simulation.

> **Returns**
>> The timesteps [s]

`get_total_mass_fraction_of_component`(*component*)

Returns the total mass fraction of the specified component in dependency of time.

> **Parameters**
>> **`component`** – The name of the component
>
> **Returns**
>> A tuple of two lists of floats (time [s], total mass fraction of the component)

`get_total_mass_fraction_of_component_in_phase`(*component*, *phase*)

Returns the total mass fraction of the specified component in the specified phase in dependency of time.

> **Parameters**
>> - **`component`** – The name of the component
>> - **`phase`** – The name of the phase
>
> **Returns**
>> A tuple of two lists of floats (time [s], total mass fraction of the component in the phase)

`get_total_mass_fraction_of_phase`(*phase*)

Returns the total mass fraction of the specified phase in dependency of the time.

> **Parameters**
>> **`phase`** – The name of the phase
>
> **Returns**
>> A tuple of two lists of floats (time [s], total mass fraction of the phase)

`get_total_mole_fraction_of_component`(*component*)

Returns the total mole fraction of the specified component in dependency of time.

> **Parameters**
>> **`component`** – The name of the component

Returns

A tuple of two lists of floats (time [s], total mole fraction of the component)

get_total_mole_fraction_of_component_in_phase(*component*, *phase*)

Returns the total mole fraction of the specified component in the specified phase in dependency of time.

Parameters

- **component** – The name of the component

- **phase** – The name of the phase

Returns

A tuple of two lists of floats (time [s], total mole fraction of the component in the phase)

get_total_mole_fraction_of_phase(*phase*)

Returns the total mole fraction of the specified phase in dependency of time.

Parameters

phase – The name of the phase

Returns

A tuple of two lists of floats (time [s], total mole fraction of the phase)

get_total_volume_fraction_of_phase(*phase*)

Returns the total volume fraction of the specified phase in dependency of the time.

Parameters

phase – The name of the phase

Returns

A tuple of two lists of floats (time [s], total volume fraction of the phase)

get_values_of(*x_axis*, *y_axis*, *plot_condition*, *independent_variable*)

Returns the specified result from the simulation, allows all possible settings.

Note: As an alternative, DICTRA Console Mode syntax can be used as well for each quantity and condition.

Warning: This is an advanced mode that is equivalent to the possibilities in the DICTRA Console Mode. Not every combination of settings will return a result.

Parameters

- **x_axis** – The first result quantity

- **y_axis** – The second result quantity

- **plot_condition** – The plot conditions

- **independent_variable** – The independent variable

Returns

A tuple of two lists of floats (the x_axis quantity result, the y_axis quantity result) [units according to the quantities]

get_velocity_of_lower_boundary_of_region(*region*)

Returns the velocity of the lower boundary of the specified region in dependency of time.

> **Parameters**
> > **region** – The name of the region
>
> **Returns**
> > A tuple of two lists of floats (time [s], velocity of lower boundary of region [m/s])

get_velocity_of_upper_boundary_of_region(*region*)

Returns the velocity of the upper boundary of the specified region in dependency of time.

> **Parameters**
> > **region** – The name of the region
>
> **Returns**
> > A tuple of two lists of floats (time [s], velocity of upper boundary of region [m/s])

get_width_of_region(*region*)

Returns the width of region, in dependency of time.

> **Parameters**
> > **region** – The name of the region
>
> **Returns**
> > A tuple of two lists of floats (time [s], width of the specified region [m])

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

Saves the result to disk. The result can later be loaded using `+tc_toolbox.server.SetUp.load_result_from_disk()`.

> **Note:** The *result data* is represented by a whole folder containing multiple files.

> **Parameters**
> > **path** – The path to the result folder, can be relative or absolute.
>
> **Returns**
> > This *DiffusionCalculationResult* object

with_continued_calculation()

Returns a *ContinuedDiffusionCalculation* that is used for continuing a diffusion calculation with altered settings.

> **Returns**
> > A *ContinuedDiffusionCalculation*

class +tc_toolbox.+diffusion.**DiffusionIsoThermalCalculation**(*back*)

Configuration for an isothermal diffusion calculation.

DiffusionIsoThermalCalculation(*back*)

Call base constructor: `tc_toolbox.AbstractCalculation`.

add_console_command(*console_command*)

Registers a DICTRA Console Mode command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands are stored and used until explicitly deleted using +tc_toolbox.diffusion.DiffusionIsoThermoCalculation.remove_all_console_commands.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw DICTRA-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, ...).

> **Parameters**
> **console_command** – The DICTRA Console Mode command
>
> **Returns**
> This *DiffusionIsoThermalCalculation* object

add_region(*region*)

Adds a region to the calculation. Regions are always added in the simulation domain from left to right.

If you want to replace an already added region, call *remove_all_regions()*, and add the regions that you want to keep.

Warning: Regions must have unique names.

> **Parameters**
> **region** – The region to be added
>
> **Returns**
> This *DiffusionIsoThermalCalculation* object

calculate(*timeout_in_minutes*)

Runs the diffusion calculation.

> **Parameters**
> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
>
> **Returns**
> A *DiffusionCalculationResult* which later can be used to get specific values from the calculated result

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
>> The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

remove_all_console_commands()

Removes all previously added Console Mode commands.

> **Returns**
>> This *DiffusionIsoThermalCalculation* object

remove_all_regions()

Removes all previously added regions.

:return This *DiffusionIsoThermalCalculation* object

set_simulation_time(*simulation_time*)

Sets the simulation time.

> **Parameters**
>> **simulation_time** – The simulation time [s]

> **Returns**
>> This *DiffusionIsoThermalCalculation* object

set_temperature(*temperature*)

Sets the temperature for the isothermal simulation.

> **Parameters**
>> **temperature** – The temperature [K]

> **Returns**
>> This *DiffusionIsoThermalCalculation* object

with_cylindrical_geometry(*first_interface_position*)

Sets geometry to *cylindrical*, corresponds to an infinitely long cylinder of a certain radius.

Default: A planar geometry

Note: With a cylindrical or spherical geometry, the system's zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the *first_interface_position*, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

> **Parameters**
>> **first_interface_position** – The position of the left-most coordinate along the axis, only necessary for modeling a tube geometry [m]

Returns
> This *DiffusionIsoThermalCalculation* object

with_left_boundary_condition(*boundary_condition*, *to*)
> Defines the boundary condition on the left edge of the system.
>
> **Default**: A closed-system boundary condition.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.

Examples

- *with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)*

- *with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*

- *with_left_boundary_condition(BoundaryCondition.closed_system())*

This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

Note: You can specify time-dependent boundary conditions by calling *with_left_boundary_condition()* many times, with different values of the "to" parameter.

> **Parameters**
> > - **boundary_condition** – The boundary condition
> > - **to** – The upper time-limit for boundary_condition.
>
> **Returns**
> > This *DiffusionIsoThermalCalculation* object

with_options(*options*, *to*)
> Sets the general simulation conditions.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
> > **Parameters**
> > > - **options** – The general simulation conditions
> > > - **to** – The upper time-limit for options.
> >
> > **Returns**
> > > This *DiffusionIsoThermalCalculation* object

with_planar_geometry()
> Sets geometry to *planar*.
>
> **This is default.**
>
> > **Returns**
> > > This *DiffusionIsoThermalCalculation* object

with_reference_state(*element*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

> For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

> If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> > **Parameters**
> >
> > - **element** – The name of the element
> >
> > - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> >
> > - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> >
> > - **pressure** – The pressure (in Pa) for the reference state
> >
> > **Returns**
> > This *DiffusionIsoThermalCalculation* object

with_right_boundary_condition(*boundary_condition*, *to*)

> Defines the boundary condition on the right edge of the system.

> **Default**: A closed-system boundary condition

> It is possible specify the upper time-point for which this setting is valid using the parameter "to".

> **Default**: The end of the simulation.

> **Examples**

> - *with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)*
>
> - *with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*
>
> - *with_right_boundary_condition(BoundaryCondition.closed_system())*

> This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

Note: You can specify time-dependent boundary conditions by calling *with_right_boundary_condition()* many times, with different values of the "to" parameter.

> **Parameters**
>
> - **boundary_condition** – The boundary condition
>
> - **to** – The upper time-limit for boundary_condition.
>
> **Returns**
>> This *DiffusionIsoThermalCalculation* object

with_solver(*solver*, *to*)

> Sets the solver to use (*Classic*, *Homogenization* or *Automatic*). **Default is Automatic**.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
>> **Parameters**
>>
>> - **solver** – The solver to use
>>
>> - **to** – The upper time-limit for solver.
>>
>> **Returns**
>>> This *DiffusionIsoThermalCalculation* object

with_spherical_geometry(*first_interface_position*)

> Sets geometry to *spherical*, corresponds to a sphere with a certain radius.
>
> **Default**: A spherical geometry

Note: With a cylindrical or spherical geometry, the system's zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the *first_interface_position*, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

> **Parameters**
>> **first_interface_position** – The position of the left-most coordinate along the axis, only necessary for modeling a hollow sphere geometry [m]
>
> **Returns**
>> This *DiffusionIsoThermalCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
>> **system_modifications** – The system modification to be performed

---

Returns
    This *DiffusionIsoThermalCalculation* object

with_timestep_control(*timestep_control*, *to*)

Sets the timestep control options.

It is possible specify the upper time-point for which this setting is valid using the parameter "to".

Default: The end of the simulation.

> **Parameters**
>
> - **timestep_control** – The new timestep control options
>
> - **to** – The upper time-limit for timestep_control.
>
> **Returns**
>   This *DiffusionIsoThermalCalculation* object

class +tc_toolbox.+diffusion.**DiffusionNonIsoThermalCalculation**(*back*)

Configuration for a non-isothermal diffusion calculation.

DiffusionNonIsoThermalCalculation(*back*)

Call base constructor: `tc_toolbox.AbstractCalculation`.

add_console_command(*console_command*)

Registers a DICTRA Console Mode command for execution. These commands are executed after all other configuration directly before the calculation starts to run. All commands are stored and used until explicitly deleted using `+tc_toolbox.diffusion.DiffusionNonIsoThermalCalculation.remove_all_console_commands`.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw DICTRA-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, …).

> **Parameters**
>   **console_command** – The DICTRA Console Mode command
>
> **Returns**
>   This *DiffusionNonIsoThermalCalculation* object

add_region(*region*)

Adds a region to the calculation. Regions are always added in the simulation domain from left to right.

If you want to replace an already added region, call *remove_all_regions()*, and add the regions that you want to keep.

> **Warning:** Regions must have unique names.

> **Parameters**
>   **region** – The region to be added

Returns

> This *DiffusionNonIsoThermalCalculation* object

calculate(*timeout_in_minutes*)

> Runs the diffusion calculation.
>
> **Parameters**
>
> > **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.
>
> **Returns**
>
> > A *DiffusionCalculationResult* which later can be used to get specific values from the calculated result

get_configuration_as_string()

> Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_system_data()

> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
>
> > The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

remove_all_console_commands()

> Removes all previously added Console Mode commands.
>
> **Returns**
>
> > This *DiffusionNonIsoThermalCalculation* object

remove_all_regions()

> Removes all previously added regions.
>
> **Returns**
>
> > This *DiffusionNonIsoThermalCalculation* object

set_simulation_time(*simulation_time*)

> Sets the simulation time.
>
> **Parameters**
>
> > **simulation_time** – The simulation time [s]
>
> **Returns**
>
> > This *DiffusionNonIsoThermalCalculation* object

with_cylindrical_geometry(*first_interface_position*)

> Sets geometry to *cylindrical*, corresponds to an infinitely long cylinder of a certain radius.
>
> **Default**: A planar geometry
>
> ---
>
> **Note:** With a cylindrical or spherical geometry, the system's zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the *first_interface_position*, a different left-most coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).
>
> ---
>
> > **Parameters**
> > > **first_interface_position** – The position of the left-most coordinate along the axis, only necessary for modeling a tube geometry [m]
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_left_boundary_condition(*boundary_condition*, *to*)

> Defines the boundary condition on the left edge of the system.
>
> **Default**: A closed-system boundary condition.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
> ### Examples
>
> - *with_left_boundary_condition(BoundaryCondition.closed_system(), to=100)*
>
> - *with_left_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*
>
> - *with_left_boundary_condition(BoundaryCondition.closed_system())*
>
> This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.
>
> ---
>
> **Note:** You can specify time-dependent boundary conditions by calling *with_left_boundary_condition()* many times, with different values of the "to" parameter.
>
> ---
>
> > **Parameters**
> > > - **boundary_condition** – The boundary condition
> > >
> > > - **to** – The upper time-limit for boundary_condition.
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_options(*options*, *to*)

> Sets the general simulation conditions.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
> > **Parameters**
> >
> > - **options** – The general simulation conditions
> >
> > - **to** – The upper time-limit for options.
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_planar_geometry()

> Sets geometry to *planar*.
>
> **This is default.**
>
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_reference_state(*element*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
>
> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.
>
> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.
>
> For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.
>
> If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.
>
> > **Parameters**
> >
> > - **element** – The name of the element
> >
> > - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.
> >
> > - **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.
> >
> > - **pressure** – The pressure (in Pa) for the reference state
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_right_boundary_condition(*boundary_condition*, *to*)

> Defines the boundary condition on the right edge of the system.
>
> **Default**: A closed-system boundary condition
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.

Examples

> - *with_right_boundary_condition(BoundaryCondition.closed_system(), to=100)*
>
> - *with_right_boundary_condition(BoundaryCondition.mixed_zero_flux_and_activity().set_activity_for_element("C", surface_activity), to=500)*
>
> - *with_right_boundary_condition(BoundaryCondition.closed_system())*
>
> This example sets an closed-system-boundary-condition from start up to 100s and a activity-boundary-condition from 100s to 500s and finally a closed-system-boundary-condition from 500s to the end of simulation.

> **Note:** You can specify time-dependent boundary conditions by calling `with_right_boundary_condition()` many times, with different values of the "to" parameter.

> > **Parameters**
> >
> > - **boundary_condition** – The boundary condition
> >
> > - **to** – The upper time-limit for boundary_condition.
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_solver(*solver*, *to*)

> Sets the solver to use (*Classic*, *Homogenization* or *Automatic*). **Default is Automatic**.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
> > **Parameters**
> >
> > - **solver** – The solver to use
> >
> > - **to** – The upper time-limit for solver.
> >
> > **Returns**
> > > This *DiffusionNonIsoThermalCalculation* object

with_spherical_geometry(*first_interface_position*)

> Sets geometry to *spherical*, corresponds to a sphere with a certain radius.
>
> **Default**: A spherical geometry

> **Note:** With a cylindrical or spherical geometry, the system's zero coordinate (left boundary) is at the centre of the cylinder or sphere by default. By specifying the *first_interface_position*, a different left-most

---

coordinate can be defined. This allows to model a tube or a hollow sphere geometry. The highest coordinate (right boundary) is defined by the cylinder or sphere radius (i.e. by the width of all regions).

> **Parameters**
>> **first_interface_position** – The position of the left-most coordinate along the axis, only necessary for modeling a hollow sphere geometry [m]
>
> **Returns**
>> This *DiffusionNonIsoThermalCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
>> **system_modifications** – The system modification to be performed
>
> **Returns**
>> This *DiffusionNonIsoThermalCalculation* object

with_temperature_profile(*temperature_profile*)

> Sets the temperature profile to use with this calculation.
>
> **Parameters**
>> **temperature_profile** – The temperature profile object (specifying time / temperature points)
>
> **Returns**
>> This *DiffusionNonIsoThermalCalculation* object

with_timestep_control(*timestep_control*, *to*)

> Sets the timestep control options.
>
> It is possible specify the upper time-point for which this setting is valid using the parameter "to".
>
> **Default**: The end of the simulation.
>
> **Parameters**
>> - **timestep_control** – The new timestep control options
>>
>> - **to** – The upper time-limit for timestep_control.
>
> **Returns**
>> This *DiffusionNonIsoThermalCalculation* object

class +tc_toolbox.+diffusion.**DoubleGeometricGrid**(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

> Represents a double geometric grid.

DoubleGeometricGrid(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

Creates a double geometric grid.

Note: Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases, a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> **Parameters**
>
> - **no_of_points** – The number of points
> - **lower_geometrical_factor** – The geometrical factor for the left half
> - **upper_geometrical_factor** – The geometrical factor for the right half

static double_geometric(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

Factory method that creates a **new** double geometric grid.

Note: Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> **Parameters**
>
> - **no_of_points** – The number of points
> - **lower_geometrical_factor** – The geometrical factor for the left half
> - **upper_geometrical_factor** – The geometrical factor for the right half
>
> **Returns**
>   A new *DoubleGeometricGrid* object

static geometric(*no_of_points*, *geometrical_factor*)

Factory method that creates a **new** geometric grid.

Note: A grid that yields a varying density of grid points in the region. A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

> **Parameters**
>
> - **no_of_points** – The number of points
> - **geometrical_factor** – The geometrical factor
>
> **Returns**
>   A new *GeometricGrid* object

get_lower_geometrical_factor()

> Returns the lower geometrical factor (for the left half).

> > **Returns**
> >
> > > The lower geometrical factor

get_no_of_points()

> Returns number of grid points.

> > **Returns**
> >
> > > The number of grid points

get_type()

> Type of the grid.

> > **Returns**
> >
> > > The type of the grid

get_upper_geometrical_factor()

> Returns the upper geometrical factor (for the right half).

> > **Returns**
> >
> > > The upper geometrical factor

static linear(*no_of_points*)

> Factory method that creates a **new** equally spaced grid.

> > **Parameters**
> >
> > > **no_of_points** – The number of points

> > **Returns**
> >
> > > A new *LinearGrid* object

set_lower_geometrical_factor(*geometrical_factor*)

> Sets the lower (left half) geometrical factor.

> ---

> **Note:** A geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> ---

> > **Parameters**
> >
> > > **geometrical_factor** – The geometrical factor for the left half

> > **Returns**
> >
> > > This *DoubleGeometricGrid* object

set_no_of_points(*no_of_points*)

> Sets the number of grid points.

> > **Parameters**
> >
> > > **no_of_points** – The number of points

> > **Returns**
> >
> > > This *DoubleGeometricGrid* object

set_upper_geometrical_factor(*geometrical_factor*)

> Sets the upper (right half) geometrical factor.

Note: A geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> **Parameters**
> > **geometrical_factor** – The geometrical factor for the right half
>
> **Returns**
> > This *DoubleGeometricGrid* object

class +tc_toolbox.+diffusion.**ElementProfile**

Factory class providing objects for configuring a step, function or linear initial concentration profile.

static **constant**(*value*)

Factory method that creates a **new** constant initial concentration profile.

> **Parameters**
> > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].
>
> **Returns**
> > A new *ConstantProfile* object

static **funct**(*dictra_console_mode_function*)

Factory method that creates a **new** initial concentration profile defined by a function in DICTRA Console Mode syntax.

Note: This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.

> **Parameters**
> > **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.
>
> **Returns**
> > A new *FunctionProfile* object

static **linear**(*start_value*, *end_value*)

Factory method that creates a **new** linear initial concentration profile.

> **Parameters**
>
> - **start_value** – Composition at the left side of the region [unit as defined in *CompositionProfile*].
>
> - **end_value** – Composition at the right side of the region [unit as defined in *CompositionProfile*].
>
> **Returns**
> > A new *LinearProfile* object

static **step**(*lower_boundary*, *upper_boundary*, *step_at*)

Factory method that creates a **new** initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.

> **Parameters**

- **lower_boundary** – Composition before the step [unit as defined in *CompositionProfile*].

- **upper_boundary** – Composition after the step [unit as defined in *CompositionProfile*].

- **step_at** – The distance where the step should be [m].

> **Returns**
>> A new *StepProfile* object

class +tc_toolbox.+diffusion.**FixFluxValue**

Contains factory methods for the the different boundary conditions available.

FixFluxValue()

Represents a boundary having a fixed flux value.

This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*. Constructs an instance of *FixFluxValue*.

static **activity_flux_function()**

Factory method that creates a **new** activity-flux-function boundary condition.

This type of boundary condition is used to take into account the finite rate of a surface reaction.

The flux for the independent components must be given in the format:

J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

f and g must be expressed in DICTRA Console Mode syntax.

Note: The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

> **Returns**
>> A new *ActivityFluxFunction* object

static **closed_system()**

Factory method that creates a **new** closed-system boundary condition.

> **Returns**
>> A new *ClosedSystem* object

static **fix_flux_value()**

Factory method that creates a **new** fix-flux-value boundary condition.

This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.

> **Returns**
>> A new *FixFluxValue* object

---

static **fixed_compositions**(*unit_enum*)

> Factory method that creates a **new** fixed-composition boundary condition.
>
> > **Parameters**
> > > **unit_enum** – The composition unit
> >
> > **Returns**
> > > A new *FixedCompositions* object

get_type()

> The type of the boundary condition.
>
> > **Returns**
> > > The type

static **mixed_zero_flux_and_activity**()

> Factory method that creates a **new** mixed zero-flux and activity boundary condition
>
> > **Returns**
> > > A new *MixedZeroFluxAndActivity* object

set_flux(*element_name*, *J*, *to_time*)

> Enter functions that yield the flux times the molar volume for the specified element. May be a function of time, temperature and pressure: *J(T,P,TIME)*.
>
> > **Parameters**
> >
> > - **element_name** – The name of the element
> >
> > - **J** – the function *J(T,P,TIME)*
> >
> > - **to_time** – The max-time for which the flux function is used.

class +tc_toolbox.+diffusion.**FixedCompositions**(*unit_enum*)

> Represents a boundary having fixed composition values.
>
> **FixedCompositions**(*unit_enum*)
>
> > Represents a boundary having fixed composition values.
> >
> > > **Parameters**
> > > > **unit_enum** – The composition unit for all compositions at the boundary
>
> static **activity_flux_function**()
>
> > Factory method that creates a **new** activity-flux-function boundary condition.
> >
> > This type of boundary condition is used to take into account the finite rate of a surface reaction.
> >
> > The flux for the independent components must be given in the format:
> >
> > J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))
> >
> > where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.
> >
> > f and g must be expressed in DICTRA Console Mode syntax.
> >
> > ---
> >
> > **Note:** The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.
> >
> > ---

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

> **Returns**
>> A new *ActivityFluxFunction* object

static closed_system()

> Factory method that creates a **new** closed-system boundary condition.

> > **Returns**
> > > A new [`ClosedSystem`](#) object

static fix_flux_value()

> Factory method that creates a **new** fix-flux-value boundary condition.

> This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.

> > **Returns**
> > > A new *FixFluxValue* object

static fixed_compositions(*unit_enum***)**

> Factory method that creates a **new** fixed-composition boundary condition.

> > **Parameters**
> > > **unit_enum** – The composition unit

> > **Returns**
> > > A new *FixedCompositions* object

get_type()

> The type of the boundary condition.

> > **Returns**
> > > The type

static mixed_zero_flux_and_activity()

> Factory method that creates a **new** mixed zero-flux and activity boundary condition

> > **Returns**
> > > A new *MixedZeroFluxAndActivity* object

set_composition(*element_name***,** *value***)**

> Sets the composition for the specified element.

Note: The boundary composition needs to be specified for each element.

> > **Parameters**
> > > - **element_name** – The name of the element
> > > - **value** – The composition value [unit according to the constructor parameter]

---

class +tc_toolbox.+diffusion.**FunctionProfile**(*dictra_console_mode_function*)

> Creates an initial concentration profile defined by a function in DICTRA Console Mode syntax.

> **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.

> **FunctionProfile**(*dictra_console_mode_function*)
>
> > Creates a initial concentration profile defined by a function in DICTRA Console Mode syntax.
> >
> > ---
> >
> > **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.
> >
> > ---
> >
> > > **Parameters**
> > > > **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.
> > >
> > > **Returns**
> > > > A new *StepProfile* object

> **static constant**(*value*)
>
> > Factory method that creates a **new** constant initial concentration profile.
> >
> > > **Parameters**
> > > > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].
> > >
> > > **Returns**
> > > > A new *ConstantProfile* object

> **static funct**(*dictra_console_mode_function*)
>
> > Factory method that creates a **new** initial concentration profile defined by a function in DICTRA Console Mode syntax.
> >
> > ---
> >
> > **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.
> >
> > ---
> >
> > > **Parameters**
> > > > **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.
> > >
> > > **Returns**
> > > > A new *FunctionProfile* object

> **get_type**()
>
> > The type of the element profile.
> >
> > > **Returns**
> > > > The type

> **static linear**(*start_value*, *end_value*)
>
> > Factory method that creates a **new** linear initial concentration profile.
> >
> > > **Parameters**

- **start_value** – Composition at the left side of the region [unit as defined in *CompositionProfile*].

- **end_value** – Composition at the right side of the region [unit as defined in *CompositionProfile*].

> **Returns**
>> A new *LinearProfile* object

static step(*lower_boundary*, *upper_boundary*, *step_at*)

> Factory method that creates a **new** initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.

> **Parameters**

- **lower_boundary** – Composition before the step [unit as defined in *CompositionProfile*].

- **upper_boundary** – Composition after the step [unit as defined in *CompositionProfile*].

- **step_at** – The distance where the step should be [m].

> **Returns**
>> A new *StepProfile* object

class +tc_toolbox.+diffusion.**GeneralLowerHashinShtrikman**

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

GeneralLowerHashinShtrikman()

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. Constructs an instance of *GeneralLowerHashinShtrikman*.

static general_lower_hashin_shtrikman()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> Parameters
>> **excluded_phases** – The excluded phases
>
> Returns
>> A new `GeneralLowerHashinShtrikmanExcludedPhase` object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikman`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> Returns
>> A new `GeneralUpperHashinShtrikman` object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> Parameters
>> **excluded_phases** – The excluded phases
>
> Returns
>> A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> Returns
>> A new `HashinShtrikmanBoundMajority` object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajorityExcludedPhase`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> Parameters
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `HashinShtrikmanBoundMajorityExcludedPhase` object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundPrescribed`.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
>> **Parameters**
>>> **matrix_phase** – The matrix phase
>>
>> **Returns**
>>> A new `HashinShtrikmanBoundPrescribed` object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundPrescribedExcludedPhase`.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>> **Parameters**
>>> - **matrix_phase** – The matrix phase
>>>
>>> - **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new `HashinShtrikmanBoundPrescribedExcludedPhase` object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type `InverseRuleOfMixtures`.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
>> **Returns**
>>> A new `InverseRuleOfMixtures` object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type `InverseRuleOfMixturesExcludedPhase`.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
>> **Parameters**
>>> **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new `InverseRuleOfMixturesExcludedPhase` object

static labyrinth_factor_f(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > **Parameters**
> >> **matrix_phase** – The matrix phase
> >
> > **Returns**
> >> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> >> **matrix_phase** – The matrix phase
> >
> > **Returns**
> >> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> >> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> >> **excluded_phases** – The excluded phases
> >
> > **Returns**
> >> A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**GeneralLowerHashinShtrikmanExcludedPhase**(*excluded_phases*)

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

GeneralLowerHashinShtrikmanExcludedPhase(*excluded_phases*)

> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

static general_lower_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new *GeneralUpperHashinShtrikman* object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Parameters**
> > **matrix_phase** – The matrix phase
>
> **Returns**
> > A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**

- **matrix_phase** – The matrix phase

- **excluded_phases** – The excluded phases

> **Returns**
>> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
>> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
>> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> >
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> >
> > > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**GeneralUpperHashinShtrikman**

> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

GeneralUpperHashinShtrikman()

> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. Constructs an instance of *GeneralUpperHashinShtrikman*.

static general_lower_hashin_shtrikman()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> >
> > > A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> >
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> >
> > > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static general_upper_hashin_shtrikman()

> Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralUpperHashinShtrikman` object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `HashinShtrikmanBoundMajority` object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajorityExcludedPhase`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `HashinShtrikmanBoundMajorityExcludedPhase` object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundPrescribed`.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

>   **Parameters**
>>   **matrix_phase** – The matrix phase
>
>   **Returns**
>>   A new *HashinShtrikmanBoundPrescribed* object

static **hashin_shtrikman_bound_prescribed_excluded_phase**(*matrix_phase*, *excluded_phases*)

>   Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
>   Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
>   Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>   **Parameters**
>
>   - **matrix_phase** – The matrix phase
>
>   - **excluded_phases** – The excluded phases
>
>   **Returns**
>>   A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static **inverse_rule_of_mixtures**()

>   Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
>   Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
>   **Returns**
>>   A new *InverseRuleOfMixtures* object

static **inverse_rule_of_mixtures_excluded_phase**(*excluded_phases*)

>   Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
>   Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
>   **Parameters**
>>   **excluded_phases** – The excluded phases
>
>   **Returns**
>>   A new *InverseRuleOfMixturesExcludedPhase* object

static **labyrinth_factor_f**(*matrix_phase*)

>   Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
>   The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
>   **Parameters**
>>   **matrix_phase** – The matrix phase
>
>   **Returns**
>>   A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *LabyrinthFactorF2* object

static **rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> > > A new *RuleOfMixtures* object

static **rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**GeneralUpperHashinShtrikmanExcludedPhase**(*excluded_phases*)

> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

GeneralUpperHashinShtrikmanExcludedPhase(*excluded_phases*)

> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases

static **general_lower_hashin_shtrikman**()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralLowerHashinShtrikman` object

static **general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralLowerHashinShtrikmanExcludedPhase`.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralLowerHashinShtrikmanExcludedPhase` object

static **general_upper_hashin_shtrikman**()

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikman`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralUpperHashinShtrikman` object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static **hashin_shtrikman_bound_majority**()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Returns

A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters

excluded_phases – The excluded phases

Returns

A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Parameters

matrix_phase – The matrix phase

Returns

A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters

- **matrix_phase** – The matrix phase

- **excluded_phases** – The excluded phases

Returns

A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

Returns

A new *InverseRuleOfMixtures* object

---

static **inverse_rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *InverseRuleOfMixturesExcludedPhase* object

static **labyrinth_factor_f**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *LabyrinthFactorF2* object

static **rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> > > A new *RuleOfMixtures* object

static **rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**GeometricGrid**(*no_of_points*, *geometrical_factor*)

> Represents a geometric grid.

> **GeometricGrid**(*no_of_points*, *geometrical_factor*)

> > A grid that yields a varying density of grid points in the region.

> > ---

> > **Note:** A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

> > ---

> > **Parameters**

> > > • **no_of_points** – The number of points

> > > • **geometrical_factor** – The geometrical factor

> **static double_geometric**(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

> > Factory method that creates a **new** double geometric grid.

> > ---

> > **Note:** Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> > ---

> > **Parameters**

> > > • **no_of_points** – The number of points

> > > • **lower_geometrical_factor** – The geometrical factor for the left half

> > > • **upper_geometrical_factor** – The geometrical factor for the right half

> > **Returns**

> > > A new *DoubleGeometricGrid* object

> **static geometric**(*no_of_points*, *geometrical_factor*)

> > Factory method that creates a **new** geometric grid.

> > ---

> > **Note:** A grid that yields a varying density of grid points in the region. A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

> > ---

> > **Parameters**

> > > • **no_of_points** – The number of points

> > > • **geometrical_factor** – The geometrical factor

> > **Returns**

> > > A new *GeometricGrid* object

> **get_geometrical_factor**()

> > Returns the geometrical factor.

> **Returns**
> The geometrical factor

get_no_of_points()

> Returns the number of grid points.
>
> > **Returns**
> > The number of grid points

get_type()

> Returns the type of grid.
>
> > **Returns**
> > The type

static linear(*no_of_points*)

> Factory method that creates a **new** equally spaced grid.
>
> > **Parameters**
> > **no_of_points** – The number of points
> >
> > **Returns**
> > A new *LinearGrid* object

set_geometrical_factor(*geometrical_factor*)

> Sets the geometrical factor.
>
> ---
>
> **Note:** A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.
>
> ---
>
> > **Parameters**
> > **geometrical_factor** – The geometrical factor
> >
> > **Returns**
> > This *GeometricGrid* object

set_no_of_points(*no_of_points*)

> Sets the number of grid points.
>
> > **Parameters**
> > **no_of_points** – The number of points
> >
> > **Returns**
> > This *GeometricGrid* object

class +tc_toolbox.+diffusion.**GridPoint**(*distance*)

> Represents a grid point, this is used in combination with grids of the type +tc_toolbox.diffusion. PointByPointGrid.

GridPoint(*distance*)

> Creates a grid point, this is used in combination with grids of the type +tc_toolbox.diffusion. PointByPointGrid.
>
> > **Parameters**
> > **distance** – Position (origin at the left side of the grid)

add_composition(*element*, *value*)

> Adds a composition for the specified element to the grid point.

> > **Parameters**

> > - **element** – The element

> > - **value** – The composition value [unit as defined for the grid]

> > **Returns**

> > This *GridPoint* object

class +tc_toolbox.+diffusion.**HashinShtrikmanBoundMajority**

> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **HashinShtrikmanBoundMajority**()

> > Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. Constructs an instance of *HashinShtrikmanBoundMajority*.

> **static general_lower_hashin_shtrikman**()

> > Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

> > General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> > > **Returns**

> > > A new *GeneralLowerHashinShtrikman* object

> **static general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

> > Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

> > General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> > > **Parameters**

> > > **excluded_phases** – The excluded phases

> > > **Returns**

> > > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

> **static general_upper_hashin_shtrikman**()

> > Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

> > General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralUpperHashinShtrikman` object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `HashinShtrikmanBoundMajority` object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajorityExcludedPhase`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `HashinShtrikmanBoundMajorityExcludedPhase` object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundPrescribed`.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> Parameters
>> matrix_phase – The matrix phase
>
> Returns
>> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > Parameters
> >
> > - matrix_phase – The matrix phase
> >
> > - excluded_phases – The excluded phases
> >
> > Returns
> >> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
> > Returns
> >> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > Parameters
> >> excluded_phases – The excluded phases
> >
> > Returns
> >> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > Parameters
> >> matrix_phase – The matrix phase
> >
> > Returns
> >> A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

>   Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

>   The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

>   >   **Parameters**
>   >   >   **matrix_phase** – The matrix phase

>   >   **Returns**
>   >   >   A new *LabyrinthFactorF2* object

static **rule_of_mixtures**()

>   Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

>   Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

>   >   **Returns**
>   >   >   A new *RuleOfMixtures* object

static **rule_of_mixtures_excluded_phase**(*excluded_phases*)

>   Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

>   Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

>   >   **Parameters**
>   >   >   **excluded_phases** – The excluded phases

>   >   **Returns**
>   >   >   A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**HashinShtrikmanBoundMajorityExcludedPhase**(*excluded_phases*)

>   Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

>   Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

>   The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

>   **HashinShtrikmanBoundMajorityExcludedPhase**(*excluded_phases*)

>   >   Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

>   >   Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

>   >   The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

>   >   >   **Parameters**
>   >   >   >   **excluded_phases** – The excluded phases

>   static **general_lower_hashin_shtrikman**()

>   >   Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

>   >   General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralLowerHashinShtrikman` object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralLowerHashinShtrikmanExcludedPhase`.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralLowerHashinShtrikmanExcludedPhase` object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikman`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new `GeneralUpperHashinShtrikman` object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
>> A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>> **Parameters**
>>> **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
>> **Parameters**
>>> **matrix_phase** – The matrix phase
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>> **Parameters**
>>> - **matrix_phase** – The matrix phase
>>>
>>> - **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
>> **Returns**
>>> A new *InverseRuleOfMixtures* object

static **inverse_rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *InverseRuleOfMixturesExcludedPhase* object

static **labyrinth_factor_f**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *LabyrinthFactorF2* object

static **rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> > > A new *RuleOfMixtures* object

static **rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**HashinShtrikmanBoundPrescribed**(*matrix_phase*)

> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> **HashinShtrikmanBoundPrescribed**(*matrix_phase*)
>
> > Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the pre-scribed phase.
> >
> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spher-ical shells of each phase.
> >
> > > **Parameters**
> > > > **matrix_phase** – The matrix phase
>
> **static general_lower_hashin_shtrikman**()
>
> > Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
> >
> > General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
> >
> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spher-ical shells of each phase.
> >
> > > **Returns**
> > > > A new *GeneralLowerHashinShtrikman* object
>
> **static general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)
>
> > Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.
> >
> > General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
> >
> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spheri-cal shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
> >
> > > **Parameters**
> > > > **excluded_phases** – The excluded phases
> > >
> > > **Returns**
> > > > A new *GeneralLowerHashinShtrikmanExcludedPhase* object
>
> **static general_upper_hashin_shtrikman**()
>
> > Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.
> >
> > General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
> >
> > Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spher-ical shells of each phase.
> >
> > > **Returns**
> > > > A new *GeneralUpperHashinShtrikman* object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

>   **Parameters**
>   **excluded_phases** – The excluded phases
>
>   **Returns**
>   A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static **hashin_shtrikman_bound_majority**()

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

>   **Returns**
>   A new *HashinShtrikmanBoundMajority* object

static **hashin_shtrikman_bound_majority_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

>   **Parameters**
>   **excluded_phases** – The excluded phases
>
>   **Returns**
>   A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static **hashin_shtrikman_bound_prescribed**(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

>   **Parameters**
>   **matrix_phase** – The matrix phase
>
>   **Returns**
>   A new *HashinShtrikmanBoundPrescribed* object

static **hashin_shtrikman_bound_prescribed_excluded_phase**(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> > **Parameters**
> >
> > > • **matrix_phase** – The matrix phase
> > >
> > > • **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static **inverse_rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> > **Returns**
> > > A new *InverseRuleOfMixtures* object

static **inverse_rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> > **Parameters**
> > > **excluded_phases** – The excluded phases

> > **Returns**
> > > A new *InverseRuleOfMixturesExcludedPhase* object

static **labyrinth_factor_f**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> > **Parameters**
> > > **matrix_phase** – The matrix phase

> > **Returns**
> > > A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
> **matrix_phase** – The matrix phase

> **Returns**
> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> **excluded_phases** – The excluded phases

> **Returns**
> A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**HashinShtrikmanBoundPrescribedExcludedPhase**(*matrix_phase*, *excluded_phases*)

HashinShtrikmanBoundPrescribedExcludedPhase(*matrix_phase*, *excluded_phases*)

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> - **matrix_phase** – The matrix phase
> - **excluded_phases** – The excluded phases

static general_lower_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *GeneralUpperHashinShtrikman* object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Parameters**
>> **matrix_phase** – The matrix phase
>
> **Returns**
>> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> - **matrix_phase** – The matrix phase
>>
>> - **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
>> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> **Parameters**
>> **matrix_phase** – The matrix phase
>
> **Returns**
>> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> **Parameters**
>> **matrix_phase** – The matrix phase
>
> **Returns**
>> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> **Returns**
>> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**HomogenizationFunction**

> *Homogenization function* used for the *homogenization solver*. Many homogenization functions are based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. **Default**: *RULE_OF_MIXTURES* (i.e. upper Wiener bounds)

class +tc_toolbox.+diffusion.**HomogenizationFunctions**

static **general_lower_hashin_shtrikman**()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > > A new *GeneralLowerHashinShtrikman* object

static **general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static **general_upper_hashin_shtrikman**()

> Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.
>
> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > > A new *GeneralUpperHashinShtrikman* object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.
>
> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static **hashin_shtrikman_bound_majority**()

> Factory method that creates a **new** homogenization function of the type
> *HashinShtrikmanBoundMajority*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase
> with the highest local volume fraction.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > > A new *HashinShtrikmanBoundMajority* object

static **hashin_shtrikman_bound_majority_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type
> *HashinShtrikmanBoundMajorityExcludedPhase*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase
> with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered
> when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static **hashin_shtrikman_bound_prescribed**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type
> *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundPrescribed* object

static **hashin_shtrikman_bound_prescribed_excluded_phase**(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type
> *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most
> sluggish kinetics.
>
> > **Parameters**
> > > • **matrix_phase** – The matrix phase
> > >
> > > • **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
>> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
>> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**HomogenizationSolver**

Solver using the *Homogenization model*.

Note: This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the +tc_toolbox.diffusion.AutomaticSolver instead if you do not need that behavior.

HomogenizationSolver()

Creating a solver using the *homogenization model*.

Note: This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the +tc_toolbox.diffusion. AutomaticSolver instead if you do not need that behavior.

Constructs an instance of *HomogenizationSolver*.

static automatic()

Factory method that creates a **new** *automatic solver*. **This is the default solver and recommended for most applications**.

Note: This solver uses the homogenization model if any region has more than one phase, otherwise it uses the classic model.

> **Returns**
>> A new *AutomaticSolver* object

static classic()

Factory method that creates a **new** *classic solver*.

Note: This solver never switches to the homogenization model even if the solver fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

> **Returns**
>> A new *ClassicSolver* object

disable_global_minimization()

Disables global minimization to be used in equilibrium calculations. **Default**: Disabled

---

Note: In general, using global minimization **significantly increases the simulation time**, but there is also a significantly reduced risk for non-converged equilibrium calculations.

Returns
> A new *HomogenizationSolver* object

disable_interpolation_scheme()
> Configures the simulation not use *any interpolation scheme*. **Default**: To use the *logarithmic interpolation scheme* with 10000 discretization steps

---

Note: The homogenization scheme can be switched on by using *with_linear_interpolation_scheme* or *with_logarithmic_interpolation_scheme*.

---

enable_global_minimization()
> Enables global minimization to be used in equilibrium calculations. **Default**: Disabled

---

Note: In general, using global minimization **significantly increases the simulation time**, but there is also a significantly reduced risk for non-converged equilibrium calculations.

Returns
> A new *HomogenizationSolver* object

get_type()
> The type of solver.

> **Returns**
> > The type

static homogenization()
> Factory method that creates a **new** *homogenization solver*.

Note: This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the `+tc_toolbox.diffusion.AutomaticSolver` instead if you do not need that behavior.

Returns
> A new *HomogenizationSolver* object

set_fraction_of_free_memory_to_use(*fraction*)
> Sets the maximum fraction of free physical memory to be used by the interpolation scheme. **Default**: 1 / 10 of the free physical memory

> **Parameters**
> > **fraction** – The maximum free physical memory fraction to be used

> **Returns**
> > A new *HomogenizationSolver* object

set_memory_to_use(*memory_in_megabytes*)
> Sets the maximum physical memory in megabytes to be used by the interpolation scheme. **Default**: 1000 MBytes of the free physical memory

> **Parameters**
> > **memory_in_megabytes** – The maximum physical memory to be used

> **Returns**
> A new *HomogenizationSolver* object

with_function(*homogenization_function*)

> Sets the *homogenization function* used by the *homogenization model*.
>
> > **Parameters**
> > **homogenization_function** – The homogenization function used by the homogenization model
> >
> > **Returns**
> > A new *HomogenizationSolver* object

with_linear_interpolation_scheme(*steps*)

> Configures the simulation to use the *linear interpolation scheme*. **Default**: To use the *logarithmic interpolation scheme* with 10000 discretization steps
>
> > **Parameters**
> > **steps** – The number of discretization steps in each dimension
> >
> > **Returns**
> > A new *HomogenizationSolver* object

with_logarithmic_interpolation_scheme(*steps*)

> Configures the simulation to use the *linear interpolation scheme*. **Default**: To use the *logarithmic interpolation scheme* with 10000 discretization steps
>
> > **Parameters**
> > **steps** – The number of discretization steps in each dimension
> >
> > **Returns**
> > A new *HomogenizationSolver* object

class +tc_toolbox.+diffusion.**InverseRuleOfMixtures**

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

InverseRuleOfMixtures()

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Constructs an instance of *InverseRuleOfMixtures*.

static general_lower_hashin_shtrikman()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new `GeneralLowerHashinShtrikmanExcludedPhase` object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikman`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new `GeneralUpperHashinShtrikman` object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new `HashinShtrikmanBoundMajority` object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajorityExcludedPhase`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
>> **Parameters**
>>> **matrix_phase** – The matrix phase
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>> **Parameters**
>>
>>> • **matrix_phase** – The matrix phase
>>>
>>> • **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
>> **Returns**
>>> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
>> **Parameters**
>>> **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > **Parameters**
> >   **matrix_phase** – The matrix phase
> >
> > **Returns**
> >   A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> >   **matrix_phase** – The matrix phase
> >
> > **Returns**
> >   A new *LabyrinthFactorF2* object

static rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> >   A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> >   **excluded_phases** – The excluded phases
> >
> > **Returns**
> >   A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**InverseRuleOfMixturesExcludedPhase**(*excluded_phases*)

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
> Excluded phases are not considered in the diffusion calculations.
>
> **InverseRuleOfMixturesExcludedPhase**(*excluded_phases*)
>
> > Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
> >
> > Excluded phases are not considered in the diffusion calculations.
> >
> > > **Parameters**
> > >   **excluded_phases** – The excluded phases

static **general_lower_hashin_shtrikman**()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > > A new *GeneralLowerHashinShtrikman* object

static **general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static **general_upper_hashin_shtrikman**()

> Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.
>
> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > > A new *GeneralUpperHashinShtrikman* object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.
>
> General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static **hashin_shtrikman_bound_majority**()

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Returns**
> > A new *HashinShtrikmanBoundMajority* object

static **hashin_shtrikman_bound_majority_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > **excluded_phases** – The excluded phases
>
> > **Returns**
> > A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static **hashin_shtrikman_bound_prescribed**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Parameters**
> > **matrix_phase** – The matrix phase
>
> > **Returns**
> > A new *HashinShtrikmanBoundPrescribed* object

static **hashin_shtrikman_bound_prescribed_excluded_phase**(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > - **matrix_phase** – The matrix phase
> >
> > - **excluded_phases** – The excluded phases
>
> > **Returns**
> > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
>> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
>> **excluded_phases** – The excluded phases

> **Returns**
>> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
>> **matrix_phase** – The matrix phase

> **Returns**
>> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
>> A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**LabyrinthFactorF**(*matrix_phase*)

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

LabyrinthFactorF(*matrix_phase*)

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
> > **matrix_phase** – The matrix phase

static **general_lower_hashin_shtrikman**()

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *GeneralLowerHashinShtrikman* object

static **general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static **general_upper_hashin_shtrikman**()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Returns
A new `GeneralUpperHashinShtrikman` object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `GeneralUpperHashinShtrikmanExcludedPhase`.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters
excluded_phases – The excluded phases

Returns
A new `GeneralUpperHashinShtrikmanExcludedPhase` object

static **hashin_shtrikman_bound_majority**()

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajority`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Returns
A new `HashinShtrikmanBoundMajority` object

static **hashin_shtrikman_bound_majority_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundMajorityExcludedPhase`.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

Parameters
excluded_phases – The excluded phases

Returns
A new `HashinShtrikmanBoundMajorityExcludedPhase` object

static **hashin_shtrikman_bound_prescribed**(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type `HashinShtrikmanBoundPrescribed`.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

Parameters
matrix_phase – The matrix phase

Returns
> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the pre-scribed phase.

> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>
> - **matrix_phase** – The matrix phase
>
> - **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
> > A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
> > **matrix_phase** – The matrix phase
>
> **Returns**
> > A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
> > **matrix_phase** – The matrix phase
>
> **Returns**
> > A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
> > A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.LabyrinthFactorF2(*matrix_phase*)

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

LabyrinthFactorF2(*matrix_phase*)

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
> > **matrix_phase** – The matrix phase

static general_lower_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *GeneralUpperHashinShtrikman* object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static hashin_shtrikman_bound_majority()

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Parameters**
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > > - **matrix_phase** – The matrix phase
> > > - **excluded_phases** – The excluded phases
> >
> > **Returns**
> > > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
> > **Returns**
> > > A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

>   Parameters
>>   **excluded_phases** – The excluded phases
>
>   Returns
>>   A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

>   Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

>   The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

>   Parameters
>>   **matrix_phase** – The matrix phase
>
>   Returns
>>   A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

>   Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

>   The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

>   Parameters
>>   **matrix_phase** – The matrix phase
>
>   Returns
>>   A new *LabyrinthFactorF2* object

static rule_of_mixtures()

>   Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

>   Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

>   Returns
>>   A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

>   Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

>   Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

>   Parameters
>>   **excluded_phases** – The excluded phases
>
>   Returns
>>   A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**LinearGrid**(*no_of_points*)

>   Represents an equally spaced grid.

>   **LinearGrid**(*no_of_points*)

>>   Creates an equally spaced grid.

>>   Parameters
>>>   **no_of_points** – The number of points

static **double_geometric**(*no_of_points*, *lower_geometrical_factor*, *upper_geometrical_factor*)

> Factory method that creates a **new** double geometric grid.

Note: Double geometric grids have a high number of grid points in the middle or at both ends of a region. One geometrical factor for the lower (left) and upper (right) half of the region need to specified. In both cases a geometrical factor of larger than one yields a higher density of grid points at the lower end of the half and vice versa for a factor smaller than one.

> **Parameters**
>
> - **no_of_points** – The number of points
> - **lower_geometrical_factor** – The geometrical factor for the left half
> - **upper_geometrical_factor** – The geometrical factor for the right half
>
> **Returns**
> A new *DoubleGeometricGrid* object

static **geometric**(*no_of_points*, *geometrical_factor*)

> Factory method that creates a **new** geometric grid.

Note: A grid that yields a varying density of grid points in the region. A geometrical factor larger than one yields a higher density of grid points at the lower end of the region and a factor smaller than one yields a higher density of grid points at the upper end of the region.

> **Parameters**
>
> - **no_of_points** – The number of points
> - **geometrical_factor** – The geometrical factor
>
> **Returns**
> A new *GeometricGrid* object

get_no_of_points()

> Returns the number of grid points.
>
> > **Returns**
> > The number of grid points

get_type()

> Type of the grid.
>
> > **Returns**
> > The type

static **linear**(*no_of_points*)

> Factory method that creates a **new** equally spaced grid.
>
> > **Parameters**
> > **no_of_points** – The number of points
> >
> > **Returns**
> > A new *LinearGrid* object

---

set_no_of_points(*no_of_points*)

> Sets the number of grid points.

> > **Parameters**
> > > **no_of_points** – The number of points

> > **Returns**
> > > This *LinearGrid* object

class +tc_toolbox.+diffusion.**LinearProfile**(*start_value*, *end_value*)

> Represents a linear initial concentration profile.

> **LinearProfile**(*start_value*, *end_value*)

> > Represents a linear initial concentration profile.

> > > **Parameters**

> > > - **start_value** – Composition at the left side of the region [unit as defined in *CompositionProfile*].

> > > - **end_value** – Composition at the right side of the region [unit as defined in *CompositionProfile*].

> **static constant**(*value*)

> > Factory method that creates a **new** constant initial concentration profile.

> > > **Parameters**
> > > > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].

> > > **Returns**
> > > > A new *ConstantProfile* object

> **static funct**(*dictra_console_mode_function*)

> > Factory method that creates a **new** initial concentration profile defined by a function in DICTRA Console Mode syntax.

> > ---

> > **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.

> > ---

> > > **Parameters**
> > > > **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.

> > > **Returns**
> > > > A new *FunctionProfile* object

> **get_type**()

> > The type of the element profile.

> > > **Returns**
> > > > The type

> **static linear**(*start_value*, *end_value*)

> > Factory method that creates a **new** linear initial concentration profile.

> > > **Parameters**

> > > - **start_value** – Composition at the left side of the region [unit as defined in *CompositionProfile*].

- **end_value** – Composition at the right side of the region [unit as defined in *CompositionProfile*].

Returns
A new *LinearProfile* object

static step(*lower_boundary*, *upper_boundary*, *step_at*)

Factory method that creates a **new** initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.

Parameters

- **lower_boundary** – Composition before the step [unit as defined in *CompositionProfile*].

- **upper_boundary** – Composition after the step [unit as defined in *CompositionProfile*].

- **step_at** – The distance where the step should be [m].

Returns
A new *StepProfile* object

class +tc_toolbox.+diffusion.**MixedZeroFluxAndActivity**

Represents a boundary having zero-flux as well as fixed-activity conditions.

Default: On that boundary for every element without an explicitly defined condition, a zero-flux boundary condition is used.

MixedZeroFluxAndActivity()

Represents a boundary having zero-flux as well as fixed-activity conditions.

Default: On that boundary for every element without an explicitly defined condition, a zero-flux boundary condition is used. Constructs an instance of *MixedZeroFluxAndActivity*.

static activity_flux_function()

Factory method that creates a **new** activity-flux-function boundary condition.

This type of boundary condition is used to take into account the finite rate of a surface reaction.

The flux for the independent components must be given in the format:

J = f(T,P,TIME) * (ACTIVITY^N - g(T,P,TIME))

where f and g may be functions of time (TIME), temperature (T), and pressure (P), and N is an integer.

f and g must be expressed in DICTRA Console Mode syntax.

Note: The activities are those with user-defined reference states. The function mass transfer coefficient is the mass transfer coefficient, activity of the corresponding species in the gas is the activity of the corresponding species in the gas and N is a stoichiometric coefficient.

Note: For more details see L. Sproge and J. Ågren, "Experimental and theoretical studies of gas consumption in the gas carburizing process" J. Heat Treat. 6, 9–19 (1988).

Returns
A new *ActivityFluxFunction* object

static closed_system()

Factory method that creates a **new** closed-system boundary condition.

> **Returns**
>> A new *ClosedSystem* object

static fix_flux_value()

Factory method that creates a **new** fix-flux-value boundary condition.

This type of boundary condition makes it possible to enter functions that yield the flux times the molar volume for the independent components. May be a function of time, temperature and pressure: *J(T,P,TIME)*.

> **Returns**
>> A new *FixFluxValue* object

static fixed_compositions(*unit_enum*)

Factory method that creates a **new** fixed-composition boundary condition.

> **Parameters**
>> **unit_enum** – The composition unit

> **Returns**
>> A new *FixedCompositions* object

get_type()

The type of the boundary condition.

> **Returns**
>> The type

static mixed_zero_flux_and_activity()

Factory method that creates a **new** mixed zero-flux and activity boundary condition

> **Returns**
>> A new *MixedZeroFluxAndActivity* object

set_activity_for_element(*element_name*, *activity*, *to_time*)

Sets an activity expression for an element at the boundary. Enter a formula that the software evaluates during the calculation.

The formula can be:

- a function of the variable *TIME*
- a constant

The formula must be written with these rules:

- a number must begin with a number (not a .)
- a number must have a dot or an exponent (*E*)

The operators +, -, *, /, ** (exponentiation) can be used and with any level of parenthesis. As shown, the following operators must be followed by open and closed parentheses *()*

- *SQRT(X)* is the square root
- *EXP(X)* is the exponential
- *LOG(X)* is the natural logarithm
- *LOG10(X)* is the base 10 logarithm
- *SIN(X)*, *COS(X)*, *TAN(X)*, *ASIN(X)*, *ACOS(X)*, *ATAN(X)*

- *SINH(X)*, *COSH(X)*, *TANH(X)*, *ASINH(X)*, *ACOSH(X)*, *ATANH(X)*

- *SIGN(X)*

- *ERF(X)* is the error function

Default: the expression entered is used for the entire simulation.

> **Parameters**
>
> - **element_name** – The name of the element
>
> - **activity** – The activity
>
> - **to_time** – The max-time for which the activity is used.

set_zero_flux_for_element(*element_name*)

> Sets a zero-flux condition for an element at the boundary. **Default for all elements at the boundary without an explicitly defined condition**
>
> **Parameters**
> element_name – The name of the element

class +tc_toolbox.+diffusion.**Options**

General simulation conditions for the diffusion calculations.

Options()

> General simulation conditions for diffusion calculations. Constructs an instance of *Options*.

disable_forced_starting_values_in_equilibrium_calculations()

> Disables forced starting values for the equilibrium calculations. **The default is 'enable_automatic_forced_starting_values_in_equilibrium_calculations'**.
>
> **Returns**
> This *Options* object

disable_save_results_to_file()

> Disables the saving of results to file during the simulation. **Default**: Saving of the results at every timestep
>
> **Returns**
> This *Options* object

enable_automatic_forced_starting_values_in_eq_calculations()

> Lets calculation engine decide if forced start values for the equilibrium calculations should be used. **This is the default setting**.
>
> **Returns**
> This *Options* object

enable_forced_starting_values_in_equilibrium_calculations()

> Enables forced start values for the equilibrium calculations. **The default is 'enable_automatic_forced_starting_values_in_equilibrium_calculations'**.
>
> **Returns**
> This *Options* object

enable_save_results_to_file(*every_nth_step*)

> Enables and configures saving of results to file during the simulation. They can be saved for every n-th or optionally for every timestep (*-1*). **Default**: Saving of the results at every timestep
>
> **Parameters**
> every_nth_step – *-1* or a value ranging from 0 to 99

Returns
> This `Options` object

enable_time_integration_method_automatic()

> Enables automatic selection of integration method. **This is the default method**.

> **Returns**
> > This `Options` object

enable_time_integration_method_euler_backwards()

> Enables *Euler backwards* integration. **The default method is enable_time_integration_method_automatic**.

> ---
> **Note:** This method is more stable but less accurate and may be necessary if large fluctuations occur in the profiles.
> ---

> **Returns**
> > This `Options` object

enable_time_integration_method_trapezoidal()

> Enables *trapezoidal* integration.

> ---
> **Note:** If large fluctuations occur in the profiles, it may be necessary to use the more stable but less accurate *Euler backwards method*.
> ---

> **Returns**
> > This `Options` object

set_default_driving_force_for_phases_allowed_to_form_at_interf(*driving_force*)

> Sets the default required driving force for phases allowed to form at the interfaces. **Default**: 1.0e-5

> ---
> **Note:** The required driving force (evaluated as *DGM(ph)*) is used for determining whether an inactive phase is stable, i.e. actually formed. *DGM* represents the driving force normalized by *RT* and is dimensionless.
> ---

> **Parameters**
> > **driving_force** – The driving force (*DGM(ph)*) [-]

> **Returns**
> > This `Options` object

class +tc_toolbox.+diffusion.**PointByPointGrid**(*unit_enum*)

> Represents a point-by-point grid. This is setting the grid and the compositions at once, it is typically used to enter a measured composition profile or the result from a previous calculation.

> ---
> **Note:** If a point-by-point grid is used, it is not necessary to specify the grid and composition profile separately.
> ---

PointByPointGrid(*unit_enum*)

> Represents a point-by-point grid.

> > **Parameters**
> > > **unit_enum** – The unit of the compositions

add_point(*grid_point*)

> Adds a grid point to the grid.

> > **Parameters**
> > > **grid_point** – The grid point

> > **Returns**
> > > This *PointByPointGrid* object

get_type()

> Type of the grid.

> > **Returns**
> > > The type

class +tc_toolbox.+diffusion.**Region**(*name*)

> Represents a region of the simulation domain that can contain more that one phase.

Note: The first added phase represents the matrix phase, while all later added phases are *spheriod phases*, i.e. precipitate phases.

Region(*name*)

> A region of the simulation domain that can contain more than one phase.

Note: The first added phase represents the matrix phase, while all later added phases are *spheriod phases*, i.e. precipitate phases.

> > **Parameters**
> > > **name** – The name of the region

add_phase(*phase_name*, *is_matrix_phase*)

> Adds a phase to the region, each region must contain at least one phase.

Note: Normally the *matrix phase* and the *precipitate phases* are automatically chosen based on the presence of all profile elements in the phase and if it has diffusion data. If multiple phases have equal properties, the phase that was added first is chosen. The matrix phase can be explicitly set by using *is_matrix_phase=True*.

Note: If multiple phases are added to a region, the *homogenization model* is applied. That means that average properties of the local phase mixture are used.

> > **Parameters**
> > > • **phase_name** – The phase name

- **is_matrix_phase** – If set to *True* this phase is explicitly set as matrix phase for the region, if no phase is set to *True*, the matrix phase is chosen automatically

> **Returns**
> This *Region* object

add_phase_allowed_to_form_at_left_interface(*phase_name*, *driving_force*)

Adds a phase allowed to form at the left boundary of the region (an *inactive phase*). The phase will only appear at the interface as a new automatic region if the driving force to form it is sufficiently high.

> **Parameters**
>
> - **phase_name** – The phase name
> - **driving_force** – The driving force for the phase to form (*DGM(ph)*)
>
> **Returns**
> This *Region* object

add_phase_allowed_to_form_at_right_interface(*phase_name*, *driving_force*)

Adds a phase allowed to form at the right boundary of the region (an *inactive phase*). The phase will only appear at the interface as a new automatic region if the driving force to form it is sufficiently high.

> **Parameters**
>
> - **phase_name** – The phase name
> - **driving_force** – The driving force for the phase to form (*DGM(ph)*)
>
> **Returns**
> This *Region* object

remove_all_phases()

Removes all previously added phases from the region.

> **Returns**
> This *Region* object

set_width(*width*)

Defined the width of the region.

Note: This method needs only to be used if a calculated grid has been defined (using *with_grid()*).

> **Parameters**
> **width** – The width [m]
>
> **Returns**
> This *Region* object

with_composition_profile(*initial_compositions*)

Defines the initial composition profiles for all elements in the region.

Note: This method needs only to be used if a calculated grid has been defined (using *with_grid()*).

> **Parameters**
> **initial_compositions** – The initial composition profiles for all elements

> **Returns**
>> This *Region* object

with_grid(*grid*)

> Defines a calculated grid in the region. If measured composition profiles or the result from a previous calculation should be used, instead `with_point_by_point_grid_containing_compositions()` needs to be applied.
>
> ---
> **Note:** The composition profiles need to be defined separately using `with_composition_profile()`, additionally the region width needs to be specified using `set_width()`.
>
> ---
>
> **Parameters**
>> **grid** – The grid
>
> **Returns**
>> This *Region* object

with_point_by_point_grid_containing_compositions(*grid*)

> Defines a point-by-point grid in the region. This is setting the grid and the compositions at once, it is typically used to enter a measured composition profile or the result from a previous calculation. If the composition profile should be calculated (linear, geometric, ...) `with_grid()` should be used instead.
>
> ---
> **Note:** If a point-by-point grid is used, `with_grid()`, `with_composition_profile()` and `set_width()` are unnecessary and must not be used.
>
> ---
>
> **Parameters**
>> **grid** – The point-by-point grid
>
> **Returns**
>> This *Region* object

class +tc_toolbox.+diffusion.**RuleOfMixtures**

> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

RuleOfMixtures()

> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Constructs an instance of *RuleOfMixtures*.

static general_lower_hashin_shtrikman()

> Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.
>
> General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> **Returns**
>> A new *GeneralLowerHashinShtrikman* object

TC-Toolbox for MATLAB® Documentation, Release 2024a

static **general_lower_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static **general_upper_hashin_shtrikman**()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *GeneralUpperHashinShtrikman* object

static **general_upper_hashin_shtrikman_excluded_phase**(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static **hashin_shtrikman_bound_majority**()

Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.

Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> > A new *HashinShtrikmanBoundMajority* object

4.1. Calculations 217

static **hashin_shtrikman_bound_majority_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> > **excluded_phases** – The excluded phases
> >
> > **Returns**
> > A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static **hashin_shtrikman_bound_prescribed**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
> > **Parameters**
> > **matrix_phase** – The matrix phase
> >
> > **Returns**
> > A new *HashinShtrikmanBoundPrescribed* object

static **hashin_shtrikman_bound_prescribed_excluded_phase**(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
> > **Parameters**
> >
> > • **matrix_phase** – The matrix phase
> >
> > • **excluded_phases** – The excluded phases
> >
> > **Returns**
> > A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static **inverse_rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.
>
> > **Returns**
> > A new *InverseRuleOfMixtures* object

static **inverse_rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.
>
> Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> >
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> >
> > > A new *InverseRuleOfMixturesExcludedPhase* object

static **labyrinth_factor_f**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.
>
> > **Parameters**
> >
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> >
> > > A new *LabyrinthFactorF* object

static **labyrinth_factor_f2**(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.
>
> The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.
>
> > **Parameters**
> >
> > > **matrix_phase** – The matrix phase
> >
> > **Returns**
> >
> > > A new *LabyrinthFactorF2* object

static **rule_of_mixtures**()

> Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.
>
> > **Returns**
> >
> > > A new *RuleOfMixtures* object

static **rule_of_mixtures_excluded_phase**(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.
>
> Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.
>
> > **Parameters**
> >
> > > **excluded_phases** – The excluded phases
> >
> > **Returns**
> >
> > > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**RuleOfMixturesExcludedPhase**(*excluded_phases*)

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Excluded phases are not considered in the diffusion calculations.

RuleOfMixturesExcludedPhase(*excluded_phases*)

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> **excluded_phases** – The excluded phases

static general_lower_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikman*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> A new *GeneralLowerHashinShtrikman* object

static general_lower_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralLowerHashinShtrikmanExcludedPhase*.

General lower Hashin-Shtrikman bounds: the outermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> **excluded_phases** – The excluded phases

> **Returns**
> A new *GeneralLowerHashinShtrikmanExcludedPhase* object

static general_upper_hashin_shtrikman()

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikman*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.

> **Returns**
> A new *GeneralUpperHashinShtrikman* object

static general_upper_hashin_shtrikman_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *GeneralUpperHashinShtrikmanExcludedPhase*.

General upper Hashin-Shtrikman bounds: the innermost shell consists of the phase with the most sluggish kinetics.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
>> **excluded_phases** – The excluded phases
>
> **Returns**
>> A new *GeneralUpperHashinShtrikmanExcludedPhase* object

static hashin_shtrikman_bound_majority()

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajority*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
>> **Returns**
>>> A new *HashinShtrikmanBoundMajority* object

static hashin_shtrikman_bound_majority_excluded_phase(*excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundMajorityExcludedPhase*.
>
> Hashin-Shtrikman bounds with majority phase as matrix phase: the outermost shell consists of the phase with the highest local volume fraction. Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.
>
>> **Parameters**
>>> **excluded_phases** – The excluded phases
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundMajorityExcludedPhase* object

static hashin_shtrikman_bound_prescribed(*matrix_phase*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribed*.
>
> Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the prescribed phase.
>
> Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spherical shells of each phase.
>
>> **Parameters**
>>> **matrix_phase** – The matrix phase
>>
>> **Returns**
>>> A new *HashinShtrikmanBoundPrescribed* object

static hashin_shtrikman_bound_prescribed_excluded_phase(*matrix_phase*, *excluded_phases*)

> Factory method that creates a **new** homogenization function of the type *HashinShtrikmanBoundPrescribedExcludedPhase*.

Hashin-Shtrikman bounds with prescribed phase as matrix phase: the outermost shell consists of the pre-scribed phase.

Based on a variant of the Hashin-Shtrikman bounds, their geometrical interpretation are concentric spheri-cal shells of each phase. The excluded phases are not considered when evaluating what phase has the most sluggish kinetics.

> **Parameters**
> - **matrix_phase** – The matrix phase
> - **excluded_phases** – The excluded phases
>
> **Returns**
> A new *HashinShtrikmanBoundPrescribedExcludedPhase* object

static inverse_rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixtures*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion.

> **Returns**
> A new *InverseRuleOfMixtures* object

static inverse_rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *InverseRuleOfMixturesExcludedPhase*.

Lower Wiener bounds: the geometrical interpretation are continuous layers of each phase orthogonal to the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> **excluded_phases** – The excluded phases
>
> **Returns**
> A new *InverseRuleOfMixturesExcludedPhase* object

static labyrinth_factor_f(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase.

> **Parameters**
> **matrix_phase** – The matrix phase
>
> **Returns**
> A new *LabyrinthFactorF* object

static labyrinth_factor_f2(*matrix_phase*)

Factory method that creates a **new** homogenization function of the type *LabyrinthFactorF2*.

The labyrinth factor functions implies that all diffusion takes place in a single continuous matrix phase. The impeding effect on diffusion by phases dispersed in the matrix phase is taken into account by multiplying the flux with the volume fraction of the matrix phase squared.

> **Parameters**
> **matrix_phase** – The matrix phase
>
> **Returns**
> A new *LabyrinthFactorF2* object

static rule_of_mixtures()

Factory method that creates a **new** homogenization function of the type *RuleOfMixtures*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion.

> **Returns**
> > A new *RuleOfMixtures* object

static rule_of_mixtures_excluded_phase(*excluded_phases*)

Factory method that creates a **new** homogenization function of the type *RuleOfMixturesExcludedPhase*.

Upper Wiener bounds: the geometrical interpretation are continuous layers of each phase parallel with the direction of diffusion. Excluded phases are not considered in the diffusion calculations.

> **Parameters**
> > **excluded_phases** – The excluded phases
>
> **Returns**
> > A new *RuleOfMixturesExcludedPhase* object

class +tc_toolbox.+diffusion.**SimulationTime**

Specifying special time steps for the evaluation of diffusion results.

Note: These placeholders should be used because especially the actual last timestep will slightly differ from the specified end time of the simulation.

---

class +tc_toolbox.+diffusion.**Solver**

Factory class providing objects representing a solver.

static automatic()

Factory method that creates a **new** *automatic solver*. **This is the default solver and recommended for most applications**.

Note: This solver uses the homogenization model if any region has more than one phase, otherwise it uses the classic model.

> **Returns**
> > A new *AutomaticSolver* object

static classic()

Factory method that creates a **new** *classic solver*.

Note: This solver never switches to the homogenization model even if the solver fails to converge. Use the +tc_toolbox.diffusion.AutomaticSolver if necessary instead.

> **Returns**
> > A new *ClassicSolver* object

static homogenization()

> Factory method that creates a **new** *homogenization solver*.

> **Note:** This solver always uses the homogenization model, even if all regions have only one phase. The solver is **significantly slower than the Classic model**. Use the +tc_toolbox.diffusion. AutomaticSolver instead if you do not need that behavior.

> > **Returns**
> > > A new *HomogenizationSolver* object

class +tc_toolbox.+diffusion.**StepProfile**(*lower_boundary*, *upper_boundary*, *step_at*)

> Represents an initial constant concentration profile with a step at the specified position.

> **StepProfile**(*lower_boundary*, *upper_boundary*, *step_at*)

> > Creates an initial concentration profile with a step at the specified position, otherwise the composition is constant at the specified values.

> > **Parameters**

> > > - **lower_boundary** – Composition before the step [unit as defined in *CompositionProfile*].
> > > - **upper_boundary** – Composition after the step [unit as defined in *CompositionProfile*].
> > > - **step_at** – The distance where the step should be [m].

> **static constant**(*value*)

> > Factory method that creates a **new** constant initial concentration profile.

> > **Parameters**
> > > **value** – The constant composition in the region. [unit as defined in *CompositionProfile*].

> > **Returns**
> > > A new *ConstantProfile* object

> **static funct**(*dictra_console_mode_function*)

> > Factory method that creates a **new** initial concentration profile defined by a function in DICTRA Console Mode syntax.

> > **Note:** This is an advanced feature, preferably a complex concentration profile should be generated using third party libraries and added to the simulation using +tc_toolbox.diffusion.PointByPointGrid.

> > **Parameters**
> > > **dictra_console_mode_function** – The function, expressed in DICTRA Console Mode syntax.

> > **Returns**
> > > A new *FunctionProfile* object

> **get_type()**

> > The type of the element profile.

> > **Returns**
> > > The type

---

static **linear**(*start_value*, *end_value*)

> Factory method that creates a **new** linear initial concentration profile.
>
> > **Parameters**
> >
> > - **start_value** – Composition at the left side of the region [unit as defined in `CompositionProfile`].
> >
> > - **end_value** – Composition at the right side of the region [unit as defined in `CompositionProfile`].
> >
> > **Returns**
> >
> > A new `LinearProfile` object

static **step**(*lower_boundary*, *upper_boundary*, *step_at*)

> Factory method that creates a **new** initial concentration profile with a step at the specified distance, otherwise the composition is constant at the specified values.
>
> > **Parameters**
> >
> > - **lower_boundary** – Composition before the step [unit as defined in `CompositionProfile`].
> >
> > - **upper_boundary** – Composition after the step [unit as defined in `CompositionProfile`].
> >
> > - **step_at** – The distance where the step should be [m].
> >
> > **Returns**
> >
> > A new `StepProfile` object

class +tc_toolbox.+diffusion.**TimestepControl**

> Settings that control the time steps in the simulation.

> **TimestepControl**()
>
> > Settings that control the time steps in the simulation. Constructs an instance of `TimestepControl`.

> **disable_check_interface_position**()
>
> > Disables checking of the interface position, i.e. the timesteps are not controlled by the phase interface displacement during the simulation. **The default setting is :func:`enable_automatic_check_interface_position`**.
> >
> > > **Returns**
> > >
> > > This `TimestepControl` object

> **enable_automatic_check_interface_position**()
>
> > Lets calculation engine decide if checking of the interface position should be used. **This is the default setting**.
> >
> > > **Returns**
> > >
> > > This `TimestepControl` object

> **enable_check_interface_position**()
>
> > Enables checking of the interface position, i.e. the timesteps are controlled by the phase interface displacement during the simulation. **The default setting is :func:`enable_automatic_check_interface_position`**.
> >
> > > **Returns**
> > >
> > > This `TimestepControl` object

> **set_initial_time_step**(*initial_time_step*)
>
> > Sets the initial timestep. **Default**: 1.0e-7 s

> **Parameters**
>> **initial_time_step** – The initial timestep [s]
>
> **Returns**
>> This *TimestepControl* object

set_max_absolute_error(*absolute_error*)

> Sets the maximum absolute error. **Default**: 1.0e-5
>
>> **Parameters**
>>> **absolute_error** – The maximum absolute error
>>
>> **Returns**
>>> This *TimestepControl* object

set_max_relative_error(*relative_error*)

> Sets the maximum relative error. **Default**: 0.05
>
>> **Parameters**
>>> **relative_error** – The maximum relative error
>>
>> **Returns**
>>> This *TimestepControl* object

set_max_timestep_allowed_as_percent_of_simulation_time(*max_timestep_allowed_as_percent_of_simulation_time*)

> The maximum timestep allowed during the simulation, specified in percent of the simulation time. **Default**: 10.0%
>
>> **Parameters**
>>> **max_timestep_allowed_as_percent_of_simulation_time** – The maximum timestep allowed [%]
>>
>> **Returns**
>>> This *TimestepControl* object

set_max_timestep_increase_factor(*max_timestep_increase_factor*)

> Sets the maximum timestep increase factor. **Default**: 2
>
> ---
>
> **Note:** For example, if 2 is entered the maximum time step is twice as long as the previous time step taken.
>
> ---
>
>> **Parameters**
>>> **max_timestep_increase_factor** – The maximum timestep increase factor
>>
>> **Returns**
>>> This *TimestepControl* object

set_smallest_time_step_allowed(*smallest_time_step_allowed*)

> Sets the smallest time step allowed during the simulation. This is required when using the automatic procedure to determine the time step. **Default**: 1.0e-7 s
>
>> **Parameters**
>>> **smallest_time_step_allowed** – The smalles timestep allowed [s]
>>
>> **Returns**
>>> This *TimestepControl* object

class +tc_toolbox.+diffusion.**Unit**

> Represents a composition unit.

4.1.7 Package "propertymodel"

class +tc_toolbox.+propertymodel.**PropertyModelCalculation**(*back*)

Configuration for a Property Model calculation.

Note: Specify the settings, the calculation is performed with *calculate()*.

> **PropertyModelCalculation**(*back*)
>
> > Call base constructor: `tc_toolbox.AbstractCalculation`.
>
> **add_poly_command**(*poly_command*)
>
> > This function has no effect and will be removed in 2024b. If you use it, please remove from your code.
> >
> > > **Parameters**
> > > > **poly_command** –
> > >
> > > **Returns**
> > > > This *PropertyModelCalculation* object
>
> **calculate**(*timeout_in_minutes*)
>
> > Runs the Property Model calculation.
> >
> > > **Parameters**
> > > > **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a CalculationEngineException will be thrown.
> > >
> > > **Returns**
> > > > A *PropertyModelResult* which later can be used to get specific values from the simulation.
>
> **get_argument_default**(*argument_id*)
>
> > Returns the default value for the specified argument. The argument id can be obtained with *get_arguments()*.
> >
> > > **Parameters**
> > > > **argument_id** – The argument id
> > >
> > > **Returns**
> > > > The default value (the type depends on the argument)
>
> **get_argument_description**(*argument_id*)
>
> > Returns the detailed description of the argument. The id can be obtained with *get_arguments()*.
> >
> > > **Parameters**
> > > > **argument_id** – The argument id
> > >
> > > **Returns**
> > > > The detailed description
>
> **get_arguments**()
>
> > Returns a list of the arguments of the Property Model.
> >
> > ---
> >
> > **Note:** The arguments are the 'UI-panel components' defined in the Property Model interface method `provide_ui_panel_components()`. They have the same id as specified in the Property Model. The naming is different because there is no UI present.
> >
> > ---

> **Returns**
> > The ids of the available arguments

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_dynamic_arguments()

Returns a list of the dynamic arguments of the Property Model.

> **Note:** Dynamic arguments are "extra" arguments created by pressing the "plus" button that can occur next to the UI-panel for some models, when running the Property Model from within Thermo-Calc. You can use them also from the API by *invoke_dynamic_argument()*.

> **Returns**
> > The ids of the available dynamic arguments

get_model_description()

Returns the description text of the current model.

> **Returns**
> > the description

get_model_parameter_value(*model_parameter_id*)

Returns the current value of an optimizable model parameter. The id can be obtained with *get_model_parameters()*.

> **Parameters**
> > **model_parameter_id** – The model parameter id
>
> **Returns**
> > The current value [unit according to the parameter meaning]

get_model_parameters()

Returns a list of the optimizable model parameters.

> **Note:** The model parameters are an optional set of variables that can be used within the Property Model. Typically they are used to provide the possibility to inject parameter values during an optimization into the model. This allows the dynamic development of Property Models that need to be fitted to experimental data. The model parameters are controlled with the Property Model interface methods *provide_model_parameters* and *set_model_parameter*.

> **Returns**
> > The ids of the optimizable model parameters

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
> > The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

invoke_dynamic_argument(*argument_id*)

> Increases the number of instances of this dynamic argument by one, the argument will have an id such as *argument_1*, *argument_2*, … if the dynamic argument is called *argument*.

Note: You can obtain all available dynamic arguments by using *get_dynamic_arguments()*.

> **Parameters**
> > **argument_id** – argument_id: The argument id
>
> **Returns**
> > This *PropertyModelCalculation* object

remove_all_conditions()

> Removes all set classic POLY conditions.

Note: This does not affect the compositions set by *set_composition()*.

> **Returns**
> > This *PropertyModelCalculation* object

remove_all_poly_commands()

> This function has no effect and will be removed in 2024b. If you use it, please remove from your code.

> **Returns**
> > This *PropertyModelCalculation* object

remove_dependent_element()

> Removes a manually set dependent element. This method does not affect the automatic choice of the dependent element if *set_composition()* is used.

> **Returns**
> > This *PropertyModelCalculation* object

set_argument(*argument*, *value*)

> Sets the specified model argument to the specified value. The id can be obtained with *get_arguments()*.

> **Parameters**
> > - **argument** – The argument id
> > - **value** – The value [unit according to the argument meaning]
>
> **Returns**
> > This *PropertyModelCalculation* object

---

set_composition(*element_name*, *value*)

Sets the composition of a element. The unit for the composition can be changed using *set_composition_unit()*.

Default: Mole percent (CompositionUnit.MOLE_PERCENT)

> **Parameters**
> - **element_name** – The element
> - **value** – The composition value [composition unit defined for the calculation]
>
> **Returns**
> This *PropertyModelCalculation* object

set_composition_unit(*unit_enum*)

Sets the composition unit.

Default: Mole percent (CompositionUnit.MOLE_PERCENT).

> **Parameters**
> **unit_enum** – The new composition unit
>
> **Returns**
> This *PropertyModelCalculation* object

set_condition(*classic_condition*, *value*)

Adds a classic POLY condition. If that method is used, all conditions need to be specified in such a way. If this method is used, it is necessary to set the dependent element manually using *set_dependent_element()*.

Default if not specified: pressure P = 1e5 Pa, system size N = 1, Temperature T = 1000 K

Note: It should not be necessary for most users to use this method, try to use *set_composition()* instead.

> **Warning:** It is not possible to mix POLY-commands and compositions using *set_composition()*.

> **Warning:** As this method runs raw POLY-commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten parenthesis, . . .).

> **Parameters**
> - **classic_condition** – The classic POLY condition (for example: *X(CR)*)
> - **value** – The value of the condition
>
> **Returns**
> This *PropertyModelCalculation* object

set_dependent_element(*dependent_element_name*)

Sets the dependent element manually.

Note: It should not be necessary for most users to use this method. Setting the dependent element manually is only necessary and allowed if *set_condition()* is used.

>   **Parameters**
>   >   **dependent_element_name** – The name of the dependent element
>
>   **Returns**
>   >   This *PropertyModelCalculation* object

set_model_parameter(*model_parameter_id*, *value*)

>   Resets an optimizable model parameter. The id can be obtained with *get_model_parameters()*.
>
>   **Parameters**
>
>   - **model_parameter_id** – The model parameter id
>
>   - **value** – The new value of the parameter
>
>   **Returns**
>   >   This *PropertyModelCalculation* object

set_temperature(*temperature*)

>   Sets the temperature.
>
>   **Default**: 1000 K
>
>   **Parameters**
>   >   **temperature** – The temperature [K]
>
>   **Returns**
>   >   This *PropertyModelCalculation* object

with_system_modifications(*system_modifications*)

>   Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

>   **Parameters**
>   >   **system_modifications** – The system modification to be performed
>
>   **Returns**
>   >   This *PropertyModelCalculation* object

class +tc_toolbox.+propertymodel.**PropertyModelResult**(*back*)

>   The result of a Property Model calculation.
>
>   **PropertyModelResult**(*back*)
>   >   Call base constructor: `tc_toolbox.AbstractResult`.
>
>   **get_result_quantities**()
>   >   Returns a list of the available result quantities defined in the Property Model.
>   >
>   >   **Returns**
>   >   >   The ids of the defined result quantities

get_result_quantity_description(*result_quantity_id*)

> Returns the detailed description of the result quantity. The id can be obtained by *get_result_quantities()*.
>
> > **Parameters**
> > > **result_quantity_id** – The result quantity id
> >
> > **Returns**
> > > The detailed description

get_single_equilibrium_result(*result_quantity_id*)

> Returns a result quantity value. The available result quantities can be obtained by *get_result_quantities()*.
>
> > **Parameters**
> > > **result_quantity_id** – The id of the result quantity.
> >
> > **Returns**
> > > The requested value [unit depending on the quantity], if the result is a SingleEquilibriumResult, is returned.

get_value_of(*result_quantity_id*)

> Returns a result quantity value. The available result quantities can be obtained by *get_result_quantities()*.
>
> > **Parameters**
> > > **result_quantity_id** – The id of the result quantity
> >
> > **Returns**
> > > The requested value [unit depending on the quantity]. If the result is parameterized, parameter-value pairs are returned.

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

> Saves the result to disk. The result can later be loaded using +tc_toolbox.server.SetUp. load_result_from_disk().

> **Note:** The *result data* is represented by a whole folder possibly containing multiple files.

> > **Parameters**
> > > **path** – The path to the result folder, can be relative or absolute.
> >
> > **Returns**
> > > This *PropertyModelResult* object

4.1.8 Package "material_to_material"

class +tc_toolbox.+material_to_material.**AbstractConstantCondition**

 The abstract base class for all constant conditions.

class +tc_toolbox.+material_to_material.**AbstractMaterialToMaterialCalculationAxis**

 The abstract base class of all calculation axis.

class +tc_toolbox.+material_to_material.**ConstantCondition**

 A constant condition.

 static fraction_of_material_b(*fraction_of_material_b*)

  Creates a constant fraction of material B condition object.

  Note: The unit depends on the composition unit setting in the calculator object.

  Parameters

   fraction_of_material_b – The fraction of material B [weight-fraction or mole-fraction]

  Returns

   The condition object

 static temperature(*temperature*)

  Creates a constant temperature condition object.

  Parameters

   temperature – The temperature [K]

  Returns

   The condition object

class +tc_toolbox.+material_to_material.**FractionOfMaterialBAxis**(*from_fraction*, *to_fraction*, *start_fraction*)

 A fraction of material B axis.

 FractionOfMaterialBAxis(*from_fraction*, *to_fraction*, *start_fraction*)

  Creates a fraction of material B axis object.

  Note: The unit depends on the composition unit setting in the calculator.

  Parameters

   • **from_fraction** – The left axis limit [weight-fraction or mole-fraction]

   • **to_fraction** – The right axis limit [weight-fraction or mole-fraction]

   • **start_fraction** – The start fraction of the calculation [weight-fraction or mole-fraction]

 static fraction_of_material_b(*from_fraction*, *to_fraction*, *start_fraction*)

  Creates a fraction of material B axis object.

  Note: The unit depends on the composition unit setting in the calculator.

> **Parameters**
>> - **from_fraction** – The left axis limit [weight-fraction or mole-fraction]
>> - **to_fraction** – The right axis limit [weight-fraction or mole-fraction]
>> - **start_fraction** – The start fraction of the calculation [weight-fraction or mole-fraction]
>
> **Returns**
>> A new *FractionOfMaterialBAxis* axis object

static temperature(*from_temperature*, *to_temperature*, *start_temperature*)

> Creates a temperature calculation axis object.
>
> **Parameters**
>> - **from_temperature** – The left axis limit [K]
>> - **to_temperature** – The right axis limit [K]
>> - **start_temperature** – The start temperature of the calculation [K]
>
> **Returns**
>> A new *TemperatureAxis* condition object

class +tc_toolbox.+material_to_material.**FractionOfMaterialBCondition**(*fraction_of_material_b*)

> A constant fraction of material B condition.
>
> **FractionOfMaterialBCondition**(*fraction_of_material_b*)
>
>> Creates a constant fraction of material B condition object.
>>
>> ---
>> **Note:** The unit depends on the composition unit setting in the calculator.
>> ---
>>
>> **Parameters**
>>> **fraction_of_material_b** – The fraction of material B [weight-fraction or mole-fraction]
>
> **static fraction_of_material_b**(*fraction_of_material_b*)
>
>> Creates a constant fraction of material B condition object.
>>
>> ---
>> **Note:** The unit depends on the composition unit setting in the calculator object.
>> ---
>>
>> **Parameters**
>>> **fraction_of_material_b** – The fraction of material B [weight-fraction or mole-fraction]
>>
>> **Returns**
>>> The condition object
>
> **static temperature**(*temperature*)
>
>> Creates a constant temperature condition object.
>>
>> **Parameters**
>>> **temperature** – The temperature [K]
>>
>> **Returns**
>>> The condition object

class +tc_toolbox.+material_to_material.**MaterialToMaterialCalculationAxis**

> A calculation axis.
>
> **static fraction_of_material_b**(*from_fraction*, *to_fraction*, *start_fraction*)
>
> > Creates a fraction of material B axis object.
> >
> > ---
> > **Note:** The unit depends on the composition unit setting in the calculator.
> > ---
> >
> > **Parameters**
> >
> > - **from_fraction** – The left axis limit [weight-fraction or mole-fraction]
> > - **to_fraction** – The right axis limit [weight-fraction or mole-fraction]
> > - **start_fraction** – The start fraction of the calculation [weight-fraction or mole-fraction]
> >
> > **Returns**
> > > A new *FractionOfMaterialBAxis* axis object
>
> **static temperature**(*from_temperature*, *to_temperature*, *start_temperature*)
>
> > Creates a temperature calculation axis object.
> >
> > **Parameters**
> >
> > - **from_temperature** – The left axis limit [K]
> > - **to_temperature** – The right axis limit [K]
> > - **start_temperature** – The start temperature of the calculation [K]
> >
> > **Returns**
> > > A new *TemperatureAxis* condition object

class +tc_toolbox.+material_to_material.**MaterialToMaterialCalculationContainer**(*back*)

> Provides access to the calculation objects for all Material to Material calculations.
>
> These are specialised calculations for mixtures of two materials A and B. Otherwise they behave identical to the corresponding regular single equilibrium, property diagram and phase diagram calculations.
>
> **MaterialToMaterialCalculationContainer**(*back*)
>
> > Constructs an instance of *MaterialToMaterialCalculationContainer*.
>
> **with_phase_diagram_calculation**(*default_conditions*, *components*)
>
> > Creates a Material to Material phase diagram (map) calculation.
> >
> > **Parameters**
> >
> > - **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*
> > - **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements.* If this option is used, **all elements** of the system need to be replaced by a component.
> >
> > **Returns**
> > > A new *MaterialToMaterialPhaseDiagramCalculation* object
>
> **with_property_diagram_calculation**(*default_conditions*, *components*)
>
> > Creates a Material to Material property diagram (step) calculation.
> >
> > **Parameters**

- **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*

- **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.

Returns

A new *MaterialToMaterialPropertyDiagramCalculation* object

with_single_equilibrium_calculation(*default_conditions*, *components*)

Creates a Material to Material single equilibrium calculation.

Parameters

- **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*

- **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.

Returns

A new *MaterialToMaterialSingleEquilibriumCalculation* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialPhaseDiagramCalculation**(*back*)

Configuration for a Material to Material phase diagram calculation.

Note: Specify the conditions, the calculation is performed with *calculate()*.

MaterialToMaterialPhaseDiagramCalculation(*back*)

Call base constructor: tc_toolbox.step_or_map_diagrams.AbstractPhaseDiagramCalculation.

add_initial_equilibrium(*initial_equilibrium*)

Add initial equilibrium start points from which a phase diagram is calculated.

Scans along the axis variables and generates start points when the scan procedure crosses a phase boundary.

It may take a little longer to execute than using the minimum number of start points, as some lines may be calculated more than once. But the core remembers all node points and subsequently stops calculations along a line when it finds a known node point.

It is also possible to create a sequence of start points from one initial equilibria.

Parameters

initial_equilibrium – The initial equilibrium

Returns

This *MaterialToMaterialPhaseDiagramCalculation* object

calculate(*keep_previous_results*, *timeout_in_minutes*)

Performs the phase diagram calculation.

Warning: If you use *keep_previous_results=True*, you must not use another calculator or even get results in between the calculations using *calculate()*. Then the previous results will actually be lost.

Parameters

- **keep_previous_results** – If *True*, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the mapping fails at a certain condition.

- **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

> **Returns**
>
> > A new *MaterialToMaterialPhaseDiagramResult* object which later can be used to get specific values from the calculated result.

disable_global_minimization()

Disables global minimization.

> **Default**: Enabled
>
> > **Returns**
> >
> > > This *MaterialToMaterialPhaseDiagramCalculation* object

dont_keep_default_equilibria()

Do not keep the initial equilibria added by default.

This is only relevant in combination with *add_initial_equilibrium()*.

This is the default behavior.

> **Returns**
>
> > This *MaterialToMaterialPhaseDiagramCalculation* object

enable_global_minimization()

Enables global minimization.

> **Default**: Enabled
>
> > **Returns**
> >
> > > This *MaterialToMaterialPhaseDiagramCalculation* object

get_components()

Returns the names of the components in the system (including all components auto-selected by the database(s)).

> **Returns**
>
> > The component names

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
>
> > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

> **Returns**
>> Gibbs energy addition to G per mole formula unit.

get_system_data()

> Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as *.tdb*-file.

>> **Returns**
>>> The system data

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

keep_default_equilibria()

> Keep the initial equilibria added by default. This is only relevant in combination with `add_initial_equilibrium()`.

> Default behavior is to not keep default equilibria.

>> **Returns**
>>> This `MaterialToMaterialPhaseDiagramCalculation` object

remove_all_initial_equilibria()

> Removes all previously added initial equilibria.

>> **Returns**
>>> This `MaterialToMaterialPhaseDiagramCalculation` object

run_poly_command(*command*)

> Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

>> **Parameters**
>>> **command** – The Thermo-Calc Console Mode command

>> **Returns**
>>> This `MaterialToMaterialPhaseDiagramCalculation` object

set_activities(*activities*)

> Sets the constant activity conditions.

> **Note:** The activity conditions are identical for both materials.

Parameters

activities – The constant activities

Returns

This *MaterialToMaterialPhaseDiagramCalculation* object

set_composition_unit(*unit*)

Sets the composition unit of **both materials A and B**.

Default: Weight percent

Parameters

unit – The composition unit of both materials A and B

Returns

This *MaterialToMaterialPhaseDiagramCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

Parameters

- **phase** – Specify the name of the (stoichiometric or solution) phase with the addition

- **gibbs_energy** – Addition to G per mole formula unit

Returns

This *MaterialToMaterialPhaseDiagramCalculation* object

set_material_a(*composition*, *dependent_component*)

Sets the composition of the material A.

The unit is set with *set_composition_unit()*.

Tip: The material can also have constant activity conditions, they are set in *set_activities()*.

Parameters

- **composition** – The composition of the material A

- **dependent_component** – The dependent component of the material A

Returns

This *MaterialToMaterialPhaseDiagramCalculation* object

set_material_b(*composition*, *dependent_component*)

Sets the composition of the material B.

The unit is set with *set_composition_unit()*.

Tip: The material can also have constant activity conditions, they are set in *set_activities()*.

Parameters

- **composition** – The composition of the material B

- **dependent_component** – The dependent component of the material B

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

> **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.

> **Parameters**

- **phase** – The phase name or *ALL_PHASES* for all phases

- **amount** – The phase fraction (between 0.0 and 1.0)

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

> **Parameters**

- **phase** – The phase name

- **amount** – The fixed phase fraction (between 0.0 and 1.0)

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

> **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

set_pressure(*pressure*)

> Sets the pressure (i.e. the condition *P*).

Note: If the flag *default_conditions=True* has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

> **Parameters**
> > **pressure** – The pressure [Pa]

Returns
> This *MaterialToMaterialPhaseDiagramCalculation* object

set_system_size(*system_size*)

> Sets the system size (i.e. the condition 'N', the number of moles).

> ---
> **Note:** If the flag *default_conditions=True* has been set during the creation of the calculator, the system size is set to 1.0 moles by default.
> ---

> **Parameters**
> > **system_size** – The system size [mole]

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

with_first_axis(*axis*)

> Sets the first axis (either temperature of fraction of material B). This calculation type requires that both temperature and fraction of material B axis are set.

> **Parameters**
> > **axis** – The axis

> **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

with_options(*options*)

> Sets the simulation options.

> **Parameters**
> > **options** – The simulation options

> **Returns**
> > This PhaseDiagramCalculation object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

> For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

> If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

> **Parameters**

- **component** – The name of the element must be given.

- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.

- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

- **pressure** – The Pressure (in Pa) for the reference state.

> **Returns**
> This *MaterialToMaterialPhaseDiagramCalculation* object

with_second_axis(*axis*)

> Sets the second axis (either temperature of fraction of material B). This calculation type requires that both temperature and fraction of material B axis are set.
>
> > **Parameters**
> > **axis** – The axis
> >
> > **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

with_system_modifications(*system_modifications*)

> Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).
>
> ---
>
> **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.
>
> ---
>
> > **Parameters**
> > **system_modifications** – The system modification to be performed
> >
> > **Returns**
> > This *MaterialToMaterialPhaseDiagramCalculation* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialPhaseDiagramResult**(*back*)

> Result of a Material to Material phase diagram calculation, it can be evaluated using quantities or Console Mode syntax.

MaterialToMaterialPhaseDiagramResult(*back*)

> Call base constructor: `tc_toolbox.step_or_map_diagrams.PhaseDiagramResult`.

add_coordinate_for_phase_label(*x*, *y*)

> Sets a coordinate in the result plot for which the stable phases will be evaluated and provided in the result data object. This can be used to plot the phases of a region into the phase diagram or just to programmatically evaluate the phases in certain regions.
>
> ---
>
> **Warning:** This method takes coordinates of the **plot** axes and not of the calculation axis.
>
> ---
>
> > **Parameters**
> >
> > - **x** – The coordinate of the first **plot** axis ("x-axis") [unit of the **plot** axis]
> >
> > - **y** – The coordinate of the second **plot** axis ("y-axis") [unit of the **plot** axis]

Returns

This *MaterialToMaterialPhaseDiagramResult* object

get_values_grouped_by_quantity_of(*x_quantity*, *y_quantity*)

Returns x-y-line data grouped by the multiple datasets of the specified quantities (for example in dependency of components). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

Note: The different datasets will contain *NaN*-values between different subsections and are not sorted (because they are unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), *MATERIAL_B_FRACTION*, or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), *MATERIAL_B_FRACTION*, or even a function (for example *'CP=HM.T'*)

Returns

The phase diagram data

get_values_grouped_by_stable_phases_of(*x_quantity*, *y_quantity*)

Returns x-y-line data grouped by the sets of "stable phases" (for example "LIQUID" or "LIQUID + FCC_A1"). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`. Usually the result data represents the phase diagram.

Note: The different datasets will contain *NaN*-values between different subsections and are not sorted (because they are unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), *MATERIAL_B_FRACTION*, or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), *MATERIAL_B_FRACTION*, or even a function (for example *'CP=HM.T'*)

Returns

The phase diagram data

---

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

remove_phase_labels()

> Erases all added coordinates for phase labels.
>
> > **Returns**
> >
> > > This *MaterialToMaterialPhaseDiagramResult* object

save_to_disk(*path*)

> Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`
>
> > **Parameters**
> >
> > > **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
> >
> > **Returns**
> >
> > > this *MaterialToMaterialPhaseDiagramResult* object

set_phase_name_style(*phase_name_style_enum*)

> Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, . . .).
>
> **Default**: PhaseNameStyle.NONE
>
> > **Parameters**
> >
> > > **phase_name_style_enum** – The phase name style
> >
> > **Returns**
> >
> > > This *MaterialToMaterialPhaseDiagramResult* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialPropertyDiagramCalculation**(*back*)

> Configuration for a Material to Material property diagram calculation.

Note: Specify the conditions and possibly other settings, the calculation is performed with *calculate()*.

MaterialToMaterialPropertyDiagramCalculation(*back*)

> Call base constructor: tc_toolbox.step_or_map_diagrams.AbstractPropertyDiagramCalculation.

calculate(*keep_previous_results*, *timeout_in_minutes*)

> Performs the Material to Material property diagram calculation.

Warning: If you use *keep_previous_results=True*, you must not use another calculator or even get results in between the calculations using *calculate()*. Then the previous results will actually be lost.

> > **Parameters**
> >
> > > • **keep_previous_results** – If *True*, results from any previous call to this method are appended. This can be used to combine calculations with multiple start points if the stepping fails at a certain condition.
> > >
> > > • **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

> **Returns**
>> A new *MaterialToMaterialPropertyDiagramResult* object which later can be used to
>> get specific values from the calculated result

disable_global_minimization()

Disables global minimization.

Default: Enabled

> **Returns**
>> This *MaterialToMaterialPropertyDiagramCalculation* object

disable_step_separate_phases()

Disables *step separate phases*. This is the **default** setting.

> **Returns**
>> This *MaterialToMaterialPropertyDiagramCalculation* object

enable_global_minimization()

Enables global minimization.

Default: Enabled

> **Returns**
>> This *MaterialToMaterialPropertyDiagramCalculation* object

enable_step_separate_phases()

Enables *step separate phases*.

Default: By default separate phase stepping is *disabled*

Note: This is an advanced option, it is used mostly to calculate how the Gibbs energy for a number of phases varies for different compositions. This is particularly useful to calculate Gibbs energies for complex phases with miscibility gaps and for an ordered phase that is never disordered (e.g. SIGMA-phase, G-phase, MU-phase, etc.).

> **Returns**
>> This *MaterialToMaterialPropertyDiagramCalculation* object

get_components()

Returns the names of the components in the system (including all components auto-selected by the database(s)).

> **Returns**
>> The component names

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

Warning: The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

---

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
> > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>
> **Returns**
> > Gibbs energy addition to G per mole formula unit.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

> **Returns**
> > The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
> > **command** – The Thermo-Calc Console Mode command
>
> **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_activities(*activities*)

Sets the constant activity conditions.

Note: The activity conditions are identical for both materials.

> **Parameters**
> > **activities** – The constant activities

> **Returns**
>> This *MaterialToMaterialPropertyDiagramCalculation* object

set_composition_unit(*unit*)

> Sets the composition unit of **both materials A and B**.
>
> **Default**: Weight percent
>
>> **Parameters**
>>> **unit** – The composition unit of both materials A and B
>>
>> **Returns**
>>> This *MaterialToMaterialPropertyDiagramCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

> Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
>> **Parameters**
>>> - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>>> - **gibbs_energy** – Addition to G per mole formula unit
>>
>> **Returns**
>>> This *MaterialToMaterialPropertyDiagramCalculation* object

set_material_a(*composition*, *dependent_component*)

> Sets the composition of the material A.
>
> The unit is set with *set_composition_unit()*.

> **Tip:** The material can also have constant activity conditions, they are set in *set_activities()*.

>> **Parameters**
>>> - **composition** – The composition of the material A
>>> - **dependent_component** – The dependent component of the material A
>>
>> **Returns**
>>> This *MaterialToMaterialPropertyDiagramCalculation* object

set_material_b(*composition*, *dependent_component*)

> Sets the composition of the material B.
>
> The unit is set with *set_composition_unit()*.

> **Tip:** The material can also have constant activity conditions, they are set in *set_activities()*.

>> **Parameters**
>>> - **composition** – The composition of the material B
>>> - **dependent_component** – The dependent component of the material B

> **Returns**
> This *MaterialToMaterialPropertyDiagramCalculation* object

set_phase_to_dormant(*phase*)

> Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.
>
> > **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_phase_to_entered(*phase*, *amount*)

> Sets the phase to the status ENTERED, that is the default state.
>
> > **Parameters**
> >
> > - **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > - **amount** – The phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_phase_to_fixed(*phase*, *amount*)

> Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.
>
> > **Parameters**
> >
> > - **phase** – The phase name
> >
> > - **amount** – The fixed phase fraction (between 0.0 and 1.0)
> >
> > **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_phase_to_suspended(*phase*)

> Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.
>
> > **Parameters**
> > **phase** – The phase name or *ALL_PHASES* for all phases
> >
> > **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_pressure(*pressure*)

> Sets the pressure (i.e. the condition *P*).

Note: If the flag *default_conditions=True* has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

> > **Parameters**
> > **pressure** – The pressure [Pa]
> >
> > **Returns**
> > This *MaterialToMaterialPropertyDiagramCalculation* object

set_system_size(*system_size*)

> Sets the system size (i.e. the condition 'N', the number of moles).

> > **Note:** If the flag *default_conditions=True* has been set during the creation of the calculator, the system size is set to 1.0 moles by default.

> > **Parameters**
> > > **system_size** – The system size [mole]

> > **Returns**
> > > This *MaterialToMaterialPropertyDiagramCalculation* object

with_axis(*axis*)

> Sets the axis (either temperature of fraction of material B). This calculation type requires that either temperature or fraction of material B is set as a constant condition - the other one is set as an axis.

> > **Parameters**
> > > **axis** – The axis

> > **Returns**
> > > This *MaterialToMaterialPropertyDiagramCalculation* object

with_constant_condition(*condition*)

> Sets the constant condition (either temperature of fraction of material B). This calculation type requires that either temperature or fraction of material B is set as a constant condition - the other one is set as an axis.

> > **Parameters**
> > > **condition** – The condition

> > **Returns**
> > > This *MaterialToMaterialPropertyDiagramCalculation* object

with_options(*options*)

> Sets the simulation options.

> > **Parameters**
> > > **options** – The simulation options

> > **Returns**
> > > This *MaterialToMaterialPropertyDiagramCalculation* object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.

> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.

> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.

For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.

If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.

Parameters

- **component** – The name of the element must be given.

- **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.

- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

- **pressure** – The Pressure (in Pa) for the reference state.

Returns
This *MaterialToMaterialPropertyDiagramCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

Parameters
system_modifications – The system modification to be performed

Returns
This *MaterialToMaterialPropertyDiagramCalculation* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialPropertyDiagramResult**(*back*)

Result of a Material to Material property diagram. It can be used to query for specific values.

MaterialToMaterialPropertyDiagramResult(*back*)

Call base constructor: `tc_toolbox.step_or_map_diagrams.PropertyDiagramResult`.

get_values_grouped_by_quantity_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the multiple datasets of the specified quantities (typically the phases). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), *MATERIAL_B_FRACTION*, or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), *MATERIAL_B_FRACTION*, or even a function (for example *'CP=HM.T'*)

- **sort_and_merge** – If *True*, the data is sorted and merged into as few subsections as possible (divided by *NaN*)

Returns
    Containing the datasets with the quantities as their keys

get_values_grouped_by_stable_phases_of(*x_quantity*, *y_quantity*, *sort_and_merge*)

Returns x-y-line data grouped by the sets of "stable phases" (for example "LIQUID" or "LIQUID + FCC_A1"). The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: The different datasets might contain *NaN*-values between different subsections and different lines of an ambiguous dataset. They might not be sorted **even if the flag `sort_and_merge` has been set** (because they might be unsortable due to their nature).

Note: Its possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function`, or by using an expression that contains '='.

Parameters

- **x_quantity** – The first quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*), *MATERIAL_B_FRACTION*, or even a function (for example *'f=T*1.01'*)

- **y_quantity** – The second quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), *MATERIAL_B_FRACTION*, or even a function (for example *'CP=HM.T'*)

- **sort_and_merge** – If *True*, the data will be sorted and merged into as few subsections as possible (divided by *NaN*)

Returns
    Containing the datasets with the quantities as their keys

get_values_of(*x_quantity*, *y_quantity*)

Returns sorted x-y-line data without any separation. Use *get_values_grouped_by_quantity_of()* or *get_values_grouped_by_stable_phases_of()* instead if you need such a separation. The available quantities can be found in the documentation of the factory class `ThermodynamicQuantity`.

Note: This method will always return sorted data without any *NaN*-values. If it is unsortable that might give data that is hard to interpret. In such a case you need to choose the quantity in another way or use one of the other methods. One example of this is to use quantities with *All*-markers, for example *MassFractionOfAComponent("All")*.

Note: Its possible to use functions as axis variables, either by using `ThermodynamicQuantity.user_defined_function()`, or by using an expression that contains '='.

> **Parameters**
>
> - **x_quantity** – The first thermodynamic quantity ("x-axis"), Console Mode syntax strings can be used as an alternative (for example *'T'*, *MATERIAL_B_FRACTION*, or even a function (for example *'f=T*1.01'*).
> - **y_quantity** – The second thermodynamic quantity ("y-axis"), Console Mode syntax strings can be used as an alternative (for example *'NV'*), *MATERIAL_B_FRACTION*, or even a function (for example *'CP=HM.T'*)
>
> **Returns**
>
> A tuple containing the x- and y-data in lists

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

save_to_disk(*path*)

> Saves the result to disc. Note that a result is a folder, containing potentially many files. The result can later be loaded with `load_result_from_disk()`
>
> **Parameters**
>
> **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
>
> **Returns**
>
> this *MaterialToMaterialPropertyDiagramResult* object

set_phase_name_style(*phase_name_style_enum*)

> Sets the style of the phase name labels that will be used in the result data object (constitution description, ordering description, …).
>
> **Default**: PhaseNameStyle.NONE
>
> **Parameters**
>
> **phase_name_style_enum** – The phase name style
>
> **Returns**
>
> This *MaterialToMaterialPropertyDiagramResult* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialSingleEquilibriumCalculation**(*back*)

Configuration for a Material to Material single fraction of B calculation.

Note: Specify the conditions and possibly other settings, the calculation is performed with *calculate()*.

MaterialToMaterialSingleEquilibriumCalculation(*back*)

> Call base constructor: tc_toolbox.single_equilibrium.AbstractSingleEquilibriumCalculation.

calculate(*timeout_in_minutes*)

> Performs the material to material calculation.

Note: The calculation result is **no** temporary result object.

Parameters

> **timeout_in_minutes** – Used to prevent the calculation from running longer than what is wanted, or from hanging. If the calculation runs longer than timeout_in_minutes, a UnrecoverableCalculationException will be thrown, the current TCPython-block will be unusable and a new TCPython block must be created for further calculations.

Returns

> A new *MaterialToMaterialSingleEquilibriumResult* object which can be used to get specific values from the calculated result. It is undefined behavior to use that object after the state of the calculation has been changed.

disable_global_minimization()

Turns the global minimization completely off.

> **Returns**
>
> > This *MaterialToMaterialSingleEquilibriumCalculation* object

enable_global_minimization()

Turns the global minimization on (using the default settings).

> **Returns**
>
> > This *MaterialToMaterialSingleEquilibriumCalculation* object

get_components()

Returns a list of components in the system (including all components auto-selected by the database(s)).

> **Returns**
>
> > The components

get_configuration_as_string()

Returns detailed information about the current state of the calculation object.

> **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.

get_gibbs_energy_addition_for(*phase*)

Used to get the additional energy term (always being a constant) of a given phase. The value given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.

It is not composition-, temperature- or pressure-dependent.

> **Parameters**
>
> > **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>
> **Returns**
>
> > Gibbs energy addition to G per mole formula unit.

get_system_data()

Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.

> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

Returns
The system data

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

Parameters
command – The Thermo-Calc Console Mode command

Returns
This *MaterialToMaterialSingleEquilibriumCalculation* object

set_activities(*activities*)

Sets the constant activity conditions.

Note: The activity conditions are identical for both materials.

Parameters
activities – The constant activities

Returns
This *MaterialToMaterialSingleEquilibriumCalculation* object

set_component_to_entered(*component*)

Sets the specified component to the status ENTERED, that is the default state.

Parameters
component – The component name or *ALL_COMPONENTS*

Returns
This *MaterialToMaterialSingleEquilibriumCalculation* object

set_component_to_suspended(*component*, *reset_conditions*)

Sets the specified component to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters

- **reset_conditions** – if 'True' also remove composition conditions for the component if they are defined

- **component** – The component name or *ALL_COMPONENTS*

> **Returns**
>> This *MaterialToMaterialSingleEquilibriumCalculation* object

set_composition_unit(*unit*)

> Sets the composition unit of **both materials A and B**.
>
> **Default**: Weight percent
>
>> **Parameters**
>>> **unit** – The composition unit of both materials A and B
>>
>> **Returns**
>>> This *MaterialToMaterialSingleEquilibriumCalculation* object

set_gibbs_energy_addition_for(*phase*, *gibbs_energy*)

> Used to specify the additional energy term (always being a constant) of a given phase. The value (*gibbs_energy*) given is added to the Gibbs energy of the (stoichiometric or solution) phase. It can represent a nucleation barrier, surface tension, elastic energy, etc.
>
> It is not composition-, temperature- or pressure-dependent.
>
>> **Parameters**
>>
>>> - **phase** – Specify the name of the (stoichiometric or solution) phase with the addition
>>>
>>> - **gibbs_energy** – Addition to G per mole formula unit
>>
>> **Returns**
>>> This *MaterialToMaterialSingleEquilibriumCalculation* object

set_material_a(*composition*, *dependent_component*)

> Sets the composition of the material A.
>
> The unit is set with *set_composition_unit()*.
>
> ---
>
> **Tip:** The material can also have constant activity conditions, they are set in *set_activities()*.
>
> ---
>
>> **Parameters**
>>
>>> - **composition** – The composition of the material A
>>>
>>> - **dependent_component** – The dependent component of the material A
>>
>> **Returns**
>>> This *MaterialToMaterialSingleEquilibriumCalculation* object

set_material_b(*composition*, *dependent_component*)

> Sets the composition of the material B.
>
> The unit is set with *set_composition_unit()*.
>
> ---
>
> **Tip:** The material can also have constant activity conditions, they are set in *set_activities()*.
>
> ---
>
>> **Parameters**
>>
>>> - **composition** – The composition of the material B
>>>
>>> - **dependent_component** – The dependent component of the material B

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_phase_to_dormant(*phase*)

Sets the phase to the status DORMANT, necessary for calculating the driving force to form the specified phase.

Parameters

phase – The phase name or *ALL_PHASES* for all phases

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_phase_to_entered(*phase*, *amount*)

Sets the phase to the status ENTERED, that is the default state.

Parameters

• **phase** – The phase name or *ALL_PHASES* for all phases

• **amount** – The phase fraction (between 0.0 and 1.0)

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_phase_to_fixed(*phase*, *amount*)

Sets the phase to the status FIXED, i.e. it is guaranteed to have the specified phase fraction after the calculation.

Parameters

• **phase** – The phase name

• **amount** – The fixed phase fraction (between 0.0 and 1.0)

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_phase_to_suspended(*phase*)

Sets the phase to the status SUSPENDED, i.e. it is ignored in the calculation.

Parameters

phase – The phase name or *ALL_PHASES* for all phases

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_pressure(*pressure*)

Sets the pressure (i.e. the condition *P*).

Note: If the flag *default_conditions=True* has been set during the creation of the calculator, the pressure is set to 1000 hPa by default.

Parameters

pressure – The pressure [Pa]

Returns

This *MaterialToMaterialSingleEquilibriumCalculation* object

set_system_size(*system_size*)

> Sets the system size (i.e. the condition 'N', the number of moles).

Note: If the flag *default_conditions=True* has been set during the creation of the calculator, the system size is set to 1.0 moles by default.

> **Parameters**
> > **system_size** – The system size [mole]
>
> **Returns**
> > This *MaterialToMaterialSingleEquilibriumCalculation* object

with_first_constant_condition(*condition*)

> Sets the first constant condition (either temperature of fraction of material B).
>
> **Parameters**
> > **condition** – The condition
>
> **Returns**
> > This *MaterialToMaterialSingleEquilibriumCalculation* object

with_options(*options*)

> Sets the simulation options.
>
> **Parameters**
> > **options** – The simulation options
>
> **Returns**
> > This SingleEquilibriumCalculation object

with_reference_state(*component*, *phase*, *temperature*, *pressure*)

> The reference state for a component is important when calculating activities, chemical potentials and enthalpies and is determined by the database being used. For each component the data must be referred to a selected phase, temperature and pressure, i.e. the reference state.
>
> All data in all phases where this component dissolves must use the same reference state. However, different databases can use different reference states for the same element/component. It is important to be careful when combining data obtained from different databases.
>
> By default, activities, chemical potentials and so forth are computed relative to the reference state used by the database. If the reference state in the database is not suitable for your purposes, use this command to set the reference state for a component using SER, i.e. the Stable Element Reference (which is usually set as default for a major component in alloys dominated by the component). In such cases, the temperature and pressure for the reference state is not needed.
>
> For a phase to be usable as a reference for a component, the component needs to have the same composition as an end member of the phase. The reference state is an end member of a phase. The selection of the end member associated with the reference state is only performed once this command is executed.
>
> If a component has the same composition as several end members of the chosen reference phase, then the end member that is selected at the specified temperature and pressure will have the lowest Gibbs energy.
>
> **Parameters**
> > - **component** – The name of the element must be given.
> > - **phase** – Name of a phase used as the new reference state. Or SER for the Stable Element Reference.

- **temperature** – The Temperature (in K) for the reference state. Or CURRENT_TEMPERATURE which means that the current temperature is used at the time of evaluation of the reference energy for the calculation.

- **pressure** – The Pressure (in Pa) for the reference state.

> **Returns**
> This *MaterialToMaterialSingleEquilibriumCalculation* object

with_second_constant_condition(*condition*)

Sets the second constant condition (either temperature of fraction of material B).

> **Parameters**
> **condition** – The condition

> **Returns**
> This *MaterialToMaterialSingleEquilibriumCalculation* object

with_system_modifications(*system_modifications*)

Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).

Note: This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

> **Parameters**
> **system_modifications** – The system modification to be performed

> **Returns**
> This *MaterialToMaterialSingleEquilibriumCalculation* object

class +tc_toolbox.+material_to_material.**MaterialToMaterialSingleEquilibriumResult**(*back*)

Result of a Material To Material calculation for a single fraction of material B, it can be evaluated using a quantity or Console Mode syntax.

MaterialToMaterialSingleEquilibriumResult(*back*)

Call base constructor: `tc_toolbox.single_equilibrium.SingleEquilibriumResult`.

change_pressure(*pressure*)

Change the pressure and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with higher performance. The properties are calculated at the new pressure using the phase amount, temperature and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.

> **Parameters**
> **pressure** – The pressure [Pa]

> **Returns**
> This *MaterialToMaterialSingleEquilibriumResult* object

change_temperature(*temperature*)

Change the temperature and re-evaluate the results from the equilibrium without minimizing Gibbs energy, i.e. with high performance. The properties are calculated at the new temperature using the phase amount, pressure and composition of phases from the initial equilibrium. Use *get_value_of()* to obtain them.

Note: This is typically used when calculating room temperature properties (e.g. density) for a material when it is assumed that the equilibrium phase amount and composition freeze-in at a higher temperature

---

during cooling.

> **Parameters**
> > **temperature** – The temperature [K]
>
> **Returns**
> > This *MaterialToMaterialSingleEquilibriumResult* object

get_components()

Returns the names of the components selected in the system (including any components auto-selected by the database(s)).

> **Returns**
> > The names of the selected components

get_conditions()

Returns the conditions.

> **Returns**
> > The selected conditions

get_phases()

Returns the phases present in the system due to its configuration. It also contains all phases that have been automatically added during the calculation, this is the difference to the method System. get_phases_in_system().

> **Returns**
> > The names of the phases in the system including automatically added phases

get_stable_phases()

Returns the stable phases (i.e. the phases present in the current equilibrium).

> **Returns**
> > The names of the stable phases

get_value_of(*quantity*)

Returns a value from a single equilibrium calculation.

> **Parameters**
> > **quantity** – The thermodynamic quantity to get the value of; a Console Mode syntax strings can be used as an alternative (for example "NPM(FCC_A1)")
>
> **Returns**
> > The requested value

invalidate()

Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

run_poly_command(*command*)

Runs a Thermo-Calc command from the Console Mode POLY module immediately in the engine. This affects only the state of the result object.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

> **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the
> program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
>> **command** – The Thermo-Calc Console Mode command
>
> **Returns**
>> This *MaterialToMaterialSingleEquilibriumResult* object

save_to_disk(*path*)

> Saves the result to disk. Note that the result is a folder, containing potentially many files. The result can
> later be loaded with `load_result_from_disk()`

> **Parameters**
>> **path** – the path to the folder you want the result to be saved in. It can be relative or absolute.
>
> **Returns**
>> this *MaterialToMaterialSingleEquilibriumResult* object

class +tc_toolbox.+material_to_material.**TemperatureAxis**(*from_temperature*, *to_temperature*,
                                                                  start_temperature)

> A temperature calculation axis.

> **TemperatureAxis**(*from_temperature*, *to_temperature*, *start_temperature*)

>> Creates a temperature calculation axis object.

>> **Parameters**

>>> - **from_temperature** – The left axis limit [K]
>>>
>>> - **to_temperature** – The right axis limit [K]
>>>
>>> - **start_temperature** – The start temperature of the calculation [K]

> **static fraction_of_material_b**(*from_fraction*, *to_fraction*, *start_fraction*)

>> Creates a fraction of material B axis object.

>> ---

>> **Note:** The unit depends on the composition unit setting in the calculator.

>> ---

>> **Parameters**

>>> - **from_fraction** – The left axis limit [weight-fraction or mole-fraction]
>>>
>>> - **to_fraction** – The right axis limit [weight-fraction or mole-fraction]
>>>
>>> - **start_fraction** – The start fraction of the calculation [weight-fraction or mole-fraction]

>> **Returns**
>>> A new *FractionOfMaterialBAxis* axis object

> **static temperature**(*from_temperature*, *to_temperature*, *start_temperature*)

>> Creates a temperature calculation axis object.

>> **Parameters**

>>> - **from_temperature** – The left axis limit [K]
>>>
>>> - **to_temperature** – The right axis limit [K]

- **start_temperature** – The start temperature of the calculation [K]

> **Returns**
> > A new *TemperatureAxis* condition object

class +tc_toolbox.+material_to_material.**TemperatureCondition**(*temperature*)

> A constant temperature condition.

> **TemperatureCondition**(*temperature*)
> > Creates a constant temperature condition object.
> >
> > > **Parameters**
> > > > **temperature** – The temperature [K]

> **static fraction_of_material_b**(*fraction_of_material_b*)
> > Creates a constant fraction of material B condition object.
> >
> > ---
> >
> > **Note:** The unit depends on the composition unit setting in the calculator object.
> >
> > ---
> >
> > > **Parameters**
> > > > **fraction_of_material_b** – The fraction of material B [weight-fraction or mole-fraction]
> > >
> > > **Returns**
> > > > The condition object

> **static temperature**(*temperature*)
> > Creates a constant temperature condition object.
> >
> > > **Parameters**
> > > > **temperature** – The temperature [K]
> > >
> > > **Returns**
> > > > The condition object

4.1.9 Package "process_metallurgy"

4.1.9.1 Package "base"

class +tc_toolbox.+process_metallurgy.+base.**AbstractAddition**

> The base class for representing an addition to an equilibrium calculation or process simulation.

> **get_composition**()
> > Returns the composition of the addition - without containing a dependent component.
> >
> > > **Returns**
> > > > The composition [in the unit provided by `getCompositionUnit()`]

> **get_composition_unit**()
> > Returns the composition unit used in this addition.
> >
> > > **Returns**
> > > > The composition unit

get_dependent_component()

    Returns the dependent component.

        Returns

            The dependent component or an empty string if no dependent component is defined

get_elements()

    Returns all elements of the addition.

        Returns

            The elements

get_id()

    Returns the unique ID of the addition.

        Returns

            The unique ID of the addition

get_temperature()

    Returns the temperature of the addition. This refers to the temperature before it is added to the process.

        Returns

            The temperature [K]

is_do_scale()

    Returns if the composition of the addition is being scaled to 100% / 1 or not.

        Returns

            If the composition is scaled

is_empty()

    Returns if the addition is "empty", i.e., has zero amount.

        Returns

            If the addition is empty

class +tc_toolbox.+process_metallurgy.+base.**ActivityReference**

    The reference for a slag activity calculation. The actual reference phase depends on the component for which the activity is request and can be obtained by using these methods on the result object: +tc_toolbox.process_metallurgy.process.ProcessSimulationResult. get_formula_for_activity_of_slag() or +tc_toolbox.process_metallurgy.equilibrium. EquilibriumResult.get_formula_for_activity_of_slag().

class +tc_toolbox.+process_metallurgy.+base.**PhaseGroup**

    The phase group, such a group is collecting all phases that belong to a certain type.

class +tc_toolbox.+process_metallurgy.+base.**ProcessDatabase**

    The database used for a Process Metallurgy calculation.

class +tc_toolbox.+process_metallurgy.+base.**ProcessMetallurgyOptions**

    The options for a process metallurgy calculation.

    ProcessMetallurgyOptions()

        The options for a process metallurgy calculation. Constructs an instance of *ProcessMetallurgyOptions*.

    disable_approximate_driving_force_for_metastable_phases()

        Disables the approximation of the driving force for metastable phases.

        Default: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
>> This *ProcessMetallurgyOptions* object

disable_control_step_size_during_minimization()

> Disables step size control during minimization (non-global).

> **Default**: Enabled

>> **Returns**
>>> This *ProcessMetallurgyOptions* object

disable_force_positive_definite_phase_hessian()

> Disables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

> **Default**: Enabled

>> **Returns**
>>> This *ProcessMetallurgyOptions* object

enable_approximate_driving_force_for_metastable_phases()

> Enables the approximation of the driving force for metastable phases.

> **Default**: Enabled

Note: When enabled, the metastable phases are included in all iterations. However, these may not have reached their most favorable composition and thus their driving forces may be only approximate.

> **Returns**
>> This *ProcessMetallurgyOptions* object

enable_control_step_size_during_minimization()

> Enables step size control during normal minimization (non-global).

> **Default**: Enabled

>> **Returns**
>>> This *ProcessMetallurgyOptions* object

enable_force_positive_definite_phase_hessian()

> Enables forcing of positive definite phase Hessian. This determines how the minimum of an equilibrium state in a normal minimization procedure (non-global) is reached. For details, search the Thermo-Calc documentation for "Hessian minimization".

> **Default**: Enabled

>> **Returns**
>>> This *ProcessMetallurgyOptions* object

set_global_minimization_max_grid_points(*max_grid_points*)

Sets the maximum number of grid points in global minimization. **Only applicable if global minimization is actually used**.

Default: 2000 points

> **Parameters**
> > **max_grid_points** – The maximum number of grid points
>
> **Returns**
> > This *ProcessMetallurgyOptions* object

set_max_no_of_iterations(*max_no_of_iterations*)

Sets the maximum number of iterations for the CALPHAD minimizer.

Default: max. 2000 iterations

Note: As some models give computation times of more than 1 CPU second/iteration, this number is also used to check the CPU time and the calculation stops if 500 CPU seconds/iterations are used.

> **Parameters**
> > **max_no_of_iterations** – The max. number of iterations
>
> **Returns**
> > This *ProcessMetallurgyOptions* object

set_process_minimization_policy(*minimization_policy*)

Sets the minimization policy for the process metallurgy calculation. It is possible to choose policies that try different methods if one method fails.

> **Parameters**
> > **minimization_policy** – The minimization policy to be used
>
> **Returns**
> > This *ProcessMetallurgyOptions* object

set_required_accuracy(*accuracy*)

Sets the required relative accuracy.

Default: 1.0E-6

Note: This is a relative accuracy, and the program requires that the relative difference in each variable must be lower than this value before it has converged. A larger value normally means fewer iterations but less accurate solutions. The value should be at least one order of magnitude larger than the machine precision.

> **Parameters**
> > **accuracy** – The required relative accuracy
>
> **Returns**
> > This *ProcessMetallurgyOptions* object

set_smallest_fraction(*smallest_fraction*)

Sets the smallest fraction for constituents that are unstable.

It is normally only in the gas phase that you can find such low fractions.

The **default value** for the smallest site-fractions is 1E-16 for all phases except for IDEAL phase with one sublattice site (such as the GAS mixture phase in many databases) for which the default value is always as 1E-30.

> **Parameters**
> > **smallest_fraction** – The smallest fraction for constituents that are unstable

> **Returns**
> > This *ProcessMetallurgyOptions* object

class +tc_toolbox.+process_metallurgy.+base.**ProcessMinimizationPolicy**

The policy for the CALPHAD minimization routine used in a calculation.

Note: This affects the runtime and stability of a calculation. Global minimization is more stable but more time-consuming. Local minimization is much faster but can miss new phases coming up. Global test is a compromise between both approaches.

---

class +tc_toolbox.+process_metallurgy.+base.**SlagProperty**

The slag property, different definitions are available. The actual definition of a certain slag property for the current system can be obtained using these methods on the result object: +tc_toolbox.process_metallurgy. process.ProcessSimulationResult.getFormulaForSlagProperty() or +tc_toolbox. process_metallurgy.equilibrium.EquilibriumResult.getFormulaForSlagProperty().

Note: If not all components required by the definition of slag property are available in a given system, the slag property will return *NaN*.

---

class +tc_toolbox.+process_metallurgy.+base.**SlagType**

The type of slag considered for a slag property calculation.

4.1.9.2 Package "equilibrium"

class +tc_toolbox.+process_metallurgy.+equilibrium.**AbstractEquilibriumAddition**

The base class for representing an addition to an equilibrium calculation.

get_composition()

Returns the composition of the addition - without containing a dependent component.

> **Returns**
> > The composition [in the unit provided by getCompositionUnit()]

get_composition_unit()

Returns the composition unit used in this addition.

> **Returns**
> > The composition unit

get_dependent_component()

Returns the dependent component.

> **Returns**
> > The dependent component or an empty string if no dependent component is defined

---

get_elements()

> Returns all elements of the addition.
>
> > **Returns**
> >
> > The elements

get_id()

> Returns the unique ID of the addition.
>
> > **Returns**
> >
> > The unique ID of the addition

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.
>
> > **Returns**
> >
> > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.
>
> > **Returns**
> >
> > If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.
>
> > **Returns**
> >
> > If the addition is empty

set_amount(*amount*)

> Change the amount of the addition.
>
> > **Parameters**
> >
> > **amount** – The new amount [in the amount unit of this addition]
> >
> > **Returns**
> >
> > This *AbstractEquilibriumAddition* object

set_component_composition(*component*, *content*)

> Change the composition of a component of the addition.
>
> > **Parameters**
> >
> > - **component** – The component to be changed
> >
> > - **content** – The new content of the component [in the composition unit defined for this addition]
> >
> > **Returns**
> >
> > This *AbstractEquilibriumAddition* object

class +tc_toolbox.+process_metallurgy.+equilibrium.**AdiabaticEquilibriumCalculation**(*back*)

> An *adiabatic* Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.

AdiabaticEquilibriumCalculation(*back*)

> Call base constructor: tc_toolbox.process_metallurgy.equilibrium.EquilibriumCalculation.

add_addition(*addition*)

Adds an addition to the calculation.

> **Parameters**
> > **addition** – A *EquilibriumAddition* or *EquilibriumGasAddition*
>
> **Returns**
> > This *AdiabaticEquilibriumCalculation* object

add_poly_command(*command*)

Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the *calculate()* method.

If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.

Note: It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.

Warning: As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
> > **command** – The POLY module command in Thermo-Calc console syntax
>
> **Returns**
> > This *AdiabaticEquilibriumCalculation* object

calculate(*timeout_in_minutes*)

Runs the Process Metallurgy equilibrium calculation.

> **Parameters**
> > **timeout_in_minutes** – The calculation will be aborted after that time, **default: no timeout**
>
> **Returns**
> > A new *EquilibriumResult* object

remove_addition(*addition*)

Removes an addition from the calculation.

> **Parameters**
> > **addition** – The addition to be removed
>
> **Returns**
> > This *AdiabaticEquilibriumCalculation* object

remove_all_additions()

Removes all additions from the calculation.

> **Returns**
> > This *AdiabaticEquilibriumCalculation* object

set_pressure(*pressure*)

Sets the pressure.

> **Parameters**
> > **pressure** – The pressure [Pa]

Returns

This *AdiabaticEquilibriumCalculation* object

update_addition(*addition*)

Replaces an already added addition with an updated one. This is usually used to change the composition or amount of an addition while iterating over them. Typically, this is done for stepping or mapping calculations.

Note: The calculation must already contain the addition object to be updated.

Parameters

addition – The new addition containing updated values

Returns

This `IsoThermalMetallurgyCalculation` object

with_options(*options*)

Sets the options for the calculation.

Parameters

options – The options

Returns

This *AdiabaticEquilibriumCalculation* object

class `+tc_toolbox.+process_metallurgy.+equilibrium.`**EquilibriumAddition**(*composition*, *amount*, *temperature*, *composition_unit*, *do_scale*)

An addition to an equilibrium calculation.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

Parameters

- **composition** – The composition
- **amount** – The amount [kg]
- **temperature** – The initial addition temperature (**default: 20 °C**) [K]
- **composition_unit** – The composition unit
- **do_scale** – If the composition is scaled to 100% / fraction of 1

EquilibriumAddition(*composition*, *amount*, *temperature*, *composition_unit*, *do_scale*)

An addition to an equilibrium calculation.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

> **Parameters**
>
> - **composition** – The composition
> - **amount** – The amount [kg]
> - **temperature** – The initial addition temperature (**default: 20 °C**) [K]
> - **composition_unit** – The composition unit
> - **do_scale** – If the composition is scaled to 100% / fraction of 1

get_amount()

Returns the amount of this addition.

> **Returns**
>
> The amount [kg]

get_composition()

Returns the composition of the addition - without containing a dependent component.

> **Returns**
>
> The composition [in the unit provided by `getCompositionUnit()`]

get_composition_unit()

Returns the composition unit used in this addition.

> **Returns**
>
> The composition unit

get_dependent_component()

Returns the dependent component.

> **Returns**
>
> The dependent component or an empty string if no dependent component is defined

get_elements()

Returns all elements of the addition.

> **Returns**
>
> The elements

get_id()

Returns the unique ID of the addition.

> **Returns**
>
> The unique ID of the addition

get_temperature()

Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> **Returns**
>
> The temperature [K]

is_do_scale()

Returns if the composition of the addition is being scaled to 100% / 1 or not.

> **Returns**
>
> If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.
>
> > **Returns**
> >
> > > If the addition is empty

set_amount(*amount*)

> Change the amount of the addition.
>
> > **Parameters**
> >
> > > **amount** – The new amount [in the amount unit of this addition]
> >
> > **Returns**
> >
> > > This *AbstractEquilibriumAddition* object

set_component_composition(*component*, *content*)

> Change the composition of a component of the addition.
>
> > **Parameters**
> >
> > > • **component** – The component to be changed
> > >
> > > • **content** – The new content of the component [in the composition unit defined for this addition]
> >
> > **Returns**
> >
> > > This *AbstractEquilibriumAddition* object

class +tc_toolbox.+process_metallurgy.+equilibrium.**EquilibriumCalculation**(*back*)

> A Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.
>
> **EquilibriumCalculation**(*back*)
>
> > Constructs an instance of *EquilibriumCalculation*.
>
> **add_addition**(*addition*)
>
> > Adds an addition to the calculation.
> >
> > > **Parameters**
> > >
> > > > **addition** – The addition
> > >
> > > **Returns**
> > >
> > > > This *EquilibriumCalculation* object
>
> **add_poly_command**(*command*)
>
> > Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the *calculate()* method.
> >
> > If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.
> >
> > ---
> >
> > **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.
> >
> > ---
> >
> > **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign).

> **Parameters**
> > command – The POLY module command in Thermo-Calc console syntax
>
> **Returns**
> > This *EquilibriumCalculation* object

calculate(*timeout_in_minutes*)

> Runs the Process Metallurgy equilibrium calculation.
>
> > **Parameters**
> > > timeout_in_minutes – The calculation will be aborted after that time, **default: no timeout**
> >
> > **Returns**
> > > A new *EquilibriumResult* object

remove_addition(*addition*)

> Removes an addition from the calculation.
>
> > **Parameters**
> > > addition – The addition to be removed
> >
> > **Returns**
> > > This *EquilibriumCalculation* object

remove_all_additions()

> Removes all additions from the calculation.
>
> > **Returns**
> > > This *EquilibriumCalculation* object

set_pressure(*pressure*)

> Sets the pressure.
>
> > **Parameters**
> > > pressure – The pressure [Pa]
> >
> > **Returns**
> > > This *EquilibriumCalculation* object

update_addition(*addition*)

> Replaces an already added addition with an updated one. This is usually used to change the composition or amount of an addition while iterating over them. Typically, this is done for stepping or mapping calculations.
>
> ---
>
> **Note:** The calculation must already contain the addition object to be updated.
>
> ---
>
> > **Parameters**
> > > addition – The new addition containing updated values
> >
> > **Returns**
> > > This IsoThermalMetallurgyCalculation object

with_options(*options*)

> Sets the options for the calculation.
>
> > **Parameters**
> > > options – The options
> >
> > **Returns**
> > > This *EquilibriumCalculation* object

class +tc_toolbox.+process_metallurgy.+equilibrium.**EquilibriumGasAddition**(*composition*,
                                                                                    amount,
                                                                                    temperature,
                                                                                    amount_unit,
                                                                                    composition_unit,
                                                                                    do_scale)

A gas addition to an equilibrium calculation.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.

EquilibriumGasAddition(*composition*, *amount*, *temperature*, *amount_unit*, *composition_unit*, *do_scale*)

A gas addition to an equilibrium calculation.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.

Parameters

- **composition** – The composition

- **amount** – The amount

- **temperature** – The initial addition temperature (**default: 20 °C**) [K]

- **amount_unit** – The amount unit

- **composition_unit** – The composition unit

- **do_scale** – If the composition is scaled to 100% / fraction of 1

get_amount()

Returns the amount of this addition.

Note: The amount unit can be obtained using *get_amount_unit()*.

Returns

The amount [in the amount unit]

get_amount_unit()

Returns the amount unit used in this addition.

Returns

The amount unit

get_composition()

Returns the composition of the addition - without containing a dependent component.

Returns

The composition [in the unit provided by getCompositionUnit()]

`get_composition_unit()`

Returns the composition unit used in this addition.

> **Returns**
>> The composition unit

`get_dependent_component()`

Returns the dependent component.

> **Returns**
>> The dependent component or an empty string if no dependent component is defined

`get_elements()`

Returns all elements of the addition.

> **Returns**
>> The elements

`get_id()`

Returns the unique ID of the addition.

> **Returns**
>> The unique ID of the addition

`get_temperature()`

Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> **Returns**
>> The temperature [K]

`is_do_scale()`

Returns if the composition of the addition is being scaled to 100% / 1 or not.

> **Returns**
>> If the composition is scaled

`is_empty()`

Returns if the addition is "empty", i.e., has zero amount.

> **Returns**
>> If the addition is empty

`set_amount`(*amount*)

Change the amount of the addition.

> **Parameters**
>> **amount** – The new amount [in the amount unit of this addition]

> **Returns**
>> This *AbstractEquilibriumAddition* object

`set_component_composition`(*component*, *content*)

Change the composition of a component of the addition.

> **Parameters**
>> - **component** – The component to be changed
>>
>> - **content** – The new content of the component [in the composition unit defined for this addition]

> **Returns**
>> This *AbstractEquilibriumAddition* object

class +tc_toolbox.+process_metallurgy.+equilibrium.**EquilibriumResult**(*back*)

> The result of a Process Metallurgy equilibrium calculation.

> **EquilibriumResult**(*back*)

>> Call base constructor: `tc_toolbox.AbstractResult`.

> **get_activity_of_slag**(*component*, *reference*)

>> Returns the activity of a component in the slag.

>> **Parameters**

>>> - **component** – The component

>>> - **reference** – The reference for the activity, can be liquid or solid slag, **default: liquid slag**

>> **Returns**
>>> The activity of the component [-]

> **get_amount**()

>> Returns the total amount.

>> **Returns**
>>> The total amount [kg]

> **get_amount_of_elements**()

>> Returns the amount of each element.

>> **Returns**
>>> The amount of the elements [kg]

> **get_amount_of_phase_groups**()

>> Returns the amount of each phase group (e.g., for example all liquid slag).

>> **Returns**
>>> The amount of the phase groups [kg]

> **get_amount_of_phases**()

>> Returns the amount of each phase.

>> **Returns**
>>> The amount of the phases [kg]

> **get_components**()

>> Returns all components defined for the elements present in this result.

>> **Returns**
>>> The components present in this result

> **get_composition**(*composition_unit*)

>> Returns the composition of the result.

>> **Parameters**
>>> **composition_unit** – The composition unit, **default: mass percent**

>> **Returns**
>>> The composition

get_composition_of_phase(*phase*, *composition_unit*, *composition_type*)

Returns the composition of a phase in the result.

> **Parameters**
>> * **phase** – The phase name
>> * **composition_unit** – The composition unit, **default: mass percent**
>> * **composition_type** – Defines if the composition is given by element (e.g., *75 wt-% Fe - 25 wt-% Cr*) or by component (e.g. *65 wt-% Al2O3 - 35 wt-% CaO*). In case of a metallic phase, the composition is given by element even if *component* is selected. **Default: by component**.
>
> **Returns**
>> The composition

get_composition_of_phase_group(*phase_group*, *composition_unit*, *composition_type*)

Returns the composition of a phase group (e.g., all liquid slag) in the result.

> **Parameters**
>> * **phase_group** – The phase group
>> * **composition_unit** – The composition unit, **default: mass percent**
>> * **composition_type** – Defines if the composition is given by element (e.g., *75 wt-% Fe - 25 wt-% Cr*) or by component (e.g. *65 wt-% Al2O3 - 35 wt-% CaO*). In case of a metallic phase, the composition is given by element even if *component* is selected. **Default: by component**.
>
> **Returns**
>> The composition

get_elements()

Returns all elements defined for the result.

> **Returns**
>> All elements present in this result

get_formula_for_activity_of_slag(*component*, *reference*)

Returns the Thermo-Calc Console syntax formula used for calculating the activity of a component in the slag (e.g. *AC(AL2O3, IONIC_LIQ)*). The actual activity can be obtained using *get_activity_of_slag()*.

> **Parameters**
>> * **component** – The component
>> * **reference** – The reference for the activity, can be liquid or solid slag, **default: liquid slag**
>
> **Returns**
>> The formula for calculating the activity

get_formula_for_slag_property(*slag_property*, *slag_type*)

Returns the Thermo-Calc Console syntax formula used for calculating a property of the slag (e.g. *B(CAO)/B(SIO2)*). The actual slag property can be obtained using *get_slag_property()*.

> **Parameters**
>> * **slag_property** – The slag property
>> * **slag_type** – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. **Default: all slag**

> **Returns**
> The formula for calculating the slag property

get_fraction_of_phase_groups(*unit*)

Returns the fraction of the phase groups (e.g., all liquid slag) in the result.

> **Parameters**
> **unit** – The unit of the fraction, **default: volume fraction**

> **Returns**
> The phase fractions

get_fraction_of_phases(*unit*)

Returns the fraction of the stable phases in the result.

> **Parameters**
> **unit** – The unit of the fraction, **default: volume fraction**

> **Returns**
> The phase fractions

get_gas_components()

Returns all components of the gas phase defined for the elements present in this result.

> **Returns**
> The components of the gas phase present in this result

get_oxygen_partial_pressure()

Returns the partial pressure of oxygen in the result.

> **Returns**
> The partial pressure [Pa]

get_pressure()

Returns the pressure in the result.

> **Returns**
> The pressure [Pa]

get_slag_property(*slag_property*, *slag_type*)

Returns a property of the slag. These properties are mostly used to describe the property of a slag to pick up sulfur.

> **Parameters**
> - **slag_property** – The slag property
>
> - **slag_type** – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. **Default: all slag**

> **Returns**
> The slag property [unit depending on the property]

get_stable_phases()

Returns the stable phases in the result.

> **Returns**
> The stable phases

get_stable_phases_in_phase_group(*phase_group*)

Returns the stable phases of a phase group (e.g., all liquid slag) in the result.

> **Parameters**
> > **phase_group** – The phase group

> **Returns**
> > The stable phases

get_temperature()

> Returns the temperature in the result.

> > **Returns**
> > > The temperature [K]

get_value_of(*classic_expression*)

> Returns a value for a thermodynamic quantity.

> ---
> **Warning:** It should normally not be required to use this method, use the appropriate method available in the API instead.
> ---

> > **Parameters**
> > > **classic_expression** – The thermodynamic quantity to get the value of in Thermo-Calc Console Mode syntax (for example "NPM(FCC_A1)")

> > **Returns**
> > > The requested value

get_viscosity_dynamic_of_phase(*phase*)

> Returns the dynamic viscosity of a phase in the result.

> > **Parameters**
> > > **phase** – The phase name

> > **Returns**
> > > The dynamic viscosity [Pa*s]

get_viscosity_kinematic_of_phase(*phase*)

> Returns the kinematic viscosity of a phase in the result.

> > **Parameters**
> > > **phase** – The phase name

> > **Returns**
> > > The kinematic viscosity [m**2/s]

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

class +tc_toolbox.+process_metallurgy.+equilibrium.**IsoThermalEquilibriumCalculation**(*back*)

> An *isothermal* Process Metallurgy equilibrium calculation. Such calculations can for example be used to determine the global equilibrium state of a process.

> **IsoThermalEquilibriumCalculation**(*back*)

> > Call base constructor: `tc_toolbox.process_metallurgy.equilibrium.EquilibriumCalculation`.

> **add_addition**(*addition*)

> > Adds an addition to the calculation.

> **Parameters**
>> **addition** – A *EquilibriumAddition* or *EquilibriumGasAddition*
>
> **Returns**
>> This *IsoThermalEquilibriumCalculation* object

add_poly_command(*command*)

> Adds a Thermo-Calc Console syntax POLY module command which will be executed when performing the calculation using the *calculate()* method.
>
> If multiple commands are added, they will be executed in the order of addition. Each command will only be executed one.
>
> ---
>
> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.
>
> ---
>
> ---
> | **Warning:** As this method runs raw Thermo-Calc commands directly in the engine, it may hang the program in case of spelling mistakes (e.g. forgotten equals sign). |
> ---
>
> **Parameters**
>> **command** – The POLY module command in Thermo-Calc console syntax
>
> **Returns**
>> This *IsoThermalEquilibriumCalculation* object

calculate(*timeout_in_minutes*)

> Runs the Process Metallurgy equilibrium calculation.
>
> **Parameters**
>> **timeout_in_minutes** – The calculation will be aborted after that time, **default: no timeout**
>
> **Returns**
>> A new *EquilibriumResult* object

remove_addition(*addition*)

> Removes an addition from the calculation.
>
> **Parameters**
>> **addition** – The addition to be removed
>
> **Returns**
>> This *IsoThermalEquilibriumCalculation* object

remove_all_additions()

> Removes all additions from the calculation.
>
> **Returns**
>> This *IsoThermalEquilibriumCalculation* object

set_pressure(*pressure*)

> Sets the pressure.
>
> **Parameters**
>> **pressure** – The pressure [Pa]
>
> **Returns**
>> This *IsoThermalEquilibriumCalculation* object

set_temperature(*temperature*)

> Sets the temperature.
>
> > **Parameters**
> > > **temperature** – The temperature [K]
> >
> > **Returns**
> > > This *IsoThermalEquilibriumCalculation* object

update_addition(*addition*)

> Replaces an already added addition with an updated one.
>
> ---
>
> **Tip:** This is usually used to change the composition or amount of an addition while iterating over multiple values. Typically, this is done for stepping or mapping calculations.
>
> ---
>
> ---
>
> **Note:** The calculation must already contain the addition object to be updated.
>
> ---
>
> > **Parameters**
> > > **addition** – A previously added addition object with the updated values
> >
> > **Returns**
> > > This *IsoThermalEquilibriumCalculation* object

with_options(*options*)

> Sets the options for the calculation.
>
> > **Parameters**
> > > **options** – The options
> >
> > **Returns**
> > > This *IsoThermalEquilibriumCalculation* object

4.1.9.3 Package "process"

class +tc_toolbox.+process_metallurgy.+process.**AbstractContinuousAddition**

> The base class representing an addition in a process simulation that is added continuously over a period of time.
>
> **get_composition**()
>
> > Returns the composition of the addition - without containing a dependent component.
> >
> > > **Returns**
> > > > The composition [in the unit provided by getCompositionUnit()]
>
> **get_composition_unit**()
>
> > Returns the composition unit used in this addition.
> >
> > > **Returns**
> > > > The composition unit
>
> **get_dependent_component**()
>
> > Returns the dependent component.
> >
> > > **Returns**
> > > > The dependent component or an empty string if no dependent component is defined

get_elements()

Returns all elements of the addition.

> **Returns**
>> The elements

get_id()

Returns the unique ID of the addition.

> **Returns**
>> The unique ID of the addition

get_temperature()

Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> **Returns**
>> The temperature [K]

is_do_scale()

Returns if the composition of the addition is being scaled to 100% / 1 or not.

> **Returns**
>> If the composition is scaled

is_empty()

Returns if the addition is "empty", i.e., has zero amount.

> **Returns**
>> If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**AbstractProcessAddition**

The base class for representing an addition in a process simulation.

get_composition()

Returns the composition of the addition - without containing a dependent component.

> **Returns**
>> The composition [in the unit provided by getCompositionUnit()]

get_composition_unit()

Returns the composition unit used in this addition.

> **Returns**
>> The composition unit

get_dependent_component()

Returns the dependent component.

> **Returns**
>> The dependent component or an empty string if no dependent component is defined

get_elements()

Returns all elements of the addition.

> **Returns**
>> The elements

get_id()

Returns the unique ID of the addition.

> **Returns**
>> The unique ID of the addition

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> > **Returns**

> > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.

> > **Returns**

> > If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.

> > **Returns**

> > If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**AbstractSingleTimeAddition**

> The base class representing an addition in a process simulation that is added at a distinct time point.

get_composition()

> Returns the composition of the addition - without containing a dependent component.

> > **Returns**

> > The composition [in the unit provided by getCompositionUnit()]

get_composition_unit()

> Returns the composition unit used in this addition.

> > **Returns**

> > The composition unit

get_dependent_component()

> Returns the dependent component.

> > **Returns**

> > The dependent component or an empty string if no dependent component is defined

get_elements()

> Returns all elements of the addition.

> > **Returns**

> > The elements

get_id()

> Returns the unique ID of the addition.

> > **Returns**

> > The unique ID of the addition

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> > **Returns**

> > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.

> > **Returns**

> > If the composition is scaled

`is_empty`()

>   Returns if the addition is "empty", i.e., has zero amount.

>   > **Returns**

>   >   If the addition is empty

`class` +tc_toolbox.+process_metallurgy.+process.**BulkZone**(*density*, *phase_group_to_transfer*, *name*)

>   A bulk zone in a process simulation, this is representing a large volume in the process, for example the steel melt or the top slag. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

>   ---

>   **Tip:** This is a generic class and seldom used directly. Use instead *MetalBulkZone* or *SlagBulkZone*.

>   ---

>   **BulkZone**(*density*, *phase_group_to_transfer*, *name*)

>   >   A bulk zone in a process simulation, this is representing a large volume in the process, for example the steel melt or the top slag. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

>   >   ---

>   >   **Tip:** This is a generic class and seldom used directly. Use instead *MetalBulkZone* or *SlagBulkZone*.

>   >   ---

>   >   **Parameters**

>   >   >   - **density** – The density of the zone [kg/m**3]

>   >   >   - **phase_group_to_transfer** – The phase group that is transferred from the attached reaction zones back to this zone after each time step, usually this is *ALL_METAL* or *ALL_OXIDES*

>   >   >   - **name** – The unique name of the zone

>   `add_addition`(*addition*, *time*)

>   >   Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

>   >   **Parameters**

>   >   >   - **addition** – A *SingleTimeAddition* or *SingleTimeGasAddition*

>   >   >   - **time** – The time point [s]

>   >   **Returns**

>   >   >   This *BulkZone* object

>   `add_continuous_addition`(*addition*, *from_time*, *to_time*)

>   >   Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

>   >   **Parameters**

>   >   >   - **addition** – A *ContinuousAddition* or *ContinuousGasAddition*

>   >   >   - **from_time** – The start time point [s]

>   >   >   - **to_time** – The end time point [s]

>   >   **Returns**

>   >   >   This *BulkZone* object

add_power(*power*, *from_time*, *to_time*)

> Adds a constant power during a specified time period to the zone (for example heating or cooling).
>
> > **Parameters**
> >
> > - **power** – The power [W]
> >
> > - **from_time** – The start time point [s]
> >
> > - **to_time** – The end time point [s]
> >
> > **Returns**
> > This *BulkZone* object

disable_degassing()

> Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.
>
> > **Returns**
> > This *BulkZone* object

enable_degassing()

> Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the *exhaust gas zone*. **This is the default**.
>
> > **Returns**
> > This *BulkZone* object

get_density()

> Returns the density of the zone
>
> > **Returns**
> > The density [kg/m**3]

get_elements()

> Returns the elements present in the zone. The elements are determined by the additions.
>
> > **Returns**
> > The elements

get_id()

> Returns the unique id of the zone. :return: The zone id

get_phase_group_to_transfer()

> Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.
>
> > **Returns**
> > The phase group

is_degassing_enabled()

> Returns if degassing is enabled in the zone.
>
> > **Returns**
> > If degassing is enabled

class +tc_toolbox.+process_metallurgy.+process.**ContinuousAddition**(*composition*, *rate*, *temperature*, *composition_unit*, *do_scale*)

An addition in a process simulation that is added continuously during a period of time.

It is assumed that the material added during that period is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

Parameters

- **composition** – The composition

- **rate** – The rate of addition [kg/s]

- **temperature** – The initial addition temperature (**default: 20 °C**) [K]

- **composition_unit** – The composition unit

- **do_scale** – If the composition is scaled to 100% / fraction of 1

ContinuousAddition(*composition*, *rate*, *temperature*, *composition_unit*, *do_scale*)

An addition in a process simulation that is added continuously during a period of time.

It is assumed that the material added during that period is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

Parameters

- **composition** – The composition

- **rate** – The rate of addition [kg/s]

- **temperature** – The initial addition temperature (**default: 20 °C**) [K]

- **composition_unit** – The composition unit

- **do_scale** – If the composition is scaled to 100% / fraction of 1

get_composition()

Returns the composition of the addition - without containing a dependent component.

Returns

The composition [in the unit provided by `getCompositionUnit()`]

get_composition_unit()

Returns the composition unit used in this addition.

Returns

The composition unit

get_dependent_component()

Returns the dependent component.

Returns

The dependent component or an empty string if no dependent component is defined

---

get_elements()

> Returns all elements of the addition.

> > **Returns**
> > > The elements

get_id()

> Returns the unique ID of the addition.

> > **Returns**
> > > The unique ID of the addition

get_rate()

> Returns the rate of addition.

> > **Returns**
> > > The addition rate [kg/s]

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> > **Returns**
> > > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.

> > **Returns**
> > > If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.

> > **Returns**
> > > If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**ContinuousGasAddition**(*composition*, *rate*, *temperature*, *rate_unit*, *composition_unit*, *do_scale*)

A gas addition in a process simulation that is added continuously during a period of time.

It is assumed that the gas added during that period is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.

ContinuousGasAddition(*composition*, *rate*, *temperature*, *rate_unit*, *composition_unit*, *do_scale*)

> A gas addition in a process simulation that is added continuously during a period of time.

> It is assumed that the gas added during that period is dissolved instantaneously.

> ---

> **Tip:** By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.

> ---

> **Parameters**
>
> - **composition** – The composition
> - **rate** – The rate of addition [kg/s]
> - **temperature** – The initial addition temperature (**default: 20 °C**) [K]
> - **rate_unit** – The amount unit
> - **composition_unit** – The composition unit
> - **do_scale** – If the composition is scaled to 100% / fraction of 1

get_composition()

Returns the composition of the addition - without containing a dependent component.

> **Returns**
>
> The composition [in the unit provided by getCompositionUnit()]

get_composition_unit()

Returns the composition unit used in this addition.

> **Returns**
>
> The composition unit

get_dependent_component()

Returns the dependent component.

> **Returns**
>
> The dependent component or an empty string if no dependent component is defined

get_elements()

Returns all elements of the addition.

> **Returns**
>
> The elements

get_id()

Returns the unique ID of the addition.

> **Returns**
>
> The unique ID of the addition

get_rate()

Returns the rate of addition.

> **Note:** The rate unit can be obtained using *get_rate_unit()*.

> **Returns**
>
> The addition rate [in the rate unit]

get_rate_unit()

Returns the rate unit used in this addition.

> **Returns**
>
> The rate unit

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.
>
> > **Returns**
> >
> > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.
>
> > **Returns**
> >
> > If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.
>
> > **Returns**
> >
> > If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**ExhaustGasResult**(*back*)

> A result representing the exhaust gas zone, here all exhaust gas generated during the process is accumulated.
>
> The data is returned for each time point of the process simulation. These time points can be obtained from this method: *ProcessSimulationResult.get_time_points()*.
>
> **ExhaustGasResult**(*back*)
>
> > Constructs an instance of *ExhaustGasResult*.
>
> **get_amount()**
>
> > Returns the amount of exhaust gas present at each time point.
> >
> > This is the amount of gas accumulated since the beginning of the process.
> >
> > > **Returns**
> > >
> > > The accumulated amount of gas at each time point [kg]
>
> **get_amount_of_components()**
>
> > Returns the amount of each exhaust gas component present at each time point.
> >
> > This is the amount of gas accumulated since the beginning of the process. This is different from the **current composition at each time point** obtained using *get_composition()*.
> >
> > > **Returns**
> > >
> > > The accumulated amount of each gas component at each time point [kg]
>
> **get_composition**(*composition_type*, *unit*)
>
> > Returns the current composition of the exhaust gas zone at each time point. This is the composition **at each time point**. This is different from the **accumulated amount** obtained using *get_amount_of_components()*.
> >
> > > **Parameters**
> > >
> > > - **composition_type** – The type of the composition, can be by gas component or by element, **default: by gas component**
> > >
> > > - **unit** – The composition unit, **default: mass percent**
> > >
> > > **Returns**
> > >
> > > The current composition of the gas components at each time point

get_pressure()

> Returns the pressure of the exhaust gas zone at each time point.

> > **Returns**
> > > The pressure [Pa]

get_stable_phases()

> Returns the stable phases within the exhaust gas zone at each time point.

> > **Returns**
> > > The stable phases

get_temperature()

> Returns the temperature of the exhaust gas at each time point.

> > **Returns**
> > > The temperature at each time point [K]

class +tc_toolbox.+process_metallurgy.+process.**MassTransferCoefficients**

> The mass transfer coefficients between a reaction zone and a bulk zone vs. time.

> **MassTransferCoefficients()**

> > The mass transfer coefficients between a reaction zone and a bulk zone vs. time. Constructs an instance of *MassTransferCoefficients*.

> **add**(*mass_transfer_coefficient*, *time*)

> > Adds the mass transfer coefficient valid beginning at a time point.

> > This value is valid until another value is defined for a later time point.

> > > **Parameters**

> > > - **mass_transfer_coefficient** – The mass transfer coefficient [m/s]

> > > - **time** – The time-point where the mass transfer coefficient begins to be valid [s]

> > > **Returns**
> > > > This *MassTransferCoefficients* object

class +tc_toolbox.+process_metallurgy.+process.**MetalBulkZone**(*density*)

> A metallic bulk zone in a process simulation.

> This is representing a large volume in the process, for example the steel melt. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

> The name of this zone is automatically defined and unique.

> **MetalBulkZone**(*density*)

> > A metallic bulk zone in a process simulation.

> > This is representing a large volume in the process, for example the steel melt. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

> > > **Parameters**
> > > > **density** – The density of the zone [kg/m**3]

> **add_addition**(*addition*, *time*)

> > Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

> > > **Parameters**

> > > - **addition** – A *SingleTimeAddition* or *SingleTimeGasAddition*

- **time** – The time point [s]

> **Returns**
>> This [`MetalBulkZone`](#) object

add_continuous_addition(*addition*, *from_time*, *to_time*)

> Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

> **Parameters**

>> - **addition** – A [`ContinuousAddition`](#) or [`ContinuousGasAddition`](#)

>> - **from_time** – The start time point [s]

>> - **to_time** – The end time point [s]

> **Returns**
>> This [`MetalBulkZone`](#) object

add_power(*power*, *from_time*, *to_time*)

> Adds a constant power during a specified time period to the zone (for example heating or cooling).

> **Parameters**

>> - **power** – The power [W]

>> - **from_time** – The start time point [s]

>> - **to_time** – The end time point [s]

> **Returns**
>> This [`MetalBulkZone`](#) object

disable_degassing()

> Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

> **Returns**
>> This [`MetalBulkZone`](#) object

enable_degassing()

> Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the *exhaust gas zone*. **This is the default**.

> **Returns**
>> This [`MetalBulkZone`](#) object

get_density()

> Returns the density of the zone

> **Returns**
>> The density [kg/m**3]

get_elements()

> Returns the elements present in the zone. The elements are determined by the additions.

> **Returns**
>> The elements

get_id()

> Returns the unique name / id of the zone.

> **Returns**
>> The zone name / id

get_phase_group_to_transfer()

> Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.

> > **Returns**
> > The phase group

is_degassing_enabled()

> Returns if degassing is enabled in the zone.

> > **Returns**
> > If degassing is enabled

class +tc_toolbox.+process_metallurgy.+process.**ProcessSimulationCalculation**(*back*)

> A Process Metallurgy process simulation. Such calculations represent complete metallurgical processes with several zones and simulate their evolution over time.

> **ProcessSimulationCalculation**(*back*)

> > Constructs an instance of *ProcessSimulationCalculation*.

> **calculate**(*timeout_in_minutes*)

> > Runs the Process Metallurgy process simulation.

> > > **Parameters**
> > > **timeout_in_minutes** – The calculation will be aborted after that time, **default: no timeout**

> > > **Returns**
> > > A new *ProcessSimulationResult* object

> **set_end_time**(*end_time*)

> > Sets the end time of a process.

> > > **Parameters**
> > > **end_time** – The end time point [s]

> > > **Returns**
> > > This *ProcessSimulationCalculation* object

> **set_initial_time_step**(*initial_time_step*)

> > Sets the initial time step used in the process simulation.

> > ---

> > **Note:** All later time steps are automatically determined to limit the expected temperature change during that step, this is controlled by *set_max_allowed_temp_change_per_step()*.

> > ---

> > > **Parameters**
> > > **initial_time_step** – The initial time step [s]

> > > **Returns**
> > > This *ProcessSimulationCalculation* object

> **set_max_allowed_temp_change_per_step**(*max_allowed_temp_change*)

> > The maximum allowed temperature change per time step. This is implicitly also limiting the composition change during a time step and required for numerical stability.

> > > **Parameters**
> > > **max_allowed_temp_change** – The maximum allowed temperature change [K]

> **Returns**
>> This *ProcessSimulationCalculation* object

set_max_time_step(*max_time_step*)

> The maximum time step chosen by the automatic time step control.
>
> ---
>
> **Note:** All time steps are automatically determined to limit the expected temperature change during that step, this is controlled by *set_max_allowed_temp_change_per_step()*.
>
> ---
>
> **Parameters**
>> **max_time_step** – The maximum time step [s]
>
> **Returns**
>> This *ProcessSimulationCalculation* object

set_min_time_step(*min_time_step*)

> The minimum time step chosen by the automatic time step control.
>
> ---
>
> **Note:** All time steps are automatically determined to limit the expected temperature change during that step, this is controlled by *set_max_allowed_temp_change_per_step()*.
>
> ---
>
> **Parameters**
>> **min_time_step** – The minimum time step [s]
>
> **Returns**
>> This *ProcessSimulationCalculation* object

set_pressure(*pressure*)

> Sets a constant pressure during the complete process.
>
> **Parameters**
>> **pressure** – The pressure [Pa]
>
> **Returns**
>> This *ProcessSimulationCalculation* object

set_pressure_in_time_period(*pressure_in_pa*, *from_time*, *to_time*)

> Sets a constant pressure during a time period.
>
> **Default: 1.0e5 Pa.**
>
> **Parameters**
>> - **pressure_in_pa** – The pressure [Pa]
>>
>> - **from_time** – The start time [s]
>>
>> - **to_time** – The end time [s]
>
> **Returns**
>> This *ProcessSimulationCalculation* object

with_options(*options*)

> Sets the options for the process simulation.
>
> **Parameters**
>> **options** – The options

Returns

This *ProcessSimulationCalculation* object

with_reaction_zone(*reaction_zone*)

Sets the reaction zone of the process simulation. The bulk zones attached to this reaction zone are configured in the reaction zone object.

Note: In the present release, **only one** reaction zone is supported.

Parameters

reaction_zone – The reaction zone object

Returns

This *ProcessSimulationCalculation* object

class +tc_toolbox.+process_metallurgy.+process.**ProcessSimulationResult**(*back*)

The result of a Process Metallurgy process simulation.

ProcessSimulationResult(*back*)

Call base constructor: `tc_toolbox.AbstractResult`.

get_activity_of_slag(*zone*, *component*, *reference*)

Returns the activity of a component in the slag in a zone at each time point.

Parameters

- **zone** – The zone object or the zone name

- **component** – The component

- **reference** – The reference for the activity, can be liquid or solid slag, **default: liquid slag**

Returns

The activity of the component at each time point [-]

get_amount(*zone*)

Returns the amount of a zone at each time point.

Parameters

zone – The zone object or the zone name

Returns

The amount at each time point [kg]

get_amount_of_elements()

Returns the total amount of each element in the simulation at each time point.

Returns

The total amount of the elements at each time point [kg]

get_amount_of_phase_groups(*zone*)

Returns the amount of each phase group (e.g., for example all liquid slag) in a zone at each time point.

Parameters

zone – The zone object or the zone name

Returns

The amount of the phase groups at each time point [kg]

get_amount_of_phases(*zone*)

> Returns the amount of each phase in a zone at each time point.

> > **Parameters**
> > > **zone** – The zone object or the zone name

> > **Returns**
> > > The amount of the phases at each time point [kg]

get_components()

> Returns all components defined in the simulation.

> > **Returns**
> > > The components

get_composition(*zone*, *composition_unit*)

> Returns the composition of a zone per element at each time point.

> > **Parameters**
> > > - **zone** – The zone object or the zone name
> > > - **composition_unit** – The composition unit, **default: mass percent**

> > **Returns**
> > > The composition at each time point

get_composition_of_phase(*zone*, *phase*, *composition_unit*, *composition_type*)

> Returns the composition of a phase in a zone at each time point.

> > **Parameters**
> > > - **zone** – The zone object or the zone name
> > > - **phase** – The phase name
> > > - **composition_unit** – The composition unit, **default: mass percent**
> > > - **composition_type** – Defines if the composition is given by element (e.g., *75 wt-% Fe - 25 wt-% Cr*) or by component (e.g. *65 wt-% Al2O3 - 35 wt-% CaO*). In case of a metallic phase, the composition is given by element even if *component* is selected. **Default: by component**.

> > **Returns**
> > > The composition at each time point

get_composition_of_phase_group(*zone*, *phase_group*, *composition_unit*, *composition_type*)

> Returns the composition of a phase group (e.g., all liquid slag) in a zone at each time point.

> > **Parameters**
> > > - **zone** – The zone object or the zone name
> > > - **phase_group** – The phase group
> > > - **composition_unit** – The composition unit, **default: mass percent**
> > > - **composition_type** – Defines if the composition is given by element (e.g., *75 wt-% Fe - 25 wt-% Cr*) or by component (e.g. *65 wt-% Al2O3 - 35 wt-% CaO*). In case of a metallic phase, the composition is given by element even if *component* is selected. **Default: by component**.

> > **Returns**
> > > The composition at each time point

`get_elements()`

    Returns all elements present in the simulation.

> **Returns**
>> The elements

`get_enthalpy()`

    Returns the total enthalpy of the process at each time point.

> **Returns**
>> The enthalpy at each time point [J]

`get_exhaust_gas()`

    Returns the result for the exhaust gas zone.

    This result object can be used to evaluate the exhaust gas zone at each time point.

> **Returns**
>> The exhaust gas zone result object.

`get_formula_for_activity_of_slag`(*zone*, *component*, *reference*)

    Returns the Thermo-Calc Console syntax formula used for calculating the activity of a component in the slag (e.g. *AC(AL2O3, IONIC_LIQ)*) in a zone at each time point. The actual activity can be obtained using *get_activity_of_slag()*.

> **Parameters**
>> - **`zone`** – The zone object or the zone name
>> - **`component`** – The component
>> - **`reference`** – The reference for the activity, can be liquid or solid slag, **default: liquid slag**

> **Returns**
>> The formula for calculating the activity at each time point

`get_formula_for_slag_property`(*zone*, *slag_property*, *slag_type*)

    Returns the Thermo-Calc Console syntax formula used for calculating a property of the slag (e.g. *B(CAO)/B(SIO2)*) in a zone at each time point. The actual slag property can be obtained using *get_slag_property()*.

> **Parameters**
>> - **`zone`** – The zone object or the zone name
>> - **`slag_property`** – The slag property
>> - **`slag_type`** – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. **Default: all slag**

> **Returns**
>> The formula for calculating the slag property at each time point

`get_fraction_of_phase_groups`(*zone*, *unit*)

    Returns the fractions of the phase groups (e.g., all liquid slag) in a zone at each time point.

> **Parameters**
>> - **`zone`** – The zone object or the zone name
>> - **`unit`** – The unit of the fraction

> **Returns**
>> The phase fractions at each time point

get_fraction_of_phases(*zone*, *unit*)

> Returns the fractions of all stable phases in a zone at each time point.

>> **Parameters**
>>> - **zone** – The zone object or the zone name
>>>
>>> - **unit** – The unit of the fraction

>> **Returns**
>>> The phase fractions at each time point

get_gas_components()

> Returns all components of the gas phase defined for the elements present in the simulation.

>> **Returns**
>>> The components of the gas phase

get_num_of_performed_steps()

> Returns the accumulated number of performed time steps at each time point.

Note: The number of performed time steps can differ from the index of the time step in the result list because time steps might have been repeated with smaller step size during a process simulation.

>> **Returns**
>>> The accumulated number of performed time steps

get_oxygen_partial_pressure(*zone*)

> Returns the partial pressure of oxygen in the zone at each time point.

>> **Parameters**
>>> **zone** – The zone object or the zone name

>> **Returns**
>>> The partial pressure [Pa]

get_pressure(*zone*)

> Returns the pressure in a zone at each time point.

>> **Parameters**
>>> **zone** – The zone object or the zone name

>> **Returns**
>>> The pressure at each time point [Pa]

get_slag_property(*zone*, *slag_property*, *slag_type*)

> Returns a property of the slag in a zone at each time point. These properties are mostly used to describe the property of a slag to pick up sulfur.

>> **Parameters**
>>> - **zone** – The zone object or the zone name
>>>
>>> - **slag_property** – The slag property
>>>
>>> - **slag_type** – The part of the slag for which the property will be calculated. Can be all slag, the liquid or the solid slag. **Default: all slag**

> **Returns**
>> The slag property at each time point [unit depending on the property]

get_stable_phases(*zone*)

> Returns the stable phases in a zone.

>> **Parameters**
>>> **zone** – The zone object or the zone name

>> **Returns**
>>> The stable phases

get_stable_phases_in_phase_group(*zone*, *phase_group*)

> Returns the stable phases of a phase group (e.g., all solid slag) in a zone.

>> **Parameters**

>>> • **zone** – The zone object or the zone name

>>> • **phase_group** – The phase group

>> **Returns**
>>> The stable phases of the phase group

get_temperature(*zone*)

> Returns the temperature of a zone at each time point.

>> **Parameters**
>>> **zone** – The zone object or the zone name

>> **Returns**
>>> The temperature at each time point [K]

get_time_points()

> Returns the time points of the process simulation. All result quantities are returned for exactly these time points.

>> **Returns**
>>> The time points [s]

get_value_of(*zone*, *classic_expression*)

> Returns a value for a thermodynamic quantity in a zone at each time point.

> **Warning:** It should normally not be required to use this method, use the appropriate method available in the API instead.

>> **Parameters**

>>> • **zone** – The zone object or the zone name

>>> • **classic_expression** – The thermodynamic quantity to get the value of in Thermo-Calc Console Mode syntax (for example "NPM(FCC_A1)")

>> **Returns**
>>> The requested value at each time point

get_viscosity_dynamic_of_phase(*zone*, *phase*)

> Returns the dynamic viscosity of a phase in a zone at each time point.

>> **Parameters**

- **zone** – The zone object or the zone name

- **phase** – The phase name

> **Returns**
>> The dynamic viscosity at each time point [Pa*s]

get_viscosity_kinematic_of_phase(*zone*, *phase*)

> Returns the kinematic viscosity of a phase in a zone at each time point.

> **Parameters**

>> - **zone** – The zone object or the zone name

>> - **phase** – The phase name

> **Returns**
>> The kinematic viscosity at each time point [m**2/s]

invalidate()

> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

class +tc_toolbox.+process_metallurgy.+process.**ReactionZone**(*area*, *left_zone*,
                                                                  mass_transfer_coefficient_left,
                                                                  right_zone,
                                                                  mass_transfer_coefficient_right)

A reaction zone in a process simulation, this is representing the interface layer between two bulk zones that are in contact and can react with each other, for example the steel melt and the top slag. The size of the reaction zone is dynamic and determined by the mass transfer coefficient. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

ReactionZone(*area*, *left_zone*, *mass_transfer_coefficient_left*, *right_zone*, *mass_transfer_coefficient_right*)

> A reaction zone in a process simulation, this is representing the interface layer between two bulk zones that are in contact and can react with each other, for example the steel melt and the top slag. The size of the reaction zone is dynamic and determined by the mass transfer coefficient. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

> **Parameters**

>> - **area** – The contact area between the bulk zones in contact [m**2]

>> - **left_zone** – The left bulk zone

>> - **mass_transfer_coefficient_left** – The mass transfer coefficient between the left bulk zone and the reaction zone, can be a constant value or time-dependent [m/s]

>> - **right_zone** – The right bulk zone

>> - **mass_transfer_coefficient_right** – The mass transfer coefficient between the right bulk zone and the reaction zone, can be a constant value or time-dependent [m/s]

add_addition(*addition*, *time*)

> Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

> **Parameters**

>> - **addition** – A *SingleTimeAddition* or *SingleTimeGasAddition*

>> - **time** – The time point [s]

Returns
> This *ReactionZone* object

add_continuous_addition(*addition*, *from_time*, *to_time*)

> Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

> **Parameters**
> - **addition** – A *ContinuousAddition* or *ContinuousGasAddition*
> - **from_time** – The start time point [s]
> - **to_time** – The end time point [s]

> **Returns**
> > This *ReactionZone* object

add_heat_transfer(*heat_transfer_coefficient*)

> Adds heat transfer through the reaction zone, i.e., between the two attached bulk zones.

> **Parameters**
> > **heat_transfer_coefficient** – The heat transfer coefficient [W/(K*m**2)]

> **Returns**
> > This *ReactionZone* object

add_power(*power*, *from_time*, *to_time*)

> Adds a constant power during a specified time period to the zone (for example heating or cooling).

> **Parameters**
> - **power** – The power [W]
> - **from_time** – The start time point [s]
> - **to_time** – The end time point [s]

> **Returns**
> > This *ReactionZone* object

add_transfer_of_phase_group(*transfer_of_phase_group*)

> Adds transfer of a certain phase group through the reaction zone during each time step, i.e. from one of the attached bulk zones to the other. This is for example used to model inclusion flotation from the steel melt to the slag.

> **Parameters**
> > **transfer_of_phase_group** – The transfer of phase group configuration, can be time-dependent.

> **Returns**
> > This *ReactionZone* object

disable_degassing()

> Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

> **Returns**
> > This *ReactionZone* object

enable_degassing()

> Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the *exhaust gas zone*. **This is the default**.

> **Returns**
>> This *ReactionZone* object

get_elements()
> Returns the elements present in the zone. The elements are determined by the additions.
>
>> **Returns**
>>> The elements

get_id()
> Returns the unique id of the zone. :return: The zone id

is_degassing_enabled()
> Returns if degassing is enabled in the zone.
>
>> **Returns**
>>> If degassing is enabled

class +tc_toolbox.+process_metallurgy.+process.**SingleTimeAddition**(*composition*, *amount*, *temperature*, *composition_unit*, *do_scale*)

An addition in a process simulation that is added at a distinct time point.

It is assumed that the addition is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

> **Parameters**
>> - **composition** – The composition
>> - **amount** – The amount [kg]
>> - **temperature** – The initial addition temperature (**default: 20 °C**) [K]
>> - **composition_unit** – The composition unit
>> - **do_scale** – If the composition is scaled to 100% / fraction of 1

SingleTimeAddition(*composition*, *amount*, *temperature*, *composition_unit*, *do_scale*)

An addition in a process simulation that is added at a distinct time point.

It is assumed that the addition is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not summing to 100% / 1. An example could be a slag addition which is provided like this: *90 wt-% CaO - 5 wt-% Al2O3 - 4 wt-% SiO2*.

> **Parameters**
>> - **composition** – The composition
>> - **amount** – The amount [kg]
>> - **temperature** – The initial addition temperature (**default: 20 °C**) [K]

- **composition_unit** – The composition unit

- **do_scale** – If the composition is scaled to 100% / fraction of 1

get_amount()

Returns the amount of this addition.

> **Returns**
>
> The amount [kg]

get_composition()

Returns the composition of the addition - without containing a dependent component.

> **Returns**
>
> The composition [in the unit provided by `getCompositionUnit()`]

get_composition_unit()

Returns the composition unit used in this addition.

> **Returns**
>
> The composition unit

get_dependent_component()

Returns the dependent component.

> **Returns**
>
> The dependent component or an empty string if no dependent component is defined

get_elements()

Returns all elements of the addition.

> **Returns**
>
> The elements

get_id()

Returns the unique ID of the addition.

> **Returns**
>
> The unique ID of the addition

get_temperature()

Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> **Returns**
>
> The temperature [K]

is_do_scale()

Returns if the composition of the addition is being scaled to 100% / 1 or not.

> **Returns**
>
> If the composition is scaled

is_empty()

Returns if the addition is "empty", i.e., has zero amount.

> **Returns**
>
> If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**SingleTimeGasAddition**(*composition*, *amount*, *temperature*, *amount_unit*, *composition_unit*, *do_scale*)

A gas addition in a process simulation that is added at a distinct time point.

It is assumed that the addition is dissolved instantaneously.

Tip: By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.

> **Parameters**
> - **composition** – The composition
> - **amount** – The amount
> - **temperature** – The initial addition temperature (**default: 20 °C**) [K]
> - **amount_unit** – The amount unit
> - **composition_unit** – The composition unit
> - **do_scale** – If the composition is scaled to 100% / fraction of 1

SingleTimeGasAddition(*composition*, *amount*, *temperature*, *amount_unit*, *composition_unit*, *do_scale*)

> A gas addition in a process simulation that is added at a distinct time point.
>
> It is assumed that the addition is dissolved instantaneously.
>
> ---
>
> **Tip:** By setting *do_scale=True*, the composition will be scaled to 100% / fraction of 1. This is useful if the composition provided is not scaling to 100% / 1. An example could be a gas addition which is provided like this: *90 vol–% Ar - 10 vol-% O2*.
>
> ---
>
> > **Parameters**
> > - **composition** – The composition
> > - **amount** – The amount
> > - **temperature** – The initial addition temperature (**default: 20 °C**) [K]
> > - **amount_unit** – The amount unit
> > - **composition_unit** – The composition unit
> > - **do_scale** – If the composition is scaled to 100% / fraction of 1

get_amount()

> Returns the amount of this addition.
>
> ---
>
> **Note:** The amount unit can be obtained using *get_amount_unit()*.
>
> ---
>
> > **Returns**
> > The amount [in the amount unit]

get_amount_unit()

> Returns the amount unit used in this addition.

> > **Returns**
> > > The amount unit

get_composition()

> Returns the composition of the addition - without containing a dependent component.

> > **Returns**
> > > The composition [in the unit provided by `getCompositionUnit()`]

get_composition_unit()

> Returns the composition unit used in this addition.

> > **Returns**
> > > The composition unit

get_dependent_component()

> Returns the dependent component.

> > **Returns**
> > > The dependent component or an empty string if no dependent component is defined

get_elements()

> Returns all elements of the addition.

> > **Returns**
> > > The elements

get_id()

> Returns the unique ID of the addition.

> > **Returns**
> > > The unique ID of the addition

get_temperature()

> Returns the temperature of the addition. This refers to the temperature before it is added to the process.

> > **Returns**
> > > The temperature [K]

is_do_scale()

> Returns if the composition of the addition is being scaled to 100% / 1 or not.

> > **Returns**
> > > If the composition is scaled

is_empty()

> Returns if the addition is "empty", i.e., has zero amount.

> > **Returns**
> > > If the addition is empty

class +tc_toolbox.+process_metallurgy.+process.**SlagBulkZone**(*density*)

> A slag bulk zone in a process simulation.

> This is representing a large volume in the process, for example the top slag. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

> The name of this zone is automatically defined and unique.

SlagBulkZone(*density*)

A slag bulk zone in a process simulation.

This is representing a large volume in the process, for example the top slag. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

The name of this zone is automatically defined and unique.

> **Parameters**
> > **density** – The density of the zone [kg/m**3]

add_addition(*addition*, *time*)

Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

> **Parameters**
>
> - **addition** – A *SingleTimeAddition* or *SingleTimeGasAddition*
>
> - **time** – The time point [s]
>
> **Returns**
> > This *SlagBulkZone* object

add_continuous_addition(*addition*, *from_time*, *to_time*)

Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

> **Parameters**
>
> - **addition** – A *ContinuousAddition* or *ContinuousGasAddition*
>
> - **from_time** – The start time point [s]
>
> - **to_time** – The end time point [s]
>
> **Returns**
> > This *SlagBulkZone* object

add_power(*power*, *from_time*, *to_time*)

Adds a constant power during a specified time period to the zone (for example heating or cooling).

> **Parameters**
>
> - **power** – The power [W]
>
> - **from_time** – The start time point [s]
>
> - **to_time** – The end time point [s]
>
> **Returns**
> > This *SlagBulkZone* object

disable_degassing()

Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

> **Returns**
> > This *SlagBulkZone* object

enable_degassing()

Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the *exhaust gas zone*. **This is the default**.

> **Returns**
> > This *SlagBulkZone* object

---

get_density()

> Returns the density of the zone
>
> > **Returns**
> >
> > > The density [kg/m**3]

get_elements()

> Returns the elements present in the zone. The elements are determined by the additions.
>
> > **Returns**
> >
> > > The elements

get_id()

> Returns the unique id of the zone. :return: The zone id

get_phase_group_to_transfer()

> Returns the phase group that is transferred from the attached reaction zones back to this zone after each time step.
>
> > **Returns**
> >
> > > The phase group

is_degassing_enabled()

> Returns if degassing is enabled in the zone.
>
> > **Returns**
> >
> > > If degassing is enabled

class +tc_toolbox.+process_metallurgy.+process.**TransferOfPhaseGroup**(*phase_group_to_transfer*, *source_zone*)

> The transfer of a percentage of a certain phase group (.e.g., solid slag) between zones during each time step. This is for example used to model inclusion flotation from the steel melt to the slag.
>
> **TransferOfPhaseGroup**(*phase_group_to_transfer*, *source_zone*)
>
> > The transfer of a percentage of a certain phase group (.e.g., solid slag) between zones during each time step. This is for example used to model inclusion flotation from the steel melt to the slag.
> >
> > > **Parameters**
> > >
> > > - **phase_group_to_transfer** – The phase group to be transferred
> > >
> > > - **source_zone** – The source zone of the transfer

add(*transfer_rate*, *time*)

> Adds the transfer rate valid beginning at a time point.
>
> This value is valid until another value is defined for a later time point.
>
> > **Parameters**
> >
> > - **transfer_rate** – The transfer rate [% of phase group amount/s]
> >
> > - **time** – The time point where the transfer of a phase group begins to be valid [s]
> >
> > **Returns**
> >
> > > This *TransferOfPhaseGroup* object

get_phase_group_to_transfer()

> Returns the phase group to be transferred
>
> > **Returns**
> >
> > > The phase group

get_transfer_source_zone_id()

>   The id of the source zone of the transfer

>   >   **Returns**
>   >   >   This source zone id

class +tc_toolbox.+process_metallurgy.+process.**Zone**

>   The base class of a zone in a process simulation. A zone is a volume in a process that has identical temperature and composition. It has well-defined boundaries to other zones.

>   **add_addition**(*addition*, *time*)

>   >   Adds a single-time addition at the specified time point to the zone. The addition will be dissolved immediately.

>   >   **Parameters**

>   >   >   • **addition** – A *SingleTimeAddition* or *SingleTimeGasAddition*

>   >   >   • **time** – The time point [s]

>   >   **Returns**
>   >   >   This *Zone* object

>   **add_continuous_addition**(*addition*, *from_time*, *to_time*)

>   >   Adds a constant addition continuously during the specified time period to the zone. All added material will be dissolved immediately.

>   >   **Parameters**

>   >   >   • **addition** – A *ContinuousAddition* or *ContinuousGasAddition*

>   >   >   • **from_time** – The start time point [s]

>   >   >   • **to_time** – The end time point [s]

>   >   **Returns**
>   >   >   This *Zone* object

>   **add_power**(*power*, *from_time*, *to_time*)

>   >   Adds a constant power during a specified time period to the zone (for example heating or cooling).

>   >   **Parameters**

>   >   >   • **power** – The power [W]

>   >   >   • **from_time** – The start time point [s]

>   >   >   • **to_time** – The end time point [s]

>   >   **Returns**
>   >   >   This *Zone* object

disable_degassing()

>   Disables degassing for this zone, i.e. all gas formed at any time step will be staying in this zone.

>   **Returns**
>   >   This *Zone* object

enable_degassing()

>   Enables degassing for this zone, i.e. any gas formed at any time step will be removed after that time step. This gas will be transferred into the *exhaust gas zone*. **This is the default**.

>   **Returns**
>   >   This *Zone* object

get_elements()

> Returns the elements present in the zone. The elements are determined by the additions.
>
> > **Returns**
> > > The elements

get_id()

> Returns the unique id of the zone. :return: The zone id

is_degassing_enabled()

> Returns if degassing is enabled in the zone.
>
> > **Returns**
> > > If degassing is enabled

4.2 Root Package

class +tc_toolbox.**AbstractCalculation**(*back*)

> Abstract base class for calculations.

> **AbstractCalculation**(*back*)
>
> > Constructs an instance of *AbstractCalculation*.

> **get_configuration_as_string**()
>
> > Returns detailed information about the current state of the calculation object.
> >
> > ---
> > **Warning:** The structure of the calculator objects is an implementation detail and might change between releases without notice. **Therefore do not rely on the internal object structure**.
> > ---

> **get_system_data**()
>
> > Returns the content of the database for the currently loaded system. This can be used to modify the parameters and functions and to change the current system by using *with_system_modifications()*.
> >
> > ---
> > **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.
> > ---
> >
> > > **Returns**
> > > > The system data

> **invalidate**()
>
> > Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation*. No data can be retrieved from the object afterwards.

> **with_system_modifications**(*system_modifications*)
>
> > Updates the system of this calculator with the supplied system modification (containing new phase parameters and system functions).
> >
> > ---
> > **Note:** This is only possible if the system has been read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.
> > ---

> **Parameters**
> `system_modifications` – The system modification to be performed

class +tc_toolbox.**AbstractResult**(*back*)

Abstract base class for results. This can be used to query for specific values .

> **AbstractResult**(*back*)
>
> Constructs an instance of *AbstractResult*.

> **invalidate**()
>
> Invalidates the object and frees the disk space used by it. *This is only required if the disk space occupied by the object needs to be released during the calculation.* No data can be retrieved from the object afterwards.

class +tc_toolbox.**CompositionType**

The type of composition.

class +tc_toolbox.**CompositionUnit**

The composition unit.

class +tc_toolbox.**Constants**

> **ALL_COMPONENTS**
>
> **ALL_PHASES**
>
> **CURRENT_TEMPERATURE**
>
> **MATERIAL_B_FRACTION**
>
> **SER**

class +tc_toolbox.**ConversionUnit**

The composition unit used in a conversion.

class +tc_toolbox.**DiffusionQuantity**

Factory class providing quantities used for defining diffusion simulations and their results.

Note: In this factory class only the most common quantities are defined, you can always use the *Console Mode* syntax strings in the respective methods as an alternative (for example: "NPM(*)").

> **static activity_of_component**(*component*, *use_ser*)
>
> Creates a quantity representing the activity of a component.
>
> > **Parameters**
> >
> > • `component` – The name of the component, use *ALL_COMPONENTS* to choose all components
> >
> > • `use_ser` – Use Stable-Element-Reference(SER). The user-defined reference state is be used if this setting is set to *False*.
> >
> > **Returns**
> > A new `ActivityOfComponent` object.

> **static chemical_diffusion_coefficient**(*phase*, *diffusing_element*, *gradient_element*, *reference_element*)
>
> Creates a quantity representing the chemical diffusion coefficient of a phase [m^2/s].

Parameters

- **phase** – The name of the phase

- **diffusing_element** – The diffusing element

- **gradient_element** – The gradient element

- **reference_element** – The reference element (for example "Fe" in a steel)

Returns

A new ChemicalDiffusionCoefficient object.

static **chemical_potential_of_component**(*component*, *use_ser*)

Creates a quantity representing the chemical potential of a component [J].

Parameters

- **component** – The name of the component, use *ALL_COMPONENTS* to choose all components

- **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to *False*.

Returns

A new ChemicalPotentialOfComponent object.

static **distance**(*region*)

Creates a quantity representing the distance [m].

Parameters

region – The name of the region or *All* to choose global.

static **intrinsic_diffusion_coefficient**(*phase*, *diffusing_element*, *gradient_element*, *reference_element*)

Creates a quantity representing the intrinsic diffusion coefficient of a phase [m^2/s].

Parameters

- **phase** – The name of the phase

- **diffusing_element** – The diffusing element

- **gradient_element** – The gradient element

- **reference_element** – The reference element (for example "Fe" in a steel)

Returns

A new IntrinsicDiffusionCoefficient object.

static **l_bis**(*phase*, *diffusing_element*, *gradient_element*, *reference_element*)

Creates a quantity representing L' of a phase [m^2/s].

Parameters

- **phase** – The name of the phase

- **diffusing_element** – The diffusing element

- **gradient_element** – The gradient element

- **reference_element** – The reference element (for example "Fe" in a steel)

Returns

A new Lbis object.

static mass_fraction_of_a_component(*component*)

> Creates a quantity representing the mass fraction of a component.
>
> > **Parameters**
> > > **component** – The name of the component or *ALL_COMPONENTS* to choose all components
> >
> > **Returns**
> > > A new `MassFractionOfAComponent` object.

static mass_fraction_of_a_phase(*phase*)

> Creates a quantity representing the mass fraction of a phase.
>
> > **Parameters**
> > > **phase** – The name of the phase or *ALL_PHASES* to choose all phases.
> >
> > **Returns**
> > > A new `MassFractionOfAPhase` object.

static mobility_of_component_in_phase(*phase*, *component*)

> Creates a quantity representing the mobility of a component in a phase [m^2/Js].
>
> > **Parameters**
> > > - **phase** – The name of the phase
> > > - **component** – The name of the component
> >
> > **Returns**
> > > A new `MobilityOfComponentInPhase` object.

static mole_fraction_of_a_component(*component*)

> Creates a quantity representing the mole fraction of a component.
>
> > **Parameters**
> > > **component** – The name of the component or *ALL_COMPONENTS* to choose all components
> >
> > **Returns**
> > > A new `MoleFractionOfAComponent` object.

static mole_fraction_of_a_phase(*phase*)

> Creates a quantity representing the mole fraction of a phase.
>
> > **Parameters**
> > > **phase** – The name of the phase or *ALL_PHASES* to choose all phases
> >
> > **Returns**
> > > A new `MoleFractionOfAPhase` object.

static position_of_lower_boundary_of_region(*region*)

> Creates a quantity representing the position of lower boundary of a region [m].
>
> > **Parameters**
> > > **region** – The name of the region
> >
> > **Returns**
> > > A new `PositionOfLowerBoundaryOfRegion` object.

static position_of_upper_boundary_of_region(*region*)

> Creates a quantity representing the position of upper boundary of a region [m].
>
> > **Parameters**
> > > **region** – The name of the region

> **Returns**
>> A new PositionOfUpperBoundaryOfRegion object.

static temperature()

> Creates a quantity representing the temperature [K].
>
> > **Returns**
> >> A new Temperature object.

static thermodynamic_factor(*phase*, *diffusing_element*, *gradient_element*, *reference_element*)

> Creates a quantity representing thermodynamic factor of a phase.
>
> > **Parameters**
> >
> > - **phase** – The name of the phase
> >
> > - **diffusing_element** – The diffusing element
> >
> > - **gradient_element** – The gradient element
> >
> > - **reference_element** – The reference element (for example "Fe" in a steel)
> >
> > **Returns**
> >> A new ThermoDynamicFactor object.

static time()

> Creates a quantity representing the time [s].

static total_mass_fraction_of_component(*component*)

> Creates a quantity representing the total mass fraction of a component.
>
> > **Parameters**
> >> **component** – The name of the component
> >
> > **Returns**
> >> A new TotalMassFractionOfComponent object.

static total_mass_fraction_of_component_in_phase(*phase*, *component*)

> Creates a quantity representing the total mass fraction of a component in a phase.
>
> > **Parameters**
> >
> > - **phase** – The name of the phase
> >
> > - **component** – The name of the component
> >
> > **Returns**
> >> A new TotalMassFractionOfComponentInPhase object.

static total_mass_fraction_of_phase(*phase*)

> Creates a quantity representing the total mass fraction of a phase.
>
> > **Parameters**
> >> **phase** – The name of the phase.
> >
> > **Returns**
> >> A new TotalMassFractionOfPhase object.

static total_mole_fraction_of_component(*component*)

> Creates a quantity representing the total mole fraction of a component.
>
> > **Parameters**
> >> **component** – The name of the component

Returns
> A new `TotalMoleFractionOfComponent` object.

static **total_mole_fraction_of_component_in_phase**(*phase*, *component*)

> Creates a quantity representing the total mole fraction of a component in a phase.
>
> **Parameters**
>
> > - **phase** – The name of the phase
> >
> > - **component** – The name of the component
>
> **Returns**
> > A new `TotalMoleFractionOfComponentInPhase` object.

static **total_volume_fraction_of_phase**(*phase*)

> Creates a quantity representing the total volume fraction of a phase.
>
> **Parameters**
> > **phase** – The name of the phase.
>
> **Returns**
> > A new `TotalVolumeFractionOfPhase` object.

static **tracer_diffusion_coefficient**(*phase*, *diffusing_element*)

> Creates a quantity representing tracer diffusion coefficient of a phase [m^2/s].
>
> **Parameters**
>
> > - **phase** – The name of the phase
> >
> > - **diffusing_element** – The diffusing element
>
> **Returns**
> > A new `TracerDiffusionCoefficient` object.

static **u_fraction_of_a_component**(*component*)

> Creates a quantity representing the u-fraction of a component.
>
> **Parameters**
> > **component** – The name of the component
>
> **Returns**
> > A new `UFractionOfAComponent` object.

static **user_defined_function**(*expression*)

> Creates a quantity representing a user-defined function.
>
> **Parameters**
> > **expression** – The function expression
>
> **Returns**
> > A new `Function` object

static **velocity_of_lower_boundary_of_region**(*region*)

> Creates a quantity representing the velocity of lower boundary of a region [m/s].
>
> **Parameters**
> > **region** – The name of the region
>
> **Returns**
> > A new `VelocityOfLowerBoundaryOfRegion` object.

static velocity_of_upper_boundary_of_region(*region*)

> Creates a quantity representing the velocity of upper boundary of a region [m/s].
>
> > **Parameters**
> > > **region** – The name of the region
> >
> > **Returns**
> > > A new `VelocityOfUpperBoundaryOfRegion` object.

static width_of_region(*region*)

> Creates a quantity representing the width of a region [m].
>
> > **Parameters**
> > > **region** – The name of the region
> >
> > **Returns**
> > > A new `WidthOfRegion` object.

class +tc_toolbox.**GasAmountUnit**

> The amount of a gas.

class +tc_toolbox.**GasCompositionUnit**

> The composition unit for a gas.

class +tc_toolbox.**GasRateUnit**

> The rate of a gas flow.

class +tc_toolbox.**IndependentVariable**

> Factory class providing quantities used for defining the independent variable in general diffusion result querying.

static distance(*region*)

> Creates an independent variable representing the distance [m].
>
> > **Returns**
> > > A new *Distance* object

static time()

> Creates an independent variable representing the time [s].
>
> > **Returns**
> > > A new *Time* object

class +tc_toolbox.**InterfacePosition**

> The position of an interface relative to its region. Only used for diffusion simulations.

class +tc_toolbox.**MetallurgyCalculations**(*back*)

> Provides access to the calculation objects for all Process Metallurgy calculations.
>
> These are specialised calculations for working with metallurgical processes. Both equilibrium calculations and kinetic process simulations (Effective Equilibrium Reaction Zone model) are available.

MetallurgyCalculations(*back*)

> Constructs an instance of *MetallurgyCalculations*.

with_adiabatic_equilibrium_calculation(*database*)

> Creates an adiabatic equilibrium calculation for Process Metallurgy.
>
> > **Parameters**
> > > **database** – The thermodynamic database used in the calculation

> **Returns**
>> A new `AdiabaticEquilibriumCalculation` object

with_adiabatic_process_calculation(*database*)

> Creates an adiabatic kinetic process simulation (EERZ, i.e. Effective Equilibrium Reaction Zone model).
>
> > **Parameters**
> >> **database** – The thermodynamic database used in the calculation
> >
> > **Returns**
> >> A new `ProcessSimulationCalculation` object

with_isothermal_equilibrium_calculation(*database*)

> Creates an isothermal equilibrium calculation for Process Metallurgy.
>
> > **Parameters**
> >> **database** – The thermodynamic database used in the calculation
> >
> > **Returns**
> >> A new `IsoThermalEquilibriumCalculation` object

class +tc_toolbox.**PhaseParameter**(*parameter_name*)

> Database phase parameter expression used by *SystemModifications.set()*.
>
> > **Parameters**
> >> **parameter_name** – The phase parameter name

PhaseParameter(*parameter_name*)

> Constructs an instance of *PhaseParameter*.

get_intervals()

> Returns the list of all defined intervals.
>
> > **Returns**
> >> The defined temperature intervals

get_lower_temperature_limit()

> Returns the lower temperature limit.
>
> > **Returns**
> >> The lower temperature limit in K

get_name()

> Returns the name of the phase parameter.
>
> > **Returns**
> >> The name of the phase parameter.

remove_all_intervals()

> Removes all previously defined temperature intervals.
>
> > **Returns**
> >> This *PhaseParameter* object

remove_interval_with_upper_limit(*upper_temperature_limit*)

> Removes a previously defined temperature interval with matching upper temperature limit.
>
> **If no such interval exists, an exception is thrown.**
>
> > **Returns**
> >> This *PhaseParameter* object

set_expression_with_upper_limit(*parameter_expression*, *upper_temperature_limit*)

> Adds/overwrites a parameter expression for a temperature interval.

> **Default value of the upper limit of the interval**: 6000 K

> **Note:** The lower temperature limit is either defined by the lower temperature limit given with *PhaseParameter.set_lower_temperature_limit()* or by the upper temperature limit of the adjacent interval.

> **Note:** If there is an existing interval with exactly the same *upper_temperature_limit*, that interval is overwritten, otherwise the interval is added.

> **Parameters**
> - **parameter_expression** – The parameter expression, example: *+V34*T*LN(T)+V35*T**2+V36*T**(-1)+V37*T**3")*
> - **upper_temperature_limit** – The upper temperature limit for which the expression should be used

> **Returns**
> > This *PhaseParameter* object

set_interval(*interval*)

> Adds/overwrites a temperature interval.

> **Note:** The lower temperature limit is either defined by the lower temperature limit given with *PhaseParameter.set_lower_temperature_limit()* or by the upper temperature limit of the adjacent interval.

> **Note:** If there is an existing interval with exactly the same *upper_temperature_limit*, that interval is overwritten, otherwise the interval is added.

> **Returns**
> > This *PhaseParameter* object

set_lower_temperature_limit(*lower_temperature_limit*)

> Sets the lower temperature limit of the phase parameter.

> **Default**: 298.15 K

> **Parameters**
> > **lower_temperature_limit** – The lower temperature limit in K

> **Returns**
> > This *PhaseParameter* object

class +tc_toolbox.**PhaseUnit**

> The units available for a phase fraction.

---

class +tc_toolbox.**PlotCondition**

> Factory class providing quantities used for defining the plot condition in general diffusion result querying.

> **Note:** In this factory class only the most common quantities are defined, you can always use the *Console Mode* syntax strings in the respective methods as an alternative (for example: "time last").

> **static distance**(*distancepoint*, *region*)
>
> > Creates a plot condition representing the distance [m].
> >
> > **Change in version 2019b**: Mandatory parameter *distancepoint* added
> >
> > > **Parameters**
> > >
> > > - **distancepoint** – The distance from the lower interface of the region
> > >
> > > - **region** – The name of the region or *All* to choose global.
> > >
> > > **Returns**
> > > A new *DistanceCondition* object
>
> **static integral**()
>
> > Creates an integral plot condition.
> >
> > > **Returns**
> > > A new *IntegralCondition* object
>
> **static interface**(*region*, *interface_position*)
>
> > Creates a plot condition representing an interface between two regions.
> >
> > > **Parameters**
> > >
> > > - **region** – The name of the region used for defining the interface
> > >
> > > - **interface_position** – The position of the interface relative to that region (lower or upper)
> > >
> > > **Returns**
> > > A new *InterfaceCondition* object
>
> **static time**(*timepoint*)
>
> > Creates a plot condition representing the time [s].
> >
> > **Change in version 2019b**: Lists of timepoints are no longer supported
> >
> > > **Parameters**
> > > **timepoint** – The timepoint. Optionally "Last" can be used for the end of the simulation
> > >
> > > **Returns**
> > > A new *TimeCondition* object

class +tc_toolbox.**ResultLoader**(*back*)

> Contains methods for loading results from previously done calculations.

> **ResultLoader**(*back*)
>
> > Constructs an instance of *ResultLoader*.

> **diffusion**(*path*)
>
> > Loads a DiffusionCalculationResult from disc.
> >
> > > **Parameters**
> > > **path** – path to the folder where result was previously saved.

> **Returns**
>> A new `DiffusionCalculationResult` object which later can be used to get specific values from the calculated result

phase_diagram(*path*)

> Loads a `PhaseDiagramResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

>> **Returns**
>>> A new `PhaseDiagramResult` object which later can be used to get specific values from the calculated result

precipitation_TTT_or_CCT(*path*)

> Loads a `PrecipitationCalculationTTTorCCTResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

>> **Returns**
>>> A new `PrecipitationCalculationTTTorCCTResult` object which later can be used to get specific values from the calculated result

precipitation_single(*path*)

> Loads a `PrecipitationCalculationSingleResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

>> **Returns**
>>> A new `PrecipitationCalculationSingleResult` object which later can be used to get specific values from the calculated result

property_diagram(*path*)

> Loads a `PropertyDiagramResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

>> **Returns**
>>> A new `PropertyDiagramResult` object which later can be used to get specific values from the calculated result

property_model(*path*)

> Loads a `PropertyModelResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

>> **Returns**
>>> A new `PropertyModelResult` object which later can be used to get specific values from the calculated result

scheil(*path*)

> Loads a `ScheilCalculationResult` from disc.

>> **Parameters**
>>> **path** – path to the folder where result was previously saved.

> **Returns**
> > A new `ScheilCalculationResult` object which later can be used to get specific values from the calculated result

single_equilibrium(*path*)

> Loads a `SingleEquilibriumResult` from disc.
>
> > **Parameters**
> > > **path** – path to the folder where result was previously saved.
> >
> > **Returns**
> > > A new `SingleEquilibriumResult` object which later can be used to get specific values from the calculated result

class +tc_toolbox.**ResultValueGroup**(*back*)

> A x-y-dataset representing a line data calculation result (i.e. a Thermo-Calc *quantity 1* vs. *quantity 2*).

Warning: Depending on the calculator, the dataset might contain *NaN*-values to separate the data between different subsets.

> > **Returns**
> > > list of floats representing the second quantity ("y-axis")

ResultValueGroup(*back*)

> Constructs an instance of *ResultValueGroup*.

get_label()

> Accessor for the line label :return the line label

get_x()

> Accessor for the x-values :return the x values

get_y()

> Accessor for the y-values :return the y values

class +tc_toolbox.**ScheilQuantity**

> Factory class providing quantities used for defining a Scheil calculation result (+tc_toolbox.scheil. ScheilCalculationResult).

static apparent_heat_capacity_per_gram()

> Creates a quantity representing the apparent heat capacity [J/g/K].
>
> > **Returns**
> > > A new `ApparentHeatCapacityPerGram` object.

static apparent_heat_capacity_per_mole()

> Creates a quantity representing the apparent heat capacity [J/mol/K].
>
> > **Returns**
> > > A new `ApparentHeatCapacityPerMole` object.

static apparent_volumetric_thermal_expansion_coefficient()

> Creates a quantity representing the apparent volumetric thermal expansion coefficient of the system [1/K].
>
> > **Returns**
> > > A new `ApparentVolumetricThermalExpansionCoefficient` object.

---

static **average_composition_of_solid_phases_as_mass_fraction**(*component*)

Creates a quantity representing the average composition of solid phases [mass fraction] at the current Scheil step.

> **Parameters**
>
> > **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> **Returns**
>
> > A new `AverageCompositionOFSolidPhasesAsMassFraction` object.

static **average_composition_of_solid_phases_as_mole_fraction**(*component*)

Creates a quantity representing the average composition of solid phases [mole fraction] at the current Scheil step.

> **Parameters**
>
> > **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> **Returns**
>
> > A new `AverageCompositionOFSolidPhasesAsMoleFraction` object.

static **composition_of_phase_as_mole_fraction**(*phase*, *component*)

Creates a quantity representing the composition of a phase [mole-fraction].

> **Parameters**
>
> > - **phase** – The name of the phase, use *ALL_PHASES* to choose all stable phases
> >
> > - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> **Returns**
>
> > A new `CompositionOfPhaseAsMoleFraction` object.

static **composition_of_phase_as_weight_fraction**(*phase*, *component*)

Creates a quantity representing the composition of a phase [weight-fraction].

> **Parameters**
>
> > - **phase** – The name of the phase, use *ALL_PHASES* to choose all stable phases
> >
> > - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> **Returns**
>
> > A new `CompositionOfPhaseAsWeightFraction` object.

static **density_of_phase**(*phase*)

Creates a quantity representing the average density of a phase [g/cm^3].

> **Parameters**
>
> > **phase** – The name of the phase or *ALL_PHASES* to choose all phases
>
> **Returns**
>
> > A new `DensityOfPhase` object.

static **density_of_system**()

Creates a quantity representing the average density of the system [g/cm^3].

> **Returns**
>
> > A new `DensityOfSystem` object.

static distribution_of_component_of_phase(*phase*, *component*)

Creates a quantity representing the (molar) fraction of the specified component being present in the specified phase compared to the overall system [-]. This corresponds to the degree of segregation to that phase.

> **Parameters**
> - **phase** – The name of the phase
> - **component** – The name of the component
>
> **Returns**
> A new `DistributionOfComponentOfPhase` object.

static heat_per_gram()

Creates a quantity representing the total heat release from the liquidus temperature down to the current temperature [J/g].

Note: The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (`latent_heat_per_mole()` and `latent_heat_per_gram()`), and the other is the heat related to the specific heat of liquid and solid phases (`heat_per_mole()` and `heat_per_gram()`).

> **Returns**
> A new `HeatPerGram` object.

static heat_per_mole()

Creates a quantity representing the total heat release from the liquidus temperature down to the current temperature [J/mol].

Note: The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (`latent_heat_per_mole()` and `latent_heat_per_gram()`), and the other is the heat related to the specific heat of liquid and solid phases (`heat_per_mole()` and `heat_per_gram()`).

> **Returns**
> A new `HeatPerMole` object.

static latent_heat_per_gram()

Creates a quantity representing the cumulated latent heat release from the liquidus temperature down to the current temperature [J/g].

Note: The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (`latent_heat_per_mole()` and `latent_heat_per_gram()`), and the other is the heat related to the specific heat of liquid and solid phases (`heat_per_mole()` and `heat_per_gram()`).

> **Returns**
> A new `LatentHeatPerGram` object.

static **latent_heat_per_mole**()

> Creates a quantity representing the cumulated latent heat release from the liquidus temperature down to the current temperature [J/mol].
>
> ---
>
> **Note:** The total or apparent heat release during the solidification process consists of two parts: one is the so-called latent heat, i.e. heat due to the liquid -> solid phase transformation (*latent_heat_per_mole()* and *latent_heat_per_gram()*), and the other is the heat related to the specific heat of liquid and solid phases (*heat_per_mole()* and *heat_per_gram()*).
>
> ---
>
> > **Returns**
> > > A new `LatentHeatPerMole` object.

static **mass_fraction_of_a_solid_phase**(*phase*)

> Creates a quantity representing the mass fraction of a solid phase.
>
> > **Parameters**
> > > **phase** – The name of the phase or *ALL_PHASES* to choose all solid phases
> >
> > **Returns**
> > > A new `MassFractionOfASolidPhase` object.

static **mass_fraction_of_all_liquid**()

> Creates a quantity representing the total mass fraction of all the liquid phase.
>
> > **Returns**
> > > A new `MassFractionOfAllLiquid` object.

static **mass_fraction_of_all_solid_phases**()

> Creates a quantity representing the total mass fraction of all solid phases.
>
> > **Returns**
> > > A new `MassFractionOfAllSolidPhase` object.

static **molar_volume_of_phase**(*phase*)

> Creates a quantity representing the molar volume of a phase [m^3/mol].
>
> > **Parameters**
> > > **phase** – The name of the phase or *ALL_PHASES* to choose all phases
> >
> > **Returns**
> > > A new `MolarVolumeOfPhase` object.

static **molar_volume_of_system**()

> Creates a quantity representing the molar volume of the system [m^3/mol].
>
> > **Returns**
> > > A new `MolarVolumeOfSystem` object.

static **mole_fraction_of_a_solid_phase**(*phase*)

> Creates a quantity representing the molar fraction of a solid phase.
>
> > **Parameters**
> > > **phase** – The name of the phase or *ALL_PHASES* to choose all solid phases
> >
> > **Returns**
> > > A new `MoleFractionOfASolidPhase` object.

static mole_fraction_of_all_liquid()

> Creates a quantity representing the total molar fraction of all the liquid phase.
>
> > **Returns**
> >   A new `MoleFractionOfAllLiquid` object.

static mole_fraction_of_all_solid_phases()

> Creates a quantity representing the total molar fraction of all solid phases.
>
> > **Returns**
> >   A new `MoleFractionOfAllSolidPhases` object.

static site_fraction_of_component_in_phase(*phase*, *component*, *sub_lattice_ordinal_no*)

> Creates a quantity representing the site fractions [-].

Note: Detailed information about the sublattices can be obtained by getting the *Phase* object of a phase from the *System* object using `+tc_toolbox.system.System.get_phase_in_system`. For each phase the sublattices are obtained by using `+tc_toolbox.system.Phase.get_sublattices`. The order in the returned list is equivalent to the sublattice ordinal number expected, **but note that the ordinal numbers start with 1**.

> > **Parameters**
> >
> > - **phase** – The name of the phase, use *ALL_PHASES* to choose all stable phases
> >
> > - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
> >
> > - **sub_lattice_ordinal_no** – The ordinal number (i.e. 1, 2, …) of the sublattice of interest, use None to choose all sublattices
> >
> > **Returns**
> >   A new `SiteFractionOfComponentInPhase` object.

static temperature()

> Creates a quantity representing the temperature [K].
>
> > **Returns**
> >   A new `Temperature` object.

static volume_fraction_of_a_solid_phase(*phase*)

> Creates a quantity representing the volume fraction of a solid phase.
>
> > **Parameters**
> >   **phase** – The name of the phase or *ALL_PHASES* to choose all solid phases
> >
> > **Returns**
> >   A new `VolumeFractionOfASolidPhase` object.

static volume_fraction_of_all_liquid()

> Creates a quantity representing the total volume fraction of all the liquid phase.
>
> > **Returns**
> >   A new `VolumeFractionOfAllLiquid` object.

static volume_fraction_of_all_solid_phases()

> Creates a quantity representing the total volume fraction of all solid phases.

> **Returns**
> A new `VolumeFractionOfAllSolidPhases` object.

class `+tc_toolbox.`**`SystemData`**(*back*)

Provides information about the parameters and functions of a user database. The obtained objects can be used to modify the database using `with_system_modifications()` of all calculators.

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.

`SystemData`(*back*)

Constructs an instance of *`SystemData`*.

`get_phase_parameter`(*parameter*)

Returns a phase parameter.

Example:

system_data.get_phase_parameter('G(HCP_A3,FE:VA;0)')

Note: Parameters can only be read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

Note: For details about the syntax search the Thermo-Calc help for *GES* (the name for the Gibbs Energy System module in Console Mode).

> **Parameters**
> **`parameter`** – The name of the phase parameter (for example: *"G(LIQUID,FE;0)"*)
>
> **Returns**
> The phase parameter

`get_phase_parameter_names`()

Returns all phase parameters present in the current system.

> **Returns**
> The list of phase parameters

`get_system_function`(*f*)

Returns a system function.

Example:

system_data.get_system_function('GHSERCR')

Note: The parameter 'f' was previously called 'function' but was renamed.

Note: Functions can only be read from unencrypted (i.e. *user*) databases loaded as a `*.tdb`-file.

Note: For details about the syntax search the Thermo-Calc help for *GES* (the name for the Gibbs Energy System module in Console Mode).

> **Parameters**
>> **f** – The name of the system function (for example: *"GHSERCR"*)
>
> **Returns**
>> The system function

get_system_function_names()

> Returns all system functions present in the current system.
>
> **Returns**
>> The list of system functions

class +tc_toolbox.**SystemFunction**(*function_name*)

> Database function expression used by *SystemModifications.set()*.
>
> **Parameters**
>> **function_name** – The function name

SystemFunction(*function_name*)

> Constructs an instance of *SystemFunction*.

get_intervals()

> Returns the list of all defined intervals.
>
> **Returns**
>> The defined temperature intervals

get_lower_temperature_limit()

> Returns the lower temperature limit.
>
> **Returns**
>> The lower temperature limit in K

get_name()

> Returns the name of the system function.
>
> **Returns**
>> The name of the system function

remove_all_intervals()

> Removes all previously defined temperature intervals.
>
> **Returns**
>> This *SystemFunction* object

remove_interval_with_upper_limit(*upper_temperature_limit*)

> Removes a previously defined temperature interval with matching upper temperature limit.
>
> **If no such interval exists, an exception is thrown.**
>
> **Returns**
>> This *SystemFunction* object

set_expression_with_upper_limit(*function_expression*, *upper_temperature_limit*)

> Adds/overwrites a function expression for a temperature interval.
>
> **Default value of the upper limit of the interval**: 6000 K

Note: The lower temperature limit is either defined by the lower temperature limit given with `SystemFunction.set_lower_temperature_limit()` or by the upper temperature limit of the adjacent interval.

Note: If there is an existing interval with exactly the same *upper_temperature_limit*, that interval is overwritten, otherwise the interval is added.

> **Parameters**
> - **function_expression** – The function expression, example: *+V34*T*LN(T)+V35*T**2+V36*T**(-1)+V37*T**3")*
> - **upper_temperature_limit** – The upper temperature limit for which the expression should be used
>
> **Returns**
>   This *SystemFunction* object

set_interval(*interval*)

Adds/overwrites a temperature interval.

Note: The lower temperature limit is either defined by the lower temperature limit given with `SystemFunction.set_lower_temperature_limit()` or by the upper temperature limit of the adjacent interval.

Note: If there is an existing interval with exactly the same *upper_temperature_limit*, that interval is overwritten, otherwise the interval is added.

> **Returns**
>   This *SystemFunction* object

set_lower_temperature_limit(*lower_temperature_limit*)

Sets the lower temperature limit of the system function.

Default: 298.15 K

> **Parameters**
>   **lower_temperature_limit** – The lower limit in K
>
> **Returns**
>   This *SystemFunction* object

class +tc_toolbox.**SystemModifications**

Functionality to modify a user database during a calculation by changing phase parameters and system functions.

The actual changes are **only applied** by using `+tc_toolbox.abstract_base.AbstractCalculation.with_system_modifications()` on a calculator object.

SystemModifications()

Constructs an instance of *SystemModifications*.

---

run_ges_command(*ges_command*)

> Sends a GES-command. **This is actually applied when running `with_system_modifications` on a calculator object.**
>
> Example: *run_ges_command("AM-PH-DE FCC_A1 C_S 2 Fe:C")* for adding a second composition set to the FCC_A1 phase with *Fe* as major constituent on first sublattice and *C* as major constituent on second sublattice.
>
> ---
> **Note:** For details about the syntax search the Thermo-Calc help for *GES* (the name for the Gibbs Energy System module in Console Mode).
>
> ---
>
> ---
> **Note:** It should not be necessary for most users to use this method, try to use the corresponding method implemented in the API instead.
>
> ---
>
> ┌───┐
> │ **Warning:** As this method runs raw GES-commands directly in the engine, it may hang the program │
> │ in case of spelling mistakes (e.g. forgotten parenthesis, . . .). │
> └───┘
>
> > **Parameters**
> > > **ges_command** – The GES-command (for example: *"AM-PH-DE FCC_A1 C_S 2 Fe:C"*)
> >
> > **Returns**
> > > This *SystemModifications* object

set(*parameter_or_function*)

> Overwrites or creates a phase parameter or system function.
>
> Example: *system_modifications.set(PhaseParameter('G(LIQUID,FE;0)').set_expression_with_upper_limit('+1.2*GFELIQ*
>
> Example: *system_modifications.set(SystemFunction("DGDEF").set_expression_with_upper_limit('+10.0-R*T', 1000).set_expression_with_upper_limit('+20.0-R*T', 3000))*
>
> ---
> **Note:** The old parameter/function is **overwritten** and any temperature intervals not defined are lost.
>
> ---
>
> ---
> **Note:** Please consult the Thermo-Calc GES-system documentation for details about the syntax.
>
> ---
>
> > **Returns**
> > > This *SystemModifications* object

class +tc_toolbox.**TCToolbox**

> TCToolbox Starting point for all calculations. This class exposes methods that have no precondition, it is used for choosing databases and elements.

TCToolbox()

> TCToolbox Construct an instance of this class

delete()

> TCToolbox Clears all resources used by the session Shuts down the API server and deletes all temporary files. The disk usage of temporary files might be significant.

---

disable_caching()

A previously set cache folder is no longer used.

Note: Within the session, caching is activated and used through the default temporary directory.

Returns

This *SetUp* object

get_database_info(*database_short_name*)

Obtains the short information available for the specified database.

Parameters

database_short_name – The name of the database (i.e. "FEDEMO", . . .)

Returns

The short information about the database

get_database_path_on_disk(*database_short_name*)

Obtains the path to the database file on disk. *TCPATH* is a placeholder for the root path of the used Thermo-Calc installation.

Note: Encrypted databases (*.TDC*) cannot be edited.

Parameters

database_short_name – The name of the database (i.e. "FEDEMO", . . .)

Returns

The path to the database on disk

get_databases()

Obtains the short names of all databases available in the used Thermo-Calc installation.

Note: Only databases with a valid license are listed.

Returns

List of the available databases

get_property_models(*path_to_models*)

Lists the names of all Property Models in the specified directory.

If the directory is not specified, the Property Model folder used by the normal Thermo-Calc application is used.

Parameters

path_to_models – The path where the Property Models are installed. If no value is entered, the Property Model folder used by the normal Thermo-Calc application is used.

Returns

Set containing all Property Model names

`load_result_from_disk()`
> Loads a previously calculated result from disk.

Note: This **only** works for results created by calling one of the `save_result()` methods on a *Result* class created from a calculation.

> **Returns**
>> A new `ResultLoader` object

`select_database_and_elements`(*database_name*, *list_of_elements*)
> Selects a first thermodynamic or kinetic database and selects the elements in it.
>
> **Parameters**
>> • **`database_name`** – The name of the database, for example "FEDEMO"
>>
>> • **`list_of_elements`** – The list of the selected elements in that database, for example ["Fe", "C"]
>
> **Returns**
>> A new `SystemBuilder` object

`select_thermodynamic_and_kinetic_databases_with_elements`(*thermodynamic_db_name*, *kinetic_db_name*, *list_of_elements*)
> Selects the thermodynamic and kinetic database at once, guarantees that the databases are added in the correct order. Further rejection or selection of phases applies to both databases.
>
> **Parameters**
>> • **`thermodynamic_db_name`** – The thermodynamic database name, for example "FEDEMO"
>>
>> • **`kinetic_db_name`** – The kinetic database name, for example "MFEDEMO"
>>
>> • **`list_of_elements`** – The list of the selected elements in that database, for example ["Fe", "C"]
>
> **Returns**
>> A new `MultiDatabaseSystemBuilder` object

`select_user_database_and_elements`(*path_to_user_database*, *list_of_elements*)
> Selects a user-defined database and selects the elements in it.

Note: By using a r-literal, it is possible to use slashes on all platforms, also on Windows: *select_user_database_and_elements(r"my path/user_db.tdb", ["Fe", "Cr"]])*

Note: On Linux and Mac the path is case-sensitive, also the file ending.

> **Parameters**
>> • **`path_to_user_database`** – The path to the database file ("database".TDB), defaults to the current working directory. Only filename is required if the database is located in the same folder as the script.

- **list_of_elements** – The list of the selected elements in that database, for example ["Fe", "C"]

Returns
A new SystemBuilder object

set_cache_folder(*path*, *precision_for_floats*)

Sets a folder where results from calculations and state of systems are saved. If at any time a calculation is run which has the exact same setting as a previous, the calculation is not re-run. The result is instead loaded from this folder.

Note: The same folder can be used in several scripts, and it can even be shared between different users. It can be a network folder.

Parameters

- **path** – path to the folder where results should be stored. It can be relative or absolute.

- **precision_for_floats** – The number of significant figures used when comparing if the calculation has the same setting as a previous.

Returns
This *SetUp* object

set_ges_version(*version*)

Setting the version of the Gibbs Energy System (GES).

Parameters
version – The GES-version (currently version 5 or 6)

Returns
This *SetUp* object

set_log_level_to_debug()

Sets log level to DEBUG

Returns
This *SetUp* object

set_log_level_to_info()

Sets log level to INFO

Returns
This *SetUp* object

with_metallurgy()

Provides access to the calculation objects for all Process Metallurgy calculations.

These are specialised calculations for working with metallurgical processes. Both equilibrium calculations and kinetic process simulations (Effective Equilibrium Reaction Zone model) are available.

class +tc_toolbox.**TemperatureInterval**(*expression*, *upper_temperature_limit*)

Temperature interval expression used by the classes *SystemFunction* and *PhaseParameter*.

Parameters

- **expression** – The temperature function expressed in Thermo-Calc database syntax.

- **upper_temperature_limit** – The upper temperature limit in K

> **TemperatureInterval**(*expression*, *upper_temperature_limit*)
>
> > Constructs an instance of *TemperatureInterval*.

> **get_expression**()
>
> > Returns the function expression of this temperature interval.
> >
> > > **Returns**
> > >
> > > > The temperature function expression

> **get_upper_temperature_limit**()
>
> > Returns the upper limit of this temperature interval.
> >
> > > **Returns**
> > >
> > > > The upper temperature limit in K

> **set_expression**(*expression*)
>
> > Sets the function expression of this temperature interval.
> >
> > > **Parameters**
> > >
> > > > **expression** – The temperature function expression

> **set_upper_temperature_limit**(*upper_temperature_limit*)
>
> > Sets the upper limit of this temperature interval.
> >
> > > **Parameters**
> > >
> > > > **upper_temperature_limit** – The upper temperature limit in K

class +tc_toolbox.**TemperatureProfile**

> Represents a time-temperature profile used by non-isothermal calculations.

Note: The total simulation time can differ from the defined temperature profile. Constant temperature is assumed for any timepoint after the end of the defined profile.

> **TemperatureProfile**()
>
> > Constructor. Constructs an instance of *TemperatureProfile*.

> **add_time_temperature**(*time*, *temperature*)
>
> > Adds a time-temperature point to the non-isothermal temperature profile.
> >
> > > **Parameters**
> > >
> > > > - **time** – The time [s]
> > > >
> > > > - **temperature** – The temperature [K]
> > >
> > > **Returns**
> > >
> > > > This *TemperatureProfile* object

class +tc_toolbox.**ThermodynamicQuantity**

> Factory class providing quantities used for defining equilibrium calculations (single equilibrium, property and phase diagrams, . . .) and their results.

Note: In this factory class only the most common quantities are defined, you can always use the *Console Mode* syntax strings in the respective methods as an alternative (for example: "NPM(*)").

---

static activity_of_component(*component*, *use_ser*)

Creates a quantity representing the activity of a component [-].

> **Parameters**
>
> - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> - **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to *False*.
>
> **Returns**
> A new `ActivityOfComponent` object.

static chemical_diffusion_coefficient(*phase*, *diffusing_element*, *gradient_element*, *reference_element*)

Creates a quantity representing the chemical diffusion coefficient of a phase [m^2/s].

> **Parameters**
>
> - **phase** – The name of the phase
>
> - **diffusing_element** – The diffusing element
>
> - **gradient_element** – The gradient element
>
> - **reference_element** – The reference element (for example "Fe" in a steel)
>
> **Returns**
> A new `ChemicalDiffusionCoefficient` object.

static chemical_potential_of_component(*component*, *use_ser*)

Creates a quantity representing the chemical potential of a component [J].

> **Parameters**
>
> - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> - **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state is used if this setting is set to *False*.
>
> **Returns**
> A new `ChemicalPotentialOfComponent` object.

static composition_of_phase_as_mole_fraction(*phase*, *component*)

Creates a quantity representing the composition of a phase [mole-fraction].

> **Parameters**
>
> - **phase** – The name of the phase, use *ALL_PHASES* to choose all stable phases
>
> - **component** – The name of the component, use *ALL_COMPONENTS* to choose all components
>
> **Returns**
> A new `CompositionOfPhaseAsMoleFraction` object.

static composition_of_phase_as_weight_fraction(*phase*, *component*)

Creates a quantity representing the composition of a phase [weight-fraction].

> **Parameters**
>
> - **phase** – The name of the phase, use *ALL_PHASES* to choose all stable phases

- **component** – The name of the component, use *ALL_COMPONENTS* to choose all components

> **Returns**
>> A new CompositionOfPhaseAsWeightFraction object.

static gibbs_energy_of_a_phase(*phase*, *use_ser*)

> Creates a quantity representing the Gibbs energy of a phase [J].

>> **Parameters**

>> - **phase** – The name of the phase or *ALL_PHASES* to choose all phases

>> - **use_ser** – Use Stable-Element-Reference(SER). The user-defined reference state will be used when this setting is set to False.

>> **Returns**
>>> A new GibbsEnergyOfAPhase object.

static mass_fraction_of_a_component(*component*)

> Creates a quantity representing the mass fraction of a component.

>> **Parameters**
>>> component – The name of the component or *ALL_COMPONENTS* to choose all components

>> **Returns**
>>> A new MassFractionOfAComponent object.

static mass_fraction_of_a_phase(*phase*)

> Creates a quantity representing the mass fraction of a phase.

>> **Parameters**
>>> phase – The name of the phase or *ALL_PHASES* to choose all phases.

>> **Returns**
>>> A new MassFractionOfAPhase object.

static mole_fraction_of_a_component(*component*)

> Creates a quantity representing the mole fraction of a component.

>> **Parameters**
>>> component – The name of the component or *ALL_COMPONENTS* to choose all components

>> **Returns**
>>> A new MoleFractionOfAComponent object.

static mole_fraction_of_a_phase(*phase*)

> Creates a quantity representing the mole fraction of a phase.

>> **Parameters**
>>> phase – The name of the phase or *ALL_PHASES* to choose all phases

>> **Returns**
>>> A new MoleFractionOfAPhase object.

static normalized_driving_force_of_a_phase(*phase*)

> Creates a quantity representing normalized driving force of a phase [-].

> **Warning:** A driving force calculation requires that the respective phase has been set to the state *DORMANT*. The parameter *All* is only reasonable if all phases have been set to that state.

> **Parameters**
> > **phase** – The name of the phase or *ALL_PHASES* to choose all phases
>
> **Returns**
> > A new `DrivingForceOfAPhase` object.

static `pressure`()

Creates a quantity representing the pressure [Pa].

> **Returns**
> > A new `Pressure` object.

static `system_size`()

Creates a quantity representing the system size [mol].

> **Returns**
> > A new `SystemSize` object.

static `temperature`()

Creates a quantity representing the temperature [K].

> **Returns**
> > A new `Temperature` object.

static `tracer_diffusion_coefficient`(*phase*, *diffusing_element*)

Creates a quantity representing tracer diffusion coefficient of a phase [m^2/s].

> **Parameters**
>
> > - **phase** – The name of the phase
> >
> > - **diffusing_element** – The diffusing element
>
> **Returns**
> > A new `TracerDiffusionCoefficient` object.

static `u_fraction_of_a_component`(*component*)

Creates a quantity representing the u-fraction of a component.

> **Parameters**
> > **component** – The name of the component
>
> **Returns**
> > A new `UFractionOfAComponent` object.

static `user_defined_function`(*expression*)

Creates a quantity representing a user-defined function.

> **Parameters**
> > **expression** – The function expression
>
> **Returns**
> > A new `Function` object

static `volume_fraction_of_a_phase`(*phase*)

Creates a quantity representing the volume fraction of a phase.

> **Parameters**
> > **phase** – The name of the phase or *ALL_PHASES* to choose all phases
>
> **Returns**
> > A new `VolumeFractionOfAPhase` object.

4.3 Package "system"

class +tc_toolbox.+system.**CompositionSet**(*phase_name*)

> Used by the method +tc_toolbox.system.SystemBuilder.with_new_composition_set() to enter two or more composition sets for a phase.
>
> > **Parameters**
> > > **phase_name** – The name of the phase for which a new composition set is required
>
> **CompositionSet**(*phase_name*)
>
> **set_major_constituents_for_sublattice**(*sublattice_index*, *major_constituents*)
>
> > Specify the new major constituent(s) for the sublattice.
> >
> > **Default**: If not specified, a default is automatically chosen based on the specified composition set.
> >
> > ---
> >
> > **Note:** This is useful in order to make calculations converge faster and more easily (because it may simplify giving start values when calculating the equilibrium as those phases with miscibility gaps should have different major constituents for each composition set). **The databases often set major constituents for several phases automatically when the data is retrieved.**
> >
> > ---
> >
> > > **Parameters**
> > >
> > > - **sublattice_index** – Index of the sublattice to set the major constituents for (starting with *1*)
> > >
> > > - **major_constituents** – Optional list of the major constituents, which must be selected from the phase constitution of the current system.
> > >
> > > **Returns**
> > > > This *CompositionSet* object

class +tc_toolbox.+system.**Element**(*back*)

> Represents an element, making detailed information about the element accessible.
>
> **Element**(*back*)
>
> > Constructs an instance of *Element*.
>
> **get_enthalpy**()
>
> > Returns the enthalpy of the element at 298 K, part of the stable element reference state (SER).
> >
> > > **Returns**
> > > > The enthalpy [J]
>
> **get_entropy_diff_0_to_298k**()
>
> > Returns the entropy difference 0 - 298 K of the element, part of the stable element reference state (SER).
> >
> > > **Returns**
> > > > The entropy difference 0 - 298 K [J/K]
>
> **get_molar_mass**()
>
> > Returns the molar mass of the element.
> >
> > > **Returns**
> > > > The molar mass [g/mol]

`get_name`()

> Returns the name of the element.
>
> > **Returns**
> > > The element name

`get_stable_element_reference`()

> Returns the stable element reference (i.e. the stable phase at 298.15 K and 1 bar, reference for all element thermodynamic data).
>
> > **Returns**
> > > The name of the stable element reference

`is_interstitial`()

> Returns if the element is interstitial.
>
> ---
>
> **Note:** In the diffusion simulations (DICTRA), the assumption that the volume is carried by the substitutional elements only is applied. The interstitial elements are assumed to have zero molar volumes.
>
> ---
>
> > **Returns**
> > > If the element is interstitial

`is_special`()

> Returns if the element is special (i.e. vacancies (VA) and electrons (denoted either as /- in gaseous, liquid or solid phases, or ZE in an aqueous solution phase)).
>
> > **Returns**
> > > If the element is special

`is_valid`()

> Returns if the element is valid. Non-valid elements are represented by an empty name.
>
> > **Returns**
> > > If the element is valid

class `+tc_toolbox.+system.MultiDatabaseSystemBuilder`(*back*)

> Used to select databases, elements, phases etc. and create a System object. The difference to the class System-Builder is that the operations are performed on all the previously selected databases. The system is then used to create calculations.

`MultiDatabaseSystemBuilder`(*back*)

> Constructs an instance of *MultiDatabaseSystemBuilder*.

`create_and_select_species`(*stoichiometry*)

> Specify a species from the already entered elements. The stoichiometry of the species is the chemical formula of the species. The created species will also be automatically selected.
>
> ---
>
> **Note:** The elements in the chemical formula are normally separated by stoichiometric numbers. Neither parenthesis *"()"* nor an underscore *"_"* is allowed in the chemical formula, while the special combination *"/-"* or *"/+"* can be used. Consult the Thermo-Calc database documentation for details about the syntax.
>
> ---
>
> > **Parameters**
> > > **stoichiometry** – The stoichiometry of the species

Returns
  This *MultiDatabaseSystemBuilder* object

deselect_constituent_on_sublattice(*phase_name*, *sublattice_no*, *constituent_name_to_deselect*)
  Rejects a constituent on a sublattice in a phase in both the thermodynamic and the kinetic database.

  Parameters

- **phase_name** – The name of the phase

- **sublattice_no** – The number of the sublattice (starting with 1)

- **constituent_name_to_deselect** – The name of the constituent to deselect

  Returns
    This *MultiDatabaseSystemBuilder* object

deselect_phase(*phase_name_to_deselect*)
  Rejects a phase for both the thermodynamic and the kinetic database.

  Parameters
    phase_name_to_deselect – The phase name

  Returns
    This *MultiDatabaseSystemBuilder* object

deselect_species(*species_name*)
  Removes the species from the system.

  Parameters
    species_name – The species

  Returns
    This *MultiDatabaseSystemBuilder* object

get_system()
  Creates a new System object that is the basis for all calculation types. Several calculation types can be defined later from the object; these are independent.

  Returns
    A new *System* object

select_constituent_on_sublattice(*phase_name*, *sublattice_no*, *constituent_name_to_select*)
  Selects a constituent on a sublattice in a phase in both the thermodynamic and the kinetic database.

  Note: Previously the third parameter *constituent_name_to_select* had a wrong name, it has been corrected in version 2021b.

  Parameters

- **phase_name** – The name of the phase

- **sublattice_no** – The number of the sublattice (starting with 1)

- **constituent_name_to_select** – The name of the constituent to select

  Returns
    This *MultiDatabaseSystemBuilder* object

select_phase(*phase_name_to_select*)

>   Selects a phase for both the thermodynamic and the kinetic database.

>   >   **Parameters**
>   >   >   **phase_name_to_select** – The phase name

>   >   **Returns**
>   >   >   This *MultiDatabaseSystemBuilder* object

select_species(*species_name*)

>   Adds the species to the system. Up to 1000 species can be defined in a single system.

>   >   **Parameters**
>   >   >   **species_name** – The species

>   >   **Returns**
>   >   >   This *MultiDatabaseSystemBuilder* object

with_new_composition_set(*composition_set*)

>   Used to enter two or more composition sets for a phase. If a phase has a miscibility gap it is necessary to have two composition sets, one for each possible composition that can be stable simultaneously.

>   The databases often create the typical composition sets for phases automatically when data are retrieved. The equilibrium calculations (using the default settings with global minimization) will usually add new composition sets if needed.

>   ---

>   **Note:** Precipitation and diffusion calculations can require the user to define additional composition sets. E.g. in the case where the new composition set is needed in the configuration of the calculation.

>   ---

>   >   **Parameters**
>   >   >   **composition_set** – the composition set

>   >   **Returns**
>   >   >   This *MultiDatabaseSystemBuilder* object

without_default_phases()

>   Rejects all the default phases from both the thermodynamic and the kinetic database, any phase now needs to be selected manually for the databases.

>   >   **Returns**
>   >   >   This *MultiDatabaseSystemBuilder* object

class +tc_toolbox.+system.**Phase**(*back*)

>   Represents a phase, making detailed information about the phase accessible.

>   **Phase**(*back*)

>   >   Constructs an instance of *Phase*.

>   **get_name**()

>   >   Returns the name of the phase.

>   >   >   **Returns**
>   >   >   >   The phase name

>   **get_species**()

>   >   Returns the species of the phase.

> **Returns**
>> A set containing the species

get_species_for_composition_profile()

Returns all species that need to be defined in a composition profile of the phase for diffusion simulations - except for one species that needs to be the dependent species.

Note: In a composition profile of a phase for diffusion simulations it is necessary to specify all non-stoichiometric and non-special species. In case of a DILUTE diffusion model, the database enforces the choice of a certain dependent species.

> **Returns**
>> Set with the species

get_sublattices()

Returns the sublattices of the phase in a well-defined contiguous order.

> **Returns**
>> A list containing the *Sublattice* objects

get_type()

Returns the type of the phase (liquid, ionic liquid, solid, gas).

> **Returns**
>> The type of a phase

has_diffusion_data()

Returns if diffusion data exists for the phase.

> **Returns**
>> If diffusion data exists for the phase

has_molar_volume_data()

Returns if molar volume data exists for the phase.

> **Returns**
>> If molar volume data exists for the phase

is_dilute_diffusion_model()

Returns if diffusion is described using the DILUTE model for the phase. This will always return *False* if no diffusion data is available.

> **Returns**
>> If the DILUTE model is used

is_gas()

Returns if the phase is a gas phase.

> **Returns**
>> If the phase is a gas phase

is_ionic_liquid()

Returns if the phase is an ionic liquid phase.

> **Returns**
>> If the phase is an ionic liquid phase

is_liquid()

> Returns if the phase is a liquid or ionic liquid phase.

> > **Returns**
> > > If the phase is a liquid phase

is_solid()

> Returns if the phase is a solid phase.

> > **Returns**
> > > If the phase is a solid phase

class +tc_toolbox.+system.**PhaseType**

> The type of a phase.

class +tc_toolbox.+system.**Species**(*back*)

> Represents a species, making detailed information about the species accessible.

> **Species**(*back*)

> > Constructs an instance of [*Species*](#).

> **get_all_elements**()

> > Returns all the elements that the species is composed of.

> > > **Returns**
> > > > List of all elements of the species and their stoichiometry

> **get_charge**()

> > Returns the charge of the species.

> > > **Returns**
> > > > The charge of the species

> **get_name**()

> > Returns the name of the species.

> > > **Returns**
> > > > The species name

> **is_element**()

> > Returns if the species actually represents an element.

> > > **Returns**
> > > > If the species represents an element

> **is_interstitial**()

> > Returns if the species is interstitial.

> > ---

> > **Note:** In the diffusion simulations (DICTRA), the assumption that the volume is carried by the substitutional elements only is applied. The interstitial elements are assumed to have zero molar volumes.

> > ---

> > > **Returns**
> > > > If the species is interstitial

> **is_special**()

> > Returns if the species is special (i.e. vacancies (VA) and electrons (denoted either as /- in gaseous, liquid or solid phases, or ZE in an aqueous solution phase)).

> **Returns**
>> If the species is special

is_valid()

> Returns if the species is valid. Non-valid species are represented by an empty name.

>> **Returns**
>>> If the species is valid

to_element()

> Returns the *Element* representation of the species - if the species actually represents an element.

>> **Returns**
>>> The *Element* object

class +tc_toolbox.+system.**Sublattice**(*back*)

> Represents a sublattice of a phase.

> **Sublattice**(*back*)

>> Constructs an instance of *Sublattice*.

> **get_constituents**()

>> Returns the constituents of the sublattice.

>>> **Returns**
>>>> A set containing the constituents

> **get_nr_of_sites**()

>> Returns the number of sites in the sublattice.

>>> **Returns**
>>>> A float number

class +tc_toolbox.+system.**System**(*back*)

> A system containing selections for databases, elements, phases etc.

> **Note:** For the defined system, different calculations can be configured and run. **Instances of this class should always be created from a SystemBuilder**.

> **Note:** The system object is **immutable**, i.e. it cannot be changed after is has been created. If you want to change the system, you must instead create a new one.

> **System**(*back*)

>> Constructs an instance of *System*.

> **convert_composition**(*input_composition*, *input_unit*, *output_unit*, *dependent_component*)

>> Provides conversion between composition units for any combination of chemical compounds. It is fast because no thermodynamic equilibrium calculation is involved.

>> **Syntax of the chemical compounds**: *"Al2O3", "FeO", "CO", "Fe", "C", …*

>> **Note:** It is not required that the chemical compounds are components of the database. The only requirement is that all elements are present in the database.

Parameters

- **input_composition** – Composition (for example: *{"Al2O3": 25.0, "FeO": 75.0}*)

- **input_unit** – Unit of the input composition

- **output_unit** – Requested output unit

- **dependent_component** – The dependent component (optional), for example: *"Fe"*. If no dependent component is specified the sum of the input composition needs to match 100% / 1

Returns

The composition in the requested output unit

get_all_elements_in_databases()

Returns the names of all elements present in the selected databases, regardless of the actual selection of elements.

Returns

A list of element names

get_all_phases_in_databases()

Returns all phase names present in the selected databases, regardless of selected elements, phases etc.

Returns

A list of phase names

get_all_species_in_databases()

Returns all species names present in the selected databases, regardless of the actual selection of elements, phases, ….

Returns

A list of species names

get_element_object(*element_name*)

Returns the *Element* object of an element. This can be used to obtain detailed information about the element.

Parameters

element_name – The element name

Returns

object

Return type

A *Element*

get_elements_in_system()

Returns the names of all elements present in the selected system.

Note: The list does not contain any elements or components that have been auto-selected by the database(s) in a calculator. Use the `get_components()` of the calculator object instead to get the complete information.

Returns

A list of element names

get_phase_object(*phase_name*)

> Returns the *Phase* object of a phase. This can be used to obtain detailed information about the phase.

> > **Parameters**
> > > **phase_name** – The phase name

> > **Returns**
> > > object

> > **Return type**
> > > A *Phase*

get_phases_in_system()

> Returns all phase names present in the system due to its configuration (selected elements, phases, etc.).

> > **Returns**
> > > A list of phase names

get_references()

> Provides a dictionary with database references per database in the selected system.

> > **Returns**
> > > The database references

get_species_in_system()

> Returns the names of all species present in the selected system.

> ---
> **Note:** The list does not contain any species or components that have been auto-selected by the database(s) in a calculator. Use the `get_components()` of the calculator object instead to get the complete information.
> ---

> > **Returns**
> > > The list of species names

get_species_object(*species_name*)

> Returns the *Species* object of an species. This can be used to obtain detailed information about the species.

> > **Parameters**
> > > **species_name** – The species name

> > **Returns**
> > > object

> > **Return type**
> > > A *Species*

get_system_data()

> Returns the content of the database. This can be used to modify the parameters and functions and to change the current system by using `with_system_modifications()`.

> ---
> **Note:** Parameters can only be read from unencrypted (i.e. *user*) databases loaded as **.tdb*-file.
> ---

> > **Returns**
> > > The system data

with_batch_equilibrium_calculation(*default_conditions*, *components*)

> Creates a batch-equilibrium calculation (a vectorized equilibrium calculation).

Note: Use this instead of looping if you want to calculate equilibria for a larger number of compositions and know the conditions in advance. This calculation type has improved performance when calculating a large number of equilibria when each individual calculations is quick. E.g. when evaluating single phase properties for thousands of compositions.

> **Parameters**
>
> - **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*
> - **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.
>
> **Returns**
> A new `BatchEquilibriumCalculation` object

with_cct_precipitation_calculation()

> Creates a CCT diagram calculation.
>
> **Returns**
> A new `PrecipitationCCTCalculation` object

with_isothermal_diffusion_calculation()

> Creates an isothermal diffusion calculation.
>
> **Returns**
> A new `DiffusionIsoThermalCalculation` object

with_isothermal_precipitation_calculation()

> Creates an isothermal precipitation calculation.
>
> **Returns**
> A new `PrecipitationIsoThermalCalculation` object

with_material_to_material()

> Provides access to all Material to Material calculations. The actual calculation needs to be chosen in the returned object.
>
> **Returns**
> A new `MaterialToMaterialCalculationContainer` object

with_non_isothermal_diffusion_calculation()

> Creates a non-isothermal precipitation calculation.
>
> **Returns**
> A new `PrecipitationNonIsoThermalCalculation` object

with_non_isothermal_precipitation_calculation()

> Creates a non-isothermal precipitation calculation.
>
> **Returns**
> A new `PrecipitationNonIsoThermalCalculation` object

with_phase_diagram_calculation(*default_conditions*, *components*)

Creates a phase diagram (map) calculation.

> **Parameters**
>
> - **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*
> - **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.
>
> **Returns**
> A new PhaseDiagramCalculation object

with_property_diagram_calculation(*default_conditions*, *components*)

Creates a property diagram (step) calculation.

> **Parameters**
>
> - **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*
> - **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.
>
> **Returns**
> A new PropertyDiagramCalculation object

with_property_model_calculation(*model*, *path_to_models*, *debug_model*)

Creates a Property Model calculation.

The parameter *debug_model* is only used when debugging self-developed models.

> **Parameters**
>
> - **model** – The Property Model to be calculated.
> - **path_to_models** – The path where the Property Models are installed. If no value is entered, the Property Models folder used by the normal Thermo-Calc application is used.
> - **debug_model** – Used when debugging self-developed models.
>
> **Returns**
> A new PropertyModelCalculation object

with_scheil_calculation()

Creates a Scheil solidification calculation.

> **Returns**
> A new ScheilCalculation object

with_single_equilibrium_calculation(*default_conditions*, *components*)

Creates a single equilibrium calculation.

> **Parameters**
>
> - **default_conditions** – If *True*, automatically sets the conditions *N=1* and *P=100000*
> - **components** – Specify here the components of the system (for example: *[AL2O3, …]*), *only necessary if they differ from the elements*. If this option is used, **all elements** of the system need to be replaced by a component.
>
> **Returns**
> A new SingleEquilibriumCalculation object

with_ttt_precipitation_calculation()

> Creates a TTT diagram calculation.

> > **Returns**
> >
> > > A new PrecipitationTTTCalculation object

class +tc_toolbox.+system.**SystemBuilder**(*back*)

> Used to select databases, elements, phases etc. and create a System object. The system is then used to create calculations.

> **SystemBuilder**(*back*)
>
> > Constructs an instance of *SystemBuilder*.

> **create_and_select_species**(*stoichiometry*)
>
> > Specify a species from the already entered elements. The stoichiometry of the species is the chemical formula of the species. The created species will also be automatically selected.
> >
> > ---
> >
> > **Note:** The elements in the chemical formula are normally separated by stoichiometric numbers. Neither parenthesis *"()"* nor an underscore *"_"* is allowed in the chemical formula, while the special combination *"/-"* or *"/+"* can be used. Consult the Thermo-Calc database documentation for details about the syntax.
> >
> > ---
> >
> > > **Parameters**
> > >
> > > > **stoichiometry** – The stoichiometry of the species
> > >
> > > **Returns**
> > >
> > > > This *SystemBuilder* object

> **deselect_constituent_on_sublattice**(*phase_name*, *sublattice_no*, *constituent_name_to_deselect*)
>
> > Rejects a constituent on a sublattice in a phase in the last specified database only.
> >
> > > **Parameters**
> > >
> > > > * **phase_name** – The name of the phase
> > > >
> > > > * **sublattice_no** – The number of the sublattice (starting with 1)
> > > >
> > > > * **constituent_name_to_deselect** – The name of the constituent to deselect
> > >
> > > **Returns**
> > >
> > > > This *SystemBuilder* object

> **deselect_phase**(*phase_name_to_deselect*)
>
> > Rejects a phase in the last specified database only.
> >
> > > **Parameters**
> > >
> > > > **phase_name_to_deselect** – The name of the phase
> > >
> > > **Returns**
> > >
> > > > This *SystemBuilder* object

> **deselect_species**(*stoichiometry*)
>
> > Removes the species from the system.
> >
> > > **Parameters**
> > >
> > > > **stoichiometry** – The species
> > >
> > > **Returns**
> > >
> > > > This *SystemBuilder* object

get_system()

Creates a new System object that is the basis for all calculation types. Several calculation types can be defined later from the object; these are independent.

> **Returns**
> > A new *System* object

select_constituent_on_sublattice(*phase_name*, *sublattice_no*, *constituent_name_to_select*)

Selects a constituent on a sublattice in a phase in the last specified database only.

Note: Previously the third parameter *constituent_name_to_select* had a wrong name, it has been corrected in version 2021b.

> **Parameters**
> > - **phase_name** – The name of the phase
> > - **sublattice_no** – The number of the sublattice (starting with 1)
> > - **constituent_name_to_select** – The name of the constituent to select
>
> **Returns**
> > This *SystemBuilder* object

select_database_and_elements(*database_name*, *list_of_element_strings*)

Selects a thermodynamic or kinetic database and its selected elements (that will be appended). After that, phases can be selected or unselected.

> **Parameters**
> > - **database_name** – The database name, for example "FEDEMO"
> > - **list_of_element_strings** – A list of one or more elements as strings, for example ["Fe", "C"]
>
> **Returns**
> > This *SystemBuilder* object

select_phase(*phase_name_to_select*)

Selects a phase in the last specified database only.

> **Parameters**
> > **phase_name_to_select** – The name of the phase
>
> **Returns**
> > This *SystemBuilder* object

select_species(*stoichiometry*)

Adds the species to the system. Up to 1000 species can be defined in a single system.

> **Parameters**
> > **stoichiometry** – The species
>
> **Returns**
> > This *SystemBuilder* object

select_user_database_and_elements(*path_to_user_database*, *list_of_element_strings*)

> Selects a thermodynamic database which is a user-defined database and select its elements (that will be appended).

> **Note:** By using a r-literal, it is possible to use slashes on all platforms, also on Windows: *select_user_database_and_elements(r"my path/user_db.tdb", ["Fe", "Cr"]])*

> **Note:** On Linux and Mac the path is case-sensitive, also the file ending.

> **Parameters**
>
> - **path_to_user_database** – The path to the database file ("database".TDB), defaults to the current working directory. Only the filename is required if the database is located in the same folder as the script.
>
> - **list_of_element_strings** – A list of one or more elements as strings, for example ["Fe", "C"]
>
> **Returns**
>> This *SystemBuilder* object

with_new_composition_set(*composition_set*)

> Used to enter composition sets for a phase. If a phase has a miscibility gap it is necessary to have two composition sets, one for each possible composition that can be stable simultaneously.
>
> **Parameters**
>> **composition_set** – The composition set
>
> **Returns**
>> This *SystemBuilder* object

without_default_phases()

> Rejects all default phases in the last specified database only, any phase needs now to be selected manually for that database.
>
> **Returns**
>> This *SystemBuilder* object

TROUBLESHOOTING

This section provides an FAQ for common problems that occur when using the TC-Toolbox for MATLAB®.

5.1 Diagnostics Script

If you have problems running TC-Toolbox, run the diagnostics script below.

```matlab
%{
    Run this script when troubleshooting TC-Toolbox

    It is important to run this script EXACTLY the same way as you run your MATLAB script

%}

clc

toolbox_version = "2024a";


disp("Testing TC-Toolbox toolbox_version: " + toolbox_version)
disp('Please make sure that the variable "toolbox_version" above, matches the release␣
↪that you want to test, if not change it and re-run this script.')

% below this line, nothing needs to be manually updated.

[matlab_version, matlab_release_data] = version;
fprintf("\n")
disp("MATLAB version: " + matlab_version)
fprintf("\n")


tc_env_variable = 'TC' + extractBetween(toolbox_version, 3, 5).upper() + '_HOME';
if isempty(getenv(tc_env_variable))
    fprintf(2, 'No Thermo-calc environment variable for ' + toolbox_version + ' was␣
↪found. (' + tc_env_variable + ')\n')
else
    disp(getenv(tc_env_variable))
end
fprintf("\n")
```

(continues on next page)

```matlab
disp('Url of license server: (if license server is NO-NET, you need a local license file)
↪')
if isempty(getenv("LSHOST"))
    disp('No Thermo-calc license server url was found. (LSHOST)')
else
    disp(getenv("LSHOST"))
end
fprintf("\n")

disp('Path to local license file: (only necessary if not using license server)')
if isempty(getenv("LSERVRC"))
    disp('No path to local license file was found. (LSERVRC)')
else
    disp(getenv("LSERVRC"))
end
fprintf("\n")

try
    session = tc_toolbox.TCToolbox();
catch e
    fprintf(2,'TC-Toolbox not properly installed !!!\n%s\n', e.message);
end

fprintf("\n")
disp('Lists the databases: (should be a complete list of the installed databases that
↪you have license for or do not require license)')
disp(transpose(session.get_databases()))

fprintf(1, 'Make sure no error messages were printed !\n\n')
```